FORMAL LOGICAL METHODS FOR SYSTEM SECURITY AND CORRECTNESS

NATO Science for Peace and Security Series

This Series presents the results of scientific meetings supported under the NATO Programme: Science for Peace and Security (SPS).

The NATO SPS Programme supports meetings in the following Key Priority areas: (1) Defence Against Terrorism; (2) Countering other Threats to Security and (3) NATO, Partner and Mediterranean Dialogue Country Priorities. The types of meeting supported are generally "Advanced Study Institutes" and "Advanced Research Workshops". The NATO SPS Series collects together the results of these meetings. The meetings are co-organized by scientists from NATO countries and scientists from NATO's "Partner" or "Mediterranean Dialogue" countries. The observations and recommendations made at the meetings, as well as the contents of the volumes in the Series, reflect those of participants and contributors only; they should not necessarily be regarded as reflecting NATO views or policy.

Advanced Study Institutes (ASI) are high-level tutorial courses to convey the latest developments in a subject to an advanced-level audience.

Advanced Research Workshops (ARW) are expert meetings where an intense but informal exchange of views at the frontiers of a subject aims at identifying directions for future action.

Following a transformation of the programme in 2006 the Series has been re-named and re-organised. Recent volumes on topics not related to security, which result from meetings supported under the programme earlier, may be found in the NATO Science Series.

The Series is published by IOS Press, Amsterdam, and Springer Science and Business Media, Dordrecht, in conjunction with the NATO Public Diplomacy Division.

Sub-Series

A.	Chemistry and Biology	Springer Science and Business Media
B.	Physics and Biophysics	Springer Science and Business Media
C.	Environmental Security	Springer Science and Business Media
D.	Information and Communication Security	IOS Press
E.	Human and Societal Dynamics	IOS Press

http://www.nato.int/science
http://www.springer.com
http://www.iospress.nl

Sub-Series D: Information and Communication Security – Vol. 14 ISSN 1874-6268

Formal Logical Methods for System Security and Correctness

Edited by

Orna Grumberg

Technion, Israel

Tobias Nipkow

Technische Universität München, Germany

and

Christian Pfaller

Technische Universität München, Germany

IOS Press

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Published in cooperation with NATO Public Diplomacy Division

Proceedings of the NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness
Marktoberdorf, Germany
31 July–12 August 2007

ISBN 978-1-58603-843-4
Library of Congress Control Number: 2008922610

Publisher
IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

Distributor in the UK and Ireland
Gazelle Books Services Ltd.
White Cross Mills
Hightown
Lancaster LA1 4XS
United Kingdom
fax: +44 1524 63232
e-mail: sales@gazellebooks.co.uk

Distributor in the USA and Canada
IOS Press, Inc.
4502 Rachael Manor Drive
Fairfax, VA 22032
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

PRINTED IN THE NETHERLANDS

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Preface

The objective of our Summer School 2007 on *Formal Logical Methods for System Security and Correctness* was to present the state-of-the-art in the field of proof technology in connection with secure and correct software. The lecturers have shown that methods of correct-by-construction program and process synthesis allow a high level programming method more amenable to security and reliability analysis and guarantees. By providing the necessary theoretical background and presenting corresponding application oriented concepts, the objective was an in-depth presentation of such methods covering both theoretical foundations and industrial practice. In detail the following courses were given:

GILLES BARTHE lectured on *Verification Methods for Software Security and Correctness*. The objective of the lectures was to present static enforcement mechanisms to ensure reliability and security of mobile code. First, he introduced a type based verifier for ensuring information flow policies and a verification condition generator for Java bytecode programs. He also described how these mechanisms have been certified using the proof assistant *Coq*. Second, he related these two enforcement mechanisms to their counterparts for Java programs.

ROBERT CONSTABLE's lectures *Logical Foundations of Computer Security* were concerned with developing correct-by-construction security protocols for distributed systems on communication networks. He used computational type theory to express logically sound cryptographic services and established them by machine generated formal proofs.

In his course *Building a Software Model-Checker* JAVIER ESPARZA introduced *jMoped*, a tool for the analysis of Java programs. He then explained the theory and algorithms behind the tool. In jMoped is assumed that variables have a finite range. He started by considering the computational complexity of verifying different classes of programs satisfying this constraint. After choosing a reasonable class of programs, he introduced a model-checking algorithm based on pushdown automata and then addressed the problem of data. He presented an approach to this problem based on BDDs and counterexample-based abstraction refinement with interpolants.

With *Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation* ORNA GRUMBERG presented a powerful model checking technique called Symbolic Trajectory Evaluation (STE), which is particularly suitable for hardware. STE is applied to a circuit M, described as a graph over nodes (gates and latches). The specification consists of assertions in a restricted temporal language. The assertions are of the form $A \Longrightarrow C$, where the antecedent A expresses constraints on nodes n at different times t, and the consequent C expresses requirements that should hold on such nodes (n, t). Abstraction in STE is derived from the specification by initializing all inputs not appearing in A to the X ("unknown") value. A refinement amounts to changing the assertion in order to present node values more accurately. A symbolic simulation and the specific type of abstraction, used in STE, was described. We proposed a technique for automatic refinement of assertions in STE, in case the model checking results in X. In

this course the notion of hidden vacuity for STE was defined and several methods for detecting it was suggested.

JOHN HARRISON lectured on *Automated and Interactive Theorem Proving*. He covered a range of topics from Boolean satisfiability checking (SAT), several approaches to first-order automated theorem proving, special methods for equations, decision procedures for important special theories, and interactive proofs. He gave some suitable references.

MARTIN HOFMANN gave a series of lectures on *Correctness of Effect-based Program Transformations* in which a type system was considered capable of tracking reading, writing and allocation in a higher-order language with dynamically allocated references. He gave a denotational semantics to this type system which allowed us to validate a number of effect-dependent program equivalences in the sense of observational equivalence. On the way we learned popular techniques such as parametrised logical relations, regions, admissible relations, etc which belong to the toolbox of researchers in principles of programming languages.

Abstract and Concrete Models of Recursion was the course of MARTIN HYLAND. Systems of information flow are fundamental in computing systems generally and security protocols in particular. One key issue is feedback (or recursion) and we developed an approach based on the notion of trace. He covered applications to fixed point theory, automata theory and topics in the theory of processes.

In his course *Security Analysis of Network Protocols* JOHN MITCHELL provided an introduction to network protocols that have security requirements. He covered a variety of contemporary security protocols and gave students information needed to carry out case studies using automated tools and formal techniques. The first lectures surveyed protocols and their properties, including secrecy, authentication, key establishment, and fairness. The second part covered standard formal models and tools used in security protocol analysis, and described their advantages and limitations.

With his lectures on *The Engineering Challenges of Trustworthy Computing* GREG MORRISETT talked about a range of language, compiler, and verification techniques that can be used to address safety and security issues in systems software today. Some of the techniques, such as software fault isolation, are aimed at legacy software and provide relatively weak but important guarantees, and come with significant overhead. Other techniques, such as proof-carrying code, offer the potential of fine-grained protection with low overhead, but introduce significant verification challenges.

The focus of TOBIAS NIPKOW's lecture series *Verified Decision Procedures for Linear Arithmetic* was on decision procedures for linear arithmetic (only $+$, no $*$) and their realization in foundational theorem provers. Although we used arithmetic for concreteness, the course was also a general introduction of how to implement arbitrary decision procedures in foundational provers. The course focused on two well-known quantifier elimination algorithms (and hence decision procedures) for linear arithmetic: Fourier-Motzkin elimination, which is complete for rationals and reals, and Cooper's method, which is complete for the integers.

In his series of lectures *Proofs with Feasible Computational Content* HELMUT SCHWICHTENBERG considered logical propositions concerning data structures. If such a proposition involves (constructive) existential quantifiers in strictly positive positions, then—according to Brouwer, Heyting and Kolmogorov—it can be seen as a computational problem. A (constructive) proof of the proposition then provides a solution to this

problem, and one can machine extract (via a realizability interpretation) this solution in the form of a lambda calculus term, which can be seen as a functional program. He concentrated on the question how to control at the proof level the complexity of the extracted programs.

STAN WAINER lectured on *Proof Systems, Large Functions and Combinatorics*. Proof-theoretic bounding functions turn out to have surprisingly deep connections with finitary combinatorics. These four lectures showed how the power of Peano Arithmetic, and various natural fragments of it, is precisely delineated by variants of the Finite Ramsey Theorem.

The contributions in this volume have emerged from these lectures of the 28th International Summer School at Marktoberdorf from July 31 to August 12, 2007. About 90 participants from 30 countries attended—including students, lecturers and staff. The Summer School provided two weeks of learning, discussion and development of new ideas, and was a fruitful event, at both the professional and social level.

We would like to thank all lecturers, staff, and hosts in Marktoberdorf. In particular special thanks goes to Dr. Katharina Spies, Silke Müller, and Sonja Werner for their great and gentle support.

The Marktoberdorf Summer School was arranged as an Advanced Study Institute of the *NATO Science for Peace and Security Programme* with support from the town and county of Marktoberdorf and the *Deutscher Akademischer Austausch Dienst (DAAD)*. We thank all authorities involved.

THE EDITORS

Contents

Preface v

Compilation of Certificates 1
Gilles Barthe, Benjamin Grégoire and Tamara Rezk

Formal Foundations of Computer Security 29
Mark Bickford and Robert Constable

Building a Software Model Checker 53
Javier Esparza

Symbolic Trajectory Evaluation (STE): Automatic Refinement and Vacuity Detection 89
Orna Grumberg

Automated and Interactive Theorem Proving 111
John Harrison

Correctness of Effect-Based Program Transformations 149
Martin Hofmann

Abstract and Concrete Models for Recursion 175
Martin Hyland

Secrecy Analysis in Protocol Composition Logic 199
Arnab Roy, Anupam Datta, Ante Derek, John C. Mitchell and Jean-Pierre Seifert

The Engineering Challenges of Trustworthy Computing 233
Greg Morrisett

Reflecting Quantifier Elimination for Linear Arithmetic 245
Tobias Nipkow

Content in Proofs of List Reversal 267
Helmut Schwichtenberg

Proof Theory, Large Functions and Combinatorics 287
Stanley S. Wainer

Author Index 319

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Compilation of Certificates

Gilles BARTHE, Benjamin GRÉGOIRE and Tamara REZK
INRIA Sophia-Antipolis Méditerranée, France

Abstract. The purpose of the course is to stress the importance of verification methods for bytecode, and to establish a strong relation between verification at source and bytecode levels. In order to illustrate this view, we shall consider the example of verification condition generation, which underlies many verification tools and plays a central role in Proof Carrying Code infrastructure, and information flow type systems, that are used to enforce confidentiality policies in mobile applications.

Keywords. Information flow typing, program verification, preservation of typing, preservation of proof obligations, proof carrying code

1. Introduction

Reliability and security of executable code is an important concern in many application domains, and in particular mobile code where code consumers run code that originate from untrusted and potentially malicious producers. While many formal techniques and tools have been developed to address this concern, it is particularly striking to notice that many of them operate on source programs, rather than on executable code; the extended static checking tool ESC/Java [8] and the information flow aware language Jif [17] are prominent examples of environments that provide guarantees about source (Java) programs. However, source code verification is not appropriate in the context of mobile code, where code consumers need automatic and efficient verification procedures that can be run locally on executable code and that dispense them from trusting code producers (that are potentially malicious), networks (that may be controlled by an attacker), and compilers (that may be buggy).

Proof Carrying Code (PCC) [20,18,19] is a security architecture for mobile code that targets automated verification of executable code and therefore does not require trust in the code producer, nor in the compiler, nor in the network. In a typical PCC architecture, programs are compiled with a certifying compiler that returns, in addition to executable code, evidence that the code satisfies a desired policy. The evidence is provided in the form of formal objects, that can be used by automatic verification procedures to verify independently that the compiled code is indeed compliant to the policy. Typically, a certifying compiler will generate both program annotations, which specify loop invariants tailored towards the desired policy, as well as proof objects, a.k.a. certificates, that the program is correct w.r.t. its specification. Early work on PCC is based on verification condition generation and exploits the Curry-Howard isomorphism to reduce proof checking to type checking; thus, upon reception of an annotated program with a certificate, the code consumer will automatically extract a set of verification conditions $\phi_1 \dots \phi_n$ using

a verification condition generator and will establish the validity of these conditions using the certificate, which should be a tuple $(M_1, \ldots, M_n)$ of λ-terms such that $M_i : \phi_i$ for $i = 1 \ldots n$.

The idea of certifying compilation is also present in typed low-level languages, where the certificates take the form of type annotations, and certificate checkers are type systems that reject all executable code that does not comply with the consumer policy. Typed assembly languages [23] and the Java Virtual Machine (JVM) [16] provide two well-known examples of typed low-level languages, in which programs come equipped with type information. In the case of the JVM, the type annotations refer to the signature of methods, the type of fields and local variables, *etc*. These type annotations are used by the bytecode verifier, which performs a dataflow analysis to ensure adherence to the JVM safety policy (no arithmetic on references, no stack underflow or overflow, correct initialization of objects before accessing them, *etc*) [15]. Lightweight bytecode verification [21] extends the idea of bytecode verification by requiring that programs also come with additional typing information (concretely the stack type at at junction points) in order to enable a verification procedure that analyzes the program in one pass.

Through their associated verification mechanisms for executable code, infrastructures based on certifying compilers and typed low-level languages suitably address the security concerns for mobile code. Nevertheless, current instances of certifying compilers mostly focus on basic safety policies and do not take advantage of the existing methods for verifying source code. Ideally, one would like to develop expressive verification methods for executable code and to establish their adequacy with respect to verification methods for source programs, so as to be able to transfer evidence from source programs to executable code. The purpose of these notes is to present two exemplary enforcement mechanisms for executable code, and to show how they connect to similar enforcement mechanisms for source code.

The first mechanism aims at ensuring information flow policies for confidentiality: it is a type-based mechanism, compatible with the principles of bytecode verification. We show that the type system is sound, i.e. enforce non-interference of typable programs, and that source programs that are typable in a standard type system for information flow are compiled into programs that are typable in our type system. The benefits of type preservation are two-fold: they guarantee program developers that their programs written in an information flow aware programming language will be compiled into executable code that will be accepted by a security architecture that integrates an information flow bytecode verifier. Conversely, they guarantee code consumers of the existence of practical tools to develop applications that will provably meet the policy enforced by their information flow aware security architecture.

The second mechanism aims at ensuring adherence of programs to a logical specification that establishes their functional correctness or their adherence to a given policy: it is a verification condition generator for programs specified with logical annotations (pre-conditions, post-conditions, etc) compatible with Proof Carrying Code. We show that the verification condition generator is sound, i.e. a program meets its specification, expressed as a pre- and post-condition, provided all verification conditions are valid, and that verification conditions are preserved by compilation. Since verification conditions for source programs and their compilation are syntactically equal (and not merely logically equivalent), one can reuse directly certificates for source programs and bundle them with the compiled program. Preservation of proof obligations provide benefits similar to

operations	$op ::= + \mid - \mid \times \mid /$	
comparisons	$cmp ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$	
expressions	$e ::= x \mid c \mid e\ op\ e$	
tests	$t ::= e\ cmp\ e$	
instructions	$i ::= x := e$	assignment
	$\mid$ $\mathsf{if}(t)\{i\}\{i\}$	conditional
	$\mid$ $\mathsf{while}(t)\{i\}$	loop
	$\mid$ $i; i$	sequence
	$\mid$ skip	skip
	$\mid$ $\mathsf{return}\ e$	return value

where $c \in \mathbb{Z}$ and $x \in \mathcal{X}$.

Figure 1. INSTRUCTION SET FOR BASIC LANGUAGE

preservation of typing and extends the applicability of PCC by offering a means to transfer evidence from source code to executable code and thus to certify complex policies of executable code using established verification infrastructure at source level.

Contents The relation between source verification and verification of executable code is established in the context of a small imperative and assembly languages, and for a non-optimizing compiler, all presented in Section 2. Section 3 is devoted to information flow whereas Section 4 is devoted to verification condition generation. Section 5 discusses the impact of program optimizations on our results, and mentions the main difficulties in extending our results to more realistic settings.

2. Setting

Although the results presented in these notes have been developed for a sequential fragment of the Java Virtual Machine that includes objects, exceptions, and method calls (see Section 5), we base our presentation on a simple imperative language, which is compiled to a stack-based virtual machine.

This section introduces the syntax and the semantics of these simple source and bytecode languages. In addition, we define a non-optimizing compiler, which in the later sections will be shown to preserve information flow typing as well as verification conditions.

Both the source and bytecode languages use named variables taken from a fixed set $\mathcal{X}$, and manipulate memories, i.e. mappings from variables to values. In our setting, values are just integers, thus a memory ρ has type $\mathcal{X} \to \mathbb{Z}$. We denote by $\mathcal{L}$ the set of memories.

2.1. The source language: IMP

Programs Figure 1 defines the basic source language IMP. We let $\mathcal{E}$ be the set of expressions, and $\mathcal{I}$ be the set of instructions. In this language, a program p is simply an instruction followed by a return (i.e. $p = i; \mathsf{return}\ e$).

$$\frac{}{x \overset{\rho}{\hookrightarrow} \rho(x)} \qquad \frac{}{c \overset{\rho}{\hookrightarrow} c} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1 \; op \; e_2 \overset{\rho}{\hookrightarrow} v_1 \; op \; v_2} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1 \; cmp \; e_2 \overset{\rho}{\hookrightarrow} v_1 \; cmp \; v_2}$$

$$\frac{e \overset{\rho}{\hookrightarrow} v}{[\rho,\, x := e] \Downarrow_{\mathcal{S}} \rho \oplus \{x \leftarrow v\}} \qquad \frac{}{[\rho,\, \mathsf{skip}] \Downarrow_{\mathcal{S}} \rho} \qquad \frac{[\rho,\, i_1] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, i_2] \Downarrow_{\mathcal{S}} \rho''}{[\rho,\, i_1; i_2] \Downarrow_{\mathcal{S}} \rho''}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [\rho,\, i_t] \Downarrow_{\mathcal{S}} \rho'}{[\rho,\, \mathsf{if}(t)\{i_t\}\{i_f\}] \Downarrow_{\mathcal{S}} \rho'} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false} \quad [\rho,\, i_f] \Downarrow_{\mathcal{S}} \rho'}{[\rho,\, \mathsf{if}(t)\{i_t\}\{i_f\}] \Downarrow_{\mathcal{S}} \rho'}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [\rho,\, i] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho''}{[\rho,\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho''} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false}}{[\rho,\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho}$$

Figure 2. SEMANTICS OF THE BASIC LANGUAGE

instruction $i ::=$	$\mathsf{push}\ c$	push value on top of stack
$\mid$	$\mathsf{binop}\ op$	binary operation on stack
$\mid$	$\mathsf{load}\ x$	load value of x on stack
$\mid$	$\mathsf{store}\ x$	store top of stack in variable x
$\mid$	$\mathsf{goto}\ j$	unconditional jump
$\mid$	$\mathsf{if}\ cmp\ j$	conditional jump
$\mid$	return	return the top value of the stack

where $c \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}$.

Figure 3. INSTRUCTION SET FOR THE BASIC BYTECODE

Operational semantics States consist of an instruction and a memory. Thus, the set $\mathrm{State}_{\mathcal{S}} = \mathcal{I} \times \mathcal{L}$ of states is defined as the set of pairs of the form $[\rho,\, i]$, where i is an instruction.

Figure 2 presents the big step semantics of the basic source language. The first relation $\overset{\rho}{\hookrightarrow} \subseteq (\mathcal{E} \times \mathcal{L}) \times \mathbb{Z}$ defines the evaluation under a memory ρ of an expression e into a value. Abusing notation, we use the same syntax for the evaluation of tests. The second relation $\Downarrow_{\mathcal{S}} \subseteq \mathrm{State}_{\mathcal{S}} \times \mathcal{L}$ defines the big-step semantics of an instruction i as a relation between an initial memory and a final memory. There is no rule for the return, as it only appears at the end of the program. We rather define the semantics of programs directly with the clause:

$$\frac{p = i; \mathsf{return}\ e \quad [\rho_0,\, i] \Downarrow_{\mathcal{S}} \rho \quad e \overset{\rho}{\hookrightarrow} v}{p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v}$$

$$\frac{\dot{p}[k] = \mathsf{push}\ c}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k+1,\ \rho,\ c :: os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{binop}\ op \quad v = v_1\ op\ v_2}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\ \rho,\ v :: os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{load}\ x}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k+1,\ \rho,\ \rho(x) :: os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{store}\ x}{\langle k,\ \rho,\ v :: os\rangle \rightsquigarrow \langle k+1,\ \rho \oplus \{x \leftarrow v\},\ os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{if}\ cmp\ j \quad v_1\ cmp\ v_2}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\ \rho,\ os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{if}\ cmp\ j \quad \neg(v_1\ cmp\ v_2)}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle j,\ \rho,\ os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{goto}\ j}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle j,\ \rho,\ os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{return}}{\langle k,\ \rho,\ v :: os\rangle \rightsquigarrow \rho, v}$$

Figure 4. SEMANTICS OF THE BASIC BYTECODE

2.2. *The Virtual Machine :* VM

Programs A bytecode program $\dot{p}$ is an array of instructions (defined in figure 3). We let $\mathcal{P}$ be the set of program points, i.e. $\mathcal{P} = \{0 \ldots n-1\}$ where n is the length of $\dot{p}$. Instructions act on the operand stack (push and load, binop perform an operation with the two top elements, store saves the top element in a variable) or on the control flow (goto for an unconditional jump and if for a conditional jump). Note that, unlike source programs, return instructions may arise anywhere in the code. We nevertheless assume that the program is well-formed, i.e. that the last instruction of the program is a return.

Operational semantics A bytecode state is a triple $\langle k,\ \rho,\ os\rangle$ where k is a program counter, i.e. an element of $\mathcal{P}$, ρ is a memory, and os the operand stack that contains intermediate values needed by the evaluation of source language expressions. We note $\mathrm{State}_{\mathcal{B}}$ the set of bytecode states.

The small-step semantics of a bytecode program is given by the relation $\rightsquigarrow \subseteq \mathrm{State}_{\mathcal{B}} \times (\mathrm{State}_{\mathcal{B}} + \mathcal{L} \times \mathbb{Z})$ which represents one step of execution. Figure 4 defines this relation.

The reflexive and transitive closure $\rightsquigarrow^* \subseteq \mathrm{State}_{\mathcal{B}} \times \mathrm{State}_{\mathcal{B}}$ of $\rightsquigarrow$ is inductively defined by

$$\frac{}{\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k,\ \rho,\ os\rangle} \qquad \frac{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k',\ \rho',\ os'\rangle \quad \langle k',\ \rho',\ os'\rangle \rightsquigarrow^* \langle k'',\ \rho'',\ os''\rangle}{\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k'',\ \rho'',\ os''\rangle}$$

Finally, the evaluation of a bytecode program $\dot{p} : \rho_0 \Downarrow \rho, v$, from an initial memory ρ_0 to a final memory ρ and a return value v is defined by

$$\frac{\langle 0,\ \rho_0,\ \emptyset\rangle \rightsquigarrow^* \langle k,\ \rho,\ os\rangle \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow \rho, v}{\dot{p} : \rho_0 \Downarrow \rho, v}$$

Remark that the last transition step is necessary done by a return instruction so the memory is unchanged.

Compilation of expressions

$$
\begin{aligned}
[\![x]\!] &= \mathsf{load}\ x \\
[\![c]\!] &= \mathsf{push}\ c \\
[\![e_1\ op\ e_2]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op
\end{aligned}
$$

Compilation of instructions

$$
\begin{aligned}
k : [\![\mathsf{skip}]\!] &= \\
k : [\![x := e]\!] &= [\![e]\!];\ \mathsf{store}\ x \\
k : [\![i_1; i_2]\!] &= k : [\![i_1]\!]; k_2 : [\![i_2]\!] \\
&\text{where } k_2 = k + |[\![i_1]\!]| \\
k : [\![\mathsf{return}\ e]\!] &= [\![e]\!];\ \mathsf{return} \\
k : [\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ l;\ k_2 : [\![i_2]\!] \\
&\text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
&\qquad k_2 = k_1 + |[\![i_1]\!]| + 1 \\
&\qquad l = k_2 + |[\![i_2]\!]| \\
k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\
&\text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
&\qquad k_2 = k_1 + |[\![i]\!]| + 1
\end{aligned}
$$

Figure 5. COMPILATION SCHEME

2.3. The compiler

The compiler from the source language to the bytecode language is defined in Figure 5. Compilation of expression $[\![e]\!]$ generates a bytecode sequence which evaluate e and store/push the result on the top of the operand stack. For the compilation of instructions $k : [\![i]\!]$ the compiler argument k indicates the starting position of the resulting bytecode sequence.

Compilation of an assignment $x := e$ is the compilation of the expression e followed by a store x. At the end of the evaluation of $[\![e]\!]$ the value of e is on the top of the operand stack, then a store x instruction stores this value in the variable x and pop the value from the stack.

The compilation of a conditional $k : [\![\mathsf{if}(e_1\ op\ e_2)\{i_1\}\{i_2\}]\!]$ starts by the sequence corresponding to the evaluation of the two expressions e_2 and e_1. After this sequence the operand stack contains on the top the values of e_1 and e_2, the if $cmp\ k_2$ instruction evaluates the comparison and pop the two value from the stack. If the test is true the evaluation continues at label k_1 corresponding to the beginning of the true branch, if the

test is false the if instruction jumps to label k_2 to the beginning of the false branch. At the end of the true branch a goto instruction jumps to the code of the false branch.

The compilation of a loop $k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!]$ evaluates the two expressions e_2 and e_1 and then performs a conditional jump. If the test is false the evaluation jumps to the code corresponding to the body of the loop, if the test is true the evaluation continue by the evaluation of the loop body and then perform a jump to the label corresponding to the beginning of the evaluation of the test.

Finally the compilation of a program $p = (i; \mathsf{return}\ e)$ is defined by:

$$[\![p]\!] \stackrel{\text{def}}{\equiv} 0 : [\![i]\!];\ [\![e]\!];\ \mathsf{return}$$

2.3.1. Correctness

The compiler is correct in the sense that the semantics of a source program is the same as its compiled version. In other words, for all source program p and initial memory ρ_0, executing p with initial memory ρ_0 terminates with final memory ρ and return value v, iff executing $[\![p]\!]$ with initial memory ρ_0 terminates with final memory ρ and return value v

The correctness proof of compiler is done by exhibiting a strong form of simulation between the evaluation of the source program and the evaluation of the bytecode program. In order to carry the proof, we define a new notion of reduction for bytecode states: $s \rightsquigarrow^n s'$, which stands for s evaluates to s' in exactly n steps of reduction $\rightsquigarrow$. Remark that the relation $s \rightsquigarrow^* s'$ can be defined by $\exists n,\ s \rightsquigarrow^n s'$.

Lemma 2.1 *For all bytecode program $\dot{p}$, expression e, memory ρ and operand stack os such that $l = |[\![e]\!]|$ and $\dot{p}[k..k+l-1] = [\![e]\!]$ the following proposition holds:*

$$\begin{array}{l}\forall\ n\ k'\ os',\\ \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow^n \langle k',\ \rho,\ os'\rangle \wedge k' \geq k + l \Rightarrow\\ \qquad \exists\ v\ n' > 0, \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n'} \langle k + l,\ \rho,\ v :: os\rangle \rightsquigarrow^{n-n'} \langle k',\ \rho,\ os'\rangle\end{array}$$

The proof is by induction over e. The base cases are trivial. The case of binary expressions is proved using the induction hypothesis.

Lemma 2.2 (Correctness for expressions) *For all bytecode program $\dot{p}$, expression e, value v, memory ρ and operand stack os such that $l = |[\![e]\!]|$ and $\dot{p}[k..k+l-1] = [\![e]\!]$*

$$e \stackrel{\rho}{\hookrightarrow} v \Longleftrightarrow \langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k + l,\ \rho,\ v :: os\rangle$$

The proof is by induction over e. The only difficult case is the $\Longleftarrow$ for binary expressions. We have $e = e_1\ op\ e_2$ and $[\![e]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op$ and

$$\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k + l,\ \rho,\ v :: os\rangle$$

Using the previous lemma there exists v_1 and v_2 such that $v = v_1 + v_2$ and

$$\begin{array}{l}\langle k,\ \rho,\ os\rangle \rightsquigarrow^*\\ \langle k + |[\![e_2]\!]|,\ \rho,\ v_2 :: os\rangle \rightsquigarrow^*\\ \langle k + |[\![e_2]\!]| + |[\![e_1]\!]|,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow^*\\ \langle k + l,\ \rho,\ v :: os\rangle\end{array}$$

We conclude using the induction hypothesis.

Lemma 2.3 *For all bytecode program $\dot{p}$, instruction i which is not a* skip*, memory ρ and operand stack os such that $l = |[\![k]\!]|$ and $\dot{p}[k..k+l-1] = k : [\![i]\!]$ the following proposition holds:*

$$\begin{array}{l}\forall\ n\ k'\ \rho'\ os\ os',\\ \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n} \langle k',\ \rho',\ os'\rangle \wedge k' \geq k + l \Rightarrow\\ \quad\quad \exists\ \rho''\ n' > 0, \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n'} \langle k + l,\ \rho'',\ os\rangle \rightsquigarrow^{n-n'} \langle k',\ \rho',\ os'\rangle\end{array}$$

The proof is done using a general induction principle on n (i.e the induction hypothesis can be applied to any $n' < n$) and by case analysis on i. Remark that this lemma can be proved with $os = \emptyset$ due to the fact that the compiler maintains the invariant that the operand stack is empty at the beginning and at the end of the evaluation of all instructions. This invariant is used in the proof of the next lemma.

Lemma 2.4 (Correctness for instructions) *For all bytecode program $\dot{p}$, instruction i, memories ρ and ρ' such that $l = |[\![i]\!]|$ and $\dot{p}[k..k + l - 1] = k : [\![i]\!]$*

$$[\rho,\ i] \Downarrow_{\mathcal{S}} \rho' \Longleftrightarrow \langle k,\ \rho,\ \emptyset\rangle \rightsquigarrow^{*} \langle k + l,\ \rho',\ \emptyset\rangle$$

The direction $\Longrightarrow$ is done by induction on the evaluation of i (i.e one the derivation of $[\rho,\ i] \Downarrow_{\mathcal{S}} \rho'$) and presents no difficulty. To prove the direction $\Longleftarrow$ we prove that :

$$\forall n,\ \langle k,\ \rho,\ \emptyset\rangle \rightsquigarrow^{n} \langle k + l,\ \rho',\ \emptyset\rangle \Longrightarrow [\rho,\ i] \Downarrow_{\mathcal{S}} \rho'$$

The proof is done using a general induction principle on n and by case analysis on i. The cases of loop and conditional use the previous lemma.

Proposition 2.5 (Correctness of the compiler) *For all source program p, initial memory ρ_0, final memory ρ and return value v,*

$$p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v \Longleftrightarrow [\![p]\!] : \rho_0 \Downarrow \rho, v$$

This is a direct application of the above lemmas.

3. Information flow

Confidentiality (also found in the literature as privacy or secrecy) policies aim to guarantee that an adversary cannot access information considered as secret. Thus, confidentiality is not an absolute concept but is rather defined relative to the observational capabilities of the adversary. In these notes, we assume that the adversary cannot observe or modify intermediate memories, and cannot distinguish if a program terminates or not. In addition, we consider that information is classified either as public, and thus visible by adversary, or secret. We refer to [22] for further details on information flow.

3.1. Security policy

In this section, we specify formally (termination insensitive) *non-interference* [7,13], a baseline information flow policy, which assumes that an adversary can read the public inputs and outputs of a run, and which ensures that adversaries cannot deduce the value of secret inputs from observing the value of public outputs. In its more general form, non-interference is expressed relative to a lattice of security levels. For the purpose of these notes, we consider a lattice with only two security levels, and let $\mathcal{S} = \{L, H\}$ be the set of security levels, where H (high) and L (low) respectively correspond to secret and public information; one provides a lattice structure by adding the subtyping constraint $L \leq H$.

A policy is a function that classifies all variables as low or high.

Definition 3.1 (Policy) *A policy is a mapping* $\Gamma : \mathcal{X} \to \mathcal{S}$.

In order to state the semantic property of non-interference, we begin by defining when two memories are indistinguishable from the point of view of an adversary.

Definition 3.2 (Indistinguishability of memories) *Two memories* $\rho, \rho' : \mathcal{L}$ *are indistinguishable w.r.t. a policy* Γ*, written* $\rho \sim_\Gamma \rho'$ *(or simply* $\rho \sim \rho'$ *when there is no ambiguity), if* $\rho(x) = \rho'(x)$ *for all* $x \in \mathcal{X}$ *such that* $\Gamma(x) = L$.

One can think about the adversary as a program with access to only low parts of the memory and that is put in sequence with the code that manipulates secret information. Its goal is to distinguish between two different executions starting with indistinguishable memories ρ_1 and ρ_2. This is stated in the following definition.

Definition 3.3 (Non-interfering program) *A bytecode program* $\dot{p}$ *is* non-interfering *w.r.t. a policy* Γ*, if for every* $\rho_1, \rho_1', \rho_2, \rho_2', v_1, v_2$ *such that* $\dot{p} : \rho_1 \Downarrow \rho_1', v_1$ *and* $\dot{p} : \rho_2 \Downarrow \rho_2', v_2$ *and* $\rho_1 \sim \rho_2$*, we have* $\rho_1' \sim \rho_2'$ *and* $v_1 = v_2$.

The definition of non-interference applies both to bytecode programs, as stated above, and source programs. Note moreover that by correctness of the compiler, a source program p is non-interfering iff its compilation $[\![p]\!]$ is non-interfering.

3.2. Examples of insecure programs

This section provides examples of insecure programs that must be rejected by a type system. For each example, we provide the source program and its compilation. In all examples, x_L is a low variable and y_H is a high variable.

Our first example shows a direct flow of information, when the result of some secret information is copied directly into a public variable. Consider the program $x_L := y_H; \mathsf{return}\ 0$ and its compilation in Figure 6(a). The program stores in the variable x_L the value held in the variable y_H, and thus leaks information.

Our second example shows an indirect flow of information, when assignments to low variables within branching instructions that test on secret information leads to information leakages. Consider the program $\mathsf{if}(y_H = 0)\{x_L := 0\}\{x_L := 1\}; \mathsf{return}\ 0$ and its compilation in Figure 6(b). The program yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Indeed, the final value of x_L depends on the

1 load y_H	1 push 0	1 push 0	1 push 0
2 store x_L	2 load y_H	2 load y_H	2 load y_H
3 push 0	3 if $=$ 7	3 if $=$ 6	3 if $=$ 6
4 return	4 prim 0	4 push 0	4 push 0
	5 store x_L	5 return	5 return
	6 goto 9	6 push 0	6 push 1
	7 prim 1	7 store x_L	7 return
	8 store x_L	8 push 0	
	9 push 0	9 return	
	10 return		

Figure 6. EXAMPLES OF INSECURE PROGRAMS

initial value of y_H. The problem is caused by an assignment to x_L in the scope of an if instruction depending on high variables.

Our third example[1] shows an indirect flow of information, caused by an abrupt termination within branching instructions that test on secret information leads to information leakages. Consider the program $\mathsf{if}(y_H = 0)\{\mathsf{return}\ 0\}\{\mathsf{skip}\}; x_L := 0; \mathsf{return}\ 0$ and its compilation in Figure 6(c); it yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Our fourth example is of a similar nature, but caused by a return whose value depends on a high expression. Consider the program $\mathsf{if}(y_H = 0)\{\mathsf{return}\ 0\}\{\mathsf{return}\ 1\}$ and its compilation in Figure 6(c). Indeed, the final value of x_L is 0 if the initial value of y_H is 0. The problem is caused by a return instruction within the scope of a high if instruction.

3.3. Information flow typing for source code

In this section, we introduce a type system for secure information flow for IMP inspired from [24]. Typing judgments are implicitly parametrized by the security policy Γ of the form:

$$\vdash e : \tau \text{ (expressions)}$$
$$\vdash i : \tau \text{ (statements)}$$

where e is an expression, i is a statement and τ is a security level. The intuitive meaning of $\vdash e : \tau$ is that τ is an upper bound of the security levels of variables that occur in e, whereas the intuitive meaning of $\vdash i : \tau$ is that i is non-interfering and does not assign to variables with security level lower than τ.

Figure 7 and Figure 8 respectively present the typing rules for expressions and instructions. The rule for assignments prevents direct flows, whereas the rule for if statements prevents indirect flows. Note that the typing rule for return is only sound because we do not allow return expressions to appear within statements: indeed, the source code of the program in Figure 6(c), in which a return statement appears in a high branching statement, is insecure.

The type system is sound, in the sense that typable programs are non-interfering.

[1] Both the third and fourth examples are not legal source programs in our syntax. We nevertheless provide the source code for readability.

$$\textsc{Var}\ \vdash x : \Gamma(x) \qquad \textsc{Val}\ \vdash c : L$$

$$\textsc{Op}\ \frac{\vdash e_i : \tau \ \text{ for } i = 1,2}{\vdash e_1\ op\ e_2 : \tau} \qquad \textsc{Test}\ \frac{\vdash e_i : \tau \ \text{ for } i = 1,2}{\vdash e_1\ cmp\ e_2 : \tau}$$

$$\textsc{SubE}\ \frac{\vdash e : \tau \qquad \tau \leq \tau'}{\vdash e : \tau'}$$

Figure 7. INFORMATION FLOW TYPING RULES FOR EXPRESSIONS

$$\textsc{Assign}\ \frac{\vdash e : \tau \qquad \tau \leq \Gamma(x)}{\vdash x := e : \Gamma(x)} \qquad \textsc{Seq}\ \frac{\vdash i : \tau \qquad \vdash i' : \tau}{\vdash i; i' : \tau} \qquad \textsc{SubC}\ \frac{\vdash i : \tau \qquad \tau' \leq \tau}{\vdash i : \tau'}$$

$$\textsc{Cond}\ \frac{\vdash t : \tau \qquad \vdash i : \tau \qquad \vdash i' : \tau}{\vdash \mathsf{if}(t)\{i\}\{i'\} : \tau} \qquad \textsc{While}\ \frac{\vdash t : \tau \qquad \vdash i : \tau}{\vdash \mathsf{while}(t)\{i\} : \tau} \qquad \textsc{Skip}\ \vdash \mathsf{skip} : \mathsf{H}$$

$$\textsc{Ret}\ \frac{\vdash e : L}{\vdash \mathsf{return}\ e : L}$$

Figure 8. INFORMATION FLOW TYPING RULES FOR STATEMENTS

Proposition 3.4 (Soundness of source type system) *If* $p = i; \mathsf{return}\ e$ *is typable, i.e.* $\vdash i : \tau$ *and* $\mathsf{return}\ e : L$*, then* p *is non-interfering.*

One can prove the above proposition directly, in the style of [24]. An alternative is to derive soundness of the source type system from soundness of the bytecode type system, as we do in the next section.

3.4. Information flow for bytecode

To prevent illicit flows in a non-structured language, one cannot simply enforce local constraints in the typing rules for branching instructions: one must also enforce global constraints that prevent low assignments and updates to occur under high guards (conditional branching that depends on high information). In order to express the global constraints that are necessary to enforce soundness, we rely on some additional information about the program. Concretely, we assume given control dependence regions (cdr) which approximate the scope of branching statements, as well as a security environment, that

attaches to each program point a security level, intuitively the upper bound of all the guards under which the program point executes.

Before explaining the typing rules, we proceed to define the concept of control dependence region, which is closely related to the standard notion used in compilers. The notion of region can be described in terms of a successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$ between program points. Intuitively, j is a successor of i, written $i \mapsto j$, if performing one-step execution from a state whose program point is i may lead to a state whose program point is j. Then, a return point is a program point without successor (corresponding to a return instruction); in the sequel, we write $i \mapsto \mathsf{exit}$ if i is a return point and let $\mathcal{P}_r$ denote the set of return points. Finally, if instructions usually have two successors; when it is the case, the program point of this instruction is referred as a branching point. Formally, the successor relation $\mapsto$ is given by the clauses:

- if $\dot{p}[i] = \mathsf{goto}\ j$, then $i \mapsto j$;
- if $\dot{p}[i] = \mathsf{if}\ cmp\ j$, then $i \mapsto i+1$ and $i \mapsto j$. Since if instructions have two successors, they are thus referred to as branching points;
- if $\dot{p}[i] = \mathsf{return}$, then i has no successors, and we write $i \mapsto \mathsf{exit}$;
- otherwise, $i \mapsto i+1$.

Control dependence regions are characterized by a function that maps a branching program point i to a set of program points $\mathsf{region}(i)$, called the region of i, and by a partial function that maps branching program points to a junction point $\mathsf{jun}(i)$. The intuition behind regions and junction points is that $\mathsf{region}(i)$ includes all program points executing under the guard of i and that $\mathsf{jun}(i)$, if it exists is the sole exit to the region of i; in particular, whenever $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. The properties to be satisfied by control dependence regions, called SOAP properties, are:

Definition 3.5 *A cdr structure* $(\mathsf{region}, \mathsf{jun})$ *satisfies the SOAP (Safe Over APproximation) properties if the following holds:*

SOAP1: *for all program points* i, j, k *such that* $i \mapsto j$ *and* $i \mapsto k$ *and* $j \neq k$*, either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{jun}(i)$*;*

SOAP2: *for all program points* i, j, k*, if* $j \in \mathsf{region}(i)$ *and* $j \mapsto k$*, then either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{jun}(i)$*;*

SOAP3: *for all program points* i, j*, if* $j \in \mathsf{region}(i)$ *and* $j \mapsto \mathsf{exit}$ *then* $\mathsf{jun}(i)$ *is undefined.*

Given a cdr structure $(\mathsf{region}, \mathsf{jun})$, it is straightforward to verify whether or not it satisfies the SOAP properties.

Definition 3.6 *A security environment is a mapping* $se : \mathcal{P} \to \mathcal{S}$.

The bytecode type system is implicitly parametrized by a policy Γ, a cdr structure $(\mathsf{region}, \mathsf{jun})$, and a security environment se. Typing judgments are of the form

$$i \vdash st \Rightarrow st'$$

where i is a program point, and st and st' are stacks of security levels. Intuitively, st and st' keep track of security levels of information on the operand stack during all possible executions.

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma(x)}{i \vdash k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow (\Gamma(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) = L}{i \vdash k :: st \Rightarrow \epsilon}$$

$$\frac{P[i] = \mathsf{if}\ cmp\ j \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \mathrm{lift}_k(st)}$$

Figure 9. TRANSFER RULES FOR VM INSTRUCTIONS

Figure 9 presents a set of typing rules that guarantee non-interference for bytecode programs, where $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$, lift_k is the point-wise extension to stack types of $\lambda l.\ k \sqcup l$. The transfer rule for if requires that the security environment of program points in a high region is high. In conjunction with the transfer rule for load, the transfer rule for if prevents implicit flows and rejects the program of Figure 6(b). Likewise, in conjunction with the transfer rule for push, which requires that the value pushed on top of the operand stack has a security level greater than the security environment at the current program point, and the typing rule for return which requires that $se(i) = L$ and thus avoids return instructions under the guard of high expressions, the transfer rule for return prevents implicit flows and rejects the program of Figure 6(c). Besides, the transfer rule for return requires that the value on top of the operand stack has a low security level, since it will be observed by the adversary. It thus rightfully rejects the program of Figure 6(d).

In addition, the operand stack requires the stack type on the right hand side of $\Rightarrow$ to be lifted by the level of the guard, i.e. the top of the input stack type. It is necessary to perform this lifting operation to avoid illicit flows through operand stack. The following example, which uses a new instruction swap that swaps the top two elements of the operand stack, illustrates why we need to lift the operand stack. This is a contrived example because it does not correspond to any simple source code, but it is nevertheless accepted by a standard bytecode verifier.

1 push 1
2 push 0
3 push 0
4 load y_H
5 if 7
6 swap
7 store x_L
8 push 0
9 return

In this example, the final value of variable x_L depends on the initial value of y_H and so the program is interfering. It is rightfully rejected by our type system, thanks to the lift of the operand stack at program point 5.

One may argue that lifting the entire stack is too restrictive, as it leads the typing system to reject safe programs; indeed, it should be possible, at the cost of added complexity, to refine the type system to avoid lifting the entire stack. Nevertheless, one may equally argue that lifting the stack is unnecessary, because as noted in Section 2 the stack at branching points only has one element in all compiled programs, in which case a more restrictive rule of the form below is sufficient:

$$\frac{P[i] = \mathsf{if}\ cmp\ j \qquad \forall j' \in \mathsf{region}(i).k \leq se(j')}{i \vdash k :: \epsilon \Rightarrow \epsilon}$$

Furthermore, there are known techniques to force that the stack only has one element at branching points.

Typability Typing rules are used to establish a notion of typability. Following Freund and Mitchell [12], typability stipulates the existence of a function, that maps program points to stack types, such that each transition is well-typed.

Definition 3.7 (Typable program) *A program $\dot{p}$ is typable w.r.t. a memory security policy Γ, and cdr structure $(\mathsf{region}, \mathsf{jun})$, and a security environment $se : \mathcal{P} \to \mathcal{S}$ iff there exists a type $S : \mathcal{P} \to \mathcal{S}^\star$ such that:*

- *$S_0 = \varepsilon$ (the operand stack is empty at the initial program point 0);*
- *for all $i \in \mathcal{P}$ and $j \in \mathcal{P} \cup \{\mathsf{exit}\}$, $i \mapsto j$ implies that there exists $st \in \mathcal{S}^\star$ such that $i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$.*

where we write S_i instead of $S(i)$ and $\sqsubseteq$ denotes the point-wise partial order on type stack with respect to the partial order taken on security levels.

The type system is sound, in the sense that typable programs are non-interfering.

Proposition 3.8 *Let $\dot{p}$ be a bytecode program and $(\mathsf{region}, \mathsf{jun})$ a cdr structure that satisfies the SOAP properties. Suppose $\dot{p}$ is typable with respect to* region *and to a memory security policy Γ. Then $\dot{p}$ is non-interfering w.r.t. Γ.*

The proof is based on the SOAP properties, and on two unwinding lemmas showing that execution of typable programs does not reveal secret information. In order to state the unwinding lemmas, one must define an indexed indistinguishability relation between stacks and states.

Definition 3.9 (Indistinguishability of states)

- *A $\mathcal{S}$-stack S is high, written $\mathsf{high}(S)$, if all levels in S are high.*
- *Indistinguishability $os \sim_{S,S'} os'$ between stacks os and os (relative to $\mathcal{S}$-stacks S and S') is defined inductively by the clauses:*

$$\frac{\mathsf{high}(S) \qquad \mathsf{high}(S') \qquad \#os = \#S \qquad \#os' = \#S'}{os \sim_{S,S'} os'}$$

$$\frac{os \sim_{S,S'} os'}{v :: os \sim_{L::S,L::S'} v :: os'} \qquad \frac{os \sim_{S,S'} os'}{v :: os \sim_{H::S,H::S'} v' :: os'}$$

where $\#$ denotes the length of a stack.

- *Indistinguishability between states $\langle i, \rho, os\rangle \sim_{S,S'} \langle i', \rho', os'\rangle$ (relative to stack of security levels S and S') holds iff $os \sim_{S,S'} os'$ and $\rho \sim \rho'$.*

We must also introduce some terminology and notation: we say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j)$ is high for all $j \in \mathsf{region}(i)$. Besides, we let $\mathsf{pc}(s)$ denote the program counter of a state s. Then, the unwinding lemmas can be stated as follows:

- *locally respects:* if $s \sim_{S,T} t$, and $\mathsf{pc}(s) = \mathsf{pc}(t) = i$, and $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$, and $i \vdash T \Rightarrow T'$, then $s' \sim_{S',T'} t'$.
- *step consistent:* if $s \sim_{S,T} t$, and $\mathsf{pc}(s) = i$, and $s \rightsquigarrow s'$ and $i \vdash S \Rightarrow S'$, and $se(i)$ is high, and S is high, then $s' \sim_{S',T} t$.

In order to repeatedly apply the unwinding lemmas, we need additional results about preservation of high contexts.

- *high branching:* if $s \sim_{S,T} t$ with $\mathsf{pc}(s) = \mathsf{pc}(t) = i$ and $\mathsf{pc}(s') \neq \mathsf{pc}(t')$, if $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$ and $i \vdash T \Rightarrow T'$, then S' and T' are high and se is high in region $\mathsf{region}(i)$.
- *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and the security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then S' is high.

The combination of the unwinding lemmas, the high context lemmas, the monotonicity lemmas and the SOAP properties enable to prove that typable programs are non-interfering. The proof proceeds by induction on the length of derivations: assume that we have two executions of a typable program $\dot{p}$, and that s_n and s'_m are final states:

$$s_0 \rightsquigarrow \cdots \rightsquigarrow s_n$$
$$s'_0 \rightsquigarrow \cdots \rightsquigarrow s'_m$$

such that $\mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$ and $s_0 \sim_{S_{\mathsf{pc}(s_0)},S_{\mathsf{pc}(s'_0)}} s'_0$. We want to establish that either the states s_n and s'_m are indistinguishable, i.e. $s_n \sim_{S_{\mathsf{pc}(s_n)},S_{\mathsf{pc}(s'_m)}} s'_m$, or that both stack types $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high. By induction hypothesis, we know that the property holds for all strictly shorter execution paths.

Define $i_0 = \mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$. By the *locally respects* lemma and typability hypothesis, $s_1 \sim_{st,st'} s'_1$ for some stack types st and st' such that $i_0 \vdash S_{i_0} \Rightarrow st$, $st \sqsubseteq S_{\mathsf{pc}(s_1)}$, $i_0 \vdash S_{i_0} \Rightarrow st'$, $st' \sqsubseteq S_{\mathsf{pc}(s'_1)}$.

- If $\mathsf{pc}(s_1) = \mathsf{pc}(s'_1)$ we can apply monotony of indistinguishability (w.r.t. indexes) to establish that $s_1 \sim_{S_{\mathsf{pc}(s_1)},S_{\mathsf{pc}(s'_1)}} s'_1$ and conclude by induction hypothesis.

- If $\mathsf{pc}(s_1) \neq \mathsf{pc}(s'_1)$ we know by the *high branching* lemma that se is high in region $\mathsf{region}(i_0)$ and st and st' are high. Hence both $S_{\mathsf{pc}(s_1)}$ and $S_{\mathsf{pc}(s'_1)}$ are high.
  Using the SOAP properties, one can prove that either $\mathsf{jun}(i_0)$ is undefined and both $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high, or that $\mathsf{jun}(i_0)$ is defined and there exists k, k', $1 \leq k \leq n$ and $1 \leq k' \leq m$ such that $k = k' = \mathsf{jun}(i_0)$ and $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{i_0}} s'_0$ $s_0 \sim_{S_{i_0}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$. Since $s_0 \sim_{S_{i_0}, S_{i_0}} s'_0$ we have by transitivity and symmetry of $\sim$, $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$ with $\mathsf{pc}(s_k) = \mathsf{pc}(s'_{k'})$ and we can conclude by induction hypothesis.

3.5. Preservation of Typability

In this section, we focus on preservation of typability by compilation. Since the bytecode type system uses both a cdr structure $(\mathsf{region}, \mathsf{jun})$ and a security environment se, we must extend the compiler of Section 2.3 so that it generates for each program p a cdr structure $(\mathsf{region}, \mathsf{jun})$ and the security environment se such that $[\![p]\!]$ is typable w.r.t. $(\mathsf{region}, \mathsf{jun})$ and se. The cdr structure of the compiled programs can be defined easily. For example, the region of if statement is given by the clause:

$$\begin{aligned} k : [\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ l;\ k_2 : [\![i_2]\!] \\ \text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\ k_2 = k_1 + |[\![i_1]\!]| + 1 \\ l = k_2 + |[\![i_2]\!]| \\ \mathsf{region}(k_1 - 1) = [k_1, l - 1] \\ \mathsf{jun}(k_1 - 1) = l \\ \mathsf{blev}(k_1 - 1) = \bigcup\{\tau \mid\vdash e_1\ cmp\ e_2 : \tau\} \end{aligned}$$

Note that in addition to the region, we define the branching level $\mathsf{blev}(k_1 - 1)$ of $k_1 - 1$ as the minimal level of its associated test, i.e. $\mathsf{blev}(k_1 - 1)$ low if all variables in e_1 and e_2 are low, and high otherwise.

Likewise, the region of while statement is given by the clause:

$$\begin{aligned} k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\ \text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\ k_2 = k_1 + |[\![i]\!]| + 1 \\ \mathsf{region}(k_1 - 1) = [k, l - 1] \\ \mathsf{jun}(k_1 - 1) = l \\ \mathsf{blev}(k_1 - 1) = \bigcup\{\tau \mid\vdash e_1\ cmp\ e_2 : \tau\} \end{aligned}$$

The security environment is derived from the cdr structure and the branching level of program points. Formally, we define

$$se(i) = \bigcup\{\mathsf{blev}(j) \mid i \in \mathsf{region}(j)\}$$

with the convention that $\bigcup\ \emptyset = L$.

Theorem 3.10 (Typability Preservation) *Let p be a typable source program. Then $[\![p]\!]$ is a typable bytecode program w.r.t. the generated cdr structure* $(\mathsf{region}, \mathsf{jun})$ *and the*

generated security environment se*. In addition, the cdr structure* $(\mathsf{region}, \mathsf{jun})$ *satisfies the SOAP properties.*

Using the fact that the compiler preserves the semantics of program, the soundness of the information flow type system for bytecode and preservation of typability, we can derive soundness of the information flow type system for the source language.

Corollary 3.11 (Soundness of source type system) *Let* p *be a typable* IMP *program w.r.t. to a memory security policy* Γ*. Then* p *is non-interfering w.r.t.* Γ*.*

By preservation of typing, $[\![p]\!]$ is typable, and thus non-interfering by soundness of the bytecode type system. By correctness of the compiler, the program p is non-interfering iff its compilation $[\![p]\!]$ is non-interfering, and therefore p is non-interfering.

3.6. Optimizations

Simple optimizations such as constant folding, dead code elimination, and rewriting conditionals whose conditions always evaluate to the same constant can be modeled as source-to-source transformations and can be shown to preserve information-flow typing. Figure 10 provides examples of transformations that preserve typing.[2] Most rules are of the form

$$\frac{P[i] = ins \quad constraints}{P[i] = ins'} \qquad \frac{P[i, i+n] = i\vec{n}s \quad constraints}{P[i, i+n] = i\vec{n}s'}$$

where ins is the original instruction and ins' is the optimized instruction. In some cases however, the rules are of the form

$$\frac{P[i, n+m] = i\vec{n}s \quad constraints}{P[i, n+m'] = i\vec{n}s'}$$

with $m \neq m'$. Therefore such rules do not preserve the number of instructions, and the transformations must recompute the targets of jumps, which is omitted here.

In the rules, we use $\mathcal{F}$ to denote a stack-preserving sequence of instructions, i.e. a sequence of instructions such that the stack is the same at the beginning and the end of $\mathcal{F}$ execution, which we denote as $\mathcal{F} \in \mathsf{StackPres}$ in the rules. We also assume that there are no jumps from an instruction in $\mathcal{F}$ outside $\mathcal{F}$, so that all executions must flow through the immediate successor of $\mathcal{F}$, and that there are no jumps from an instruction outside $\mathcal{F}$ inside $\mathcal{F}$, so that all executions enter $\mathcal{F}$ through its immediate predecessor. In other words, we assume that $ins :: \mathcal{F} :: ins'$ is a program fragment, where ins and ins' are the instructions preceding and following $\mathcal{F}$.

The last rule uses $\mathcal{VAL}(x, i)$ to denote the safe approximation of the value of x at program point i; this approximation can be statically computed through, e.g., symbolic analysis. The optimizations use two new instructions nop and dup, the first one simply jump to the next program point, the second duplicates the top value of the stack and continues the execution to the next program point.

[2] Thanks to Salvador Cavadini for contributing to these examples.

$$\frac{P[i, i+n+2] = i :: \mathcal{F} :: \mathsf{pop} \qquad i \in \{\mathsf{load}\ x, \mathsf{push}\ n\}}{P[i, i+n] = \mathcal{F}}$$

$$\frac{P[i, i+n] = \mathsf{binop}\ op :: \mathcal{F} :: \mathsf{pop}}{P[i, i+n] = \mathsf{pop} :: \mathcal{F} :: \mathsf{pop}}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad x \text{ is dead at } P[i]}{P[i] = \mathsf{pop}}$$

$$\frac{P[i, i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F}}{P[i, i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{dup}}$$

$$\frac{P[i, i+n] = \mathsf{store}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F}}{P[i, i+n] = \mathsf{dup} :: \mathsf{store}\ x :: \mathcal{F}}$$

$$\frac{P[i, i+2+n] = \mathsf{store}\ x :: \mathsf{load}\ x :: \mathcal{F} :: \mathsf{store}\ x \qquad \mathsf{store}\ x, \mathsf{load}\ x \notin \mathcal{F}}{P[i, i+n] = \mathcal{F} :: \mathsf{store}\ x}$$

$$\frac{P[i, i+2] = \mathsf{push}\ c_1 :: \mathsf{push}\ c_2 :: \mathsf{binop}\ op}{P[i] = \mathsf{push}\ (c_1\ op\ c_2)}$$

$$\frac{P[i] = \mathsf{load}\ x \qquad \mathcal{VAL}(x, i) = n}{P[i] = \mathsf{push}\ n}$$

In all rules, we assume that $\mathcal{F}$ is stack-preserving.

Figure 10. OPTIMIZING TRANSFORMATION RULES

As noted in [4], more aggressive optimizations may break type preservation, even though they are semantics preserving, and therefore security preserving. For example, applying common subexpression elimination to the program

$$x_H := n_1 * n_2; y_L := n_1 * n_2$$

where n_1 and n_2 are constant values, will result in the program

$$x_H := n_1 * n_2; y_L := x_H$$

Assuming that variable x_H is a high variable and y_L is a low variable, the original program is typable, but the optimized program is not, since the typing rule for assignment will detect an explicit flow $y_L := x_H$. For this example, one can recover typability by creating a low auxiliary variable z_L in which to store the result of the computation $n_1 * n_2$, and assign z_L to x_H and y_L, i.e.

$$z_L := n_1 * n_2; x_H := z_L; y_L := z_L$$

$$\begin{array}{ll}
\text{source logical expressions} & \bar{e} ::= \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \\
\text{source logical tests} & \bar{t} ::= \bar{e}\ cmp\ \bar{e} \\
\text{source propositions} & \phi ::= \bar{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \exists x.\ \phi \mid \forall x.\ \phi
\end{array}$$

Figure 11. SPECIFICATION LANGUAGE FOR SOURCE PROGRAMS

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{skip}, \psi) = \psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(x := e, \psi) = \psi\{x \leftarrow e\}, \emptyset}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(i_2, \psi) = \phi_2, \theta_2 \quad \mathsf{wp}_{\mathcal{S}}(i_1, \phi_2) = \phi_1, \theta_1}{\mathsf{wp}_{\mathcal{S}}(i_1; i_2, \psi) = \phi_1, \theta_1 \cup \theta_2}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(i_t, \psi) = \phi_t, \theta_t \quad \mathsf{wp}_{\mathcal{S}}(i_f, \psi) = \phi_f, \theta_f}{\mathsf{wp}_{\mathcal{S}}(\mathsf{if}(t)\{i_t\}\{i_f\}, \psi) = (t \Rightarrow \phi_t) \wedge (\neg t \Rightarrow \phi_t), \theta_t \cup \theta_f}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(i, I) = \phi, \theta}{\mathsf{wp}_{\mathcal{S}}(\mathsf{while}_I(t)\{i\}, \psi) = I, \{I \Rightarrow (t \Rightarrow \phi) \wedge (\neg t \Rightarrow \psi)\} \cup \theta}$$

Figure 12. WEAKEST PRE-CONDITION FOR SOURCE PROGRAMS

The above examples show that a more systematic study of the impact of program optimizations on information flow typing is required.

4. Verification conditions

Program logics are expressive frameworks that enable reasoning about complex properties as well as program correctness. Early program verification techniques include Hoare logics, and weakest pre-condition calculi, which are concerned with proving program correctness in terms of triples, i.e. statements of the form $\{P\}c\{Q\}$, where P and Q are respectively predicates about the initial and final states of the program c. The intended meaning of a statement $\{P\}c\{Q\}$ is that any terminating run of the program c starting with a state s satisfying the pre-condition P will conclude in a state s' that satisfies the post-condition Q. In these notes, we focus on a related mechanism, called verification condition generation, which differs from the former by operating on annotated programs, and is widely used in program verification environments.

4.1. Source language

The verification condition generator VCgen operates on annotated source programs, i.e. source programs that carry a pre-condition, a post-condition, and an invariant for each

stack expressions	$\bar{os} ::= \mathsf{os} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os}$
bytecode logical expressions	$\bar{e} ::= \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \bar{os}[k]$
bytecode logical tests	$\bar{t} ::= \bar{e}\ cmp\ \bar{e}$
bytecode propositions	$\phi ::= \bar{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \exists x.\ \phi \mid \forall x.\ \phi$

where os is a special variable representing the current stack operand stack.

Figure 13. SPECIFICATION LANGUAGE FOR BYTECODE PROGRAMS

loop.

Definition 4.1 (Annotated source program)

- *The set of propositions is defined in Figure 11, where $\bar{x}$ is a special variable representing the initial value of the variable x, and* res *is a special value representing the final value of the evaluation of the program.*
- *A pre-condition is a proposition that only refers to the initial values of variables. An invariant is a proposition that refers to the initial and current values of variables (not to the final result). A post-condition is a proposition.*
- *An annotated program is a triple (p, Φ, Ψ), where Φ is a pre-condition, Ψ is a post-condition, and p is a program in which all* while *loops are annotated (we note* $\mathsf{while}_I(t)\{s\}$ *for a loop annotated with invariant I).*

The VCgen computes a set of verification conditions (VC). Their validity ensure that the program meets its contract, i.e. that every terminating run of a program starting from a state that satisfies the program pre-condition will terminate in a state that satisfies the program post-condition, and that loop invariants hold at the entry and exit of each loops.

Definition 4.2 (Verification conditions for source programs)

- *The weakest pre-condition calculus $\mathsf{wp}_{\mathcal{S}}(i, \psi)$ relative to a instruction i and a post-condition ψ is defined by the rules of Figure 12.*
- *The verification conditions of an annotated program (p, Φ, Ψ) with $p = i; \mathsf{return}\ e$ is defined as*

$$\mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi) = \{\Phi \Rightarrow \phi\{\vec{x} \leftarrow \vec{\bar{x}}\}\} \cup \theta$$

where $\mathsf{wp}_{\mathcal{S}}(i, \Psi\{\mathsf{res} \leftarrow e\}) = \phi, \theta$.

4.2. Target language

As for the source language, the verification condition generator operates on annotated bytecode programs, i.e. bytecode that carry a pre-condition, a post-condition and loop invariants. In an implementation, it would be reasonable to store invariants in a separate annotation table, the latter being a partial function form program points to propositions. Here we find it more convenient to store the annotations directly in the instructions.

Definition 4.3 (Annotated bytecode program)

- *The set of bytecode propositions is defined in Figure 13.*
- *An annotation is a proposition that does not refer to the operand stack. A pre-condition is an annotation that only refers to the initial value of variables. An invariant is an annotation that does not refer to the result of the program. A post-condition is an annotation.*
- *An annotated bytecode instruction is either an bytecode instruction or a bytecode proposition and a bytecode instruction:*

$$\bar{i} ::= i \mid \phi : i$$

- *An annotated program is a triple $(\dot{p}, \dot{\Phi}, \dot{\Psi})$, where $\dot{\Phi}$ is a pre-condition, $\dot{\Psi}$ is a post-condition, and $\dot{p}$ is a bytecode program in which some instructions are annotated.*

At the level of the bytecode language, the predicate transformer wp is a partial function that computes, from a partially annotated program, a fully annotated program in which all labels of the program have an explicit pre-condition attached to them. However, wp is only defined on programs that are sufficiently annotated, i.e. through which all loops must pass through an annotated instruction. The notion of sufficiently annotated is characterized by an inductive and decidable definition and does not impose any specific structure on programs.

Definition 4.4 (Well-annotated program) *A annotated program $\dot{p}$ is well-annotated if every program point satisfies the inductive predicate $\mathsf{reachAnnot}_{\dot{p}}$ defined by the clauses:*

$$\frac{\dot{p}[k] = \phi : i}{k \in \mathsf{reachAnnot}_{\dot{p}}} \qquad \frac{\dot{p}[k] = \mathsf{return}}{k \in \mathsf{reachAnnot}_{\dot{p}}} \qquad \frac{\forall k'.\ k \mapsto k' \Rightarrow k' \in \mathsf{reachAnnot}_{\dot{p}}}{k \in \mathsf{reachAnnot}_{\dot{p}}}$$

Given a well-annotated program, one can generate an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the pre-condition that an initial state before the execution of the corresponding label should satisfy for the function to terminate only in a state satisfying its post-condition.

Definition 4.5 (Verification conditions for bytecode programs) *Let $(\dot{p}, \dot{\Phi}, \dot{\Psi})$ be a well-annotated program.*

- *The weakest pre-condition $\mathsf{wp}_{\mathcal{L}}(k)$ of a program point k and the weakest pre-condition $\mathsf{wp}_i(k)$ of its corresponding instruction are defined in Figure 14.*
- *The verification conditions $\mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})$ is defined by the clauses:*

$$\frac{}{\dot{\Phi} \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)\{\vec{x} \leftarrow \vec{\tilde{x}}\}) \in \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})} \qquad \frac{\dot{p}[k] = \phi : i}{\phi \Rightarrow \mathsf{wp}_i(k)) \in \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})}$$

$$
\begin{array}{lcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow c :: \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{push}\ c \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow (\mathsf{os}[0]\ op\ \mathsf{os}[1]) ::\uparrow^2 \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{binop}\ op \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow x :: \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{load}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os}, x \leftarrow \uparrow \mathsf{os}, \mathsf{os}[0]\} & \text{if } \dot{p}[k] = \mathsf{store}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } \dot{p}[k] = \mathsf{goto}\ l \\
\mathsf{wp}_i(k) & = & (\mathsf{os}[0]\ cmp\ \mathsf{os}[1] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow \uparrow^2 \mathsf{os}\}) & \text{if } \dot{p}[k] = \mathsf{if}\ cmp\ l \\
 & & \wedge\ (\neg(\mathsf{os}[0]\ cmp\ \mathsf{os}[1]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)\{\mathsf{os} \leftarrow \uparrow^2 \mathsf{os}\}) & \\
\mathsf{wp}_i(k) & = & \dot{\Psi}\{\mathsf{res} \leftarrow \mathsf{os}[0]\} & \text{if } \dot{p}[k] = \mathsf{return}
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } \dot{p}[k] = \phi : i \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Figure 14. WEAKEST PRE-CONDITION FOR BYTECODE PROGRAMS

$$
\frac{}{\bar{\rho}, os, \rho \vdash \mathsf{os} \mapsto os} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{e} \mapsto e \quad \bar{\rho}, os, \rho \vdash \bar{os} \mapsto os'}{\bar{\rho}, os, \rho \vdash \bar{e} :: \bar{os} \mapsto v :: os'}
$$

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto v_1 :: \ldots :: v_k :: os'}{\bar{\rho}, os, \rho \vdash \uparrow^k \bar{os} \mapsto os'} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto v_0 :: \ldots :: v_k :: os'}{\bar{\rho}, os, \rho \vdash \bar{os}[k] \mapsto v_k}
$$

$$
\frac{}{\bar{\rho}, os, \rho \vdash \bar{x} \mapsto \bar{\rho}(x)} \qquad \frac{}{\bar{\rho}, os, \rho \vdash x \mapsto \rho(x)} \qquad \frac{}{\bar{\rho}, os, \rho \vdash c \mapsto c}
$$

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1\ op\ \bar{e}_2 \mapsto v_1\ op\ v_2}
$$

Figure 15. INTERPRETATION OF BYTECODE EXPRESSIONS

4.3. Soundness

Bytecode (resp. source) propositions can be interpreted as predicates on bytecode (resp. source) states. In the case of bytecode, the interpretation builds upon a partially defined interpretation of expressions (partiality comes from the fact that some expressions refer to the operand stack and might not be well defined w.r.t. particular states).

Definition 4.6 (Correct program)

- *The evaluation of logical bytecode expressions $\bar{e}$ in an initial memory $\bar{\rho}$, a current operand stack os and a current memory ρ to a value v is defined by the rules of Figure 15. This evaluation is naturally extended to bytecode propositions $\bar{\rho}, os, \rho \vdash P \mapsto \phi_v$, where ϕ_v is a boolean formula, with the following rule for tests:*

$$\frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1 \; cmp \; \bar{e}_2 \mapsto v_1 \; cmp \; v_2}$$

- *An initial memory $\bar{\rho}$, a current operand stack os and a current memory ρ validate a logical bytecode proposition ϕ,$\bar{\rho}, os, \rho \vdash \phi$, if $\bar{\rho}, os, \rho \vdash \phi \mapsto \phi_v$ and ϕ_v is a valid boolean formula.*
- *A well-annotated bytecode program $(\dot{p}, \dot{\Phi}, \dot{\Psi})$ is correct, written $\vdash \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})$, if all the verification conditions are valid.*

Soundness establishes that the VCgen is a correct backwards abstraction of one step execution.

Lemma 4.7 (One step soundness of VCgen**)** *For all correct programs $(\dot{p}, \dot{\Phi}, \dot{\Psi})$:*

$$\left.\begin{array}{l} \langle k,\, \rho,\, os\rangle \rightsquigarrow \langle k',\, \rho',\, os'\rangle \\ \bar{\rho}, os, \rho \vdash \mathsf{wp}_i(k) \end{array}\right\} \Rightarrow \bar{\rho}, os', \rho' \vdash \mathsf{wp}_{\mathcal{L}}(k')$$

Furthermore, if the evaluation terminates $\langle k,\, \rho,\, os\rangle \rightsquigarrow \rho, v$ (i.e the instruction at position k is a return*) then $\bar{\rho}, \emptyset, \rho' \vdash \dot{\Psi}\{\mathsf{res} \leftarrow v\}$*

Soundness of the VCgen w.r.t. pre-condition and post-condition follows.

Corollary 4.8 (Soundness of VCgen**)** *For all correct programs $(\dot{p}, \dot{\Phi}, \dot{\Psi})$, initial memory $\bar{\rho}$, final memory ρ and final value v, if $\dot{p} : \bar{\rho} \Downarrow \rho, v$ and $\bar{\rho}, \emptyset, \emptyset \vdash \dot{\Phi}$ then $\bar{\rho}, \emptyset, \rho \vdash \dot{\Psi}\{\mathsf{res} \leftarrow v\}$*

The proof proceeds as follows. First, we prove by induction on n that

$$\left.\begin{array}{l} \langle k,\, \rho,\, os\rangle \rightsquigarrow^n \langle k',\, \rho',\, os'\rangle \\ \bar{\rho}, os, \rho \vdash \mathsf{wp}_i(k) \end{array}\right\} \Rightarrow \bar{\rho}, os', \rho' \vdash \mathsf{wp}_i(k')$$

If $n = 0$, it is trivial. If $n = 1+m$, we have $\langle k,\, \rho,\, os\rangle \rightsquigarrow \langle k_1,\, \rho_1,\, os_1\rangle \rightsquigarrow^n \langle k',\, \rho',\, os'\rangle$. It is sufficient to prove that $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_i(k_1)$, since then we can conclude the proof using the induction hypothesis. Using the previous lemma, we get $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_{\mathcal{L}}(k_1)$.

We now conclude with a case analysis:

- if the program point k_1 is not annotated then $\mathsf{wp}_{\mathcal{L}}(k_1) = \mathsf{wp}_i(k_1)$, and we are done;
- if the program point k_1 is annotated, say $\dot{p}[k_1] = \phi : i$, then $\mathsf{wp}_{\mathcal{L}}(k_1) = \phi$. Since the program is correct the proposition $\phi \Rightarrow \mathsf{wp}_i(k_1)$ is valid and so $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_i(k_1)$.

Second, since $\dot{p} : \bar{\rho} \Downarrow \rho, v$ there exists n such that

$$\langle 0,\, \rho_0,\, \emptyset\rangle \rightsquigarrow^n \langle k,\, \rho,\, os\rangle \rightsquigarrow \rho, v$$

By step one above, we have $\rho_0, os, \rho \vdash \mathsf{wp}_i(k)$. Furthermore, $\dot{p}[k] = \mathsf{return}$ so $\mathsf{wp}_i(k) = \dot{\Psi}\{res \leftarrow \mathsf{os}[0]\}$. This concludes the proof.

4.4. Preservation of proof obligations

We now extend our compiler so that it also inserts annotations in bytecode programs, and show that it transforms programs into well-annotated programs, and that furthermore it transforms correct source programs into correct bytecode programs. In fact, we show a stronger property, namely that the proof obligations at source and bytecode level coincide.

The compiler of Section 2.3 is modified to insert invariants in bytecode:

$$k : [\![\mathsf{while}_I(e_1\ cmp\ e_2)\{i\}]\!] = I : [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k$$
$$\text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1$$
$$k_2 = k_1 + |[\![i]\!]| + 1$$

As expected, the compiler produces well-annotated programs.

Lemma 4.9 (Well-annotated programs) *For all annotated source program (p, Φ, Ψ), the bytecode program $[\![p]\!]$ is well-annotated.*

In addition, the compiler "commutes" with verification condition generation. Furthermore, the commutation property is of a very strong form, since it claims that proof obligations are syntactically equal.

Proposition 4.10 (Preservation of proof obligations – PPO) *For all annotated source program (p, Φ, Ψ):*

$$\mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi) = \mathsf{VCgen}_{\mathcal{B}}([\![p]\!], \Phi, \Psi)$$

Thus, correctness proofs of source programs can be used directly as proof of bytecode programs without transformation. In particular, the code producer can directly prove the source program and send the proofs and the compiled program to the code consumer without transforming the proofs.

Using the fact that the compiler preserves the semantics of program, the soundness of the verification condition generator for bytecode and PPO, we can derive soundness of the source verification condition generator. (The notion of correct source program is defined in the same way as that of bytecode program).

Corollary 4.11 (Soundness of $\mathsf{VCgen}_{\mathcal{S}}$) *If $\vdash \mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi)$ then for all initial memories ρ_0 satisfying Φ, if $p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v$ then $\rho_0, \rho \vdash \Psi$.*

By correctness of the compiler, $[\![p]\!] : \rho_0 \Downarrow \rho, v$. By preservation of proof obligations, $\vdash \mathsf{VCgen}_{\mathcal{B}}([\![p]\!], \Phi, \Psi)$. By correctness of the bytecode VCgen, $\rho_0, \rho \vdash \Psi$.

4.5. Optimizations

Preservation of proof obligations does not hold in general for program optimizations, as illustrated by the following example:

$r_1 := 1$	$r_1 := 1$
$\{\mathsf{true}\}$	$\{\mathsf{true}\}$
$r_2 := r_1$	$r_2 := 1$
$\{r_1 = r_2\}$	$\{r_1 = r_2\}$

The proof obligations related to the sequence of code containing the assignment $r_2 := r_1$ is $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$ for the original and optimized version respectively. The second proof obligation is unprovable, since this proof obligation is unrelated to the sequence of code containing the assignment $r_1 := 1$.

In order to extend our results to optimizing compilers, we are led to consider certificate translation, whose goal is to transform certificates of original programs into certificates of compiled programs. Given a compiler $[\![\cdot]\!]$, a function $[\![\cdot]\!]_{\mathrm{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ", written $c : P \models \phi$), a certificate translator is a function $[\![\cdot]\!]_{\mathrm{cert}}$ such that for all programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow [\![c]\!]_{\mathrm{cert}} : [\![p]\!] \models [\![\phi]\!]_{\mathrm{spec}}$$

In [2], we show that certificate translators exist for most common program optimizations, including program transformations that perform arithmetic reasoning. For such transformations, one must rely on certifying analyzers that generate automatically certificates of correctness for the analysis, and then appeal to a weaving process to produce a certificate of the optimized program.

Whereas [2] shows the existence of certificate translators on a case-by-case basis, a companion work [3] uses the setting of abstract interpretation [10,11] to provide sufficient conditions for transforming a certificate of a program p into a certificate of a program p', where p' is derived from p by a semantically justified program transformation, such as the optimizations considered in [2].

5. Extensions to sequential Java

The previous sections have considered preservation of information flow typing and preservation of proof obligations for a simple language. In reality, these results have been proved for a sequential Java-like language with objects, exceptions, and method calls. The purpose of this section is to highlight the main issues of this extension. The main difficulties are three-fold:

- *dealing with object-orientation*: Java and JVM constructs induce a number of well-known difficulties for verification. For instance, method signatures (for type systems) or specifications (for logical verification) are required for modular verification. In addition, signatures and specifications must account for all possible termination behaviors; in the case of method specifications, it entails providing exceptional post-conditions as well as normal post-conditions. Furthermore, signatures and specifications must be compatible with method overriding;
- *achieving sufficient precision*: a further difficulty in scaling up our results to a Java-like language is precision. The presence of exceptions and object-orientation yields a significant blow-up in the control flow graph of the program, and, if no care is taken, may lead to overly conservative type-based analyses and to an explosion of verification conditions. In order to achieve an acceptable degree of usability, both the information flow type system and the verification condition generator need to rely on preliminary analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will

perform safety analyses such as class analysis, null pointer analysis, exception analysis, and array out-of-bounds analysis. These analyses drastically improve the quality of the approximation of the control flow graph (see [6] forthe case of null pointer exceptions). In particular, one can define a tighter successor relation $\mapsto$ that leads to more precise cdr information and thus typing in the case of information flow [5], and to more compact verification conditions in the case of functional verification [14];

- *guaranteeing correctness of the verification mechanisms*: the implementation of type-based verifiers and verification condition generators for sequential Java bytecode are complex programs that form the cornerstone of the security architectures that we propose. It is therefore fundamental that their implementation is correct, since flaws in the implementation of a type system or of a verification condition generator can be exploited to launch attacks. We have therefore used the Coq proof assistant [9] to certify both verification mechanisms. The verification is based on Bicolano, a formal model of a fragment of the Java Virtual Machine in the Coq proof assistant. In addition to providing strong guarantees about the correctness of the type system and verification condition generator, the formalization serves as a basis for a Foundational Proof Carrying Code architecture. A distinctive feature of our architecture is that both the type system and the verification condition generator are executable inside higher order logic and thus one can use reflection for verifying certificates. As compared to Foundational Proof Carrying Code [1], which is deductive in nature, reflective Proof Carrying Code exploits the interplay between deduction and computation to support efficient verification procedures and compact certificates.

6. Conclusion

Popular verification environments such as Jif (for information flow) and ESC/Java (for functional verification) target source code, and thus do not address directly the concerns of mobile code, where code consumers require guarantees on the code they download and execute. The purpose of these notes has been to demonstrate in a simplified setting that one can bring the benefits of source code verification to code consumers by developing adequate verification methods at bytecode level and by relating them suitably to source code verification.

Acknowledgments This work is partially supported by the EU project MOBIUS, and by the French ANR project PARSEC.

References

[1] A. W. Appel. Foundational Proof-Carrying code. In *Proceedings of LICS'01*, pages 247–258. IEEE Computer Society, 2001.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, Seoul, Korea, August 2006. Springer-Verlag.

[3] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, number xxxx in Lecture Notes in Computer Science. Springer-Verlag, 2008.

[4] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[5] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 125 – 140. Springer-Verlag, 2007.

[6] P. Chalin and P.R. James. Non-null references by default in java: Alleviating the nullity annotation burden. In Erik Ernst, editor, *Proceedings of ECOOP'07*, volume 4609 of *Lecture Notes in Computer Science*, pages 227–247. Springer, 2007.

[7] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[8] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML — progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[9] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.

[12] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.

[13] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[14] B. Grégoire and J.-L. Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In G. Barthe and C. Fournet, editors, *Proceedings og TGC'07*, volume 4912 of *Lecture Notes in Computer Science*, pages xxx–xxx. Springer, 2007.

[15] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.

[17] A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.

[18] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[19] G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[20] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

[21] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3-4):303–334, 2003.

[22] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.

[23] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of PLDI'96*, pages 181–192, 1996.

[24] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Formal Foundations of Computer Security

Mark BICKFORD and Robert CONSTABLE
Department of Computer Science, Cornell University

1. Introduction

We would like to know with very high confidence that private data in computers is not unintentionally disclosed and that only authorized persons or processes can modify it. Proving security properties of software systems has always been hard because we are trying to show that something bad cannot happen no matter what a hostile adversary tries and no matter what coding errors are made. For a limited interactive program like a chess player, it might be possible to "prove" that all rules are followed and king is safe in a certain board position, but in a large complex system, like the Internet, the many points of attack and many ways to introduce errors seem to defy absolute guarantees.

Services such as *public key cryptography, digital signatures*, and *nonces* provide the means to secure selected Internet communications and specific transactions. However, public key cryptography depends on mathematically deep computational complexity assumptions, and nonces depend on statistical assumptions. When formal methods researchers include complexity theory and probability theory in the formal mathematical base, the task of giving a formal logical account of security properties and proving them formally becomes daunting.

In this paper we explain a significantly less costly and more abstract way of providing adequate formal guarantees about cryptographic services. Our method is based on properties of the data type of Atoms in type theory and it is most closely related to the use of types in abstract cryptographic models [2,4,1,16,3]. Atoms provide the basis for creating "unguessable tokens" as shown in Bickford [10]. Essential to the use of atoms is the concept that an expression e of type theory is *computationally independent* of atom a, written $e\|a$. Bickford [10] discovered this concept and has shown that this logical primitive is sufficient for defining cryptographic properties of processes in the rich model of distributed computing formalized by the authors in [11,12] and elaborated and implemented in Nuprl [7]. We will explain independence in detail.

The goal of this paper is to elaborate this approach to proving security properties and illustrate it on simple examples of security specifications and protocols of the kind presented in John Mitchell's lectures [29] including Needham-Schroeder-Lowe protocol [25,19] and the core of the SSL protocol. We illustrate new ways of looking at proofs in terms of assertions about events that processes "believe' and "conjecture," and we relate them to what is constructively known. We will show how to provide formal proofs of security properties using a *logic of events*. It is possible that proofs of this type could be *fully automated* in provers such as Coq, HOL, MetaPRL, Nuprl, and PVS – those based on type theory. We also exposit elements of our formal *theory of event structures* needed

to understand these simple applications and to understand how they can provide guaranteed security services in the standard computing model of asynchronous distributed computing based on message passing.

Mark Bickford has implemented the theory of event structures and several applications in the Nuprl prover and posted them in its Formal Digital Library [7], the theory of event structures could be implemented in the other provers based on type theories, though it might be less natural to treat some concepts in intensional theories. At certain points in this article we will refer to the formal version of concepts and the fundamental notions of type theory in which they are expressed.

2. Intuitive Account of Event Structures.

2.1. The Computing Model – General Features

Our computing model resembles the Internet. It is a network of asynchronously communicating sequential processes. Accommodating processes that are multi-threaded sharing local memory is accomplished by building them from sequential processes if the need arises - it won't in our examples.

At the lowest level, processes communicate over directed reliable links, point to point. Links have unique labels, and message delivery is not only reliable but messages arrive at their destination in the order sent (FIFO). Messages are not blocked on a link, so the link is a queue of messages. These are the default assumptions of the basic model, and it is easy to relax them by stipulating that communication between P_i and P_j goes through a virtual process P_k which can drop messages and reorder them. Moreover, we can specify communications that do not mention specific links or contents, thus modeling Internet communications.

Messages are tagged so that a link can be used for many purposes. The content is typed, and polymorphic operations are allowed. We need not assume decidability of equality on message contents, but it must be possible to *decide* whether tags are equal.

Processes In our formal computing model, processes are called *message automata* (MA). Their state is potentially infinite and contents are accessed by typed identifiers, x_1, x_2,... The contents are typed, using types from the underlying type theory. We use an extremely rich collection of types, capable of defining those of any existing programming language as well as the types used in computational mathematics, e.g. higher-order functions, infinite precision computable real numbers, random variables, etc.

Message automata are finite sets of *clauses* (order is semantically irrelevant). Each clause is either a *declaration* of the types of variables, an *action* or a *frame condition*. The frame conditions are a unique feature distinguishing our process model from others, so we discuss them separately in the next subsection. The other clauses are standard for process models such as I/O automata [21,20] or distributed programs [8] or distributed state machines, or distributed abstract state machines.

The action clauses can be labeled so that we speak of action a. Actions also have kinds k_1, k_2... so that we can classify them. One of the essential kinds is a *receive action*. Such an action will receive a tagged message on a link (by reading a message queue), but we can classify an action without naming the link. All actions can send messages. An initial action will *initialize* a state variable, and an internal action will *update* a state vari-

able, possibly depending on a received value or a random value, $x_i := f(state, value)$. We will not discuss the use of random values. Actions that update variables can be guarded by a boolean expression, called the *precondition* for the action. (Note, receive actions are not guarded.) The complete syntax for message automata is given by the Nuprl definition [12]. There is also an abstract syntax for them in which message automata are called *realizers*.

For each base level receive action a, we can syntactically compute the *sender* by looking at the link. For a receive action routed through the network, we may not be able to compute the sender. For each action that updates the state, if it is not the first, we can compute its *predecessor* from a trace of the computation (discussed below).

2.2. Composition and Feasibility

Frame Conditions One of the novel features of our process model is that execution can be constrained by *frame conditions* which limit which actions can send and receive messages of a certain type on which links and access the state using identifiers. For example, constraints on state updates have the form *only a can affect* x_i.

Composition of Processes Processes are usually built from subprocesses that are composed to create the main process. Suppose $S_1, S_2..., S_n$ are sub processes. Their composition is denoted $S_1 \oplus ... \oplus S_n$. In our account of processes, composition is extremely simple, namely the union of the actions and frame conditions, but the result is a *feasible process* only if the subprocesses are compatible. For S_i and S_j to be *compatible*, the frame conditions must be consistent with each other and with the actions. For example, if S_1 requires that only action k can change x_1, then S_2 can't have an action k' that also changes x_1.

We say that a process is *feasible* if and only if it has at least one execution. If S_1 and S_2 are feasible and compatible with each other, then $S_1 \oplus S_2$ is feasible. If we allow any type expression from the mathematical theory to appear in the code, then we can't decide either feasibility or compatibility. So in general we use only standard types in the process language, but it is possible to use any type as long as we take the trouble to prove compatibility "by hand" if it is not decided by an algorithm.

2.3. Execution

Scheduling Given a network of processes $\mathcal{P}$ (distributed system, DSystem) it is fairly clear how they compute. Each process P_i with state S_i with a schedule sch_i and message queues m_i at its links executes a basic action a_i; it can *receive* a message by reading a queue, *change* its state, and *send* a list of tagged messages (appending them to outgoing links). Thus given the queues, m_i, state s_i, and action a_i, the action produces new queues, m'_i, state s'_i, and action a'_i. This description might seem deterministic

$$m_i, s_i, a_i \rightarrow m'_i, s'_i, a'_i$$

But in fact, the new message queues can be changed by the processes that access them, and in one transition of P_i, an arbitrary number of connected processes P_j could have taken an arbitrary number of steps, so the production of m'_i is not deterministic even given the local schedule sch_i. Also, the transition from s_i to s'_i can be the execution of

any computable function on the state, so the *level of atomicity* in our model can be very "large," in terms of the number of steps that it takes to compute the state transition.

Given the whole network and the schedules, sch_i, we can define deterministic execution indexed by natural numbers t.

$$\langle m_i, s_i, a_i\rangle @t \ \ \overrightarrow{sch_i} \ \ \langle m_i', s_i', a_i'\rangle @t+1$$

for each i in $\mathbb{N}$, a process might "stutter" by not taking any action ($s_i' = s_i, a_i' = a_i$, and *outbound* message links are unchanged). If we do not provide the scheduler, then the computation is underdetermined.

Fair Scheduling We are interested in *all possible fair executions*, i.e. all possible *fair schedules*. A fair schedule is fair if each action is tried infinitely often. If a guard is true when an action is scheduled, then the action is executed. A round-robin scheduler is fair. Note, if a guard is always true, then the action is eventually taken. This gives rise to a set of possible executions, $\mathcal{E}$. We want to know those properties of systems that are true in all possible fair executions.

2.4. Events

As a distributed system executes, we focus on the changes, the "things that happen." We call these *events*. Events happen at a processes P_i. In the base model, events have no duration, but there is a model wtih time where they have duration. We allow that the code a process executes may change over time as sub-processes are added, so we imagine processes as *locations* where local state is kept; they are a *locus of action* where events are sequential. This is a key idea in the concept of cooperating (or communicating) sequential processes [Dijkstra, Hoare]. So in summary, we say that all events happen at a location (locus, process) which can be assigned to the event as $loc(e)$. We also write $e@i$ for events at i.

Events have two dimensions, or aspects, local and communication. An update is local and can send a message, and a receive event can update state, modify the state and send a message. These dimensions are made precise in terms of the order relations induced. The dimensional is sequentially ordered, say at P_i, $e_0 < e_1 < e_2 < ...$ starting from an initial event. The communication events are between processes, say $e@i$ receives a message from $e'@j$, than $e' < e$ and in general sender$(e) < e$. (Note, a process can send to itself so possibly $i = j$.) The *transitive closure* of these distinct orderings defines Lamport's causal order, $e < e'$. To say $e < e'$ means that there is a sequence of local events and communication events such that

$$e = e_1 < e_2, < ... < e_n = e'.$$

One of the most basic concepts in the theory of events is that *causal order is well-founded*. If $f(e)$ computes the immediate predecessor of e, either a local action or a send, then $e > f(e) > f(f(e)) > e_0$. The order is *strongly well founded* in that we can compute the number of steps until we reach an initial event.

Event Structures There are statements we cannot make about an asynchronous message passing system. For example, there is no global clock that can assign an absolute time t to every event of the system. It is not possible to know the exact time it takes from sending a message to its being read. We don't always know how long it will take before a pending action will be executed.

What can we say about executions? *What relationships can we reason about*? The simplest relationship is the *causal order* among events at locations. We can say that a state change event e at i causes another one e' at i, so $e < e$! We can even say that e is the predecessor of e', say $pred(e') = e$. We can say that event e at i sends a message to j which is received at event e' at j. We will want to say $sender(e') = e$. Also $pred(e') = e$.

The language for causal order involves events e in the type E of all events. These occur at a location from the type Loc. If we talk about $E, Loc, \leq$ we have *events with order*. This is a very spare and abstract account of some of the expressible relationships in an execution of a distributed system.

Given an execution (or computation) *comp*, we can pick out the locations say $P_1, P_2, P_3,$ and the events - all the actions taken, say $e_1, e_2, e_3, ...$ Each action has a location apparent from its definition, say $loc(e)$. Some of the events are comparable $e_i < e_j$ and others aren't, e.g. imagine two processes that never communicate $e_1, e_2, ...$ at i and $e'_i, e'_2, ...$ at j. Then never do we have $e_i \leq e'_j$ nor $e_j \leq e_i$. These events and their relationship define an *event structure* with respect to $E, Loc, \leq$.

It is natural to talk about the *value* of events which receive messages, the value, $val(e)$, is the message sent. For the sake of uniformity, we might want all events to have a value, including internal events. In the full theory implemented in Nuprl, this is the case, but we do not need those values in this work.

Temporal Relationships We can use events to define temporal relationships. For example, when an event occurs at location i, we can determine the value of any identifier x referring to the state just as the event occurs, x **when** e. To pin down the value exactly, we consider the kind of action causing e. If e is a state change, say $x := f(state)$, then x **when** e is the value of x used by f. If the action is reading the input queue at link $< i, j >$ labeled by e, the value of x is the value during the read which is the same as just before or just after the read because reads are considered atomic actions that do not change the state. They do change the message queue.

In the same way we can define x **after** e. This gives us a way to say when an event changes a variable, namely "e changes x", written $x\Delta e$ is defined as

$$x\Delta e \text{ iff } \textbf{after } e \neq x \textbf{ when } e.$$

This language of "temporal operators" is very rich. At a simpler level of granularity we can talk about the *network topology* and its labeled communication links $\ell < i, j >$. This is a layer of language independent of the state, say the network layer. The actions causing events are *send* and *receive* on links.

Computable Event Structures We are interested in event structures that arise from computation, called *computable event structures*. There is a class of those arising from the execution of distributed systems. We say that these structures are *realizable* by the sys-

tem, and we will talk about statements in the language of events that are realizable. For example, we can trivially realize this statement at any process: there is an event e_i at P that sends a natural number, and after each such event there is a subsequent event e_{i+1} that sends a larger number, so at the receiving process there will be corresponding events e.g. $val(e'_i) < val(e'_{i+1}) over\ \mathbb{N}$. To realize this assertion, we add to P a clause that initializes a variable $counter$ of type (N) and another clause that sends the counter value and then increments the counter.

2.5. Specifying Properties of Communication

Processes must communicate to work together. Some well studied tasks are forming *process groups*, electing group *leaders*, attempting to reach *consensus*, synchronizing actions, achieving *mutually exclusive* access to resources, tolerating *failures* of processes, taking snapshots of state and keeping secrets in a group. Security properties usually involve properties of communication, and at their root are descriptions of simple handshake protocols that govern how messages are exchanged. These are the basis of *authentication* protocols. As an example, suppose we want a system of processes $\mathcal{P}$ with the property that two of its processes, say S and R connected by link ℓ_1 from S to$R$ and ℓ_2 from R to S should operate using explicit acknowledgement. So when S sends to R on ℓ_1 with tag tg, it will not send again on ℓ_1 with this tag until receiving an acknowledgement tag, ack, on ℓ_2. The specification can be stated as a theorem about event structures arising from extensions $mathcal(P)'$ of $\mathcal{P}$, namely:

Theorem 1 *For any distributed system $\mathcal{P}$ with two designated processes S and R linked by $S \overset{\ell_1}{\rightarrow} R$ and $R \overset{\ell_2}{\rightarrow} S$ with two new tags, tg and ack, we can construct an extension $\mathcal{P}'$ of $\mathcal{P}$ such that the following* ***specification*** *holds: $\forall e_1, e_2 : E. loc(e_1) = loc(e_2) = S\ \&\ kind\ (e_1) = kind(e_2) = send(\ell_1, tg).\ e_1 < e_2 \Rightarrow \exists r : E.\ loc(r) = S\ \&\ kind\ (r) = rcv(\ell_2, ack).\ e_1 < r < e_2$.*

This theorem is true because we know how to add clauses to processes S and R to achieve the specification, which means that the specification is constructively *achievable*. We can prove the theorem constructively and in the process define the extension $\mathcal{P}'$ implicitly. Here is how.

Proof: What would be required of $\mathcal{P}'$ to meet the specification? Suppose in $\mathcal{P}'$ we have $e_1 < e_2$ as described in the theorem. We need to know more than the obvious fact that two send events occurred namely $< tg, m_1 >, < tg, m_2 >$ were sent to R. One way to have more information is to remember the first event in the state. Suppose we use a new Boolean state variable of S, called rdy, and we require that a send on ℓ_1 with tag tg happens only if $rdy = true$ and that after the send, $rdy = false$. Suppose we also stipulate in a frame condition that *only a receive on ℓ_2* sets ready to true, then we know that rdy **when** $e_1 = true$, rdy **after** $e_1 = false$ and rdy **when** $e_2 = true$. So between e_1 and e_2, some event e' must happen at S that sets rdy to true. But since only a $rcv(\ell_2, ack)$ can do so, then e' must be the receive required by the specification.

This argument proves constructively that $\mathcal{P}'$ exists, and it is clear that the proof shows how to extend process S namely add these clauses:

a : **if** $rdy = true$**then**
$\quad send(\ell_1, < tg, m >); rdy := false$
r : **rcv**(ℓ_2, ack) **effect** $rdy := true$
$\quad only[a, r]$ **affect** rdy

QED

We could add a liveness condition that a send will occur by initializing rdy to true. If we want a live dialogue we would need to extend R by

$$\textbf{rcv}(\ell_1, < tg, m >) \textbf{ effect send } (\ell_2, ack)$$

but our theorem did not require liveness.

Now suppose that we don't want to specify the communication at the basic level of links but prefer to route messages from S to R and back by a *network* left unspecified assuming no attackers. In this case, the events e_1, e_2 have destinations, say $kind(e_2) = sendto(R, tag)$ and $kind(r) = rcvfrom(R, ack)$. The same argument just given works assuming that there is a delivery system, and the frame conditions govern the message content not the links.

We might want to know that the communication is actually between S and R even when there is potential eavesdropper. What is required of the delivery system to authenticate that messages are going between S and R? This question already requires some security concepts to rule out that the messages are being intercepted by another party pretending to be R for example.

Here is a possible requirement. Suppose S and R have process identifiers, $uid(S), uid(R)$. Can we guarantee that if S sends to R at event e_1 then sends to R again at e_2, there will be a message r such that $e_1 < r < e_2$ and from receiving r, S has evidence that at R there was an event v_1 which received the $uid(S)$ and tg from e_1 and an event v_2 at R that sent back to S an acknowledgement of receiving tag from S? This can be done if we assume that the processes S and R have access to a Signature Authority (SA). This is a *cryptographic service* that we will describe in the next section building it using nonces.

A plausible communication is that S will sign $uid(S)$ and $sendto(R)$. Let $sign_S(m)$ be a message signed by S. Any recipient can ask the signature authority SA to verify that $sign_S(m)$ is actually signed by S. So when R receives $sign_S(uid(s))$ it verifies this and sends $sign_R(uid(R))$ back as acknowledgement. An informal argument shows that this is possible, and only R can acknowledge and does so only if S has signed the message. We take up this issue in the next section.

3. Intuitive Account of Cryptographic Services

In the previous section we suggested how a signature authority can be used to guarantee that a message came from the process which signed it. That argument can be the basis for guaranteeing that the signing process receives a message. If the message includes a nonce created by another process, say by S at event e, then it is certain that the signing event s

came after e, thus $e < s$. Thus nonces can be used to signal the start of a communication exchange as we saw earlier, and in addition, for us they are the basis of the signature authority as well as a public key service. Thus we will examine a Nonce Service and the concept of a nonce first.

3.1. Nonce Services and Atoms

Informal definitions of nonce might say "a bit string or random number used only once, typically in authentication protocols." This is not a very precise or suggestive definition, and it is not sufficiently abstract. We will provide a precise definition and explain other uses of nonces. One way to implement them is using a long bit string that serves as a random number that is highly unlikely to be guessed. We will not discuss implementation issues, nevertheless, the standard techniques will serve well to implement our abstract definition.

Nonce Server We will define a Nonce Server to be a process NS that can produce on demand an element that no other process can create or guess. Moreover, the NS produces a specific nonce exactly once on request. So it is not that a nonce is "used only once," it is created exactly once, but after it is created, it might be sent to other processes or combined with other data. But how can a Nonce Server be built, how can it provide an element n that no other process can create, either by guessing it or computing it out of other data? To answer this question, we must look at the concept of an Atom in a type theory such as Computational Type Theory (CTT), a logic implemented by Nuprl and MetaPRL. Then we will explain how a Nonce Server is built using atoms.

The Type of Atoms The elements of the type Atoms in CTT are abstract and unguessable. Their semantics is explained by Stuart Allen [6]. There is only one operation on the type Atom, it is to decide whether two of them are equal by computing $atomeq(a, b)$ whose type is a Boolean. The canonical elements of Atom are $tok(a), tok(b)$,...where the names a, b,... are not accessible except to the equality test and are logically indistinguishable.

A precise way to impose indistinguishability is to stipulate a permutation rule for all judgements, $J(a, b, \ldots)$ of CTT logic. Any such judgement is a finite expression which can thus contain only a finite number of atoms, say $a, b, \ldots$. The permutation rule asserts that if $J(a, b, \ldots)$ is true, then so is $J(a', b', \ldots)$ where $a', b' \ldots$ are a permutation of the atoms. Moreover, the evaluation rules for expressions containing atoms can only involve comparing them for equality and must reduce to the same result under permutation.

It is important to realize that any finite set of atoms, A, say $a_1, \ldots, a_n$ can be enumerated by a function f from $\{1, ..., n\}$ onto A. However, any such function is essentially a table of values, e.g. $if\ x = 1\ then\ a,\ else\ if\ x = 2\ then\ a_2 \ldots$. Thus the function f will *explicitly mention* the atoms $a_1, \ldots, a_n$. Thus if we stipulate that a process does not mention an atom a_i, then there is no way it can compute it.

The type of all atoms, Atom, is not enumerable because it is unbounded, so any enumerating function expression would be an *infinite table* which is not expressible in any type theory. On the other hand, Atom is not finite either because for any enumeration

$a_1, \ldots a_n$ there is an atom a not among them. Notice that in standard classical set theory such as ZFC, there is no set such as Atom because any set A is either finite or infinite. Even if we add an infinite set of urelements to ZFC, say ZFC(U), this set U is finite or infinite.

In the Nuprl implementation of Atom, we use $tak(a)$ where a is a string, but each session of Nuprl is free to permute these strings so the user never knows what they are from one computation to the next.

A Nonce Server can be built as a process NS that has in its state a large finite list of atoms, say L. The process can access the list only by a pointer variable ptr which is initialized to the head of L. When a request is made for a nonce, the atom $L(ptr)$ is returned and the pointer is advanced. The frame conditions on Nonce Server are that only ptr can access L, the ptr only increases, no other state variable has type Atom, and the code at Nonce Server does not contain an atom. Moreover, no other process P has the atoms of L in its state initially nor in its transition function.

The fundamental property on NC is called the *Fundamental Nonce Theorem*. It is defined in Bickford [10] and proved using Nuprl by Bickford. Informally it says this.

Theorem 2 *If $Nonce(n)$ is the value returned by a request to NS and e is any event, then either $val(e)$ does not contain $Nonce(n)$ or $n \leq e$.*

To build a Signature Authority (SA) we use a similar mechanism. When process $uid(P)$ wants to sign data d, the SA process puts $< uld(P), d >$ in a row of a table indexed by new atom a. The atom is the signed data to verify that $uid(P)$ signed d, the verifying process sends a and the data $< uid(P), d >$ to SA. The SA process checks that row a consists of $< uid(P), d >$.

To understand how we precisely describe a Nonce Server and a Signature Authority, we need to be precise about the idea that a value v of type theory does not mention an atom a. We will say that v *is independent of* a, written $v \parallel a$, or more fully, specifying the type of v, $v : T \| a$.

3.2. Independence

Urelements As we mentioned earlier, there is an analogue of the Atom type in set theory, and a simple definition of independence. The set of *urelements* in set theory is like the type of atoms in the sense that urelements are unstructured nonsets. Let ZFC(U) be ZFC set theory with a set U whose elements are nonsets. We could even take ZFC(Atom). In this theory, to say that a set x is independent of atom a is to say that $\neg(a \in x)$.

To say that an expression exp of type T is independent of atom a, written $exp : T \parallel a$, is to say that "exp does not contain a." In the case where exp is a closed term such as 17 or $\lambda(x.x)$ or $< 17, a >$ to say it is independent of a is to say that a does not occur in the expression. Clearly $17 : \mathbb{N} \| a$, $\lambda(xx) : A \to A \| a$ but $< 17, a >$ is not independent of a.

Extensional Type Theories In an extensional type theory such as CTT, independence is more involved than is apparent from the case of closed terms. Types are essentially equivalence classes of terms, and to say that $t\|a$ is to say that some "member of $t's$ equivalence class" does not mention a. The expression $\lambda(x.\lambda(y.x)(a))$ is extensionally equal to $\lambda(x.x)$, and even though it is closed and equal to $\lambda(x.x)$, a occurs in it. However, if we apply the expression to an integer like 17, we get 17. So, unless it is applied to an atom, the result is not an atom, and we can say

$$\lambda(x.\lambda(y.x)(a)) : \mathbb{N} \to \mathbb{N}\|a.$$

We will see below that if $f = f'$ and $f' : S \to T\|a$ then $f : S \to T\|a$. It is also clear that if $f : S \to T\|a$ and $s : T\|a$, then $f(s) : T\|a$.

In Bickford [10], there is a simple axiomatization of the concept of independence that has worked well in practice and is the basis for a collection of tactics that automates reasoning about independence. We present this small complete set of axioms in the next section, table 6. Here we describe them informally.

3.3. Rules for Independence from Atoms

The most basic rule about independence is called *Independence Base*, and it says that any closed term t of type T is independent of atom a exactly when a does not occur in t. The most obvious case includes instances such as a does not occur in the number 0. We write this as $0 : \mathbb{N}\|a$ or when the type is not important, $0\|a$.

Another basic rule is that if $t = t'$ in T, and $t : T\|a$, then $t' : T\|a$. This is quite important as a basis for extending independence from the obvious base case to more subtle examples such as $\lambda(x.0)(a) : \mathbb{N}\|a$ even though a does occur in the expression $\lambda(x.0)(a)$. Since $\lambda(x.0)(a)$ reduces to 0, we have $\lambda(x.0)(a) = 0\ in\ \mathbb{N}$. The rule is called *Independence Equality*.

The equality rule is justified by the basic fact of type theory, following from Martin-Löf's semantics, that in any type T, if t reduces to a canonical term t' and $t' \in T$, then $t = t'$ and $t \in T$. The canonical members of a type determine its properties, and these properties must all respect equality. So if proposition P is true of t, $P(t)$, and t reduces to t' in T, then $P(t')$ as well. So for $t : T\|a$ to be a proposition of type theory, it must obey this equality rule. Indeed, the rule requires that if $T = T'$ and $a = a'$, then $t' : T'\|a'$ holds.

The application rule says that if $f : s \to T\|a$ and $s : S\|a$, then $f(s) : T\|a$. The basic idea is that if atom a does not appear in a closed program for f, then the computation can't produce a result with a in it. This rule is clear under the set theoretic idea of a function, however in type theory it is much more subtle, depending on extensional equality. it also depends on the type types. For example, consider this function

$$\lambda(x.if\ even(x)\ then\ 0\ else\ a).$$

As a function from even numbers, it is independent of a, because it is equal to $\lambda(x.0)$ on even number. But over $\mathbb{N}$, it is not independent of a.

The above three rules are critical ones. We also need a way to say that $\neg(a : Atom \| a)$ which we take as an axiom, and that if $\neg(a = b\ in Atom)$, then $b : Atom \| a$. The full type theory needs to treat other constructors explicitly, such as subtyping, $A \sqsubseteq B$, set types, $\{x : A | P(x)\}$, quotient types $A//Eq.$, the Top type, and others which we won't discuss.

3.4. Authentication Example

Suppose we have a group (G) of processes that agree on how to authenticate communication, that is, for any pair A (Alice) and B (Bob) distinct processes, they have a way to "know" after an exchange of messages that

1. A sent a nonce to B that A created and only B received.
2. B sent a nonce to A that B created and only A received. On this basis, other messages m can be sent with the same property:

   A sent, only B received in this exchange.
   B sent, only A received in this exchange.

The processes in G all have an authentication protocol, AP and events from this protocol can be recognized as satisfying a predicate $AP(e)$. Other processes not in G can recognize these events as well they have a distinct tag. The processes in G can draw conclusions from messages satisfying AP as long as each process follows the protocol - some authors [29] say as long as the processes in G are *honest*.

Here is the original *Needham-Shroeder* authentication protocol treated in event logic, and we reveal the flaw found by Lowe in the original Needham-Schroeder correctness argument, creating the correct *Needham-Schroeder-Lowe (NSL) protocol*. LetK_B encrypt using B's public key. Our method is to show what statements A and B *believe* and *conjecture* at various events.

If B receives an initiation message $K_B(< n_A, uid(A) >)$ at b encrypted by B's public key, then B decodes the message and checks that $uid(A) \in G$ and if so conjectures that

B_1 : n_A is a nonce created by A at some event $a < b$.
B_2 : A encrypted the nonce at some a', $a < a' < b$.
B_3 : A sent the nonce to B at some a'', $a < a' < a'' < b$.
B_4 : No other process E received the nonce at e for any e satisfying $a < a' < e < b$.

Thus by the nonce property, no other process has the nonce in its state, e.g. only A and B have it. B acts in such a way as to prove these properties $B_1 - B_4$.

B already knows

- Some process E encrypted $< n_A, uid(A) >$ and sent it to B.

Thus there is $event\ e_1$ such that $e_1 < b,\ e_1$ sends $K_B(< n_A, uid(A) >)$. Note E is by definition $loc(e_1)$, and B is trying to prove $E = A$.

B creates a nonce n_B and knows $Nonce(n_B)$. B encrypts $< n_A, n_B >$ with A's public key, following the NS protocol. B sends $K_A(< n_A, n_B >)$ to A.

A already knows that there is an event a at which it created n_A and a_1 after a at which it sent $K_E(< n_A, uid(A) >)$ to E. A assumes E is the target for an authentication pair A, E, and believes:

A_1 : There will be an event $e_1,\ \ a < e_1$ at which
E receives $K_A(< n_A, uid(A) >)$.
A_2 : There will be an event $e_2,\ \ a < e_1 < e_2$ at which E decrypts.
A_3 : Since E knows the protocol, it will send a nonce,
n_E along with n_A encrypted by K_A back to A.

A receives $K_A(< n_A, n_B >)$, decodes to find n_B, it knows E received n_A and sent it back. If E follows the NS protocol, then n_B is a nonce created by E.
Thus A conjectures as follows:

A_4: n_B is a nonce created by E.
A_5: If E receives $K_E(n_B)$ it will know A, E is an authentication pair.

A sends $K_E(n_B)$.
B receives $K_B(n_B)$ and knows:

- A received, decrypted, and resent n_B.
- A recognized its nonce n_A, and is in (G), thus following NS.

So B now deduces that:

- B_1 is true since A continued to follow NS.
- B_2 is true since A followed NS.

The original protocol assumed that B would know B_3 as well and hence deduce B_4. But B can't get beyond B_2. B does not have the evidence that A encrypted $< n_A, uid(A) >$.

What evidence is missing? There is no evidence at B that $E = A$, just as there is no evidence at A so far that $E = B$. So the NS protocol allows a process that is neither A nor B to have both n_A and n_B. This is a mistake as Lowe [19] noticed (in trying to formally prove NS correct). He proposed a fix, that B be required to send $K_A(< n_A, n_B, uid(B) >)$, so that A can check who sent n_B.

Notice, if A received $K_A(< n_A, n_B, uid(B) >)$ then it would see a problem because it used K_E to send $< n_a, uid(A) >$. If $E = B$, there is no problem for A, so it will continue the protocol, and then if B receives $K_B(n_B)$ it knows that A sent the nonce, so it knows B_3 and hence B_4. So the Lowe fix gives a correct protocol for creat-

ing an authentication pair under NSL. We can turn this into an event logic proof which shows that what A and B believe is actually true in all fair executions of this protocol, thus in the corresponding event structures. Other processes might guess the message being encrypted, but they will not receive the atom which is the encryption of it for the same reason that they do not receive the nonce used in authentication.

Deriving Needham-Schroeder-Lowe from a theorem . Assume (G) is a group of processes that "seek the capability" to have private communications based on shared nonces and plan to establish that an arbitrary pair of them can authenticate each other, A "knows" it is communicating with B and B "knows" it is communicating with A, and no third party can "listen in" by means of software.

Theorem 3 *If there is an exchange of messages between A and B such that*

1. *A creates a new nonce n_A at a_0 and sends $K_B(< n_A, uid(A) >)$ to B at a_1 after 4_0 and*
2. *B receives $K_B(< n_A, uid(A) >)$ at b_1 after a_1 and decodes it at b_2, creates a new nonce n_B at b_3, and sends $K_A(< n_A, n_B, uid(A) >)$ at b_4 and*
3. *A receives $K_A(< n_A, n_B, uid(A) >)$ at a' after b_4 and decodes the contents at a'_2 after a'_1, and sends $K_B(n_B)$ to B at a'_3 after a'_1 after b'_0*

Then $C1$: only A and B have the nonces n_A and n_B, and $C2$: the corresponding event pairs are matching sends and receives.

$$\begin{array}{ll} a_1 & b_1 \\ a'_1 & b_4 \\ a_3 & b'_1 \end{array}$$

Also at a'_2 and b'_2 each process believes the conclusions $C1$ and $C2$, so the beliefs are correct.

4. Technical Details

Our goal is, to model processes as “all distributed programs” and to carry out security proofs in a general purpose logic of distributed systems. By doing this, security theorems have a greater significance since we will be proving impossibility results for adversaries in a general computation system. We will also be using the same logical framework for all proofs about programs – security proofs will not be done in a special logical system. Therefore the confidence in the soundness of our logic and the correctness of our tools that we gain from using them for proofs that programs satisfy any kind of specification will apply also to our security proofs. If a security property depends on cryptography as well as a non-trivial distributed algorithm, then we can verify both parts of the protocol in one system.

4.1. Event Structures

In this section we first present enough of the language of event structures (which we call *event logic*) to explain the semantics of all but the read-frame clause. Then we will

discuss some extra structure that must be added to allow us to give the semantics for the read-frame clause.

The details of how a mathematical structure is represented in type theory (as a dependent product type which includes its axioms via the *propositions as types isomorphism*) is not relevant to this paper, so we present event structures by giving the signatures of the operators it provides and describing the axioms. In the following, $\mathbb{D}$ denotes a universe of types that have decidable equality tests, and the types **Loc**, **Act**, and **Tag** are all names for the same type **Id** of identifiers. This type is actually implemented, for software engineering reasons related to namespace management and abstract components, as another space of atoms like the ones used here for protected information. Here **Id** is just a type in $\mathbb{D}$. The type **Lnk** is the product **Loc** $\times$ **Loc** $\times$ **Id**, so a link l is a triple $\langle i, j, x\rangle$ with **src**$(l) = i$ and **dst**$(l) = j$.

Basic Event Structures The account will be layered, starting with the most basic properties of events and adding layer by layer more expressiveness.

Table 1. Signatures in the Logic of Events

Events with Order	**Definitional extensions**
E: $\mathbb{D}$	**loc:** $E \rightarrow Loc$
pred?: $E \rightarrow (E + Loc)$	**first:** $E \rightarrow Bool$
sender?: $E \rightarrow (E + Unit)$	**isrcv:** $E \rightarrow Bool$
	$x < y\,, \quad x <_{loc} y$
and with Values	
Kind $= Loc \times Act + Lnk \times Tag$	**sender:** $\{e : E \mid isrcv(e)\} \rightarrow E$
vtyp: $Kind \rightarrow Type$	**link:** $\{e : E \mid isrcv(e)\} \rightarrow Link$
kind: $e : E \rightarrow Kind$	**tag:** $\{e : E \mid isrcv(e)\} \rightarrow Tag$
val: $e : E \rightarrow vtyp(kind(e))$	
and with State	
typ: $Id \rightarrow Loc \rightarrow Type$	**state**$(i) = x : Id \rightarrow typ(x, i)$
initially: $x : Id \rightarrow i : Loc \rightarrow typ(x, i)$	
when: $x : Id \rightarrow e : E \rightarrow typ(x, loc(e))$	**state-when:** $e : E \rightarrow state(loc(e))$
after: $x : Id \rightarrow e : E \rightarrow typ(x, loc(e))$	**state-after:** $e : E \rightarrow state(loc(e))$

Events with Order Events are the atomic units of the theory. They are the occurrences of atomic actions in space/time. The structure of *event space* is determined by the organization of events into discrete *loci*, each a separate locus of actions through time at which events are sequentially ordered. Loci abstract the notion of an agent or a process. They do not share state. All actions take place at these locations.

There are two kinds of events, internal actions and signal detection (message reception). These events are *causally ordered*, e before e', denoted $e < e'$. As Lamport postulated, *causal order* is the structure of time. Causal order is defined in terms of two primitive functions, $pred?$ and $sender?$ which compute respectively the previous action at its locus (if the event is not the first at that location) and the sender of a received message.

To give an idea of how these layers are formally presented, we show in table 1 the signature of some of the layers. In these definitions we use the *disjoint union* of two sets or types, $A + B$ and the computable function space operator $A \rightarrow B$. The type $Unit$ has a single distinguished element.

The signature of events with order requires only two discrete types E and Loc, and two partial functions. The function $pred?$ finds the predecessor event of e if e is not the first event at a locus or it returns the *location* if e is the first event. The *sender?(e)* is the event that sent e if e is a *receive*, otherwise it is a unit. We can find the location of an event by tracing back the predecessors until the value of *pred* belongs to *Loc*. This is a kind of partial function on E. From *pred? and sender?* we can define Boolean valued functions that identify the first event and receive events. We can define a function loc that returns the location of an event. Causal order, $e < e'$, is defined as the transitive closure of the relations $e = pred?(e')$ and $e = sender?(e')$. We can also define the local linear ordering of events at a location, $<_{loc}$, the restriction of causal order, $<$, to a location.

Events with Value We next classify events by their kind, by introducing the type $Kind$ and a function $kind$ from events to kinds. The type $Kind$ is a disjoint union that represents our two basic kinds: an internal action at a location, or the receive of a message on a link with a given tag. Each kind of action has a value associated with it. The value of a receive event is the message received. The value of an internal action can be chosen randomly or nondeterministically. The value of an event e is $val(e)$ and its type depends only on $kind(e)$.

Events with State We are interested in actions with observable results. Observables are known by *identifiers* and have *types*. At a fixed location or agent, the map of identifiers to values is its *state*. Relations **when**, **after**, and **initially** (which we write with infix notation) connect events to the values of identifiers, e.g. x **when** e is the value of the variable x at the location $loc(e)$ when event e occurs. For the basic event structures, we need only the six simple axioms listed in table 2.

Table 2. Axioms of Basic Event Structures

1. On any link, an event sends boundedly many messages; formally: $\forall l : Link. \forall e : E. \exists e' : E. \forall e'' : E. R(e'', e) \Rightarrow e'' < e' \wedge loc(e') = dst(l)$ where $R(e'', e) \equiv isrcv(e'') \wedge sender(e'') = e \wedge link(e'') = l$
2. The predecessor function is one-to-one; formally: $\forall e_1, e_2 : E.\ pred?(e_1) = pred?(e_2) \Rightarrow e_1 = e_2$
3. Causal order is (strongly) well-founded; formally: $\exists f : E \rightarrow \mathbb{N}. \forall e_1, e_2 : E.\ e_1 < e_2 \Rightarrow f(e_1) < f(e_2)$
4. The location of the sender of an event is the source of the link on which the message was received; formally: $\forall e : E.\ isrcv(e) \Rightarrow loc(sender(e)) = src(link(e)$
5. Links deliver messages in FIFO (first in first out) order; formally: $\forall e_1, e_2 : E.\ link(e_1) = link(e_2) \Rightarrow sender(e_1) < sender(e_2) \Rightarrow e_1 < e_2$
6. State variables change only at events, so that: $\forall e : E. \neg\ first(e) \Rightarrow (x \textbf{ when } e) = (x \textbf{ after } pred(e))$

4.2. Message Automata

In our theory, all processes can be built out of nine basic clauses by composition. We call the resulting family of realizers *message automata*. As we said in the informal intro-

Table 3. Message Automaton Frame Clauses

only k1,k2,... affect x **at location** i Only actions with kind in the given list may affect the state variable x at location i.
only k1,k2,... send on link l **with tag** tg Only actions with kind in the given list may send messages tagged tg on link l .
only k1,k2,... read x **at location** i Only actions with kind in the given list may read the state variable x at location i.
k **affects only** x,y,... **at location** i An action of kind k affects only state variables in the given list.
k **sends only on links** l1,l2,... An action of kind k sends only on links in the given list.

duction, a message automaton is a representation of a distributed program. The behavior of such a program will be an infinite history (which we will abstract to form an *event structure*) but the program itself is a finite object. We may define a *message automaton* as a *finite set of clauses* and a *clause* as an instance of one of the following nine schemes, which we partition into four *active clauses* and five *frame clauses*. In this abstract syntax, *locations* i,j,... and *state variables*, x,y,... are simply identifiers, while an *action kind*, k,k',... is either a *internal action* $internal(i, a)$ (where a is an identifier) at some location i, or a *receive action*, $rcv(l, tag)$, where l is a *link*, which is a triple (source, destination, name) of identifiers, and tag is an identifier (used to partition the messages received on the link into classes of different types or different meanings). Every action kind k has a unique location: if k is $internal(i, a)$ then its location is i and if k is $rcv(l, tag)$ then its location is the destination of link l. The full syntax of the message automaton clauses includes abstract syntax for declaring the types of state variables and tagged messages, but to avoid overloading the reader with details we will omit those parts of the syntax since the essential concepts can be understood without them. In tables 4 and 3 we indicate, for each of the nine clause schemes, the name we use for it, and its syntactic form, followed by its intended meaning. The formal meaning is defined by the event structures that are consistent with the clause, which we discuss in the next section. Message automata A and B are sets of these clauses, and we write $A \oplus B$ for the union of the clauses from both A and B, and call it the *join* of A and B. The join is the basic composition operator on automata. We can generate runnable code (we currently generate Java) from a message automaton (and a configuration table that maps locations to hosts and link names to ports) [13]. Only the active clauses generate any code; the frame clauses only restrict the set of message automata that are feasible. A message automaton is *feasible* if there is at least one event structure consistent with it, and in particular, a feasible automaton must obey all of its own frame clauses. So, for example, an automaton that contained both clauses

effect of internal(i,a) **is** x:=f(state,val)
only [rcv(l,tag), internal(i,b)] **affect** x **at location** i

or both clauses

effect of internal(i,a) **is** y:= x + 1
only [rcv(l,tag), internal(i,b)] **read** x **at location** i

would be infeasible (unless a = b), since, in the first case, $internal(i, a)$ affects state variable x but is not listed in the frame clause given for x at location i, and, in the second case, $internal(i, a)$ reads variable x to update variable y, but is not listed in the read-

frame clause given for x at location i. There is an essentially syntactic check for feasibility (modulo type checking, which, in an expressive logic, can require theorem proving), so we could implement a compiler that refuses to generate code for an "illegal" program that fails the feasibility test.

Table 4. Message Automaton Active Clauses

at location i **initially** x = v In the initial state of agent i, the state variable x has value v.
effect of k **is** x:=f(state,val) Every action of kind k with a value v, updates the state variable x, at the location of k, to the value $f(s, v)$ where s is the current state.
k **sends on link** l : f(state,v) Every action of kind k with a value v, sends on link l a (possibly empty) list of tagged messages, $f(s, v)$ where s is the current state.
precondition internal(i,a) **is** P(state) An internal action of kind $internal(i, a)$ may not occur at i unless P is true in the current state, and, infinitely often, the agent either checks that P is false or performs an action of kind $internal(i, a)$.

Semantics of Message Automata The logical semantics of a message automaton M is the set of event structures es that are *consistent* with it, so the semantics can be defined by a relation $Consistent(es, M)$. We define the semantics so that an event structure is consistent with an automaton if and only if it is consistent with each of its clauses. This reduces the definition of the semantics to a base case for each clause scheme, and gives use the rule that

$$Consistent(es, A \oplus B) \Leftrightarrow Consistent(es, A) \wedge Consistent(es, B)$$

The semantics of a clause C is given by a formula, ΨC, in event logic, that describes how the clause C constrains the observable history es of the system. The relation $Consistent(es, C)$ is then defined by

$$Consistent(es, C) \Leftrightarrow (es \models \Psi C)$$

We give a simplified version of the semantics in table 5. The simplifications are that we suppress those parts of the constraints relating to the type declarations of the state variables and action values that we have omitted from the simplified syntax. We also treat all the state variables as *discrete* variables; in the full theory we also allow state variables to be functions of time, so that we can reason about clocks and real-time processes. Also, the syntax for the precondition clause allows the value of a local action to be chosen *randomly* from a finite probability space (like $[1/3, 1/6, 1/2]$) and the semantics of this is in terms of a theory of *independent random processes*, given the precise definition of independence of the previous section.

Feasibility and Realizability We have a formal definition of a predicate $Feasible(M)$ on message automata that defines an essentially syntactic check that M is internally consistent. We have a (rather difficult) fully formalized, constructive proof that

$$Feasible(M) \;\Rightarrow\; \exists es.\, Consistent(es, M)$$

We say that M *realizes* specification ψ if M is feasible and every event structure consistent with M satisfies ψ.

$$M \textbf{ realizes } \psi \equiv (Feasible(M) \;\wedge\; \forall\, es.\, Consistent(es, M) \;\Rightarrow\; es \models \psi)$$

If M **realizes** ψ then any feasible extension $M \oplus X$ of M also realizes ψ, so when ψ is a security specification we can see that a proof of M **realizes** ψ shows that adversaries expressible as a message automaton, X, cannot violate ψ unless they are able to violate the frame clauses in M.

Table 5. Semantics of Message Automaton Clauses (except read-frame)

$\forall e@i.\ P[e] \equiv \forall e : E.\ loc(e) = i\ \Rightarrow\ P[e]$ **state when** $e \equiv \lambda x.\ (x$ **when** $e)\ msgs(e,l) \equiv$
$[\ \langle tag(e'), val(e')\rangle \mid sender(e') = e \wedge link(e') = l\ ]$

at location i **initially** x = v $\forall e@i.\ first(e)\ \Rightarrow\ x$ **when** $e = v$
effect of k **is** x:=f(state,val) $\forall e : E.kind(e) = k\ \Rightarrow\ x$ **after** $e = f($**state when** $e, val(e))$
k **sends on link** l : f(state,v) $\forall e : E.\ kind(e) = k\ \Rightarrow\ msgs(e,l) = f($**state when** $e, val(e))$
precondition internal(i,a) **is** P(state) $(\forall e : E.\ kind(e) = internal(i,a)\ \Rightarrow$
$P($**state when** $e)) \wedge\ (\forall e@i.\ \exists e' \geq e.\ kind(e') = internal(i,a)\ \vee\ \neg P($**state after** $e'))$
only ks affect x **at location** $i\ \forall e@i.\ kind(e) \in ks\ \vee\ x$ **after** $e = x$ **when** e
only ks send on link l **with tag** tg $\forall e : E.\ kind(e) = rcv(l, tg)\ \Rightarrow\ kind(sender(e)) \in ks$
k **affects only** xs **at location** $i\ \forall e@i.\ kind(e) = k\ \Rightarrow\ \forall x : Id.\ x \in xs\ \vee\ x$ **after** $e = x$ **when** e
k **sends only on links** ls $\forall e.\ (isrcv(e)\ \wedge\ kind(sender(e))) = k\ \Rightarrow\ link(e) \in ls$

4.3. Independence

As we said in the introduction, our theory of processes admits all possible computable functions. Any theory of "all programs" must allow a program to apply any computable function and surely, for any data-type T that the secure agents can use to store protected information in their state, there must be an onto function $f : list(bit) \rightarrow T$, or since $list(bit)$ is equipotent with the natural numbers, a surjection $f : \mathbb{N} \rightarrow T$? An adversary, then, only has to discover, by eavesdropping, what the type T of protected information is and then start enumerating the range of f. Our solution to this problem is to base security on a very simple notion of what it means for an *agent to learn a secret* - namely that the secret is coded by atoms and the atoms are present in the state of the process. We will say that an atom a is unknown to a process i if for all events e at i, $state$ **when** e is independent of the atom a. We now make this idea precise. We have added a new primitive expression to CTT, the logic of Nuprl. The meaning of the new primitive is that the element x of type T is independent of the atom a; we write this as $(x : T \| a)$. Such an expression will be a type if $a \in Atom$, $T \in Type$[1], $x \in T$. Two expressions $(x : T \| a)$ and $(x' : T' \| a')$ represent the same type if $a = a' \in Atom$, $T = T' \in Type$, $x = x' \in T$.

Definition The proposition $(x : T \| a)$ is true if and only if a evaluates to $tok(b)$ for some name b, and there exists a term y in type T such that $x = y \in T$ and y does not mention name b. As is standard for propositions with no computational content, the members of the type $(x : T \| a)$ will be just the terms that evaluate to a fixed term Ax (for "axiom"), if the proposition is true, and the type will be empty if the proposition is false. This completes our definition of the new primitive proposition $(x : T \| a)$, and once we justify a few simple inference rules about it we have everything we need for our security application.

[1] In Nuprl. there is no type $Type$ of all types; instead there is a cumulative hierarchy of *universes*.In this paper, we use the symbol $Type$ for an arbitrary universe.

Lemma 4 (apply independence)

$$((f : (x : A \to B[x])\|a) \wedge (x : A\|a)) \Rightarrow (f(x) : B[x]\|a)$$

Proof If a evaluates to $tok(b)$ and $f = f' \in (x : A \to B[x])$ and f' does not mention b, and if $x = x' \in A$ and x' does not mention b, then the term $f'(x')$ does not mention b and by the definition of the dependent function type $x : A \to B[x]$, we have $f(x) = f'(x') \in B[x]$. □

Lemma 5 (independence absurdity)

$$(a : Atom\|a) \;\Rightarrow\; False$$

Proof If a evaluates to $tok(b)$ and $y = a \in Atom$ and y does not mention b, then we have $y = tok(b) \in Atom$, by computation, and $y = tok(c) \in Atom$, by the permutation rule for names. Thus $tok(b) = tok(c) \in Atom$, and this implies $False$. □

The complete set of rules for independence as implemented in the current Nuprl system are listed in table 6. The lemmas in this section prove the validity of the application and absurdity rules. In general, independence is not preserved by subtypes. If $(x : B\|a)$ and x is a member of a subtype A of B, then it may not be true that $(x : A\|a)$. A simple example of this is that for $a \in Atom$ we have $(a : Top\|a)$, $Atom$ a subtype of Top, but $\neg(a : Atom\|a)$. This is because Top is the type which has every closed term as a member but in which any two members are equal, so $a = 17 \in Top$ and 17 mentions no names. If, however, A is a subtype of B in which equality is just the restriction of the equality in B to the members of A, then independence is preserved. This is the justification for the last of the rules in table 6. Complete proofs can be found in [10].

Table 6. Rules for Independence

INDEPENDENCEEQUALITY

$$\frac{H \models T_1 = T_2 \in TypeH \models x_1 = x_2 \in T_1 H \models a_1 = a_2 \in Atom}{H \models (x_1 : T_1\|a_1) = (x_2 : T_2\|a_2) \in Type}$$

INDEPENDENCEBASE

$$\frac{H \models x \in TH \models a \in Atom\text{closed x mentions no names}}{H \models (x : T\|a)}$$

INDEPENDENCEATOMS

$$\frac{H \models \neg(x = a \in Atom)}{H \models (x : Atom\|a)}$$

INDEPENDENCEAPPLICATION

$$\frac{H \models (f : (v : A \to B)\|a)H \models (x : A\|a)}{H \models (f(x) : B[x/v]\|a)}$$

INDEPENDENCEABSURDITY

$$H, (a : Atom\|a), J \models T$$

INDEPENDENCESET

$$\frac{H \models (x : T\|a)H \models x \in \{v : T|\ P\}}{H \models (x : \{v : T|\ P\}\|a)}$$

To make these rule valid, the whole logic is constrained in several ways. The first constraint is that the names $a, b, \ldots$ are *unhideable*. This means that a definition like $f(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is not allowed because the names a and b occur on the righthand side of the definition but not on the lefthand side. If this were allowed, then we could prove a judgement that $f(1) = tok(a)$, and use the permutation

rule to conclude $f(1) = tok(b)$ and then conclude that $tok(a) = tok(b)$, which will compute to $False$. Definitions of this kind must include any names they mention among the parameters on the lefthand side, so $f\{a, b\}(x) = (\textbf{ if } x = 1 \textbf{ then } tok(a) \textbf{ else } tok(b))$ is an allowed definition.

Here is a sample theorem about independence.

Theorem 6

$$\forall a : Atom.\, \forall i : \mathbb{Z}.\, (i : \mathbb{Z} \| a)$$

Proof By induction on i: the case $i = 0$ follows from the Base rule. Assume $(i : \mathbb{Z} \| a)$ and show $(i + 1 : \mathbb{Z} \| a)$ and $(i - 1 : \mathbb{Z} \| a)$: $i + 1$ is $(\lambda x.\, x + 1)(i)$ so this case follow from the Application rule, the base rule, and the induction hypothesis. The $i - 1$ case is the same. **Qed**

Repeated use of the method of proof used in this theorem allows us to prove a very general theorem about event structures, namely that if the initial state of an agent is independent of atom a, and all messages received prior to event e are independent of a, then the **state when** e is independent of a. This basic result allows us to use the inductive approach to verifying cryptographic protocols initiated by Paulson [27] and elaborated by Millen and Ruess [23].

Using independence, we can formulate and prove the Fundamental Nonce Theorem in Nuprl. Here is the exact theorem proved.

```
∀i,i',a,nonce,L,ptr:Id.
((¬(ptr = L))
⇒ (∀as:Atom1 List. ∀es:ES.
(nonce-p(es;i;i';L;nonce;a;ptr;as)
⇒ nonce-assumption(es;i;L;as)
⇒ (∀e:E(Nonce(i;i'))
let a = Nonce(i;i')(e) in ∀e':E
((¬e c≤ e')
⇒ ((((¬(loc(e') = i))
⇒ es_state_when(es;e'):es_state(es;loc(e'))||a)
∧ ((loc(e') = i)
⇒ es_state_update(es_state_when(es;e');L;
λt.[]):es_state(es;loc(e'))||a))
∧ val(e'):valtype(e')||a
∧ e' sends || a))))))
```

Here are the two key definitions used in the proof. The second gives the frame conditions for the Nonce Server.

```
nonce-assumption(es;i;L;as) ==
no_repeats(Atom1;as)
∧ (∀a∈as.(∀j:Id. j || a)
∧ (∀e:E
((↑first(e))
⇒ (((¬(loc(e) = i))
              ⇒ es_state_when(es;e):es_state(es;loc(e))||a)
∧ ((loc(e) = i)
⇒ (((vartype(i;L) ⊆r (Atom1 List))
```

```
              ∧ ((Atom1 List) ⊆r vartype(i;L)))
∧ es_state_update(es_state_when(es;e);L;λt.[]):
              es_state(es;loc(e))||a))))))

nonce-p(es;i;i';L;nonce;a;ptr;as) ==
@i ptr initially 0:ℕ
∧ @i L initially as:Atom1 List
∧ @i events of kind rcv((link a from i' to i),nonce) change
              ptr to λs,v.
(s.ptr
+ 1) State(ptr : ℕ) (val:Top)
∧ @i only events in [] change
L : Atom1 List
∧ @i only events in [rcv((link a from i' to i),nonce)] change
ptr : ℕ
∧ @i: only members of [rcv((link a from i' to i),nonce)] read L
∧ @i: rcv((link a from i' to i),nonce) affects only [ptr]
∧ (@i:rcv((link a from i' to i),nonce) sends only on
              link/tags in  [<(link a from i to i'), nonce>]
∧ rcv((link a from i' to i),nonce)(v:Top)
sends on (link a from i to i') [nonce:Atom1, λsv.let s,v = sv
in
if s.ptr <z ||s.L||
then inl s.L[s.ptr]
else inr ·
fi  <state, v>]?[])
```

4.4. Verification

We are interested in problems of the form: find processes that exhibit behavior ϕ in a network G. We say that the processes *realize* the specification ϕ, and that the specification is *realizable*. We state this as the problem of proving that such a network exists, and create the proof in such a manner that we can *effectively find* the processes. This is the *process design problem* stated in a logical way. One way to do the proof is to explicitly write abstract code for the processes and prove it satisfies ϕ. The message automata defined below are the terms in the logic of events that correspond to code; *they include clauses that support reasoning about security*. We will see later that the processes can be defined implicitly as well, and our concepts for controlling access to information correspond to the security clauses in the automata.

Specifying Security Properties The general form of a *security assertion* will be that a program R (which is a feasible message automaton) satisfies a given property ψ, no matter how it is extended to another feasible automaton. So we may think of *adversaries* as any set of clauses X such that $R \oplus X$ is feasible, and R satisfies its security property ψ no matter what these adversaries X are. Thus, the only constraints that the adversaries must obey are the feasibility constraints, and these are essentially the frame conditions in R. These frame conditions are all local constraints. They constrain only how the agents that contribute clauses to R (call these the *agents in* R) read and write their own state variables and the links that they send on, so an adversary could only violate these constraints by adding code inside of the "process space" of one of the secure agents. We allow for

the possibility that the "adversary" X is actually some other code running at the same location (in the same process) as one of the agents in R, so when we prove that R satisfies a security property we will be saying that provided all the agents in R guarantee that all the code running at their location obey the local constraints in R, the security property will hold, no matter what extra code, at these locations or any other locations, has been added. We mention here that the last three frame clauses, the read-frame, action-frame, and action-send-frame, are needed only in the proof of security properties. The active clauses and the first two frame clauses suffice for a logic of program development for distributed systems when the specifications are properties like consensus, mutual exclusion, etc. This is because the first two frame conditions alow us to constrain programs enough to prove the usual kinds of state invariants needed to prove normal, non-security, kinds of specifications. It was only in attempting to specify and prove security properties that we discovered the need for the other frame clauses. These clauses all constrain "data-flow" and are essential in proofs that secret information will not be leaked.

Proof and Verification Formal proofs are very useful data objects in systems that have tools for manipulating them as our provers do. Proof objects can be modified to create many variants of an argument and to extract important dependency information; they can be transformed to different kinds of proofs, and any algorithms and processes that are *implicit* in them can be *extracted*. We have created completely formal proofs that several protocol specifications are achievable. These are organized in the style of the sequent calculus that underlies Coq, HOL, MetaPRL, Nuprl, PVS etc. Most elements of event logic can easily be formalized in any of these provers. Our proofs use tools from software model checking and SAT solvers as well as powerful tactics that automate much of the proof construction. There is a well-established theory and practice for creating *correct-by-construction* functional programs by extracting them from constructive proofs of assertions of the form $\forall x : A.\exists y : B.R(x, y)$ [14,9,15,22,26,18,17,28]. There have been several efforts to extend this methodology to concurrent programs [5,24], but there is no practice and the results are limited. In this subsection, we explain a *practical refinement method* for creating correct-by-construction and secure-by-construction processes (protocols).

Refinement proofs Suppose that we want to prove that ϕ is realizable, and we start a proof of the top-level goal $\models \phi$. From the form of the goal, the proof system knows that we must produce a feasible distributed system D that realizes ϕ so it adds a new abstraction $D(x, \ldots, z)$ to the library (where $x, \ldots, z$ are any parameters mentioned in ϕ). The new abstraction has no definition initially—that will be filled in automatically as the proof proceeds. This initial step leads to a goal where from the hypothesis that an event structure es is consistent with $D(x, \ldots, z)$ we must show the conclusion that $\phi(es)$, i.e., that es satisfies ϕ. Now, suppose that we can prove a lemma stating that in any event structure, es,

$\psi_1(es) \wedge \psi_2(es) \Rightarrow \phi(es)$.

In this case, the user can refine the initial goal $\phi(es)$ by asserting the two subgoals $\psi_1(es)$ and $\psi_2(es)$ (and then finishing the proof of $\phi(es)$ using the lemma). If ψ_1 is already known to be realizable, then there is a lemma $\models \psi_1$ in the library and, there is a realizer A_1 for ψ_1. Thus to prove $\psi_1(es)$, it is enough to show that es is consistent with A_1, and since this follows from the fact that es is consistent with $D(x, \ldots, z)$ and that $A_1 \subset D(x, \ldots, z)$, the system will automatically refine the goal $\psi_1(es)$ to

$A_1 \subset D(x, \ldots, z)$. If ψ_2 is also known to be realizable with realizer A_2 then the system produces the subgoal $A_2 \subset D(x, \ldots, z)$, and if not, the user uses other lemmas about event structures to refine this goal further.

Whenever the proof reaches a point where the only remaining subgoals are that $D(x, \ldots, z)$ is feasible or have the form $A_i \subset D(x, \ldots, z)$, then it can be completed automatically by defining $D(x, \ldots, z)$ to be the join of all the A_i. In this case, all the subgoals of the form $A_i \subset D(x, \ldots, z)$ are automatically proved, and only the feasibility proof remains. Since each of the realizers A_i is feasible, the feasibility of their join follows automatically from the pairwise compatibility of the A_i and the system will prove the compatibility of the realizers A_i automatically if they are indeed compatible.

Compatible realizers Incompatibilities can arise when names for variables, local actions, links, locations, or message tags that may be chosen arbitrarily and independently, happen to clash. Managing all of these names is tedious and error prone, so we have added automatic support for managing them. We are able to ensure that the names inherent in any term are always visible as explicit parameters. The logic provides a *permutation rule* mentioned in Section 2.1 that says that if proposition $\phi(x, y, \ldots, z)$ is true, where $x, y, \ldots, z$ are the names mentioned in ϕ, then proposition $\phi(x', y', \ldots, z')$ is true, where $x', y', \ldots, z'$ is the image of $x, y, \ldots, z$ under a permutation of all names. Using the permutation rule, our automated proof assistant will always permute any names that occur in realizers brought in automatically as described above.

Acknowledgements: We would like to thank Melissa Totman for helping prepare the manuscript and Stuart Allen for prior discussions about the use of atoms as nonces.[2]

References

[1] Y. G. A. Blass and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:1–3, 1999.

[2] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[3] M. Abadi, R. Corin, and C. Fournet. Computational secrecy by typing for the pi calculus. In *Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*. Springer-Verlag Heidelberg, 2006.

[4] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[5] S. Abramsky. Proofs as processes. *Journal of Theoretical Computer Science*, 135(1):5–9, 1994.

[6] S. Allen. An abstract semantics for atoms in nuprl. Technical report, 2006.

[7] S. F. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. To appear in 2006, 2006.

[8] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley Interscience, New York, 2nd edition, 2004.

[9] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[10] M. Bickford. Unguessable atoms: A logical foundation for security. Technical report, Cornell University, Ithaca, NY, 2007.

[11] M. Bickford and R. L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[12] M. Bickford and R. L. Constable. A causal logic of events in formalized computational type theory. In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktober-*

[2]We would like to thank the National Science Foundation for their support on CNS #0614790.

dorf 2005, to Appear 2007. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[13] M. Bickford and D. Guaspari. A programming logic for distributed systems. Technical report, ATC-NY, 2005.

[14] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[15] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[16] N. S. Dan Rosenzweig, Davor Runje. Privacy, abstract encryption and protocols: an asm model - part i. In *Abstract State Machines - Advances in Theory and Applications: 10th International Workshop, ASM*, volume 2589. Springer-Verlag, 2003.

[17] C. P. Gomes, D. R. Smith, and S. J. Westfold. Synthesis of schedulers for planned shutdowns of power plants. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference*, pages 12–20. IEEE Computer Society Press, 1996.

[18] C. Green, D. Pavlovic, and D. R. Smith. Software productivity through automation and design knowledge. In *Software Design and Productivity Workshop*, 2001.

[19] G. Lowe. An attack on the needham-schroeder public key encryption protocol. 56(3):131–136, 1995.

[20] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[21] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, Sept. 1989.

[22] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[23] J. Millen and H. Rue. Protocol-independent secrecy. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[24] R. Milner. Action structures and the π-calculus. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[25] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–998, 1978.

[26] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[27] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[28] D. Pavlovic and D. R. Smith. Software development by refinement. In B. K. Aichernig and T. S. E. Maibaum, editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2003.

[29] A. Roy, A. Datta, A. Derek, J. C. Mitchell, and J.-P. Seifert. Secrecy analysis in protocol composition logic. 2007. Draft Report.

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Building a Software Model Checker

Javier ESPARZA
Technische Universität München
esparza@in.tum.de

Abstract. In this paper we introduce jMoped, a tool for the analysis of Java programs based on model-checking techniques. We then proceed to introduce and explain the theory underlying the tool, and how it shaped some design choices.

1. Introduction

Model checking is a technique for the automatic verification of computer systems based on exhaustive exploration of the space of reachable states [15]. It has been very successfully applied to large hardware systems. The extension and application of model checking to software is one of the main challenges of today's research on formal verification.

In this paper we discuss some of the issues involved in building a software model checker. We follow a perhaps unusual presentation order. Instead of going from basic theory to algorithms and then to a tool, we proceed the other way round. In Section 2 we introduce jMoped, a tool for the analysis of Java programs[1]. jMoped is based on the Moped model checker developed by Stefan Schwoon in his PhD thesis [32] (see also [20]), which was later given a Java front-end by Dejvuth Suwimonteerabuth and coauthors [33,35,34]. In the rest of the paper we discuss the theory behind jMoped, and how it influenced the design decisions. In Section 3 we study the computational complexity of the fundamental problem in software verification: does some execution reach a given program point? We work under the (strong but very useful) assumption that variables only have a finite domain; conceptually, one can even assume that all variables are booleans. While one could think that under this assumption the problem is trivially decidable, we show that this is not the case. Even if all variables are boolean, a program can still have an infinite number of reachable states, due to features like (unbounded) recursion and thread generation. To address this we give programs a formal semantics in terms of *rewrite systems*, and apply several theoretical results. The main result of our analysis is that unbounded recursion is easier to handle than thread generation. Finally, in Section 4 we study the core algorithms behind jMoped in more detail. These algorithms efficiently solve the reachability of program points for (possibly recursive) sequential programs with procedures.

The paper is a survey that draws heavily from the following sources: [35,34] for Section 2, [3] for Section 3, and [19,20,32] for Section 4.

[1] jMoped is freely available. For download and documentation type "jMoped download" in a search engine.

How to read this paper. If you only want to gain an impression of how jMoped works, read Section 2. If you are interested in the problem of modelling and analysing programs, read Section 3; the section is written in survey style, and no proofs are given. Section 3 contains Theorem 3.1, the theoretical basis of jMoped. The algorithms that allow to prove the theorem are presented in Section 4 and illustrated by examples. Finally, if you want to understand these algorithms "to the bone", read the two appendices, which contain detailed correctness proofs.

2. jMoped in a nutshell

Given a Java method (say an implementation of a sorting algorithm for arrays of integers), and a finite input range (say, arrays of length at most 10 whose elements are either 0 or 1), jMoped computes all reachable states of the program for all possible input arguments within the range (2^{10} in our example) and generates coverage information for these executions. Moreover, jMoped checks for some standard errors (null pointer exceptions and array bound violations) and for errors defined by the user by means of assertions inserted in the code. When an error is found, jMoped finds out the arguments that lead to the error. A JUnit [25] test case can also be automatically generated for further testing.

From the architectural point of view, jMoped consists of a graphical user interface, a translator that converts the Java program into a so-called *symbolic pushdown system* (SPDS), and Moped [32], a model checker for SPDSs at the back-end. The graphical interface is a plug-in for Eclipse [17], the well-known environment for the development of Java programs. That is, the user experiences jMoped as an additional feature that complements the editing and simulation facilities of the Eclipse environment. The translator supports all fundamental basic features of Java, e.g. assignments, method calls, and recursion, inheritance, abstraction, and polymorphism. On the other hand, the current version still fails to translate floats and multithreaded programs.

2.1. jMoped as a Testing Tool

Traditionally, testing and model checking are seen as distinct methodologies; testing can detect bugs but not prove their absence, and model checking seeks to establish the absence of bugs, possibly at the cost of taking very long to complete (or not finishing at all). We think that model checking can and should support the testing task. jMoped is conceived as a tool in which a model checker supports the task of testing a program.

The size of variables is bounded by user-defined (artificial) ranges. Thus, the model-checking procedure can be thought of as an extended symbolic testing procedure, which is still incomplete (because only runs within the given bounds are considered); however, once the bounds are established, the model checker will perform exact checks on *all* executions within these bounds. This is very suitable for finding boundary cases, i.e. inputs with special properties that are easily forgotten during testing, but are prone to cause bugs. E.g., two boundary cases for a sorting procedure would be an array where all elements are the same, or an array that is already sorted. Even relatively small bounds on the inputs are likely to contain many interesting boundary cases, and the model checker will test all of them (and find the faulty ones). Thus, the approach can greatly enhance the confidence in the correctness of a program, without strictly guaranteeing it.

The results of a model-checking procedure can support testing in other ways, too. The quality of a set of test cases is usually measured by so-called coverage metrics, e.g., counting how many lines of code were exercised by the test cases. Such metrics can also be obtained by running a model checker on a set of inputs and checking which lines were found to be reachable. In jMoped, the user can observe the progress of these metrics while the checker is running. Moreover, the user may stop the checker at any time (e.g., if the attained level of coverage is deemed satisfactory), or ask it to specifically search for inputs that can reach a certain target in the program. Moreover, if the checker finds that bugs are caused by certain inputs, those inputs can be saved in a library of JUnit test cases, where they may be useful for future test runs.

2.2. *Working with jMoped*

As mentioned above, jMoped consists of three parts: a graphical user interface, a translator from Java bytecode into SPDS, and an SPDS model checker. The checker is capable of handling programs with thousands of lines, provided that the data complexity is low, as is the case, for instance, with device drivers [32]. However, the graphical interface was developed for unit testing, with smaller, more data-intensive programs in mind of at most a few hundred lines. The interface is also described in more detail in [2].

The graphical interface takes the form of an Eclipse plug-in. Figure 1 shows a screenshot of the interface. In the following we guide the reader through a session with jMoped using this figure.

The user starts by editing or loading a Java program in the editing window, (on the right of the figure). In this case, the program is an implementation for Quicksort taken from [29]. The program has been annotated with an *assertion* (line 45), i.e., an statement that is expected to hold in all executions when the control reaches the line. The assertion states that a call to the method `isSorted` is expected to return true.

The user selects a method from which the analysis should start (the method `sort` starting at line 43 was chosen) and chooses the bounds for the simulation of the program. There are two bounds, whose values can be set on the left-hand-side of the figure. The first one is the number of bits of every number that appears in a program. This includes constants, integer variables, and the lengths of arrays or strings. In Java this number is 32. In jMoped the user sets the length to a smaller value (typically between 3 and 7). The second bound is the heap size, which directly affects the number of objects that can be instantiated. jMoped simulates the Java heap when manipulating objects. Indirectly, a bound on the length of integers implies a bound on the heap size. The reason is that memory addresses must be integers. However, this indirect bound is almost always too large, and so the user must choose a smaller one. A typical heap bound is 32 bytes. The analysis in Figure 1 is performed with 3 bits and heap size 7.

It is also possible to specify how many bits to use for individual variables, parameters, or fields. The annotation at line 42 of Figure 1 means that the length of array `a` has two bits, and each of its elements has one bit.

jMoped can now exhaustively explore the program for all inputs within the bounds provided by the user. The user launches this process using a pop-up menu. The state space exploration is done in two steps. First, the program (which reads inputs from its user) is transformed into another program that does not read any input; instead, it nondeterministically generates one. In the second step the checker exhaustively explores all behaviours of the transformed program.

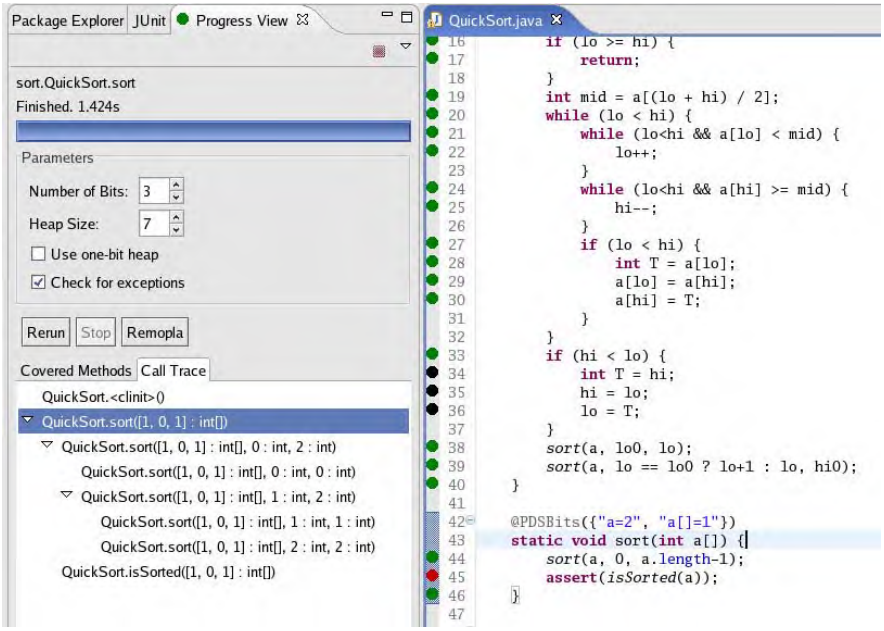


Figure 1. A view of the plug-in.

During the analysis, jMoped graphically displays its progress. First, black markers are placed in front of all statements. While the checker is running, the parts of the state space found to be reachable are mapped back to the Java program, and the appearance of the corresponding markers is changed. When a black marker turns green, it means that the corresponding Java statement is reachable from *some* argument values. A red marker means that an assertion statement has been violated by some argument values. Other markers indicate null pointer exceptions, array bound violations, and heap overflows (see below). The green markers indicate the degree of *coverage* reached by exploring the executions within the bounds.

After the analysis, users can either create a *call trace* or a JUnit test case that reaches a given statement or violates some assertion. An example of the call trace can be seen in lower left part of Figure 1, where the assertion violation occurred when the method `sort` was called with the array `[1,0,1]`.

jMoped has another mode of operation, in which it explores the set of reachable states *backwards*. Given a postcondition, jMoped computes the set of all states (within the given bounds) from which the states of the postcondition can be reached. In a typical scenario, a user will want to achieve 100% coverage, i.e. the checker should test a set of inputs such that every statement is exercised at least once. The idea for achieving this is to combine the two modes of operation. First, one uses the standard mode to cover as many instructions as possible. Say all but three instructions were covered. Then, in a second phase, one applies three backward searches starting from these three particular instructions. Since these are specific searches with the "difficult" instructions as goal, the hope is that they have better success chances than the "blind" forward search.

2.3. Motivation for the next sections

jMoped and method invocation. The input language of model checkers usually allows for the definition of procedures. However, most model checkers translate a program with procedures into a program without procedures by means of inlining. Inlining is the process of replacing a procedure call by the body of the procedure. In the absence of recursive procedures, every program is equivalent to another one without procedures, obtained by exhaustive inlining. However, the procedure-free program may be exponentially longer than the original one; an example is a program with procedures $P_1, \ldots, P_n$, in which procedure P_i calls procedure P_{i+1} twice for $1 \leq i \leq n-1$. Moreover, in the presence of recursive procedures the inlining procedure does not terminate.

jMoped does not inline methods, and it can deal with recursive methods. For this, jMoped has to solve a difficult problem: an execution of a recursive program may visit an infinite number of different program states, even if the range of all variables is finite. The reason is that the stack of activation records (in Java parlance, the stack of frames of the JVM) can become arbitrarily large during the execution, and, since the stack is part of the program state, the execution may visit infinitely many different states. It can be argued that this can only occur if the execution is non-terminating. This is true, but erroneous programs (sometimes even correct ones) may exhibit such behaviours, and jMoped does not and should not work only under the assumption that the program always terminates.

How does jMoped manage to exhaustively examine infinitely many program states? By means of a data structure allowing to represent certain infinite sets of states. The theory behind this idea is discussed in Section 3, and the algorithm implementation in Section 4.

jMoped and multithreaded programs. Currently jMoped cannot analyze multithreaded programs. There are several reasons for this. A fundamental reason is that multithreading introduces a second "source of infinity" in the state space, on top of recursion. If no bound is put on the number of threads, then, since the program state includes the local states of the threads, there can be infinitely many reachable states. While analysis problems concerning the infinity due to recursion are decidable, even very simple problems become undecidable if both recursion and multithreading are present. This problem has been very throughly discussed in the literature, and we discuss it in detail in Section 3.

jMoped and exhaustive testing. As mentioned above, jMoped simulates the execution of the program for all possible inputs within the range given by the user. It could be argued that such a brute-force approach cannot be practicable. It must be mentioned that jMoped does not proceed by simulating the executions *one by one*. It uses *binary decision diagrams* (BDDs), a compact data structure that, loosely speaking, allows to simultaneously conduct all executions "in one go". This considerably extends the possibilities of a purely sequential simulation. For example, our tool can test a Quicksort implementation for all arrays of length 60 whose elements are either 0 or 1. This means checking 2^{60} test cases, which obviously cannot be achieved in reasonable time by sequentially simulating the program on each test case. The use of these data structures will be discussed in Section 4.5.

3. Rewrite Models of Boolean Programs

The basic problem solved by jMoped is the reachability of a given program point in a given Java program by means of an execution respecting the bounds selected by the user on the ranges of the variables. Conceptually, a program in which variables have been restricted to have a finite range can be seen as a *boolean program*, an imperative, possibly nondeterministic program acting on variables of boolean type. An instruction of the real program is simulated by an instruction acting on a number of boolean variables, one for each bit needed to represent the finite range. For instance, if we restrict the range of an integer variable v to the interval $[0..3]$ we can simulate an assignment to v by a simultaneous assignment to two boolean variables; the same can be achieved, with more coding, for other types of variables. The executions of the boolean program correspond to the executions of the program in which the values of the variables stay within the specified range. We say that the boolean program is an *underapproximation* of the real program: every execution of the boolean program is an execution of the real program, but not necessarily the other way round.

In order to analyse and verify boolean programs we need to find semantic mappings linking them to formal models with a strong theory and powerful analysis algorithms. This has been the subject of intensive research since the late 90s.

We show in this section that semantic models for boolean programs can be elegantly formulated as rewrite systems. In this approach, program states are formalised as terms, and program instructions as rewrite rules. A step of the program is matched by a rewrite step in its corresponding rewrite system. The nature of the program determines the class of terms we use. In particular, we use string-rewriting and multiset-rewriting as special cases of term rewriting.

We refrain from giving a formal, general definition of rewrite system. Informally, in this paper a rewrite system is a finite set of rules of the form $t_1 \rightarrow t_2$, where t_1 and t_2 are terms over some signature. The rule $t_1 \rightarrow t_2$ indicates that a term t containing an occurrence of t_1 as a subterm can be rewritten into the term t' obtained by replacing the occurrence of t_1 by t_2. However, the term rewriting system may also have a *rewrite policy*, specifying which occurrences can be replaced.

Once we have a rewrite model of a program, we wish to analyze it. From the model-checking or program-analysis point of view questions like termination and confluence play a minor rôle. One is far more interested in the reachability problem, and actually on a generalisation of it: Given a rewriting system and two (possibly infinite!) sets of terms T and T', can some element of T be rewritten into an element of T'? The software model checking community has attacked this question by studying *symbolic reachability* techniques. In this approach, one tries to find data structures providing finite representations of a sufficiently interesting class of infinite sets of terms, and satisfying at least one of the two following properties: (1) if a set T is representable, then the set of terms reachable from T by means of an arbitrary number of rewriting steps is also representable; moreover, its representation can be effectively computed from the representation of T, and (2) same property with the set of terms that can be rewritten into terms of T.

We survey a number of results on symbolic reachability algorithms for different classes of programs. We start with sequential programs, move to concurrent programs without recursion and, finally, consider the difficult case of concurrent programs with recursive procedures. For each class we give a small example of a program and its semantics, and then present analysis results.

```
procedure p();
p0:   if (?) then
p1:     call main();                    p0 → p1
p2:     if ? then call p() fi           p0 → p3
      else                              p1 → m0 p2
p3:       call p()                      p2 → p0 p4
      fi;                               p2 → p4
p4:   return                            p3 → p0 p4
                                        p4 → ε
procedure main();                       m0 → ε
m0:   if ? then return fi;              m0 → m1
m1:   call p;                           m1 → p0 m2
m2:   return                            m2 → ε
```

Figure 2. A sequential program and its semantics.

Notation. In the rest of the paper, given a set R of rewrite rules we denote by $post_R(R)$ the set of terms reachable from T by one application of one of the rules of R (a *rewriting step*), and by $post^*{}_R(T)$ the set of terms reachable from T by means of an arbitrary number of rewriting steps. Similarly, $pre_R(T)$ denotes the set of terms that can be rewritten into terms of T in one rewriting step, and $pre^*{}_R(T)$ the set of terms that can be rewritten into terms of T in an arbitrary number of rewriting steps. Since the set of rules will always be clear from the context, we will omit the subscript.

3.1. Sequential programs

Consider the program of Figure 2. It consists of two procedures, *main*() and $p()$, and has no variables. The intended semantics of **if** ? **then** c_1 **else** c_2 **fi** is a nondeterministic choice between c_1 and c_2. The program state is not determined by the current value of the program counter only; we also need information about the procedure calls that have not terminated yet. This suggests to represent a state of the program as a *string* $p_0 p_1 \dots p_n$ where p_0 is the current value of the program counter and $p_1 \dots p_n$ is the stack of return addresses of the procedure calls whose execution has not terminated yet. For instance, the initial state of the program of Figure 2 is m_0, but the state reached after the execution of m_1 : **call**$p()$ is not p_0, it is the string $p_0 m_2$.

We can capture the behaviour of the program by the set of *string-rewriting* rules on the right of Figure 2. A procedure call is modelled by a rule of the form $X \to YZ$, where X is the current program point, Y the initial program point of the callee, and Z the return address of the caller. A return instruction is modelled by a rule $X \to \varepsilon$, where ε denotes the empty string. However, with the ordinary rewriting policy of string-rewriting systems

$$\frac{X \to w}{u X v \xrightarrow{r} u w v}$$

where $\xrightarrow{r}$ denotes a rewrite step, we have $m_0 p_2 m_2 \xrightarrow{r} m_0 p_0 p_4 m_2$ (rule $p_2 \to p_0 p_4$), which is not allowed by the intuitive semantics. We need to use the *prefix-rewriting policy*

```
bool function foo(l);
f0: if l then
f1:     return false
    else
f2:     return true
    fi

procedure main();
m0: while b do
m1:     b := foo(b)
    od
m2: return
```

$$
\begin{aligned}
b\langle t, f_0\rangle &\to b\langle t, f_1\rangle \\
b\langle f, f_0\rangle &\to b\langle f, f_2\rangle \\
b\langle l, f_1\rangle &\to f \\
b\langle l, f_2\rangle &\to t \\
t\, m_0 &\to t\, m_1 \\
f\, m_0 &\to f\, m_2 \\
b\, m_1 &\to b\langle b, f_0\rangle\, m_0 \\
b\, m_2 &\to b
\end{aligned}
$$

Figure 3. A sequential program with global and local variables and its semantics.

$$\frac{X \to w}{X\, v \xrightarrow{r} w\, v}$$

instead. We also need to interpret ε as the empty string. With these changes we have for instance the rewriting chain

$$m_0 \xrightarrow{r} m_1 \xrightarrow{r} p_0\, m_2 \xrightarrow{r} p_1\, m_2 \xrightarrow{r} m_0\, p_2\, m_2 \xrightarrow{r} p_2\, m_2 \xrightarrow{r} p_4\, m_2 \xrightarrow{r} m_2 \xrightarrow{r} \varepsilon$$

Notice that the string-rewriting system of Figure 2 is *monadic*, i.e., the left-hand-side of the rewrite rules consists of one single symbol.

3.1.1. Adding variables

Consider the program of Figure 3, where b is a global variable and l is a local variable of the function $foo()$. In the presence of variables, a state of a sequential program can be modelled as a string over the alphabet containing

- a symbol for every valuation of the global variables; and
- a symbol $\langle v, p\rangle$ for every program point p and for every valuation v of the local variables of the procedure p belongs to.

If the procedure for the control point p has no local variables, then we adopt the convention that there is a unique valuation $\perp$ of the variables, and most of the time shorten $\langle \perp, p\rangle$ to just p. States are modelled by strings of the form $g\,\langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle$, where g encodes the current values of the global variables, and each pair $\langle v_i, p_i\rangle$ corresponds to a procedure call that has not terminated yet (recall that we can have $v_i = \perp$ and then we write p_i instead of $\langle \perp, p_i\rangle$). can b. The symbol v_i encodes the values of the local variables of the caller right before the call takes place, while p_i encodes the return address at which execution must be resumed once the callee terminates. It is straightforward to assign rewrite rules to the program instructions. For instance, the call to $foo(b)$ in $main()$ is modelled by the rules

$$t\, m_1 \to t\,\langle t, f_0\rangle\, m_0 \qquad \text{and} \qquad f\, m_1 \to f\,\langle f, f_0\rangle\, m_0$$

indicating that control is transferred to f_0, that the local variable l gets assigned the current value of the global variable b, and that the return address is m_0. The complete set of rules is shown on the right of Figure 3. The symbols b and l stand in the rules for either **true** or **false**.

3.1.2. Pushdown systems

Notice that the rewrite system of a program with global variables is no longer monadic. However, the left-hand-sides of the rules are strings of length 2. The string-rewriting systems satisfying this condition are called *pushdown systems*, due to their similarity with pushdown automata. A string $g\langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle$ modelling a program state can be interpreted as the current configuration of a pushdown automaton. The valuation g of the global variables is interpreted as the current *control location* of the automaton, and the rest of the string as the current *stack content*. A rewrite rule like $t\,m_1 \rightarrow t\,\langle t, f_0\rangle\,m_0$ corresponds to a *transition rule* of the automaton: if the current control location is t and the topmost stack symbol is m_l, then the automaton can stay in state t, and replace m_1 by the word $\langle t, f_0\rangle m_0$. This is an example of a push-rule, which increases the length of the stack by 1. A rule like $b\langle l, f_1\rangle \rightarrow f$ expresses that if the automaton is in control state b and $\langle l, f_1\rangle$ is the current topmost stack symbol,then the automaton can move to the control state f and pop $\langle l, f_1\rangle$ from the stack.

So, formally, we define a pushdown system as a triplet $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*. Clearly, every pushdown system corresponds to a prefix-rewriting, string-rewriting system. Moreover, every rewrite system coming from a sequential boolean program is a pushdown system.

3.1.3. Analysis

String-rewriting systems with prefix-rewriting have an interesting story. They seem to have been studied for the first time by Büchi [9], who called them *canonical systems* (see also Chapter 5 of his unfinished book [10]). Büchi proved the fundamental result that given a regular set S of strings, the sets $pre^*(S)$ and $post^*(S)$ are also regular. The result was rediscovered by Caucal [12]. Book and Otto (who were also unaware of Büchi's work) proved that $pre^*(S)$ is regular for *monadic* string-rewriting systems with *ordinary* rewriting and presented a very simple algorithm that transform a finite automaton accepting S into another one accepting $pre^*(S)$. This algorithm was adapted to pushdown systems in [4,22], and its performance was improved in [19].

Theorem 3.1 *[4,22,19] Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ and a finite-state automaton A over the alphabet $P \cup \Gamma$, the sets $post^*(L(A))$ and $pre^*(L(A))$ are regular and effectively constructible in polynomial time in the sizes of $\mathcal{P}$ and A.*

More precisely, let n_P and n_Δ be the number of control states and transition rules of $\mathcal{P}$, and let n_Q and n_δ be the number of states and transitions of A, respectively. Let $n = \max\{n_Q, n_P\}$. An automaton recognising $post^*(L(A))$ can be constructed in $O(n_P|n_\Delta(n+n_\Delta)+n_P n_\delta)$ time and space, and the automaton recognising $pre^*(L(A))$ can be constructed in $O(n^2 n_\Delta)$ time and $O(nn_\Delta + n_\delta)$ space.

The theory of pushdown systems and related models (canonical systems, monadic string-rewriting systems, recursive state machines, context-free processes, Basic Process

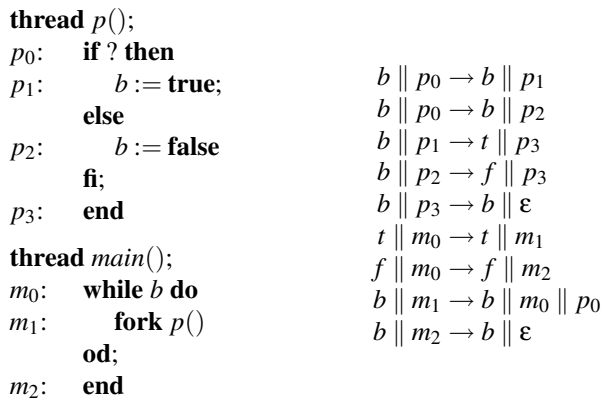


Figure 4. A program with dynamic thread generation and its semantics.

Algebra, etc.) is very rich, and even a succinct summary would exceed the scope of this paper. A good summary of the results up to the year 2000 can be found in [11].

The algorithms behind Theorem 3.1 are the core of jMoped. They are also at the basis of the MOPS tool [14]. The algorithms are described in detail in Section 4.

3.2. Multithreaded programs without procedures

Programming languages deal with concurrency in many different ways. Java uses *threads*, and, since jMoped is targeted at Java, we study threads here. (In scientific computing *cobegin-coend* sections are a popular primitive.) Languages also differ in their synchronisation and communication mechanisms: shared variables, rendezvous, asynchronous message passing. This point is less relevant for this paper, and we only consider the shared variables paradigm. In this section we study programs without procedures. The combination of concurrency and procedures is harder to analyze, and we delay it to the next section.

3.2.1. Threads

The program on the left of Figure 4 spawns a new thread $p()$ each time the while loop of *main*() is executed. This thread runs concurrently with *main*() and with the other instances of $p()$ spawned earlier. Threads communicate with each other through shared variables, in this case the global variable b. Since $p()$ nondeterministically decides whether b should be set to **true** or **false**, *main*() can create an unbounded number of instances of $p()$.

The state of the program can be modelled as a *multiset* containing the following elements:

- the current value of the global variable b,
- the current value of the program counter for the *main*() thread, and
- the current value of the program counter for each thread $p()$.

For instance, the multiset $\{0, m_1, p_1, p_2, p_2\}$ is a possible (and in fact reachable) state of the program with four threads. In order to model the program by means of rewrite

rules we introduce a parallel composition operator $\|$ and model the state as $(0 \parallel m_1 \parallel p_1 \parallel p_2 \parallel p_2)$. Intuitively, we consider a global variable as a process running in parallel with the program and communicating with it. We rewrite modulo the equational theory of $\|$, which states that $\|$ is associative, commutative, and has the empty multiset (denoted again by ε) as neutral element:

$$u \parallel (v \parallel w) = (u \parallel v) \parallel w \qquad u \parallel v = v \parallel u \qquad u \parallel \varepsilon = u\,.$$

Observe that, since we rewrite modulo the equational theory, it does not matter which rewriting policy we use (ordinary or prefix-rewriting). The complete set of rewrite rules for the program of Figure 4 is shown on the right of the figure. As in the non-concurrent case, if the program has no global variables then the rewrite system is monadic. Observe that without global variables *no communication between threads is possible*.

Notice that instructions like $p : \mathbf{wait}(b); p' : \ldots$ forcing a thread to wait until the global variable b becomes true can be modelled by the rule $t \parallel p \to t \parallel p'$.

3.2.2. Analysis.

While the reachability problem for pushdown systems can be solved in polynomial time (Theorem 3.1), it becomes harder for multiset rewriting.

Theorem 3.2 *[24,18] The reachability problem for monadic multiset-rewriting systems is NP-complete.*

NP-hardness can be proved by a straightforward reduction to SAT, while membership in NP requires a little argument. We can also prove a result similar to Theorem 3.1. In order to formulate the result, observe first that a multiset M over an alphabet $A = \{a_1, \ldots, a_n\}$ can be represented by the vector $\langle x_1, \ldots, x_n \rangle \in \mathbf{N}^n$, where x_i, $i \in \{1, \ldots, n\}$, is the number of occurrences of a_i in M. This encoding allows to represent sets of multisets by means of arithmetical constraints on integer vectors. The sets of vectors definable by formulas of Presburger arithmetic are called *semi-linear sets*. This name is due to the fact that every semi-linear set is a finite union of *linear sets*, defined as follows. A set $V \subseteq \mathbf{N}^n$ is linear if there is a *root vector* $v_0 \in \mathbf{N}^n$ and a finite number of *periods* $v_1, \ldots, v_k \in \mathbf{N}^n$ such that

$$V = \{v_0 + n_1 v_1 + \ldots, n_k v_k \mid n_1, \ldots, n_k \in \mathbf{N}\}\,.$$

Semi-linear sets share many properties with regular sets. They are closed under boolean operations. Moreover, if we associate to each word w of a regular language its *Parikh image* (the multiset containing as many copies of each symbol a as there are occurrences of a in w) we get a semi-linear set of multisets[2]. Conversely, every semi-linear set is the Parikh image of some regular language.

Intuitively, the following theorem states that semi-linear sets are to monadic multiset-rewriting what regular sets are to prefix-rewriting (see Theorem 3.1).

Theorem 3.3 *[18] Given a monadic multiset-rewriting system and a semi-linear set of states S, the sets $post^*(S)$ and $pre^*(S)$ are semi-linear and effectively constructible.*

[2] Parikh's theorem states the same result for context-free languages.

Unfortunately, Theorem 3.3 does not hold for non-monadic multiset-rewriting systems. It is easy to see that these systems are equivalent to (place/transition) Petri nets. In a nutshell, a rewrite rule

$$(X_1 \parallel \ldots \parallel X_n) \rightarrow (Y_1 \parallel \ldots \parallel Y_m)$$

corresponds to a Petri net transition that takes a token from the places $X_1, \ldots, X_n$ and puts a token on the places $Y_1, \ldots, Y_m$. It is well-known that for Petri nets $post^*(S)$ can be a non semi-linear set of states even when S is a singleton [23].

The reachability problem for multiset-rewriting systems is equivalent to the reachability problem for Petri nets, and so, using well-known results of net theory we obtain:

Theorem 3.4 *[28,26,27] The reachability problem for multiset-rewriting systems is decidable and EXPSPACE-hard.*

The known algorithms for the reachability problem of Petri nets are too complicated for practical use (not to speak of their complexity, which exceeds any primitive-recursive function). However, many program analysis problems can be stated as *control point reachability* problems in which we wish to know if a program point can be reached by a thread, independently of which or how many other threads run in parallel with it. In multiset-rewriting terms, the question is if the rewrite system associated to the program can reach a state of the form $X \parallel t$ for some multiset t. This target set of states is *upward-closed*: if some term t belongs to the set, then $t \parallel t'$ also belongs to the set for every multiset t'. Moreover, multiset-rewriting systems have the following important property: if $t \xrightarrow{r} t'$, then $t \parallel t'' \xrightarrow{r} t' \parallel t''$ for every multiset t''. This makes them *well-structured* systems in the sense of [1,21], and allows to apply a generic backward reachability algorithm to the control-reachability problem. More precisely, one can show that (1) every upward-closed set admits a finite representation (its set of minimal multisets), (2) if U is upward-closed then $U \cup pre(U)$ is upward-closed, where $pre(U) = \{t \mid \exists u \in U : t \xrightarrow{r} u\}$, and (3) every sequence $U_1 \subseteq U_2 \subseteq U_3 \ldots$ of upward-closed sets reaches a fixed point after finitely many steps. The generic backwards reachability algorithm iteratively computes (the finite representations of) $U, U \cup pre(U), U \cup pre(U) \cup pre^2(U) \ldots$ until the fixed point is reached. So we have:

Theorem 3.5 *Given a multiset-rewriting system and an upward-closed set of states S, the set $pre^*(S)$ is upward-closed and effectively constructible.*

The approach we described above has been adopted for instance in [16] for the verification of multithreaded Java programs. It has also influenced the Magic and the Spade checkers [13,30].

3.3. Putting procedures and threads together

The analysis of programs containing both procedures and concurrency is notoriously difficult. It is easy to show that a two-counter machine can be simulated by a boolean program consisting of two recursive procedures running in parallel and accessing one single global boolean variable. Intuitively, the two recursion stacks associated to the two procedures are used to simulate the two counters; the depth of the stack corresponds to

```
process p();
p0:   if (?) then
p1:     call p()
      else
p2:     skip
      fi;
p3:   return

process main();
m0:   if (?) then
m1:     fork p()
      else
m2:     call main()
      fi;
m3:   return
```

$$
\begin{aligned}
\#p_0 &\rightarrow \#p_1\\
\#p_0 &\rightarrow \#p_2\\
\#p_1 &\rightarrow \#p_0\,p_3\\
\#p_2 &\rightarrow \#p_3\\
\#p_3 &\rightarrow \#\varepsilon\\
\#m_0 &\rightarrow \#m_1\\
\#m_0 &\rightarrow \#m_2\\
\#m_1 &\rightarrow \#p_0\#m_3\\
\#m_2 &\rightarrow \#m_0\,m_3\\
\#m_3 &\rightarrow \#\varepsilon\\
\#\# &\rightarrow \#
\end{aligned}
$$

Figure 5. A program with dynamic thread generation and its semantics.

the current value of the counter. Increments and decrements can be simulated by calls and returns. The global variable is used as a semaphore indicating which counter has to be accessed next. Since two-counter machines are Turing powerful, all interesting analysis problems about these programs are bound to be undecidable.

In multithreaded programs with procedures the same code unit can be called following different policies: procedural call (caller waits until callee terminates), or thread call (caller runs concurrently with callee). We use the keyword **process** to denote such a unit.

Consider the program of Figure 5. A way of describing its semantics was proposed by Bouajjani, Müller-Olm and Touili in [8]. The idea is to represent a state at which n threads are active by a string $\#w_n\#w_{n-1}\ldots\#w_1$. Here, $w_1,\ldots w_n$ are the strings modelling the states of the threads, and they are ordered according to the following criterion: for every $1 \leq i < j \leq n$, the i-th thread (i.e., the thread in state w_i) must have been created no later than the j-th thread. The reason for putting younger threads *to the left* of older ones will be clear in a moment.

We can now try to capture the semantics of the program by string-rewriting rules. Notice however that we cannot use the prefix-rewriting policy. Loosely speaking, a thread in the middle of the string should also be able to make a move, and this amounts to rewriting "in the middle", and not only "on the left"'. So we must go back to the ordinary rewriting policy

$$\frac{X \rightarrow w}{u\,X\,v \xrightarrow{r} u\,w\,v}$$

Instructions not involving thread creation are modelled as in the non-concurrent case, with one difference: Since we can only rewrite on the left of a w_i substring, we "anchor" the rewrite rules, and use for instance $\#p_1 \rightarrow \#p_0\,p_3$ instead of $p_1 \rightarrow p_0\,p_3$. The thread creation at program point m_1 is modelled by the rule $\#m_1 \rightarrow \#p_0\#m_3$. Notice that we would not be able to give a rule if we wanted to place the new thread to the right of

its creator, because the stack length of the creator at the point of creating the new thread can be arbitrarily large. This class of string-rewriting systems is called *dynamic networks of pushdown systems* (DPN) in [8]. The complete set of rewrite rules for the program of Figure 5 is shown on the right of the same figure.

3.3.1. Analysis.

Notice that DPNs are neither prefix-rewriting nor monadic. However, we still have good analisability results. First of all, it can be proved that the pre^* operation preserves regularity:

Theorem 3.6 *[8] For every regular set S of states of a DPN, the set $pre^*(S)$ is regular and a finite-state automaton recognising it can be effectively constructed in polynomial time.*

The $post^*$ operation, however, does not preserve regularity. To see this, consider a program which repeatedly creates new threads and counts (using its stack) the number of threads it has created. The set of reachable states is not regular, because in each of them the number of spawned threads must be equal to the length of the stack. Nevertheless, the $post^*$ operation preserves context-freeness.

Theorem 3.7 *[8] For every context-free (pushdown automata definable) set S of states of a DPN, the set $post^*(S)$ is context-free and a pushdown automaton recognising it can be effectively constructed in polynomial time.*

Since intersection of a regular language with a context-free language is always context-free, and since the emptiness problem of context-free languages is decidable, this result allows to solve the reachability problem between a context-free initial set of configurations and a regular set of target configurations.

So far we have only considered the variable-free case. The results above can be extended to the case in which processes have local variables, but global variables make the model Turing powerful. In this case over/underapproximate analysis approaches can be adopted, which are outside the scope of this paper (see, e.g., [6,7,31,5]).

3.4. Discussion

We have studied rewriting models for sequential and concurrent boolean programs where concurrent processes communicate through shared variables.

Sequential boolean programs with procedure calls can be modelled by prefix-rewriting systems (pushdown systems). The reachability problem and the symbolic reachability problem for regular sets of states can be solved in polynomial time.

Concurrent programs with dynamic thread creation but without procedures can be modelled by multiset-rewriting systems (Petri nets). The reachability problem is decidable, but the algorithm is not useful in practice. The control reachability problem can be solved by a simple algorithm based on the theory of well-structured systems. The symbolic reachability problem can be solved for the monadic fragment and semi-linear

sets. The monadic fragment corresponds to programs without global variables, and so to absence of communication between threads.

Concurrent programs with thread creation and procedures, but without communication between threads, can be model by dynamic networks of pushdown systems [8], a class of string-rewriting systems. The reachability problem can be solved in polynomial time. The pre^* operation preserves regularity (and can be computed in polynomial time), while the $post^*$ operation preserves context-freeness.

Concurrent programs with procedures and one single global variable are already Turing powerful, and so very difficult to analyze. Several approximate analysis have been proposed based on the automata techniques presented in this paper (see e.g. [6,7,31,5]). The constrained dynamic networks of [8] replace global variables by a more restricted form of communication in which a process can wait for a condition on the threads it created, or for a result computed by a procedure it called.

Summarising: the problem of analyzing sequential programs is much simpler from a computational point of view than the problem of analyzing multithreaded programs. This is one of the reasons why jMoped does not currently support multithreading. Any extension of jMoped to multithreading will have to cope with the undecidability of the reachability problem.

4. Algorithms for reachability in pushdown systems

We have seen in the previous section that sequential boolean programs can be translated into pushdown systems. Pushdown systems are string-rewriting systems using the prefix-rewriting policy, and satisfying an additional property: the left-hand side of a rule is a string of length two. The first letter of the string encodes the valuation of the global variables, while the second letter encodes the current value of the program counter and the valuation of the active local variables (see Figure 2). Theorem 3.1 states that given a pushdown system and a finite-state automaton A, the sets $post^*(L(A))$ and $pre^*(L(A))$ are regular and effectively constructible in polynomial time in the sizes of the pushdown system and A. Moreover, the theorem gives a complexity estimate.

In this section we describe the algorithms behind Theorem 3.1. The algorithms for computing $pre^*(L(A))$ and $post^*(L(A))$ were presented in [4], but their complexity was not evaluated (it was only said that they run in polynomial time). Efficient implementations and detailed complexity analysis were first given in [19]. In Section 4.1 we introduce basic definitions, and recall some results of [4]. The 'abstract' solution of [4] is described in Section 4.2, and the efficient algorithms of [19] are discussed in Section 4.4. The correctness proofs and complexity analyses are given in appendices.

4.1. Pushdown systems and $\mathcal{P}$-automata

Recall that a pushdown system (already defined in Section 3) is a triplet $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*. In the following, if $((q,\gamma),(q',w)) \in \Delta$ then we write $\langle q,\gamma\rangle \hookrightarrow \langle q',w\rangle$, i.e., we reserve the arrow $\hookrightarrow$ for transition rules in order to avoid confusion.

We introduce some further notions. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w\rangle$ where $p \in P$ is a control location and $w \in \Gamma^*$ is a *stack content*. The set of all configurations is denoted by $\mathcal{C}$.

If $\langle q, \gamma\rangle \hookrightarrow \langle q', w\rangle$, then for every $v \in \Gamma^*$ the configuration $\langle q, \gamma v\rangle$ is an *immediate predecessor* of $\langle q', wv\rangle$, and $\langle q', wv\rangle$ an *immediate successor* of $\langle q, \gamma v\rangle$. This corresponds exactly to the idea that $\langle q, \gamma v\rangle$ can be rewritten into $\langle q', wv\rangle$.

The *reachability relation* $\Rightarrow$ is the reflexive and transitive closure of the immediate successor relation. A *run* of $\mathcal{P}$ is a maximal sequence of configurations such that for each two consecutive configurations $c_i c_{i+1}$, c_{i+1} is an immediate successor of c_i. A run corresponds to a maximal sequence of rewrites.

The predecessor function $pre\colon 2^C \to 2^C$ of $\mathcal{P}$ is defined as expected: c belongs to $pre(C)$ if some immediate successor of c belongs to C. As in the previous section, pre^* denotes the reflexive and transitive closure of pre. Clearly, $pre^*(C) = \{c \in \mathcal{C} \mid \exists c' \in C.\ c \Rightarrow c'\}$. Similarly, we define $post(C)$ as the set of immediate successors of elements in C and $post^*$ as the reflexive and transitive closure of $post$.

4.1.1. $\mathcal{P}$-Automata

We fix a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ for the rest of the section.

We wish to use finite automata in order to recognise possibly infinite sets of configurations of $\mathcal{P}$. For technical reasons it is convenient to introduce a variant of the usual automata model.

A *$\mathcal{P}$-automaton* is an automaton with Γ as alphabet, and P as set of initial states (we consider automata with possibly many initial states). Formally, a $\mathcal{P}$-automaton is an automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ where Q is the finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is the set of *transitions*, P is the set of *initial states* and $F \subseteq Q$ the set of *final states*. We define the *transition relation* $\rightarrow \subseteq Q \times \Gamma^* \times Q$ as the smallest relation satisfying:

- $q \xrightarrow{\varepsilon} q$ for every $q \in Q$,
- if $(q, \gamma, q') \in \delta$ then $q \xrightarrow{\gamma} q'$, and
- if $q \xrightarrow{w} q''$ and $q'' \xrightarrow{\gamma} q'$ then $q \xrightarrow{w\gamma} q'$.

All the automata used in this section are $\mathcal{P}$-automata, and so we drop the $\mathcal{P}$ from now on. An automaton *accepts* or *recognises* a configuration $\langle p, w\rangle$ if $p \xrightarrow{w} q$ for some $p \in P$, $q \in F$. The set of configurations recognised by an automaton $\mathcal{A}$ is denoted by $Conf(\mathcal{A})$. A set of configurations of $\mathcal{P}$ is *regular* if it is recognized by some automaton.

Notice that given a $\mathcal{P}$ automaton one can easily construct a normal automaton accepting the same language. It suffices to add a new initial state to the $\mathcal{P}$-automaton, say q_0, and a transition $q_0 \xrightarrow{p_i} p_i$ for every old initial state p_i. $\mathcal{P}$-automata just have some technical advantages as data structures.

Notation In the paper, we use the symbols p, p', p'' etc., eventually with indices, to denote initial states of an automaton (i.e., the elements of P). Non-initial states are denoted by s, s', s'' etc., and arbitrary states, initial or not, by q, q', q''.

4.2. Computing $pre^*(C)$ for a regular language C

Our input is an automaton $\mathcal{A}$ accepting C. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We compute $pre^*(C)$ as the language

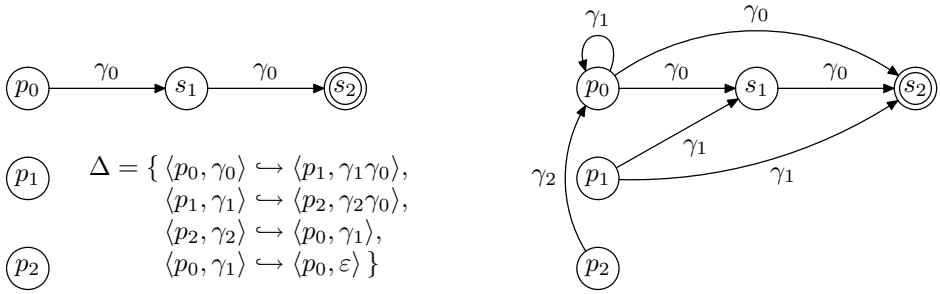


Figure 6. The automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$ (right)

accepted by an automaton $\mathcal{A}_{pre^*}$ obtained from $\mathcal{A}$ by means of a saturation procedure. The procedure adds new transitions to $\mathcal{A}$, but no new states. New transitions are added according to the following *saturation rule*:

> If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w} q$ in the current automaton,
> add a transition (p, γ, q) .

Notice that all new transitions start at initial states. Let us illustrate the procedure by an example. Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system with $P = \{p_0, p_1, p_2\}$ and Δ as shown in in the left half of Figure 6. Let $\mathcal{A}$ be the automaton that accepts the set $C = \{\langle p_0, \gamma_0 \gamma_0 \rangle\}$, also shown in the figure. The result of the algorithm is shown in the right half of Figure 6.

The saturation procedure eventually reaches a fixed point because the number of possible new transitions is finite. The correctness is proved in Appendix 1.

4.3. Computing post(C) for a regular set C*

We provide a solution for the case in which each transition rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ of Δ satisfies $|w| \leq 2$. This restriction is not essential, but leads to a simpler solution. Moreover, any pushdown system can be transformed into an equivalent one in this form. Moreover, the pushdown systems derived from sequential programs, as discussed in Section 3, satisfy this condition: we have $|w| = 2$ for transition rules modelling procedure calls, $|w| = 0$ for transition rules modelling return instructions, and $|w| = 1$ for the rest.

Our input is an automaton $\mathcal{A}$ accepting C. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We compute $post^*(C)$ as the language accepted by an automaton $\mathcal{A}_{post^*}$ with ε-moves. We denote the relation $(\xrightarrow{\varepsilon})^* \xrightarrow{\gamma} (\xrightarrow{\varepsilon})^*$ by $\stackrel{\gamma}{\Longrightarrow}$. $\mathcal{A}_{post^*}$ is obtained from $\mathcal{A}$ in two stages:

- Add to $\mathcal{A}$ a new state r for each transition rule $r \in \Delta$ of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$, and a transition (p', γ', r).
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

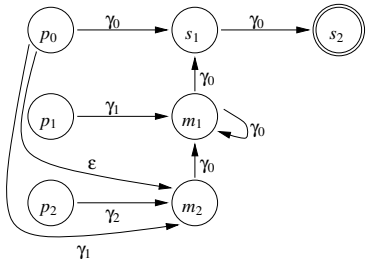


Figure 7. $\mathcal{A}_{post^*}$

If $\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition (p',ε,q).

If $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition (p',γ',q).

If $r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition (r,γ'',q).

Correctness and termination are proved in Appendix 1. Consider again the pushdown system $\mathcal{P}$ and the automaton $\mathcal{A}$ from Figure 6. Then the automaton shown in Figure 7 is the result of the algorithm above and accepts $post^*(\{\langle p_0,\gamma_0\gamma_0\rangle\})$.

4.4. Efficient Algorithms

In this section we present an efficient implementation of the abstract algorithm of Sections 4.2 and 4.3. We restrict ourselves to pushdown systems which satisfy $|w| \leq 2$ for every rule $\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle$. As already mentioned in Section 4.3, any pushdown system can be put into such a normal form with linear size increase, and the pushdown systems derived from sequential programs satisfy this condition.

4.4.1. An Efficient Algorithm for Computing $pre^*(C)$

Given an automaton $\mathcal{A}$ accepting the set of configurations C, we want to compute $pre^*(C)$ by constructing the automaton $\mathcal{A}_{pre^*}$.

Algorithm 1 computes the transitions of $\mathcal{A}_{pre^*}$ by implementing the saturation rule from section 4.2. The sets *rel* and *trans* contain the transitions that are known to belong to $\mathcal{A}_{pre^*}$; *rel* contains the transitions that have already been examined. No transition is examined more than once.

The idea of the algorithm is to avoid unnecessary operations. When we have a rule $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle$, we look out for pairs of transitions $t_1 = (p',\gamma',q')$ and $t_2 = (q',\gamma'',q'')$ (where q',q'' are arbitrary states) so that we may insert (p,γ,q'') – but we don't know in which order such transitions appear in *trans*. If every time we see a transition like t_2 we check the existence of t_1, many checks might be negative and waste time to no avail. However, once we see t_1 we know that all subsequent transitions (q',γ'',q'') must lead to (p,γ,q''). It so happens that the introduction of an extra rule $\langle p,\gamma\rangle \hookrightarrow \langle q',\gamma'\rangle$ is enough to take care of just these cases. We collect these extra rules in a set called Δ'; this notation should make it clear that the pushdown system itself is not changed. Δ' is merely needed for the computation and can be thrown away afterwards.

Algorithm 1
Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form;
a $\mathcal{P}$-Automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ without transitions into P
Output: the set of transitions of $\mathcal{A}_{pre^*}$

```
1  rel ← ∅; trans ← δ; Δ' ← ∅;
2  for all ⟨p,γ⟩ ↪ ⟨p',ε⟩ ∈ Δ do trans ← trans ∪ {(p,γ,p')};
3  while trans ≠ ∅ do
4    pop t = (q,γ,q') from trans;
5    if t ∉ rel then
6      rel ← rel ∪ {t};
7      for all ⟨p1,γ1⟩ ↪ ⟨q,γ⟩ ∈ (Δ ∪ Δ') do
8        trans ← trans ∪ {(p1,γ1,q')};
9      for all ⟨p1,γ1⟩ ↪ ⟨q,γγ2⟩ ∈ Δ do
10       Δ' ← Δ' ∪ {⟨p1,γ1⟩ ↪ ⟨q',γ2⟩};
11       for all (q',γ2,q'') ∈ rel do
12         trans ← trans ∪ {(p1,γ1,q'')};
13 return rel
```

For a better illustration, consider again the example shown in Figure 6.

The initialisation phase evaluates the ε-rules and adds (p_0, γ_1, p_0). When the latter is taken from *trans*, the rule $\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle$ is evaluated and (p_2, γ_2, p_0) is added. This, in combination with (p_0, γ_0, s_1) and the rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_0 \rangle$, leads to (p_1, γ_1, s_1), and Δ' now contains $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_0, \gamma_0 \rangle$. We now have $p_1 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_0} s_2$, so the next step adds (p_0, γ_0, s_2), and Δ' is extended by $\langle p_0, \gamma_0 \rangle \hookrightarrow \langle s_1, \gamma_0 \rangle$. Because of Δ', (p_0, γ_0, s_2) leads to (p_1, γ_1, s_2). Finally, Δ' is extended by $\langle p_0, \gamma_0 \rangle \hookrightarrow \langle s_2, \gamma_0 \rangle$, but no other transitions can be added and the algorithm terminates.

Theorem 4.1 *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system and let $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ be an automaton. There exists an automaton $\mathcal{A}_{pre^*}$ recognising $pre^*(Conf(\mathcal{A}))$. Moreover, $\mathcal{A}_{pre^*}$ can be constructed in $O(n_Q^2 n_\Delta)$ time and $O(n_Q n_\Delta + n_\delta)$ space, where $n_Q = |Q|$, $n_\delta = |\delta|$, and $n_\Delta = |\Delta|$.*

The Theorem is proved in Appendix 2. It is easy to see that a naive implementation of the abstract procedure of Section 4.2 leads to an $O(n_\mathcal{P}^2 n_\mathcal{A}^3)$ time and $O(n_\mathcal{P} n_\mathcal{A})$ space algorithm, where $n_\mathcal{P} = |P| + |\Delta|$, and $n_\mathcal{A} = |Q| + |\delta|$.

4.4.2. An Efficient Algorithm for Computing post(C)*

Given a regular set of configurations C, we want to compute $post^*(C)$, i.e. the set of successors of C. Without loss of generality, we assume that $\mathcal{A}$ has no ε-transitions.

Algorithm 2 calculates the transitions of $\mathcal{A}_{post^*}$, implementing the saturation rule from section 4.3. The approach is in some ways similar to the solution for pre^*; again we use *trans* and *rel* to store the transitions that we need to examine. Note that transitions from states outside of P go directly to *rel* since these states cannot occur in rules.

The algorithm is very straightforward. We start by including the transitions of $\mathcal{A}$; then, for every transition that is known to belong to $\mathcal{A}_{post^*}$, we find its successors. A noteworthy difference to the algorithm in Section 4.3 is the treatment of ε-moves: ε-transitions are eliminated and simulated with non-ε-transitions; we maintain the sets $eps(q)$ for every state q with the meaning that whenever there should be an ε-transition going from p to q, $eps(q)$ contains p.

Algorithm 2
Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form;
a $\mathcal{P}$-Automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ without transitions into P
Output: the automaton $\mathcal{A}_{post^*}$

```
1  trans ← δ ∩ (P × Γ × Q);
2  rel ← δ \ trans; Q' ← Q; F' ← F;
3  for all r = ⟨p,γ⟩ ↪ ⟨p',γ1γ2⟩ ∈ Δ do
4     Q' ← Q' ∪ {q_r};
5     trans ← trans ∪ {(p',γ1,q_r)};
6  for all q ∈ Q' do eps(q) ← ∅;
7  while trans ≠ ∅ do
8     pop t = (p,γ,q) from trans;
9     if t ∉ rel then
10       rel ← rel ∪ {t};
11       for all ⟨p,γ⟩ ↪ ⟨p',ε⟩ ∈ Δ do
12          if p' ∉ eps(q) then
13             eps(q) ← eps(q) ∪ {p'};
14             for all (q,γ',q') ∈ rel do
15                trans ← trans ∪ {(p',γ',q')};
16             if q ∈ F' then F' ← F' ∪ {p'};
17       for all ⟨p,γ⟩ ↪ ⟨p',γ1⟩ ∈ Δ do
18          trans ← trans ∪ {(p',γ1,q)};
19       for all r = ⟨p,γ⟩ ↪ ⟨p',γ1γ2⟩ ∈ Δ do
20          rel ← rel ∪ {(q_r,γ2,q)};
21          for all p'' ∈ eps(q_r) do
22             trans ← trans ∪ {(p'',γ2,q)};
23 return (Γ,Q',rel,P,F')
```

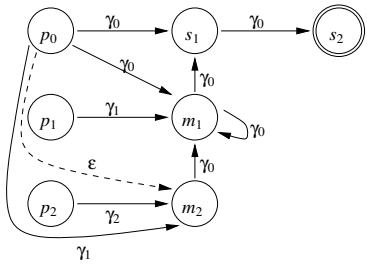


Figure 8. $\mathcal{A}_{post^*}$ as computed by Algorithm 2.

Again, consider the example in Figure 7. In that example, m_1 is the node associated with the rule $\langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_0 \rangle$, and m_2 is associated with $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_0 \rangle$. The transitions (p_1, γ_1, m_1) and (m_1, γ_0, s_1) are a consequence of (p_0, γ_0, s_1); the former leads to (p_2, γ_2, m_2) and (m_2, γ_0, m_1) and, in turn, to (p_0, γ_1, m_2). Because of $\langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle$, we now need to simulate an ε-move from p_0 to m_2. This is done by making copies of all the transitions that leave m_2; in this example, (m_2, γ_0, m_1) is copied and changed to (p_0, γ_0, m_1). The latter finally leads to (p_1, γ_1, m_1) and (m_1, γ_0, m_1). Figure 8 shows the result, similar to Figure 7 but with the ε-transition resolved.

Theorem 4.2 *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system, and $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ be an automaton. There exists an automaton $\mathcal{A}_{post^*}$ recognising $post^*(Conf(\mathcal{A}))$. Moreover,*

$\mathcal{A}_{post^*}$ can be constructed in $O(n_P n_\Delta (n_Q + n_\Delta) + n_P n_\delta)$ time and space, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, and $n_\delta = |\delta|$.

The proofs are given in Appendix 2. In [22] the same problem was considered (with different restrictions on the rules in the pushdown system).

4.5. Symbolic Pushdown Systems

In Section 3 we have shown how to translate a sequential boolean program into a pushdown system. Recall that states are modelled by strings of the form $g\,\langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle$, where g encodes the current values of the global variables, and each pair $\langle v_i, p_i\rangle$ corresponds to a procedure call that has not terminated yet. In the terminology of pushdown systems, the string corresponds to a configuration $\langle g\,,\, \langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle\rangle$. So, more precisely:

- the control location of the configuration is g, the current values of the global variables;
- the topmost stack symbol is $\langle v_1, p_1\rangle$ i.e., the current values v_1 of the local variables of the procedure being executed, and the current value p_1 of the program counter;
- the rest of the stack content corresponds to activation records for each procedure call; an activation record is a pair $\langle v_i, p_i\rangle$, where v_i stores the values of the local variables before a procedure call, and p_i stores the return address.

It follows immediately from this description that the number of transition rules of a pushdown system grows exponentially in the number of variables of the program. An assignment, being executed at program point l_1, after which execution continues at l_2, is modelled by a set of rules of the form

$$\langle g_1, \langle v_1, l_1\rangle\rangle \hookrightarrow \langle g_2, \langle v_2, l_2\rangle\rangle.$$

For each pair g_1, l_1 of valuations we get a different rule (or more if the program is nondterminstic). If the program has 20 global variables and the procedure being currently executed has 20 local variables, all of them boolean, the assignment is modelled by 2^{40} pushdown rules.

Obviously, 2^{40} rules cannot be stored or manipulated one by one. For this reason we use *symbolic pushdown systems* (SPSD). A SPDS is just a compact representation of a PDS. It has symbolic rules, which correspond to sets of "similar" rules. Instead of the 2^{40} rules above, a SPDS has one single symbolic rule $\langle p, l_1\rangle \xhookrightarrow{R} \langle p, l_2\rangle$, where R is the set of all fourtuples (g_1, l_1, g_2, l_2) for which $\langle g_1, \langle v_1, l_1\rangle\rangle \hookrightarrow \langle g_2, \langle v_2, l_2\rangle\rangle$ is a rule. The key point is that (g_1, l_1, g_2, l_2) can be seen as a bitvector, and we can now use adequate data structures to compactly represent sets of bitvectors. In jMoped we use binary decision diagrams (BDDs). Since BDDs are a very popular data structure, we refrain from explaining the technique in large detail. We only sketch how the algorithm for the computation of predecessors can be implemented for symbolic pushdown systems.

4.5.1. Symbolic Computation of Predecessors

We briefly explain how to lift the algorithm for computing the predecessors of a regular set of configurations so that it can profit from BDD technology.

Recall: given a regular set C of configurations of $\mathcal{P}$, we want to compute $pre^*(C)$. Let $\mathcal{A}$ be a $\mathcal{P}$-automaton that accepts C. We modify $\mathcal{A}$ to an automaton that accepts $pre^*(C)$ by adding new transitions according to the saturation rule:

> If $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma_1 \dots \gamma_n\rangle$ and $p' \xrightarrow{\gamma_1} q_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} q$ in the current automaton, add a transition (p,γ,q).

For the symbolic case, the saturation rule becomes:

> If $\langle p,\gamma\rangle \overset{R}{\hookrightarrow} \langle p',\gamma_1\gamma_2 \dots \gamma_n\rangle$ and $p' \xrightarrow[R_1]{\gamma_1} q_1 \xrightarrow[R_2]{\gamma_2} \cdots \xrightarrow[R_n]{\gamma_n} q$ in the current automaton, replace $p \xrightarrow[R']{\gamma} q$ by $p \xrightarrow[R'']{\gamma} q$ where
>
> $$R'' = R' \cup \{\, (g,l,g_n) \mid (g,l,g_0,l_1,\dots,l_n) \in R \wedge \exists g_1,\dots,g_{n-1} \colon \forall 1 \le i \le n \colon (g_{i-1},l_i,g_i) \in R_i \,\}.$$

Clearly, in both cases we are carrying the same computation. The key observation is that R'' can be computed without having to enumerate the bitvectors of R or R'. Given BDDs for R and R', we directly construct a BDD for R''. In favourable cases this leads to an enormous speed-up.

5. Conclusions

In this paper we have presented jMoped, discussed the basic theory behind it, and described its core algorithm. Some of the ideas embodied in the tool seem to have passed the test of time, and it can be useful to summarize them.

Working with model checkers can be very frustrating. Usually, after launching an analysis the user can only wait for an answer, which may or may not arrive; the tools do not allow the user to monitor the progress of the analysis. Moreover, if the user interrupts the analysis before it is completed, the tool does not return any information, however partial. One of the design choices behind jMoped was to give the user as much control of the tool as possible, and to keep him or her as much informed as possible. That is why jMoped was designed to have the feeling of a testing tool: during the analysis the green markers describe the progress made in covering the program, and if the analysis is interrupted, one can see how far it went.

Fom the theoretical point of view, we think that our choice of giving programs a rewrite semantics has paid off. While this choice may look very natural in retrospect, it took quite some time to develop. The program-analysis community tended to give semantics by directly defining the set of executions, which in the presence of recursion and parallelism can become very cumbersome. The rewrite style is close to the operational semantics of process algebras, but even here there is also a substantial difference. Traditional operational semantics puts much emphasis on notions of encapsulation and abstraction. While these are essential for many applications, they also make the connection to simple abstract machines (like pushdown systems or Petri nets) less clear, and hinders the development of verification algorithms.

Acknowledgments

This survey is based on papers I coauthored with Felix Berger, Ahmed Bouajjani, David Hansel, Peter Rossmanith, Stefan Schwoon, and Dejvuth Suwimonteerabuth. My warmest thanks to all of them.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

[2] F. Berger. A test and verification environment for Java programs. Master's thesis, University of Stuttgart, 2007.

[3] A. Bouajjani and J. Esparza. Rewriting models of boolean programs. In *RTA*, volume 2280 of *Lecture Notes in Computer Science*, Seattle, USA, 2006.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[5] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551–, 2003.

[7] A. Bouajjani, J. Esparza, and T. Touili. Reachability analysis of synchronized pa systems. *Electr. Notes Theor. Comput. Sci.*, 138(3):153–178, 2005.

[8] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*. Springer, 2005.

[9] J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlag.*, 6:91–111, 1964.

[10] J. R. Büchi. *The collected works of J. Richard Büchi.* Springer-Verlag, New-York, 1990.

[11] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on Infinite Structures. In *Handbook of Process Algebra*. North-Holland Elsevier, 2001.

[12] D. Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1):61–86, 1992.

[13] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, pages 334–349, 2006.

[14] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[15] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[16] G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded java programs. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2002.

[17] Eclipse. An open development platform, `http://www.eclipse.org`.

[18] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Inform.*, 31(1):13–25, 1997.

[19] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[20] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, July 2001.

[21] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[22] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electr. Notes Theor. Comput. Sci.*, 9, 1997.

[23] J. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.

[24] D. Huynh. Commutative grammars: The complexity of uniform word problems. *Information and Control*, 57(1):21–39, 1983.
[25] JUnit. Testing resources for extreme programming, http://www.junit.org/.
[26] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281. ACM, 1982.
[27] R. Lipton. The Reachability Problem Requires Exponential Space. Technical Report 62, Yale University, 1976.
[28] E. Mayr. An algorithm for the general Petri net reachability problem. In *STOC*, pages 238–246. ACM, 1981.
[29] ParForCE Project Workshop. Performance comparison between Prolog and Java, http://www.clip.dia.fi.upm.es/Projects/ParForce/Final_review/slides/intro/node4.html.
[30] G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, 2007.
[31] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
[32] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
[33] D. Suwimonteerabuth. Verifying java bytecode with the moped model checker. Master's thesis, University of Stuttgart, 2004.
[34] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jMoped: A test environment for Java programs. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 164–167, Berlin, Germany, July 2007. Springer.
[35] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Proc. TACAS*, LNCS 3440, pages 541–545, 2005. Tool paper.

Appendix 1: Correctness of the abstract algorithms

Computing $pre^*(C)$: Correctness

Let $\mathcal{A}$ be a $\mathcal{P}$-automaton, and let $\mathcal{A}_{pre^*}$ be the automaton obtained from $\mathcal{A}$ by means of the saturation rule defined in Section 4.2:

> If $\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle$ and $p' \xrightarrow{w} q$ in the current automaton,
> add a transition (p,γ,q).

Observe that all new transitions added by the procedure start at an initial state, i.e., an element of P. In the sequel we use the following notation:

- $q \xrightarrow[i]{w} q'$ denotes that the automaton obtained after adding i transitions to $\mathcal{A}$ contains a path labelled by w leading from q to q'; in particular, $q \xrightarrow[0]{w} q'$ denotes that $\mathcal{A}$ contains such a path.
- $q \xrightarrow{w} q'$ denotes that there is an index i satisfying $q \xrightarrow[i]{w} q'$. Equivalently, it denotes that $\mathcal{A}_{pre^*}$ contains a path labelled by w leading from q to q'.

We show that $\mathcal{A}_{pre^*}$ recognises the set $pre^*(Conf(\mathcal{A}))$. The result is proved in Theorem 5.1 at the end of this subsection. We need two preliminary lemmata.

Lemma 5.1 *For every configuration $\langle p,v\rangle \in Conf(\mathcal{A})$, if $\langle p',w\rangle \Rightarrow \langle p,v\rangle$ then $p' \xrightarrow{w} q$ for some final state q of $\mathcal{A}_{pre^*}$.*

Proof: Let $\langle p',w\rangle \stackrel{k}{\Longrightarrow} \langle p,v\rangle$ denote that we derive $\langle p,v\rangle$ from $\langle p',w\rangle$ in k steps. We proceed by induction on k.

Basis. $k=0$. Then $p'=p$ and $w=v$. Since $\langle p,v\rangle \in Conf(\mathcal{A})$, we have $p \xrightarrow[0]{v} q$ for some final state q, and so $p \xrightarrow{v} q$, which implies $p' \xrightarrow{w} q$.

Step. $k>0$. Then, by the definition of $\stackrel{k}{\Longrightarrow}$, there is a configuration $\langle p'',u\rangle$ such that

$$\langle p',w\rangle \stackrel{1}{\Longrightarrow} \langle p'',u\rangle \stackrel{k-1}{\Longrightarrow} \langle p,v\rangle .$$

We apply the induction hypothesis to $\langle p'',u\rangle \stackrel{k-1}{\Longrightarrow} \langle p,v\rangle$, and obtain

$$p'' \xrightarrow{u} q \text{ for some } q \in F .$$

Since $\langle p',w\rangle \stackrel{1}{\Longrightarrow} \langle p'',u\rangle$, there are γ, w_1, v_1 such that

$$w=\gamma w_1,\ u=u_1 w_1, \text{ and } \langle p',\gamma\rangle \hookrightarrow \langle p'',u_1\rangle.$$

Let q_1 be a state of $\mathcal{A}_{pre^*}$ such that

$$p'' \xrightarrow{u_1} q_1 \xrightarrow{w_1} q.$$

By the saturation rule, we have

$$p' \xrightarrow{\gamma} q_1 \xrightarrow{w_1} q ,$$

which implies

$$p' \xrightarrow{\gamma w_1} q ,$$

and since $w=\gamma w_1$, we are done. □

Lemma 5.2 *Let $p \xrightarrow{w} q$ be a path of $\mathcal{A}_{pre^*}$. The following properties hold:*

(a) *$\langle p,w\rangle \Rightarrow \langle p',w'\rangle$ for a configuration $\langle p',w'\rangle$ such that $p' \xrightarrow[0]{w'} q$; moreover,*
(b) *if q is an initial state, then $w'=\varepsilon$.*

Proof:

Let i be an index such that $p \xrightarrow[i]{w} q$. We prove (a) and (b) simultaneously by induction on i.

Basis. $i=0$. Then $p \xrightarrow{w} q$ is a path of $\mathcal{A}$.

(a) Take $p'=p$ and $w'=w$.

(b) Since $\mathcal{A}$ contains no transitions leading to an initial state, we have $w=\varepsilon$ and $p'=p$. So $\langle p,w\rangle \Rightarrow \langle p',\varepsilon\rangle$.

Step. $i \geq 1$. The proofs of (a) and (b) have a common initial part. Let $t = (p_1, \gamma, q')$ be the i-th transition added to $\mathcal{A}$. (Notice that we can safely write (p_1, γ, q') instead of (q_1, γ, q') because all new transitions start at an initial state.) Let j be the number of times that t is used in $p \xrightarrow[i]{w} q$.

The proof is by induction on j. If $j = 0$, then we have $p \xrightarrow[i-1]{w} q$, and (a), (b) follow from the induction hypothesis (induction on i). So assume that $j > 0$. Then there exist u and v such that $w = u\gamma v$ and

$$p \xrightarrow[i-1]{u} p_1 \xrightarrow[i]{\gamma} q' \xrightarrow[i]{v} q \tag{0}$$

The application of the induction hypothesis (induction on i) to $p \xrightarrow[i-1]{u} p_1$ yields (notice that p_1 is an initial state, and so both (a) and (b) can be applied):

$$\langle p, u \rangle \Rightarrow \langle p_1, \varepsilon \rangle \tag{1}$$

Since the transition (p_1, γ, q') has been added by applying the saturation rule, there exist p_2 and w_2 such that

$$\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, w_2 \rangle \tag{2.1}$$

$$p_2 \xrightarrow[i-1]{w_2} q' \tag{2.2}$$

From this point on the proofs for (a) and (b) diverge.

(a) From (0) and (2.2) we get

$$p_2 \xrightarrow[i-1]{w_2} q' \xrightarrow[i]{v} q \tag{3}$$

Since the transition t is used in (3) less often than in (0), we can apply the induction hypothesis (induction on j) to (3), and obtain

$$\langle p_2, w_2 v \rangle \Longrightarrow \langle p', w' \rangle \tag{4.1}$$

$$p' \xrightarrow[0]{w'} q \tag{4.2}$$

Putting (1), (2.1), and (4.1) together, we get

$$\langle p, u\gamma v \rangle \overset{(1)}{\Longrightarrow} \langle p_1, \gamma v \rangle \overset{(2.1)}{\Longrightarrow} \langle p_2, w_2 v \rangle \overset{(4.1)}{\Longrightarrow} \langle p', w' \rangle \tag{5}$$

We obtain (a) from (5) and (4.2).

(b) Since q is an initial state, and $\mathcal{A}_{pre^*}$ contains no transitions leading from non-initial to initial states, q' is an initial state. The application of the induction hypothesis (induction on i) to (2.2) yields:

$$\langle p_2, w_2 \rangle \Longrightarrow \langle q', \varepsilon \rangle \tag{6}$$

Since t appears less often in $q' \xrightarrow[i]{v} q$ than in (0), we can apply the induction hypothesis (induction on j) to $q' \xrightarrow[i]{v} q$. We get

$$\langle q', v\rangle \Longrightarrow \langle q, \varepsilon\rangle \tag{7}$$

Putting (1), (2.1), (6) and (7) together, we get

$$\langle p, u\gamma v\rangle \overset{(1)}{\Longrightarrow} \langle p_1, \gamma v\rangle \overset{(2.1)}{\Longrightarrow} \langle p_2, w_2 v\rangle \overset{(6)}{\Longrightarrow} \langle q', v\rangle \overset{(7)}{\Longrightarrow} \langle q, \varepsilon\rangle$$

Since q is an initial state, (b) is obtained by taking $p' = q$. □

Theorem 5.1 *Let $\mathcal{A}_{pre^*}$ be the automaton obtained from $\mathcal{A}$ by exhaustive application of the saturation rule defined in Section 4.2. $\mathcal{A}_{pre^*}$ recognises the set $pre^*(Conf(\mathcal{A}))$.*

Proof: Let $\langle p, w\rangle$ be a configuration of $pre^*(Conf(\mathcal{A}))$. Then $\langle p, w\rangle \Rightarrow \langle p', w'\rangle$ for a configuration $\langle p', w'\rangle \in Conf(\mathcal{A})$. By Lemma 5.1, $p \xrightarrow{w} q$ for some final state q of $\mathcal{A}_{pre^*}$. So $\langle p, w\rangle$ is recognised by $\mathcal{A}_{pre^*}$.

Conversely, let $\langle p, w\rangle$ be a configuration recognised by $\mathcal{A}_{pre^*}$. Then $p \xrightarrow{w} q$ in $\mathcal{A}_{pre^*}$ for some final state q. By Lemma 5.2(a), $\langle p, w\rangle \Rightarrow \langle p', w'\rangle$ for a configuration $\langle p', w'\rangle$ such that $p' \xrightarrow[0]{w'} q$. Since q is a final state, $\langle p', w'\rangle \in Conf(\mathcal{A})$, and so $\langle p, w\rangle \in pre^*(Conf(\mathcal{A}))$. □

Computing $post^*(C)$: Correctness

Let $\mathcal{A}$ be a $\mathcal{P}$-automaton, and let $\mathcal{A}_{post^*}$ be the automaton obtained from $\mathcal{A}$ by means of the saturation rules defined in Section 4.3:

> If $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton,
> add a transition (p', ε, q).
>
> If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton,
> add a transition (p', γ', q).
>
> If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton,
> add a transition (r, γ'', q).

In the sequel we use the following notation:

- $q \xrightarrow[i]{w} q'$ denotes that the automaton obtained after the i-th application of the saturation rule contains a path labelled by w leading from q to q'; in particular, $q \xrightarrow[0]{w} q'$ denotes that $\mathcal{A}$ contains such a path (unless q' is a state added by the algorithm).
- $q \xrightarrow{w} q'$ denotes that there is an index i satisfying $q \xrightarrow[i]{w} q'$. Equivalently, it denotes that $\mathcal{A}_{post^*}$ contains a path labelled by w leading from q to q'.

We show that $\mathcal{A}_{post^*}$ recognises the set $post^*(Conf(\mathcal{A}))$. The result is proved in Theorem 5.2 at the end of this subsection. We need two preliminary lemmata.

Lemma 5.3 *For every configuration $\langle p,v\rangle \in Conf(\mathcal{A})$, if $\langle p,v\rangle \Rightarrow \langle p',w\rangle$ then $p' \xrightarrow{w} q$ for some final state q of $\mathcal{A}_{post^*}$.*

Proof: Let $\langle p,v\rangle \stackrel{k}{\Longrightarrow} \langle p',w\rangle$ denote that we derive $\langle p',w\rangle$ from $\langle p,v\rangle$ in k steps. We proceed by induction on k.

Basis. $k=0$. Then $p'=p$ and $w=v$. Since $\langle p,v\rangle \in Conf(\mathcal{A})$, we have $p \xrightarrow[0]{v} q$ for some final state q, and so $p \xrightarrow{v} q$, which implies $p' \xrightarrow{w} q$.

Step. $k>0$. Then, by the definition of $\stackrel{k}{\Longrightarrow}$, there is a configuration $\langle p'',u\rangle$ such that

$$\langle p,v\rangle \stackrel{k-1}{\Longrightarrow} \langle p'',u\rangle \stackrel{1}{\Longrightarrow} \langle p',w\rangle .$$

We apply the induction hypothesis to $\langle p,v\rangle \stackrel{k-1}{\Longrightarrow} \langle p'',u\rangle$, and obtain

$$p'' \xrightarrow{u} q \text{ for some } q \in F .$$

Since $\langle p'',u\rangle \stackrel{1}{\Longrightarrow} \langle p',w\rangle$, there are γ,u_1,v_1,w_1 such that

$$u=\gamma u_1,\ w=w_1u_1, \text{ and } \langle p'',\gamma\rangle \hookrightarrow \langle p',w_1\rangle.$$

There are three possible cases, according to the length of w_1. We consider only the case $|w_1|=2$, the others being simpler. Since $|w_1|=2$, we have $w_1=\gamma'\gamma''$. Let q_1 be a state of $\mathcal{A}_{pre^*}$ such that

$$p'' \xrightarrow{\gamma} q_1 \xrightarrow{u_1} q.$$

By the initialisation and the saturation rule, we have

$$p' \xrightarrow{\gamma'} r \xrightarrow{\gamma''} q_1 \xrightarrow{u_1} q ,$$

which implies

$$p' \xrightarrow{w_1u_1} q ,$$

and since $w=w_1u_1$, we are done. □

Lemma 5.4 *Let $p \xrightarrow{w} q$ be a path of $\mathcal{A}_{post^*}$. Then the following property holds: $\langle p',w'\rangle \Rightarrow \langle p,w\rangle$ for a configuration $\langle p',w'\rangle$ such that $p' \xrightarrow[0]{w'} q$.*

Proof: Let i be an index such that $p \xrightarrow[i]{w} q$. We prove the lemma by induction on i.

Basis. $i=0$. Take $p'=p$ and $w'=w$.

Step. $i \geq 1$. Let t be the transition added to $\mathcal{A}$ in the i-th step. Let j be the number of times that t is used in $p \xrightarrow[i]{w} q$. $\mathcal{A}$ does not have any transitions leading to initial states, and the algorithm does not add any such transitions; therefore, if t starts in an initial state, t can be used at most once and only at the start of the path.

The proof is by induction on j. If $j = 0$, then we have $p \xrightarrow[i-1]{w} q$, and we apply the induction hypothesis (induction on i). So assume that $j > 0$. We distinguish the three possible cases of the saturation rule:

(i) and (ii): $t = (p_1, v, q_1)$, where $v = \varepsilon$ or $v = \gamma_1$: Then $j = 1$ and there exists w_1 such that $w = vw_1$ and q_1 such that

$$p = p_1 \xrightarrow[i]{v} q_1 \xrightarrow[i-1]{w_1} q \tag{0}$$

Since t was added via the saturation rule, there exist p_2 and γ_2 such that

$$\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, v \rangle \tag{1.1}$$

$$p_2 \xrightarrow[i-1]{\gamma_2} q_1 \tag{1.2}$$

From (0) and (1.2) we get

$$p_2 \xrightarrow[i-1]{\gamma_2} q_1 \xrightarrow[i-1]{w_1} q \tag{2}$$

t is not used in (2), so applying the induction hypothesis (on j) we obtain

$$\langle p', w' \rangle \Longrightarrow \langle p_2, \gamma_2 w_1 \rangle \tag{3.1}$$

$$p' \xrightarrow[0]{w'} q \tag{3.2}$$

Combining (1.1) and (3.1) we have

$$\langle p', w' \rangle \overset{(3.1)}{\Longrightarrow} \langle p_2, \gamma_2 w_1 \rangle \overset{(1.1)}{\Longrightarrow} \langle p_1, v w_1 \rangle = \langle p, w \rangle \tag{4}$$

The lemma follows from (3.2) and (4).

(iii) Let $t = (r, \gamma'', q')$ be a transition resulting from the application of the third part of the saturation rule. Then there exist u, v such that $w = u\gamma''v$ and

$$p \xrightarrow[i-1]{u} r \xrightarrow[i]{\gamma''} q' \xrightarrow[i]{v} q \tag{5}$$

Because t was added via the saturation rule, we conclude that there exists some rule of the form

$$\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, \gamma' \gamma'' \rangle \tag{6.1}$$

$$\text{with } p_2 \xrightarrow[i-1]{\gamma_2} q' \tag{6.2}$$

i.e. (6.1) is the rule associated with r. Application of the induction hypothesis (on i) yields

$$\langle p_3, w_3 \rangle \Rightarrow \langle p, u \rangle \tag{7.1}$$

$$p_3 \xrightarrow[0]{w_3} r \tag{7.2}$$

for some pair $\langle p_3, w_3 \rangle$; due to the construction of the automaton it holds that $\langle p_3, w_3 \rangle = \langle p_1, \gamma' \rangle$. Combining (5) and (6.2) we get

$$p_2 \xrightarrow[i-1]{\gamma_2} q' \xrightarrow[i]{v} q$$

Since t occurs less often than j in this path, we can apply the induction hypothesis (on j) to obtain the existence of some $\langle p', w' \rangle$ such that

$$\langle p', w' \rangle \Rightarrow \langle p_2, \gamma_2 v \rangle \tag{8.1}$$

$$p' \xrightarrow[0]{w'} q \tag{8.2}$$

Finally, if we put together (6.1), (7.1) and (8.1), we get

$$\langle p', w' \rangle \overset{(8.1)}{\Longrightarrow} \langle p_2, \gamma_2 v \rangle \overset{(6.1)}{\Longrightarrow} \langle p_1, \gamma' \gamma'' v \rangle = \langle p_3, w_3 \gamma'' v \rangle \overset{(7.1)}{\Longrightarrow} \langle p, u \gamma'' v \rangle = \langle p, w \rangle \tag{9}$$

and (a) follows from (8.2) and (9). □

Theorem 5.2 *Let $\mathcal{A}_{post^*}$ be the automaton obtained from $\mathcal{A}$ by exhaustive application of the saturation rule defined in Section 4.3. $\mathcal{A}_{post^*}$ recognises the set $post^*(Conf(\mathcal{A}))$.*

Proof:

Let $\langle p, w \rangle$ be a configuration of $post^*(Conf(\mathcal{A}))$. Then $\langle p', w' \rangle \Rightarrow \langle p, w \rangle$ for a configuration $\langle p', w' \rangle \in Conf(\mathcal{A})$. By Lemma 5.3, $p \xrightarrow{w} q$ for some final state q of $\mathcal{A}_{post^*}$. So $\langle p, w \rangle$ is recognised by $\mathcal{A}_{post^*}$.

Conversely, let $\langle p, w \rangle$ be a configuration recognised by $\mathcal{A}_{post^*}$. Then $p \xrightarrow{w} q$ in $\mathcal{A}_{post^*}$ for some final state q. By Lemma 5.4, $\langle p', w' \rangle \Rightarrow \langle p, w \rangle$ for a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow[0]{w'} q$. Since q is a final state, $\langle p', w' \rangle \in Conf(\mathcal{A})$, and so $\langle p, w \rangle \in post^*(Conf(\mathcal{A}))$. □

Appendix 2: Correctness and complexity of the implementations

Algorithm 1: Correctness and Complexity

In this section we prove the correctness of the algorithm for pre^* given in section 4.4.1 along with the complexity bounds given in Theorem 4.1.

Termination: *rel* is empty initially and can only grow afterwards. Q and Γ are finite sets, therefore *rel* can only take finitely many elements. Similarly, *trans* can only be of finite size. Once no more transitions can be added to *rel*, *trans* can no longer grow and will be empty eventually.

Because of the finiteness of *rel* and Δ there will be finitely many members of Δ', and so the loop at line 7 is traversed only finitely often.

Lemma 5.5 *For every transition t, if $t \in$ trans at any time during the execution of the algorithm, then t will be "used" exactly once, i.e. the part between lines 6 and 12 will be entered exactly once for t.*

Proof: *rel* is empty initially and contains exactly the used transitions later. Until t is removed from *trans*, the algorithm cannot terminate. When t is removed from *trans*, it is used if and only if it is not yet a used transition, otherwise it will be ignored. □

Correctness:

(1) Throughout the algorithm $rel \subseteq \delta_{pre^*}$ holds. *rel* contains only elements from *trans*, so we inspect the lines that change *trans*, and show that all the additions are in compliance with the algorithm given in section 4.2 the correctness of which has already been proved. More precisely, we show that all additions model that algorithm's saturation rule:

If $\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle$ and $p' \xrightarrow{w} q$, then add (p,γ,q).

- In line 1, *trans* is assigned δ which allows us to recognise C.
- Lines 2 and 12 directly model the saturation rule.
- In line 8, if the rule $\langle p_1,\gamma_1\rangle \hookrightarrow \langle q,\gamma\rangle$ is taken from Δ, then we directly model the saturation rule. Otherwise, $\langle p_1,\gamma_1\rangle \hookrightarrow \langle q,\gamma\rangle$ was added to Δ' because $\langle p_1,\gamma_1\rangle \hookrightarrow \langle p'',\gamma'\gamma\rangle \in \Delta$ and (p'',γ',q) in *trans* for some p'',γ' and again the saturation rule applies.

(2) After termination $\delta_{pre^*} \subseteq rel$ holds. Initially, *trans* contains δ, and because of Lemma 5.5 all of δ will eventually end up in *rel*. Moreover, we prove that all possible applications of the saturation rule are used. Assume that we have $\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle$ and $p' \xrightarrow{w} q$ w.r.t. *rel*.

- If $w = \varepsilon$, then $q = p'$, and (p,γ,p') is added in line 2.
- If $w = \gamma_1$, then there is some transition $t_1 = (p',\gamma_1,q)$ in *rel*. Because *rel* contains only the used transitions, and because of line 8, (p,γ,q) will be added to *trans*.
- If $w = \gamma_1\gamma_2$, then *rel* contains $t_1 = (p',y_1,q')$ and $t_2 = (q',\gamma_2,q)$.
  - If t_1 is used before t_2, Δ' will have the rule $\langle p,\gamma\rangle \hookrightarrow \langle q',\gamma_2\rangle$. Then, when t_2 is used, (p,γ,q) is added in line 8.
  - If t_2 is used before t_1, it is in *rel* when t_1 is used. Then (p,γ,q) is added in line 12.

Complexity: Let $n_Q = |Q|$, $n_\delta = |\delta|$, and $n_\Delta = |\Delta|$, i.e. the number of states and transitions in $\mathcal{A}$, and the number of rules in the pushdown system. The size of $\mathcal{A}$ can be written as $n_Q + n_\delta$.

Imagine that prior to running the algorithm, all rules of the form $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma' w\rangle$ have been put into "buckets" labelled (p',γ'). If the buckets are organised in a hash table this can be done in $O(n_\Delta)$ time and space. Similarly, all rules in Δ' can be put into such buckets at run-time, and the addition of one rule takes only constant time.

If *rel* and δ are implemented as hash tables, then addition and membership test take constant time. Moreover, if *trans* is a stack, then addition and removal of transitions take constant time, too.

When (q,γ,q') is used, we need to regard just the rules that are in the (q,γ)-bucket. Because of Lemma 5.5, every possible transition is used at most once. Based on these observations, let us compute how often the statements inside the main loop are executed.

Line 10 is executed once for each combination of rules $\langle p_1,\gamma_1\rangle \hookrightarrow \langle q,\gamma\gamma_2\rangle$ and transitions (q,γ,q'), i.e. $O(n_Q n_\Delta)$ times, therefore the size of Δ' is $O(n_Q n_\Delta)$, too. For the loop starting at line 11, q' and γ_2 are fixed, so line 12 is executed $O(n_Q^2 n_\Delta)$ times.

Line 8 is executed once for each combination of rules $\langle p_1,\gamma_1\rangle \hookrightarrow \langle q,\gamma\rangle$ in $(\Delta \cup \Delta')$ and transitions (q,γ,q'). As stated previously, the size of Δ' is $O(n_Q n_\Delta)$, so line 8 is executed $O(n_Q^2 n_\Delta)$ times.

Let us now count the iterations of the main loop, i.e. how often line 4 is executed. This directly depends on the number of elements that are added to *trans*. Initially, there are $n_\delta + O(n_\Delta)$ elements from lines 1 and 2. Notice that $n_\delta = O(n_Q \cdot n_\Delta \cdot n_Q)$. We already know that the other additions to *trans* are no more than $O(n_Q^2 n_\Delta)$ in number. As a conclusion, the whole algorithm takes $O(n_Q^2 n_\Delta)$ time.

Memory is needed for storing *rel*, *trans* and Δ'.

- Line 1 adds n_δ transitions to *trans*.
- In line 2, there are $O(n_\Delta)$ additions.
- In lines 8 and 12, p_1 and γ_1 are taken from the head of a rule in Δ. This means that these lines can only add $O(n_\Delta n_Q)$ transitions to *rel*.
- The size of Δ' is $O(n_Q n_\Delta)$ (see above).

From these facts it follows that the algorithm takes $O(n_Q n_\Delta + n_\delta)$ space, the size needed to store the result. Algorithm 1 is therefore optimal with respect to memory usage.

The result of the analysis is summarised in Theorem 4.1.

Algorithm 2: Correctness and Complexity

We prove the correctness of the algorithm for $post^*$ given in section 4.4.2 along with the complexity bounds given in Theorem 4.2.

For better understanding of the following paragraphs it is useful to consider the structure of the transitions in $\mathcal{A}_{post^*}$. Let $Q_1 = (Q \setminus P)$ and $Q_2 = (Q' \setminus Q)$.

In the beginning, there are no transitions into P, i.e. we just have transitions from P into Q_1, and from Q_1 into Q_1. After line 5 is executed once, we also have transitions from P to Q_2. All the other additions to *rel* are now either from P to $Q_1 \cup Q_2$ except for line 20; here we have transitions from Q_2 to $Q_1 \cup Q_2$. We can summarise these observations in the following facts:

- After execution of the algorithm, *rel* contains no transitions leading into P.

- The algorithm does not add any transitions starting in Q_1.

Termination: The algorithm terminates. This can be seen from the fact that the size of Q' is bounded by $|Q| + |\Delta|$; hence, *rel*'s maximum size is $|Q'| \cdot |\Gamma| \cdot |Q'|$, and we can use a similar reasoning as in the algorithm for pre^*.

Correctness: We show that the algorithm is an implementation of the construction given earlier. In section 4.3 we defined δ_{post^*} to be the smallest set of transitions containing δ, containing a transition (p_1, γ_1, q_r) for every rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_2 \rangle$ and satisfying the following saturation properties:

- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$, then $(p', \varepsilon, q) \in \delta_{post^*}$.
- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$, then $(p', \gamma', q) \in \delta_{post^*}$.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$, then $(r, \gamma'', q) \in \delta_{post^*}$.

The automaton constructed in Algorithm 2 does not have ε-transitions, so we cannot show that $rel = \delta_{post^*}$. Instead, we show that after execution, it holds that $((q, \gamma, q') \in rel) \iff (q \stackrel{\gamma}{\Longrightarrow} q')$ for all $q, q' \in Q$, $\gamma \in \Gamma$.

"$\Rightarrow$" Since elements from *trans* flow into *rel*, we inspect all the lines that change either *trans* or *rel*:

- Lines 1 and 2 add elements from δ which is a subset of δ_{pre^*}.
- Line 5 is a consequence of the initialisation rule.
- In line 15 we have (p, γ, q) and (q, γ', q') in *rel*, $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ in Δ, hence $p' \xrightarrow{\varepsilon} q \xrightarrow{\gamma'} q'$, so (p, γ', q') does not change the desired property.
- Line 18 is a direct implementation of the second saturation property.
- Line 20 is a direct implementation of the third saturation property.
- In line 22 it holds that $p'' \in eps(q_r)$; from this we conclude the existence of some rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p'', \varepsilon \rangle$ and some transition (p_1, γ_1, q_r). So, $p'' \xrightarrow{\varepsilon} q_r \xrightarrow{\gamma_2} q$, and the addition of (p'', γ_2, q) doesn't change the property.

"$\Leftarrow$" By the same argumentation as in Lemma 5.5 we can say that all the elements in *trans* eventually end up in *rel*. Therefore it is sufficient to prove that any element of δ_{post^*} is added to either *rel* or *trans* during execution of the algorithm.
We observe the following: Since there are no transitions leading into P, the ε-transitions can only go from states in P to states in $Q_1 \cup Q_2$; therefore no two ε-transitions can be adjacent. The relation $p \stackrel{\gamma}{\Longrightarrow} q$ from section 4.3 can be written as follows:

$$(p \stackrel{\gamma}{\Longrightarrow} q) \iff (p, \gamma, q) \vee \exists q' : ((p, \varepsilon, q') \wedge (q', \gamma, q))$$

The desired property follows from the following facts:

- Because of lines 1 and 2, after execution $\delta \subseteq rel$ holds.
- If $\langle p', \gamma' \rangle \hookrightarrow \langle p, \varepsilon \rangle$ and $(p', \gamma', q) \in rel$, then p is added to $eps(q)$. We will see that whenever there is p in $eps(q')$ and (q', γ, q) in *rel* for some $p, q', q \in Q'$ and $y \in \Gamma$, eventually (p, γ, q) will be in *rel*.

  - Either (q', γ, q) is known before $p \in eps(q')$ is known. Then (p, γ, q) is added in line 15;
  - Or $p \in eps(q')$ is known before (q', γ, q). Recall that $q' \in Q_1 \cup Q_2$. ε-transitions are added only after the initialisation phase, and the only transitions starting in $Q_1 \cup Q_2$ added after initialisation are those in line 20. In this case (p, γ, q) is added in line 22.

- If $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma \rangle$ and $(p', \gamma', q) \in rel$, then (p, γ, q) is added in line 18.
- If $r = \langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ then (p, γ, r) is added in line 5. If moreover $(p', \gamma', q) \in rel$, (r, γ'', q) is added in line 20.

Complexity: Let $n_P, n_Q, n_\Delta, n_\delta$ be the sizes of P, Q, Δ and δ, respectively. Once again let *rel* and δ be implemented as a hash table and *trans* as a stack, so that all the needed addition, membership test and removal operations take constant time. The sets *eps* can be implemented as bit-arrays with one entry for each state in P; again addition and membership test cost only constant time.

The rules in Δ can be sorted into buckets according to their left-hand side (at the cost of $O(\Delta)$ time and space); i.e. put every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ into the bucket labelled (p, γ). The transitions in *rel* can be sorted into buckets according to the source state (i.e. a transition (q, γ, q') would be sorted into a bucket labelled q); since no transition is added to *rel* more than once, this costs no more than $O(|rel|)$ time and space.

For every transition $t = (p, \gamma, q) \in rel$, the part from line 10 and 22 is executed only once. Because we just need to take a look at the (p, γ)-bucket for rules, we can make the following statements:

- Line 5 is executed $O(n_\Delta)$ times.
- Line 18 is executed once for every combination of rules $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and transitions (p, γ, q) of which there are at most $|\Delta|\,|Q'|$ many, i.e. there are $O(n_\Delta(n_Q + n_\Delta))$ many executions.
- Line 20 is executed $O(n_\Delta(n_Q + n_\Delta))$ times (like line 18).
- Since $eps(q)$, $q \in Q'$ can contain at most n_P entries, line 22 is executed $O(n_P n_\Delta(n_Q + n_\Delta))$ times.
- For line 15, analysis becomes more complicated. First, observe that the part from line 13 to 16 is executed only once for every $(p, \varepsilon, q) \in \delta_{post^*}$. Let us distinguish two cases:
  - $q \in Q_1$: Altogether, there are $O(n_\delta)$ many transitions going out from the states in Q_1, and each of them can be copied at most once to every state in P which means $O(n_P n_\delta)$ many operations (remember that the algorithm adds no transitions leaving states in Q_1!)
  - $q \in Q_2$: If $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$, then all the transitions leaving q_r are of the form (q_r, γ_3, q'') where q'' may be in $Q_1 \cup Q_2$. There are $O(n_\Delta)$ many states in Q_2, so we end up with $O(n_P n_\Delta(n_Q + n_\Delta))$ many operations.
- Line 8 is executed at most once for every transition in δ and for every transition added in the lines discussed above, i.e. $O(n_P n_\Delta(n_Q + n_\Delta) + n_P n_\delta)$ times. This is also an upper bound for the size of *rel* resp. *trans*.

- The initialisation phase can be completed in $O(n_\delta + n_\Delta + n_Q)$ (for the transitions in δ we just need to decide whether the source state is in P and add the transition to either *trans* or *rel*).
- We need $O(n_Q + n_\Delta)$ space to store Q', $O(n_Q)$ for F' and $O(n_P(n_Q + n_\Delta))$ for *eps*.

The result of this subsection is summarised in Theorem 4.2.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Symbolic Trajectory Evaluation (STE): Automatic Refinement and Vacuity Detection

Orna GRUMBERG
Technion – Israel Institute of Technology, Haifa, Israel

Abstract. Symbolic Trajectory Evaluation (STE) is a powerful technique for hardware model checking. It is based on combining 3-valued abstraction with symbolic simulation, using 0,1 and X ("unknown"). The X value is used to abstract away parts of the circuit. The abstraction is derived from the user's specification. Currently the process of refinement in STE is performed manually. This paper presents an automatic refinement technique for STE. The technique is based on a clever selection of constraints that are added to the specification so that on the one hand the semantics of the original specification is preserved, and on the other hand, the part of the state space in which the "unknown" result is received is significantly decreased or totally eliminated. In addition, this paper raises the problem of vacuity of passed and failed specifications. This problem was never discussed in the framework of STE. We describe when an STE specification may vacuously pass or fail, and propose a method for vacuity detection in STE.

Keywords. Symbolic Trajectory Evaluation (STE), model checking, abstraction-refinement, vacuity

1. Introduction

This paper is an overview of the work presented in [30] and [29]. It presents the framework of Symbolic Trajectory Evaluation (STE) and describes automatic refinement and vacuity detection in this context.

Symbolic Trajectory Evaluation (STE) [26] is a powerful technique for hardware model checking. STE combines 3-valued abstraction with symbolic simulation. It is applied to a circuit M, described as a graph over *nodes* (gates and latches). Specifications in STE consist of assertions in a restricted temporal language. The assertions are of the form $A \Longrightarrow C$, where the *antecedent* A expresses constraints on nodes n at different times t, and the *consequent* C expresses requirements that should hold on such nodes (n, t). For each node, STE computes a symbolic representation, often in the form of a Binary Decision Diagram (BDD) [8]. The BDD represents the value of the node as a function of the values of the circuit's inputs. For precise symbolic representation, memory requirements might be prohibitively high. Thus, in order to handle very large circuits, it is necessary to apply some form of abstraction.

Abstraction in STE is derived from the specification by initializing all inputs not appearing in A to the X ("unknown") value. The rest of the inputs are initialized according

to constraints in A to the values 0 or 1 or to symbolic variables. A fourth value, $\perp$, is used in STE for representing a contradiction between a constraint in A on some node (n, t) and the actual value of node n at time t in the circuit.

In [18], a 4-valued truth domain $\{0, 1, X, \perp\}$ is defined for the temporal language of STE, corresponding to the 4-valued domain of the values of circuit nodes. Thus, STE assertions may get one of these four values when checked on a circuit M. The values 0 and 1 indicate that the assertion fails or passes on M, respectively. The $\perp$ truth value indicates that no computation of M satisfies A. Thus, the STE assertion passes vacuously. The X truth value indicates that the antecedent is too coarse and underspecifies the circuit.

In the latter case a *refinement* is needed. Refinement in STE amounts to changing the assertion in order to present node values more accurately.

STE has been in active use in the hardware industry, and has been very successful in verifying huge circuits containing large data paths [27,25,34]. Its main drawback, however, is the need for manual abstraction and refinement, which can be labor-intensive.

In this work we propose a technique for automatic refinement of assertions in STE. In our technique, the initial abstraction is derived, as usual in STE, from the given specification. The refinement is an iterative process, which stops when a truth value other than X is achieved. Our automatic refinement is applied when the STE specification results with X. We compute a set of input nodes, whose refinement is sufficient for eliminating the X truth value. We further suggest heuristics for choosing a small subset of this set.

Selecting a "right" set of inputs has a crucial role in the success of the abstraction and refinement process: selecting too many inputs will add many variables to the computation of the symbolic representation, and may result in memory and time explosion. On the other hand, selecting too few inputs or selecting inputs that do not affect the result of the verification will lead to many iterations with an X truth value.

We point out that, as in any automated verification framework, we are limited by the following observations. First, there is no automatic way to determine whether the provided specification is in accord with the user intention. Therefore, we assume that it is, and we make sure that our refined assertion passes on the concrete circuit if and only if the original assertion does. Second, bugs cannot automatically be fixed. Thus, counterexamples are analyzed by the user.

Another important contribution of our work is identifying that STE results may hide vacuity. This possibility was never raised before. Hidden vacuity may occur since an abstract execution of M on which the truth value of the specification is 1 or 0, might not correspond to any concrete execution of M. In such a case, a pass is *vacuous*, while a counterexample is *spurious*. We propose two algorithms for detecting these cases.

We implemented our automatic refinement technique within Intel's Forte environment [27]. We ran it on two nontrivial circuits with several assertions. Our experimental results show success in automatically identifying a set of inputs that are crucial for reaching a definite truth value. Thus, a small number of iterations were needed.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3 we give some background and basic definitions and notations. Section 4 describes the inherent limitations of automatic refinement of specifications versus manual refinement, and characterizes our proposed refinement technique. Section 5 presents heuristics for choosing a subset of inputs to be refined. Section 6 defines the vacuity problem in STE and suggests several methods for vacuity detection. Section 7 briefly summa-

rizes experimental results of our refinement technique. Finally, Section 8 concludes and suggests directions for future research.

2. Related Work

Abstraction is a well known methodology in model checking for fighting the state explosion problem. Abstraction hides certain details of the system in order to result in a smaller model. Two types of semantics are commonly used for interpreting temporal logic formulas over an abstract model. In the two-valued semantics, a formula is either true or false in the abstract model. When the formula is true, it is guaranteed to hold for the concrete model as well. On the other hand, false result may be spurious, meaning that the result in the concrete model may not be false. In the three-valued semantics [7,28], a third truth value is introduced: the unknown truth value. With this semantics, the true and false truth values in the abstract model are guaranteed to hold also in the concrete model, whereas the unknown truth value gives no information about the truth value of the formula in the concrete model.

In both semantics, when the model checking result on the abstract model is inconclusive, the abstract model is refined by adding more details to it, making it more similar to the concrete model. This iterative process is called Abstraction-Refinement, and has been investigated thoroughly in the context of model checking [14,10,21,15,3].

The work presented in this paper is the first attempt to perform automatic refinement in the framework of STE. In [13], it is shown that the abstraction in STE is an abstract interpretation via a Galois connection. However, [13] is not concerned with refinement. In [32], an automatic abstraction-refinement for symbolic simulation is suggested. The main differences between our work and [32] is that we compute a set of sufficient inputs for refinement and that our suggested heuristics are significantly different from those proposed in [32].

Recently, two new refinement methods have been suggested. The automatic refinement presented in [12] is based on a notion of responsibility and can be combined with the method presented here. The method in [16] is applicable only in the SAT-based STE framework developed there. In [1], a method for automatic *abstraction* without refinement is suggested.

Generalized STE (GSTE) [36] is a significant extension of STE that can verify all ω-regular properties. Two manual refinement methods for GSTE are presented in [35]. In the first method, refinement is performed by changing the specification. In the second method, refinement is performed by choosing a set of nodes in the circuit, whose values and the relationship among them are always represented accurately. In [33], SAT-based STE is used to get quick feedback when debugging and refining a GSTE assertion graph. However, the debugging and refinement process itself is manual. An automatic refinement for GSTE has recently been introduced in [11].

An additional source of abstraction in STE is the fact that the constraints of A on internal nodes are propagated only forward through the circuit and through time. We do not deal with this source of abstraction. In [36], they handle this problem by the Bidirectional (G)STE algorithm, in which backward symbolic simulation is performed, and new constraints implied by the existing constraints are added to A. STE is then applied on the enhanced antecedent. Our automatic refinement can be activated at this stage.

Vacuity refers to the problem of trivially valid formulas. It was first noted in [4]. Automatic detection of vacuous pass under symbolic model checking was first proposed in [5] for a subset of the temporal logic ACTL called w-ACTL. In [5], vacuity is defined as the case in which, given a model M and a formula ϕ, there exists a sub formula ξ of ϕ which does not affect the validity of ϕ. Thus, replacing ξ with any other formula will not change the truth value of ϕ in M. In [19,20] the work of [5] has been extended by presenting a general method for detecting vacuity for specifications in CTL*. Further extensions appear in [2,9].

In the framework of STE, vacuity, sometimes referred to as *antecedent failure*, is discussed in [18,26]. Roughly speaking, it refers to the situation in which a node is assigned with a $\perp$ value, implying that there are no concrete executions of the circuit that satisfy all the constraints in A. As a result, $A \Longrightarrow C$ is trivially satisfied. This is in fact a special case of vacuity as defined in [5]. The work presented here is the first to raise the problem of hidden vacuity, in which the formula is trivially satisfied despite the fact that no nodes are assigned with the $\perp$ value.

3. Background

3.1. Circuits

There are different levels in which hardware circuits can be modeled. We concentrate on a synchronous gate-level view of the circuit, in which the circuit is modeled by logical gates such as AND and OR and by delay elements (latches). Aspects such as timing, asynchronous clock domains, power consumption and physical layout are ignored, making the gate-level model an abstraction of the real circuit.

More formally, a circuit M consists of a set of nodes $\mathcal{N}$, connected by directed edges. A node is either an input node or an internal node. Internal nodes consist of latches and combinational nodes. Each combinational node is associated with a Boolean function. The nodes are connected by directed edges, according to the wiring of the electric circuit. We say that a node n_1 enters a node n_2 if there exists a directed edge from n_1 to n_2. The nodes entering a certain node are its ***source nodes***, and the nodes to which a node enters are its ***sink nodes***. The value of a latch at time t can be expressed as a Boolean expression over its source nodes at times t and $t-1$, and over the latch value at time $t-1$. The value of a latch at time 0 is determined by a given initial value. The *outputs* of the circuit are designated internal nodes whose values are of interest. We restrict the set of circuits so that the directed graph induced by M may contain loops but no combinational loops.
Throughout the paper we refer to a node n at a specific time t as (n, t).

An example of a circuit is shown in Figure 1. It contains three inputs In1, In2 and In3, two OR nodes N1 and N2, two AND nodes N3 and N6, and two latches N4 and N5. For simplicity, the clocks of the latches were omitted and we assume that at each time t the latches sample their data source node from time $t-1$. Note the negation on the source node In2 of N2.

The ***bounded cone of influence*** **(BCOI)** of a node (n, t) contains all nodes (n', t') with $t' \leq t$ that may influence the value of (n, t), and is defined recursively as follows: the BCOI of a combinational node at time t is the union of the BCOI of its source nodes

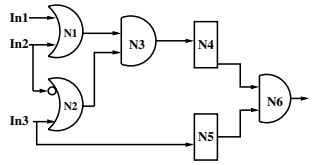


Figure 1. A Circuit

at time t, and the BCOI of a latch at time t is the union of the BCOI of its source nodes at times t and $t-1$ according to the latch type.

3.2. Four-Valued Symbolic Simulation

Usually, the circuit nodes receive Boolean values, where the value of a node can be described by a Boolean expression over its inputs. In STE, a third value, X ("unknown"), is introduced. Attaching X to a certain node represents lack of information regarding the Boolean value of that node. The motivation for the introduction of X is that its use decreases the size of the Boolean expressions of the circuit nodes. This, however, is done at the expense of the possibility of receiving unknown values for the circuit outputs.

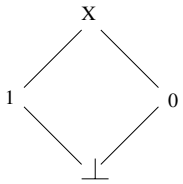


Figure 2. The $\sqsubseteq$ partial order

A fourth value, $\perp$, is also added to represent the over-constrained value, in which a node is forced both to 0 and to 1. This value indicates that a contradiction exists between external assumptions on the circuit and its actual behavior. The set of values $\mathcal{Q} \equiv \{0, 1, X, \perp\}$ forms a complete lattice with the partial order $0 \sqsubseteq X$, $1 \sqsubseteq X$, $\perp \sqsubseteq 0$ and $\perp \sqsubseteq 1$ (see Figure 2 [1]. This order corresponds to set inclusion, where X represents the set $\{0, 1\}$, and $\perp$ represents the empty set. As a result, the *greatest lower bound* (the lattice's meet) $\sqcap$ corresponds to set intersection and the *least upper bound* (the lattice's join) $\sqcup$ corresponds to set union. The Boolean operations AND, OR and NOT are extended to the domain $\mathcal{Q}$ as shown in Figure 3.

AND	X	0	1	$\perp$
X	X	0	X	$\perp$
0	0	0	0	$\perp$
1	X	0	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

OR	X	0	1	$\perp$
X	X	X	1	$\perp$
0	X	0	1	$\perp$
1	1	1	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

NOT	
X	X
0	1
1	0
$\perp$	$\perp$

Figure 3. Quaternary operations

[1] Some works refer to the partial order in which X is the smallest element in the lattice and $\perp$ is the greatest.

A ***state*** s of the circuit M is an assignment of values from $\mathcal{Q}$ to all circuit nodes, $s : \mathcal{N} \rightarrow \mathcal{Q}$. Given two states s_1, s_2, we say that $s_1 \sqsubseteq s_2 \iff ((\exists n \in \mathcal{N} : s_1(n) = \bot) \vee (\forall n \in \mathcal{N} : \ s_1(n) \sqsubseteq s_2(n)))$. A state is ***concrete*** if all nodes are assigned with values out of $\{0, 1\}$. A state s is an abstraction of a concrete state s_c if $s_c \sqsubseteq s$.

A ***sequence*** σ is any infinite series of states. We denote by $\sigma(i), i \in \mathbb{N}$, the state at time i in σ, and by $\sigma(i)(n), i \in \mathbb{N}, n \in \mathcal{N}$, the value of node n in the state $\sigma(i)$. $\sigma^i, i \in \mathbb{N}$, denotes the suffix of σ starting at time i. We say that $\sigma_1 \sqsubseteq \sigma_2 \iff ((\exists i \geq 0, n \in \mathcal{N} : \sigma_1(i)(n) = \bot) \vee (\forall\, i \geq 0 : \sigma_1(i) \sqsubseteq \sigma_2(i)))$. Note that we refer to states and sequences that contain $\bot$ values as least elements w.r.t $\sqsubseteq$.

In addition to the quaternary set of values Q, STE uses Boolean symbolic variables which enable to simulate many runs of the circuit at once. Let V be a set of symbolic Boolean variables over the domain $\{0, 1\}$. A ***symbolic expression*** over V is an expression consisting of quaternary operations, applied to $V \cup \mathcal{Q}$. A ***symbolic state*** over V is a mapping which maps each node of M to a symbolic expression. Each symbolic state represents a set of states, one for each assignment to the variables in V. A ***symbolic sequence*** over V is a series of symbolic states. It represents a set of sequences, one for each assignment to V. Given a symbolic sequence σ and an assignment ϕ to V, $\phi(\sigma)$ denotes the sequence that is received by applying ϕ to all symbolic expressions in σ. Given two symbolic sequences σ_1,σ_2 over V, we say that $\sigma_1 \sqsubseteq \sigma_2$ if for all assignments ϕ to V, $\phi(\sigma_1) \sqsubseteq \phi(\sigma_2)$.

Sequences may be incompatible with the behavior of M. A ***(symbolic) trajectory*** π is a (symbolic) sequence that is compatible with the behavior of M [24]: let $val(n, t, \pi)$ be the value of a node (n, t) as computed according to the values of its source nodes in π. It is required that for all nodes (n, t), $\pi(t)(n) \sqsubseteq val(n, t, \pi)$ (strict equality is not required in order to allow external assumptions on nodes values to be embedded into π). A trajectory is ***concrete*** if all its states are concrete. A trajectory π is an abstraction of a concrete trajectory π_c if $\pi_c \sqsubseteq \pi$.

The difference between assigning an input with a symbolic variable and assigning it with X is that a symbolic variable is used to obtain an accurate representation of the value of the input. For example, the negation of a variable v is $\neg v$ whereas the negation of X is X. In addition, if two different inputs are assigned with the same variable v in a symbolic sequence σ, then it implies that the two inputs have the same value in every concrete sequence derived from σ by applying to it an assignment ϕ. However, if the inputs are assigned with X, then it does not imply that they have the same value in any concrete sequence corresponding to σ.

Figure 4 describes a symbolic trajectory of the circuit from Figure 1 up to time 1. The values given by the user are marked in bold, and include the input values and the initial values of the latches. The notation $v_3?1 : X$ stands for "if v_3 holds then 1 else X".

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	$\mathbf{v_1}$	**1**	$\mathbf{v_2}$	1	v_2	v_2	X	**1**	X
1	$\mathbf{v_3}$	X	**0**	$v_3?1 : X$	X	X	v_2	v_2	v_2

Figure 4. Four-valued Symbolic Simulation

3.3. Trajectory Evaluation Logic (TEL)

We now describe the Trajectory Evaluation Language (TEL) used to specify properties for STE. This logic is a restricted version of the Linear Temporal Logic (LTL) [23], where only the next time temporal operator is allowed.
A ***Trajectory Evaluation Logic*** (TEL) formula is defined recursively over V as follows:

$$f ::= n \text{ is } p \mid f_1 \wedge f_2 \mid p \rightarrow f \mid \mathbf{N}f$$

where $n \in \mathcal{N}$, p is a Boolean expression over V and $\mathbf{N}$ is the next time operator. Note that TEL formulas can be expressed as a finite set of constraints on values of specific nodes at specific times. N^t denotes the application of t next time operators. The constraints on (n,t) are those appearing in the scope of N^t. The ***maximal depth*** of a TEL formula f, denoted depth(f), is the maximal time t for which a constraint exists in f on some node (n,t), plus 1.

Usually, the satisfaction of a TEL formula f on a symbolic sequence σ is defined in the 2-valued truth domain [26], i.e., f is either satisfied or not satisfied. In [18], $\mathcal{Q}$ is used also as a 4-valued truth domain for an extension of TEL. We also use a 4-valued semantics. However, our semantic definition is different from [18] w.r.t $\perp$ values. In [18], a sequence σ containing $\perp$ values could satisfy f with a truth value different from $\perp$. In our definition this is not allowed. We believe that our definition captures better the intent behind the specification w.r.t contradictory information about the state space. The intuition behind our definition is that a sequence that contains $\perp$ value does not represent any concrete sequence, and thus vacuously satisfies all properties.
Given a TEL formula f over V, a symbolic sequence σ over V, and an assignment ϕ to V, we define the satisfaction of f as follows:

$$\begin{aligned}
&[\phi, \sigma \models f] = \perp && \leftrightarrow \quad \exists i \geq 0, n \in \mathcal{N} : \phi(\sigma)(i)(n) = \perp. \text{ Otherwise:} \\
&[\phi, \sigma \models n \text{ is } p] = 1 && \leftrightarrow \quad \phi(\sigma)(0)(n) = \phi(p) \\
&[\phi, \sigma \models n \text{ is } p] = 0 && \leftrightarrow \quad \phi(\sigma)(0)(n) \neq \phi(p) \text{ and } \phi(\sigma)(0)(n) \in \{0,1\} \\
&[\phi, \sigma \models n \text{ is } p] = X && \leftrightarrow \quad \phi(\sigma)(0)(n) = X \\
&[\phi, \sigma \models p \rightarrow f] && = \quad (\neg\phi(p) \vee \phi, \sigma \models f) \\
&[\phi, \sigma \models f_1 \wedge f_2] && = \quad (\phi, \sigma \models f_1 \wedge \phi, \sigma \models f_2) \\
&[\phi, \sigma \models \mathbf{N}f] && = \quad \phi, \sigma^1 \models f
\end{aligned}$$

Note that given an assignment ϕ to V, $\phi(p)$ is a constant (0 or 1). In addition, the $\perp$ truth value is determined only according to ϕ and σ, regardless of f.
We define the truth value of $\sigma \models f$ as follows:

$$\begin{aligned}
&[\sigma \models f] = 0 && \leftrightarrow \quad \exists \phi : [\phi, \sigma \models f] = 0 \\
&[\sigma \models f] = X && \leftrightarrow \quad \forall \phi : [\phi, \sigma \models f] \neq 0 \text{ and } \exists \phi : [\phi, \sigma \models f] = X \\
&[\sigma \models f] = 1 && \leftrightarrow \quad \forall \phi : [\phi, \sigma \models f] \notin \{0, X\} \text{ and } \exists \phi : [\phi, \sigma \models f] = 1 \\
&[\sigma \models f] = \perp && \leftrightarrow \quad \forall \phi : [\phi, \sigma \models f] = \perp
\end{aligned}$$

It has been proved in [18] that the satisfaction relation is monotonic, i.e., for all TEL formulas f, symbolic sequences σ_1, σ_2 and assignments ϕ to V, if $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$ then $[\phi, \sigma_2 \models f] \sqsubseteq [\phi, \sigma_1 \models f]$. This also holds for our satisfaction definition.

Theorem 3.1 *[29] Given a TEL formula f and two symbolic sequences σ_1 and σ_2, if $\sigma_2 \sqsubseteq \sigma_1$ then $[\sigma_2 \models f] \sqsubseteq [\sigma_1 \models f]$.*

It has been proved in [18] that every TEL formula f has a ***defining sequence***, which is a symbolic sequence σ^f so that $[\sigma^f \models f] = 1$, and for all σ, $[\sigma \models f] \in \{1, \bot\}$ if and only if $\sigma \sqsubseteq \sigma^f$. For example, $\sigma^{q \to (n \text{ is } p)}$ is the sequence $s_{(n,q\to p)} s_x s_x s_x ...$, where $s_{(n,q\to p)}$ is the state in which n equals $(q \to p) \wedge (\neg q \to X)$, and all other nodes equal X, and s_x is the state in which all nodes equal X. σ^f may be incompatible with the behavior of M.

The ***defining trajectory*** π^f of M and f is a symbolic trajectory so that $[\pi^f \models f] \in \{1, \bot\}$ and for all trajectories π of M, $[\pi \models f] \in \{1, \bot\}$ if and only if $\pi \sqsubseteq \pi^f$. The $\bot$ may arise in case of a contradiction between M and f. (Similar definitions for σ^f and π^f exist in [26] with respect to a 2-valued truth domain).

Given σ^f, π^f is computed iteratively as follows: For all i, $\pi^f(i)$ is initialized to $\sigma^f(i)$. Next, the value of each node (n, i) is calculated according to its functionality and the values of its source nodes. The calculated value is then incorporated into $\pi^f(i)(n)$ using the $\sqcap$ operator. The computation of $\pi^f(i)$ continues until no new values are derived at time i. Note that since there are no combinational loops in M, it is guaranteed that eventually no new node values at time i will be derived. An example of a computation of π^f is given in Example 1.

3.4. Verification in STE

Specification in STE is given by STE assertions. STE ***assertions*** are of the form $A \Longrightarrow C$, where A (the ***antecedent***) and C (the ***consequent***) are TEL formulas. A expresses constraints on circuit nodes at specific times, and C expresses requirements that should hold on circuit nodes at specific times. $M \models (A \Longrightarrow C)$ if and only if for all concrete trajectories π of M and assignments ϕ to V, $[\phi, \pi \models A] = 1$ implies that $[\phi, \pi \models C] = 1$.

A natural verification algorithm for an STE assertion $A \Longrightarrow C$ is to compute the defining trajectory π^A of M and A and then compute the truth value of $\pi^A \models C$. If $[\pi^A \models C] \in \{1, \bot\}$ then it holds that $M \models (A \Longrightarrow C)$. If $[\pi^A \models C] = 0$ then it holds that $M \not\models (A \Longrightarrow C)$. If $[\pi^A \models C] = X$, then it cannot be determined whether $M \models (A \Longrightarrow C)$.

The case in which there is ϕ so that $\phi(\pi^A)$ contains $\bot$ is known as an ***antecedent failure***. The default behavior of most STE implementations is to consider antecedent failures as illegal, and the user is required to change A in order to eliminate any $\bot$ values. In this paper we take the approach that supports the full semantics of STE as defined above. That is, concrete trajectories $\phi(\pi^A)$ which include $\bot$ are ignored, since they do not satisfy A and therefore vacuously satisfy $A \Longrightarrow C$.

Note that although π^A is infinite, it is sufficient to examine only a bounded prefix of length depth(A) in order to detect $\bot$ values in π^A. The first $\bot$ value in π^A is the result of the $\sqcap$ operation on some node (n, t), where the two operands have contradicting assignments 0 and 1. Since $\forall i > \text{depth}(A) : \sigma^A(i) = s_x$, it must hold that $t \leq \text{depth}(A)$.

The truth value of $\pi^A \models C$ is determined as follows:

1. If for all ϕ, there exists i, n so that $\phi(\pi^A)(i)(n) = \perp$, then $[\pi^A \models C] = \perp$.
2. Otherwise, if there exists ϕ such that for some $i \geq 0, n \in \mathcal{N}$, $\phi(\pi^A)(i)(n) \in \{0,1\}$ and $\phi(\sigma^C)(i)(n) \in \{0,1\}$, and $\phi(\pi^A)(i)(n) \neq \phi(\sigma^C)(i)(n)$, and $\phi(\pi^A)$ does not contain $\perp$, then $[\pi^A \models C] = 0$.
3. Otherwise, if there exists ϕ such that for some $i \geq 0, n \in \mathcal{N}$, $\phi(\pi^A)(i)(n) = X$ and $\phi(\sigma^C)(i)(n) \in \{0,1\}$, and $\phi(\pi^A)$ does not contain $\perp$, then $[\pi^A \models C] = X$.
4. Otherwise, $[\pi^A \models C] = 1$.

Note that, similarly to detecting $\perp$, in order to determine the truth value of $\pi^A \models C$, it is sufficient to examine only a bounded prefix of length depth(C), since $\forall i >$ depth$(C) : \sigma^C(i) = s_x$.

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	0	X	v_1	X	$v_1?1 : X$	1	X	X	X
1	X	X	X	X	X	X	1	v_1	v_1

Figure 5. The Defining Trajectory π^A

Example 1 *Consider again the circuit M in Figure 1. Also consider the STE assertion $A \Longrightarrow C$, where $A =$ (In1 is 0) $\wedge$ (In3 is v_1) $\wedge$ (N3 is 1), and $C =$ **N**(N6 is 1). Figure 5 describes the defining trajectory π^A of M and A, up to time 1. It contains the symbolic expression of each node at time 0 and 1. The state $\pi^A(i)$ is represented by row i. The notation $v_1?1 : X$ stands for "if v_1 holds then 1 else X". σ^C is the sequence in which all nodes at all times are assigned X, except for node N6 at time 1, which is assigned 1. $[\pi^A \models C] = 0$ due to those assignments in which $v_1 = 0$. We will return to this example in Section 6.*

3.5. STE Implementation

Most widely used STE implementations are BDD-based (e.g., [27]). BDDs are used to represent the value of each node (n,t) as a function of the circuit's inputs. Since node values range over the quaternary domain $\{0,1,X,\perp\}$, two BDDs are used to represent the function of each node (n,t). This representation is called ***dual rail***.

The dual rail of a node (n,t) in π^A consists of two functions defined from V to $\{0,1\}$: $f^1_{n,t}$ and $f^0_{n,t}$, where V is the set of symbolic variables appearing in A. For each assignment ϕ to V, if $f^1_{n,t} \wedge \neg f^0_{n,t}$ holds under ϕ, then (n,t) equals 1 under ϕ. Similarly, $\neg f^1_{n,t} \wedge f^0_{n,t}$, $\neg f^1_{n,t} \wedge \neg f^0_{n,t}$ and $f^1_{n,t} \wedge f^0_{n,t}$ stand for 0, X and $\perp$ under ϕ, respectively. Likewise, $g^1_{n,t}$ and $g^0_{n,t}$ denote the dual rail representation of (n,t) in σ^C. Note that $g^1_{n,t} \wedge g^0_{n,t}$ never holds, since we always assume that C is not self-contradicting.

The BDDs for $f^1_{n,t}$ and $f^0_{n,t}$ can be computed for each node (n,t), based on the node's functionality and the BDDs of its input nodes. Usual BDD operations are sufficient. Once this computation terminates, the BDDs for $f^1_{n,t}$, $f^0_{n,t}$ are compared with $g^1_{n,t}$, $g^0_{n,t}$ in order to determine the truth value of the specification $A \Longrightarrow C$ on M. In the following section, we further elaborate on the use of the dual rail representation in computing the STE result.

Example 2 *Consider the symbolic trajectory described in Figure 4, where* $V = \{v_1, v_2, v_3\}$.

- *The value of node* $(In1, 1)$*,* v_3*, is given by the dual rail representation* $f^1_{In1,1}(V) = v_3$*,* $f^0_{In1,1}(V) = \neg v_3$.
- *The value of node* $(In2, 1)$*,* X*, is given by the dual rail representation* $f^1_{In2,1}(V) = 0$*,* $f^0_{In2,1}(V) = 0$.
- *The value of node* $(N1, 1)$*,* $v_3?1 : X$*, is given by the dual rail representation* $f^1_{N1,1}(V) = v_3$*,* $f^0_{N1,1}(V) = 0$.

4. Choosing Our Automatic Refinement Methodology

Intuitively, the defining trajectory π^A of a circuit M and an antecedent A is an abstraction of all concrete trajectories of M on which the consequent C is required to hold. This abstraction is directly derived from A. If $[\pi^A \models C] = X$, then A is too coarse, that is, it contains too few constraints on the values of circuit nodes. Our goal is to automatically refine A (and subsequently π^A) in order to eliminate the X truth value.

In this section we examine the requirements that should be imposed on automatic refinement in STE. We then describe our automatic refinement methodology, and formally state the relationship between the two abstractions, derived from the original and the refined antecedent. We refer only to STE implementations that compute π^A. We assume that antecedent failures are handled as described in Section 3.

We first describe the handling of $\perp$ values which is required for the description of the general abstraction and refinement process in STE. In the dual-rail notation given earlier, the Boolean expression $\neg f^1_{n,t} \vee \neg f^0_{n,t}$ represents all assignments ϕ to V for which $\phi(\pi^A)(t)(n) \neq \perp$. Thus, the Boolean expression $nbot \equiv \bigwedge_{(n,t)\in A}(\neg f^1_{n,t} \vee \neg f^0_{n,t})$ represents all assignments ϕ to V for which $\phi(\pi^A)$ does not contain $\perp$. It is sufficient to examine only nodes (n, t) on which there exists a constraint in A. This is because there exists a node (n, t) and an assignment ϕ to V such that $\phi(\pi^A)(t)(n) = \perp$ only if there exists a node (n', t') on which there exists a constraint in A and $\phi(\pi^A)(t')(n') = \perp$. That is, the constraint on (n', t') in A contradicts the behavior of M. Thus, $[\pi^A \models C] = \perp$ if and only if $nbot \equiv 0$.

We now describe how the abstraction and refinement process in STE is done traditionally, with the addition of supporting $\perp$ in π^A. The user writes an STE assertion $A \Longrightarrow C$ for M, and receives a result from STE. If $[\pi^A \models C] = 0$, then the set of all ϕ so that $[\phi, \pi^A \models C] = 0$ is provided to the user. This set, called the ***symbolic counterexample***, is given by the Boolean expression over V:

$$\Big(\bigvee_{(n,t)\in C} ((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))\Big) \wedge nbot.$$

Each assignment ϕ in this set represents a counterexample $\phi(\pi^A)$. The counterexamples are given to the user to analyze and fix.

If $[\pi^A \models C] = X$, then the set of all ϕ so that $[\phi, \pi^A \models C] = X$ is provided to the user. This set, called the ***symbolic incomplete trace***, is given by the Boolean expression over V:

$$(\bigvee_{(n,t)\in C}((g^1_{n,t} \vee g^0_{n,t}) \wedge \neg f^1_{n,t} \wedge \neg f^0_{n,t})) \wedge nbot.$$

The user decides how to refine the specification in order to eliminate the partial information that causes the X truth value. If $[\pi^A \models C] = \bot$, then the assertion passes vacuously. Otherwise, $[\pi^A \models C] = 1$ and the verification completes successfully.

We point out that, as in any automated verification framework, we are limited by the following observations. First, there is no automatic way to determine whether the provided specification is in accord with the user's intention. Therefore, we assume it is, and we make sure that our refined assertion passes on the concrete circuit if and only if the original assertion does. Second, bugs cannot automatically be fixed. Thus, counterexamples are analyzed by the user.

We emphasize that automatic refinement is valuable even when it eventually results in a fail. This is because counterexamples present specific behaviors of M and are significantly easier to analyze than incomplete traces.

As mentioned before, we must assume that the given specification is correct. Thus, automatic refinement of A must preserve the semantics of $A \Longrightarrow C$: Let $A_{new} \Longrightarrow C$ denote the refined assertion. Let $runs(M)$ denote the set of all concrete trajectories of M. We require that $A_{new} \Longrightarrow C$ holds on $runs(M)$ if and only if $A \Longrightarrow C$ holds on $runs(M)$.

In order to achieve the above preservation, we choose our automatic refinement as follows. Whenever $[\pi^A \models C] = X$, we add constraints to A that force the value of input nodes at certain times (and initial values of latches) to the value of *fresh symbolic variables*, that is, symbolic variables that do not already appear in V. By initializing an input (in, t) with a fresh symbolic variable instead of X, we represent the value of (in, t) accurately and add knowledge about its effect on M. However, we do not constrain input behavior that was allowed by A, nor do we allow input behavior that was forbidden by A. Thus, the semantics of A is preserved. In Section 5, a small but significant addition is made to our refinement technique.

We now formally state the relationship between the abstractions derived from the original and the refined antecedents. Let A be the antecedent we want to refine. A is defined over a set of variables V. Let V_{new} be a set of symbolic variables so that $V \cap V_{new} = \emptyset$. Let PI_{ref} be the set of inputs at specific times, selected for refinement. Let A_{new} be a refinement of A over $V \cup V_{new}$, where A_{new} is received from A by attaching to each input $(in, t) \in PI_{ref}$ a unique variable $v_{in,t} \in V_{new}$ and adding conditions to A as follows:

$$A_{new} = A \wedge \bigwedge_{(in,t)\in PI_{ref}} N^t(p \rightarrow (in \text{ is } v_{in,t})),$$

where $p = \neg q$ if (in, t) has a constraint $N^t(q \rightarrow (in \text{ is } e))$ in A for some Boolean expressions q and e over V, and $p = 1$ otherwise ((in, t) has no constraint in A). The reason we consider A is to avoid a contradiction between the added constraints and the original ones, due to constraints in A of the form $q \rightarrow f$.

Let $\pi^{A_{new}}$ be the defining trajectory of M and A_{new}, over $V \cup V_{new}$. Let ϕ be an assignment to V. Then $runs(A_{new}, M, \phi)$ denotes the set of all concrete trajectories π for which there is an assignment ϕ' to V_{new} so that $(\phi \cup \phi')(\pi^{A_{new}})$ is an abstraction of π. Since for all concrete trajectories π, $[(\phi \cup \phi'), \pi \models A_{new}] = 1 \iff \pi \sqsubseteq$

$(\phi \cup \phi')(\pi^{A_{new}})$, we get that $runs(A_{new}, M, \phi)$ are exactly those π for which there is ϕ' so that $[(\phi \cup \phi'), \pi \models A_{new}] = 1$.

The reason the trajectories in $runs(A_{new}, M, \phi)$ are defined with respect to a single extension to the assignment ϕ rather than all extensions to ϕ is that we are interested in the set of all concrete trajectories that satisfy $\phi(A_{new})$ with the truth value 1. Since every trajectory $\pi \in runs(A_{new}, M, \phi)$ is concrete, it can satisfy $\phi(A_{new})$ with the truth value 1 only with respect to a single assignment to V_{new}. The fact that there are other assignments to V_{new} for which π does not satisfy $\phi(A_{new})$ with the truth value 1 is not a concern, since the truth value of $A_{new} \Longrightarrow C$ is determined only according to the concrete trajectories π and assignments ϕ to $V \cup V_{new}$ so that $[\phi, \pi \models A_{new}] = 1$.

Theorem 4.1
1. *For all assignments ϕ to V, $runs(A, M, \phi) = runs(A_{new}, M, \phi)$.*
2. *If $[\pi^{A_{new}} \models C] = 1$ then for all ϕ it holds that $\forall \pi \in runs(A, M, \phi) : [\phi, \pi \models C] = 1$.*
3. *If there is ϕ' to V_{new} and $\pi \in runs(A_{new}, M, \phi \cup \phi')$ so that $[(\phi \cup \phi'), \pi \models A_{new}] = 1$ but $[(\phi \cup \phi'), \pi \models C] = 0$ then $\pi \in runs(A, M, \phi)$ and $[\phi, \pi \models A] = 1$ and $[\phi, \pi \models C] = 0$.*

Theorem 4.1 implies that if $A_{new} \Longrightarrow C$ holds on all concrete trajectories of M, then so does $A \Longrightarrow C$. Moreover, if $A_{new} \Longrightarrow C$ yields a concrete counterexample ce, then ce is also a concrete counterexample w.r.t $A \Longrightarrow C$. The proof of Theorem 4.1 can be found in [29].

5. Selecting Inputs for Refinement

After choosing our refinement methodology, we need to describe how exactly the refinement process is performed. We assume that $[\pi^A \models C] = X$, and thus automatic refinement is activated. Our goal is to add a small number of constraints to A forcing inputs to the value of fresh symbolic variables, while eliminating as many assignments ϕ as possible so that $[\phi, \pi^A \models C] = X$. The refinement process is incremental - inputs (in, t) that are switched from X to a fresh symbolic variable will not be reduced to X in subsequent iterations.

5.1. Choosing Our Refinement Goal

Assume that $[\pi^A \models C] = X$, and the symbolic incomplete trace is generated. This trace contains all assignments ϕ for which $[\phi, \pi^A \models C] = X$. For each such assignment ϕ, the trajectory $\phi(\pi^A)$ is called an ***incomplete trajectory***. In addition, this trace may contain multiple nodes that are required by C to a definite value (either 0 or 1) for some assignment ϕ, but equal X. We refer to such nodes as ***undecided nodes***. We want to keep the number of added constraints small. Therefore, we choose to eliminate one undecided node (n, t) in each refinement iteration, since different nodes may depend on different inputs. Our motivation for eliminating only part of the undecided nodes is that while it is not sufficient for verification it might be sufficient for falsification. This is because an eliminated X value may be replaced in the next iteration with a definite value that contradicts the required value (a counterexample).

Algorithm 1 EliminateIrrelevantPIs((n,t))

sinks_relevant $\leftarrow \bigvee_{(m,t')\in out(n,t)}$ relevant$_{m,t'}$
relevant$_{n,t}$ $\leftarrow$ sinks_relevant $\wedge \neg f^0_{n,t} \wedge \neg f^1_{n,t}$

We suggest to choose an undecided node (n,t) with a minimal number of inputs in its BCOI. Out of those, we choose a node with a minimal number of nodes in its BCOI. Our experimental results support this choice. The chosen undecided node is our ***refinement goal*** and is denoted $(root,tt)$. We also choose to eliminate at once all incomplete trajectories in which $(root,tt)$ is undecided. These trajectories are likely to be eliminated by similar sets of inputs. Thus, by considering them all at once we can considerably reduce the number of refinement iterations, without adding too many variables.

The Boolean expression $(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt})) \wedge nbot$ represents the set of all ϕ for which $(root,tt)$ is undecided in $\phi(\pi^A)$. Our goal is to add a small number of constraints to A so that $(root,tt)$ will not be X whenever $(g^1_{root,tt} \vee g^0_{root,tt})$ holds.

5.2. Eliminating Irrelevant Inputs

Once we have a refinement goal $(root,tt)$, we need to choose inputs (in,t) for which constraints will be added to A. Naturally, only inputs in the BCOI of $(root,tt)$ are considered, but some of these inputs can be safely disregarded.

Consider an input (in,t), an assignment ϕ to V and the defining trajectory π^A. We say that (in,t) is ***relevant*** to $(root,tt)$ under ϕ, if there is a path of nodes P from (in,t) to $(root,tt)$ in (the graph of) M, so that for all nodes (n,t') in P, $\phi(\pi^A)(t')(n) = X$. (in,t) is ***relevant*** to $(root,tt)$ if there exists ϕ so that (in,t) is relevant to $(root,tt)$ under ϕ.

For each (in,t), we compute the set of assignments to V for which (in,t) is relevant to $(root,tt)$. The computation is performed recursively starting from $(root,tt)$. $(root,tt)$ is relevant when it is X and is required to have a definite value:

$$(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt})) \wedge nbot.$$

A source node (n,t) of $(root,tt)$ is relevant whenever $(root,tt)$ is relevant and (n,t) equals X. Let $out(n,t)$ return the sink nodes of (n,t) that are in the BCOI of $(root,tt)$. Proceeding recursively as described in Algorithm 1, we compute for each (in,t) the set of assignments in which it is relevant to $(root,tt)$.

For all ϕ that are not in relevant$_{in,t}$, changing (in,t) from X to 0 or to 1 in $\phi(\pi^A)$ can never change the value of $(root,tt)$ in $\phi(\pi^A)$ from X to 0 or to 1. To see why this is true, note that if ϕ is not in relevant$_{in,t}$ it means that there is at least one node (n',t') on a path in M from (in,t) to $(root,tt)$ whose value under ϕ is definite (0 or 1). Since all nodes in M represent monotonic functions, changing the value of (in,t) in ϕ from X to 0 or 1 will not change the value of (n',t') and therefore will not change the value of $(root,tt)$.

Consequently, if (in,t) is chosen for refinement, we can optimize the refinement by associating (n,t) with a fresh symbolic variable only when relevant$_{in,t}$ holds. This can be done by adding the following constraint to the antecedent:

$$\text{relevant}_{in,t} \rightarrow \mathbf{N^t}(in \text{ is } v_{in,t}).$$

If (in, t) is chosen in a subsequent iteration for refinement of a new refinement goal $(root', tt')$, then the previous constraint is extended by disjunction to include the condition under which (in, t) is relevant to $(root', tt')$. Theorem 4.1 holds also for the optimized refinement. Let PI be the set of inputs of M. The set of all inputs that are relevant to $(root, tt)$ is

$$PI_{(root,tt)} \;=\; \{(in, t) \mid in \in PI \wedge \text{relevant}_{in,t} \not\equiv 0\}.$$

Adding constraints to A for all relevant inputs (in, t) will result in a refined antecedent A_{new}. In the defining trajectory of M and A_{new}, it is guaranteed that $(root, tt)$ will not be undecided. Note that $PI_{(root,tt)}$ is sufficient but not minimal for elimination of all undesired X values from $(root, tt)$. Namely, adding constraints for all inputs in $PI_{(root,tt)}$ will guarantee the elimination of all cases in which $(root, tt)$ is undecided. However, adding constraints for only a subset of $PI_{(root,tt)}$ may still eliminate all such cases.

The set $PI_{(root,tt)}$ may be valuable to the user even if automatic refinement does not take place, since it excludes inputs that are in the BCOI of $(root, tt)$ but will not change the verification results w.r.t $(root, tt)$.

5.3. Heuristics for Selection of Important Inputs

If we add constraints to A for all inputs $(in, t) \in PI_{(root,tt)}$, then we are guaranteed to eliminate all cases in which $(root, tt)$ was equal to X while it was required to have a definite value. However, such a refinement may add many symbolic variables to A, thus significantly increase the complexity of the computation of the defining trajectory. We can reduce the number of added variables at the cost of not guaranteeing the elimination of all undesired X values from $(root, tt)$, by choosing only a subset of $PI_{(root,tt)}$ for refinement. As mentioned before, a 1 or a 0 truth value may still be reached even without adding constraints for all relevant inputs.

We apply the following heuristics in order to select a subset of $PI_{(root,tt)}$ for refinement. Each node (n, t) selects a subset of $PI_{(root,tt)}$ as candidates for refinement, held in $\text{candidates}_{n,t}$. The final set of inputs for refinement is selected out of $\text{candidates}_{root,tt}$. PI denotes the set of inputs (in, t) of M. Each input in $PI_{(root,tt)}$ selects itself as a candidate. Other inputs have no candidates for refinement. Let $\text{out}(n, t)$ return the sink nodes of (n, t) that are in the BCOI of $(root, tt)$, and let $\text{degin}(n, t)$ return the number of source nodes of (n, t) that are in the BCOI of $(root, tt)$. Given a node (n, t), $\text{sourceCand}_{n,t}$ denotes the sets of candidates of the source nodes of (n, t), excluding the source nodes that do not have candidates. The candidates of (n, t) are determined according to the following conditions:

1. If there are candidate inputs that appear in all sets of $sourceCand_{n,t}$, then they are the candidates of (n, t).
2. Otherwise, if (n, t) has source nodes that can be classified as control and data, then the candidates of (n, t) are the union of the candidates of its control source nodes, if this union is not empty. For example, a latch has one data source node and at least one control source node - its clock. The identity of control source nodes is automatically extracted from the netlist representation of the circuit.

3. If none of the above holds, then the candidates of (n,t) are the inputs with the largest number of occurrences in $sourceCand_{n,t}$.

We prefer to refine inputs that are candidates of most source nodes along paths from the inputs to the refinement goal, i.e., influence the refinement goal over several paths. The logic behind this heuristic is that an input that has many paths to the refinement goal is more likely to be essential to determine the value of the refinement goal than an input that has less paths to the refinement goal.

We prefer to refine inputs that affect control before those that affect data since the value of control inputs has usually more effect on the verification result. Moreover, the control inputs determine when data is sampled. Therefore, if the value of a data input is required for verification, it can be restricted according to the value of previously refined control inputs. In the final set of candidates, sets of nodes that are entries of the same vector are treated as one candidate. Since the heuristics did not prefer one entry of the vector over the other, then probably only their joint value can change the verification result. Additional heuristics choose a fixed number of l candidates out of the final set.

6. Detecting Vacuity and Spurious Counterexamples

In this section we raise the problem of hidden vacuity and spurious counterexamples that may occur in STE. This problem was never addressed before in the context of STE.

In STE, the antecedent A functions both as determining the level of the abstraction of M, and as determining the trajectories of M on which C is required to hold. An important point is that the constraints imposed by A are applied (using the $\sqcap$ operator) to *abstract* trajectories of M. If for some node (n,t) and assignment ϕ to V, there is a contradiction between $\phi(\sigma^A)(t)(n)$ and the value propagated through M to (n,t), then $\phi(\pi^A)(t)(n) = \bot$, indicating that there is no concrete trajectory π so that $[\phi, \pi \models A] = 1$.

In this section we point out that due to the abstraction in STE, it is possible that for some assignment ϕ to V, there are no concrete trajectories π so that $[\phi, \pi \models A] = 1$, but still $\phi(\pi^A)$ does not contain $\bot$ values. This is due to the fact that an abstract trajectory may represent more concrete trajectories than the ones that actually exist in M. Consequently, it is possible to get $[\phi, \pi^A \models C] \in \{1, 0\}$ without any indication that this result is vacuous, i.e., for all concrete trajectories π, $[\phi, \pi \models A] = 0$. Note that this problem may only happen if A contains constraints on internal nodes of M. Given a constraint a on an input, there always exists a concrete trajectory that satisfies a (unless a itself is a contradiction, which can be easily detected). This problem exists also in STE implementations that do not compute π^A, such as [24].

Example 3 *We return to Example 1 from Section 3. Note that the defining trajectory π^A does not contain $\bot$. In addition, $[\pi^A \models C] = 0$ due to the assignments to V in which $v_1 = 0$. However, A never holds on concrete trajectories of M when $v_1 = 0$, since N3 at time 0 will not be equal to 1. Thus, the counterexample is spurious, but we have no indication of this fact. The problem occurs when calculating the value of (N3,0) by computing $X \sqcap 1 = 1$. If A had contained a constraint on the value of In2 at time 0, say (In2 is v_2), then the value of (N3,0) in π^A would have been $(v_1 \wedge v_2) \sqcap 1 = (v_1 \wedge v_2 ? 1 : \bot)$, indicating that for all assignments in which $v_1 = 0$ or $v_2 = 0$, π^A does not correspond to any concrete trajectory of M.*

Vacuity may also occur if for some ϕ to V, C under ϕ imposes no requirements. This is due to constraints of the form $p \rightarrow f$ where $\phi(p)$ is 0.

An STE assertion $A \Longrightarrow C$ is ***vacuous*** in M if for all concrete trajectories π of M and assignments ϕ to V, either $[\phi, \pi \models A] = 0$, or C under ϕ imposes no requirements. This definition is compatible with the definition in [5] for ACTL.

We say that $A \Longrightarrow C$ ***passes vacuously*** on M if $A \Longrightarrow C$ is vacuous in M and $[\pi^A \models C] \in \{\bot, 1\}$. A counterexample π is ***spurious*** if there is no concrete trajectory π_c of M so that $\pi_c \sqsubseteq \pi$. Given π^A, the symbolic counterexample ce is ***spurious*** if for all assignments ϕ to V in ce, $\phi(\pi^A)$ is spurious. We believe that this definition is more appropriate than a definition in which ce is spurious if there exists ϕ that satisfies ce and $\phi(\pi^A)$ is spurious. The reason is that the existence of at least one non-spurious counterexample represented by ce is more interesting than the question whether each counterexample represented by ce is spurious or not.

We say that $A \Longrightarrow C$ ***fails vacuously*** on M if $[\pi^A \models C] = 0$ and ce is spurious.

As explained before, vacuity detection is required only when A constrains internal nodes. It is performed only if $[\pi^A \models C] \in \{0, 1\}$ (if $[\pi^A \models C] = \bot$ then surely $A \Longrightarrow C$ passes vacuously). In order to detect non-vacuous results in STE, we need to check whether there exists an assignment ϕ to V and a concrete trajectory π of M so that C under ϕ imposes some requirement and $[\phi, \pi \models A] = 1$. In case the original STE result is fail, namely, $[\pi^A \models C] = 0$, π should also constitute a counterexample for $A \Longrightarrow C$. That is, we require that $[\phi, \pi \models C] = 0$.

We propose two different algorithms for vacuity detection. The first algorithm uses Bounded Model Checking (BMC) [6] and runs on the concrete model. The second algorithm uses STE and requires automatic refinement. The algorithm that uses STE takes advantage of the abstraction in STE, as opposed to the first algorithm which runs on the concrete model. In case non-vacuity is detected, the trajectory produced by the second algorithm (which constitutes either a witness or a counterexample) may not be concrete. However, it is guaranteed that there exists a concrete trajectory of which the produced trajectory is an abstraction. The drawback of the algorithm that uses STE, however, is that it requires automatic refinement.

6.1. Vacuity Detection using Bounded Model Checking

Since A can be expressed as an LTL formula, we can translate A and M into a Bounded Model Checking (BMC) problem. The bound of the BMC problem is determined by the depth of A. Note that in this BMC problem we search for a satisfying assignment for A, not for its negation. Additional constraints should be added to the BMC formula in order to fulfill the additional requirements on the concrete trajectory.

For detection of vacuous pass, the BMC formula is constrained in the following way: Recall that $(g^1_{n,t}, g^0_{n,t})$ denotes the dual rail representation of the requirement on the node (n, t) in C. The Boolean expression $g^1_{n,t} \vee g^0_{n,t}$ represents all assignments ϕ to V under which C imposes a requirement on (n, t). Thus, $\bigvee_{(n,t)\in C} g^1_{n,t} \vee g^0_{n,t}$ represents all assignments ϕ to V under which C imposes some requirement. This expression is added as an additional constraint to the BMC formula. If BMC finds a satisfying assignment to the resulting formula, then the assignment of BMC to the nodes in M constitutes a witness indicating that $A \Longrightarrow C$ passed non-vacuously. Otherwise, we conclude that $A \Longrightarrow C$ passed vacuously.

For detection of vacuous fail, the BMC formula is constrained by conjunction with the (symbolic) counterexample ce. For STE implementations that compute π^A, $ce = \bigvee_{(n,t)\in C}((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))$. There is no need to add the $nbot$ constraint that guarantees that none of the nodes equals $\perp$, since the BMC formula runs on the concrete model, and thus the domain of the nodes in the BMC formula is Boolean. If BMC finds a satisfying assignment to the resulting formula, the assignment of BMC to the nodes in M constitutes a concrete counterexample for $A \Longrightarrow C$. Otherwise, we conclude that $A \Longrightarrow C$ failed vacuously.

Vacuity detection using BMC is an easier problem than solving the original STE assertion $A \Longrightarrow C$ using BMC. The BMC formula for $A \Longrightarrow C$ contains the following constraints on the values of nodes:

- The constraints of A.
- The constraints of M on nodes appearing in A.
- The constraints of M on nodes appearing in C.
- A constraint on the values of the nodes appearing in C that guarantees that at least one of the requirements in C does not hold.

On the other hand, the BMC formula for vacuity detection contains only the first two types of constraints on the values of nodes. Therefore, for vacuity detection using BMC, only the BCOI of the nodes in A is required, whereas for solving the original STE assertion $A \Longrightarrow C$ using BMC, both the BCOI of the nodes appearing in A and the BCOI of the nodes appearing in C are required.

6.2. Vacuity Detection using Symbolic Trajectory Evaluation

For vacuity detection using STE, the first step is to split A into two different TEL formulas: A^{in} is a TEL formula that contains exactly all the constraints of A on inputs, and A^{out} is a TEL formula that contains exactly all the constraints of A on internal nodes. If there exists an assignment ϕ to V so that $[\phi, \pi^{A^{in}} \models A^{out}] = 1$, then we can conclude that there exists a concrete trajectory of M that satisfies A. Note that since A^{in} does not contain constraints on internal nodes, it is guaranteed that no hidden vacuity occurs. However, it is also necessary to guarantee that in case $[\pi^A \models C] = 1$, C under ϕ imposes some requirement, and in case $[\pi^A \models C] = 0$, then $\phi(\pi^{A^{in}})$ should constitute a counterexample. Namely, $\phi \wedge ce \not\equiv 0$, where ce is the symbolic counterexample.

If we cannot find such an assignment ϕ, this does not necessarily mean that the result of $A \Longrightarrow C$ is vacuous: if there are assignments ϕ to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = X$, then the trajectory $\phi(\pi^{A^{in}})$ is potentially an abstraction of a witness or a concrete counterexample for $A \Longrightarrow C$. However, it is too abstract in order to determine whether or not A^{out} holds on it. If we refine A^{in} to a new antecedent as described in Section 4, then it is possible that the new antecedent will yield more refined trajectories that contain enough information to determine whether they indeed represent a witness or concrete counterexample.

Algorithm 2 describes vacuity detection using STE. It receives the original antecedent A and consequent C. In case $[\pi^A \models C] = 0$, it also receives the symbolic counterexample ce. inputConstraints is a function that receives a TEL formula A and returns a new TEL formula that consists of the constraints of A on inputs. Similarly, internalConstraints returns a new TEL formula that consists of the constraints of A on

internal nodes. Note that since A^{in} does not contain constraints on internal nodes, then $\pi^{A^{in}}$ does not contain $\perp$ values, and thus we can assume that $f^1_{n,t} \wedge f^0_{n,t}$ never holds. By abuse of notation, $f^1_{n,t}$ and $f^0_{n,t}$ are here the dual rail representation of a node (n,t) in $\pi^{A^{in}}$. Similarly, we use $g^1_{n,t}$ and $g^0_{n,t}$ for the dual rail representation of a node (n,t) in the defining sequence of either C or A^{out}, according to the context.

Algorithm 2 STEVacuityDetection(A, C, ce)

1: $A^{in} \leftarrow$ inputConstraints(A)
2: $A^{out} \leftarrow$ internalConstraints(A)
3: $\Phi \leftarrow \bigwedge_{(n,t)\in A^{out}}((g^1_{n,t} \wedge f^1_{n,t}) \vee (g^0_{n,t} \wedge f^0_{n,t}))$
   $\{\Phi$ represents all assignments to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = 1\}$
4: **if** $[\pi^A \models C] = 1 \wedge ((\bigvee_{(n,t)\in C}(g^1_{n,t} \vee g^0_{n,t})) \wedge \Phi) \not\equiv 0$ **then**
5: return non-vacuous
6: **else if** $[\pi^A \models C] = 0 \wedge ((\Phi \wedge ce) \not\equiv 0)$ **then**
7: return non-vacuous
8: **end if**
9: **if** $\exists \phi : [\phi, \pi^{A^{in}} \models A^{out}] = X$ **then**
10: $A^{in} \leftarrow$ refine(A^{in})
11: goto 3
12: **else**
13: return vacuous
14: **end if**

The algorithm computes the set Φ, which is the set of all assignments to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = 1$. Lines 4 and 6 check whether there exists a suitable assignment ϕ in Φ that corresponds to a witness or to a counterexample. If such a ϕ exists, then the result is non-vacuous. If no such ϕ exists, then if there exist assignments for which the truth value of A^{out} on $\pi^{A^{in}}$ is X, then A^{in} is refined and Φ is recomputed. Otherwise, the result is vacuous.

Note that in case $[\pi^A \models C] = 0$, we check whether Φ contains an assignment that constitutes a counterexample by checking that the intersection between Φ and the symbolic counterexample ce produced for $[\pi^A \models C]$ is not empty. However, as a result of the refinement, Φ may contain new variables that represent new constraints of the antecedent that were not taken into account when computing ce. The reason that checking whether $(\Phi \wedge ce) \not\equiv 0$ still returns a valid result is as follows. By construction, we know that for all assignments $\phi \in \Phi$, $[\phi, \pi^{A^{in}} \models A^{out}] = 1$. Since $[\phi, \pi^{A^{in}} \models A^{in}] = 1$, we get that $[\phi, \pi^{A^{in}} \models A^{in} \cup A^{out}] = 1$, where $A^{in} \cup A^{out}$ is the TEL formula that contains exactly all the constraints in A^{in} and A^{out}. Since $[\phi, \pi^{A^{in}} \models A^{out}] = 1$, we get that $\phi(\pi^{A^{in}})$ does not contain $\perp$ values. Therefore, for all nodes (n,t) so that $\phi(\pi^A)(t)(n) = b, b \in \{0,1\}$ it holds that $\phi(\pi^{A^{in}})(t)(n) = b$. Thus, for all $\phi' \in ce$, ϕ' is a counterexample also with respect to the antecedent $A^{in} \cup A^{out}$.

Besides the need for refinement, an additional drawback of Algorithm 2 in comparison with vacuity detection using BMC, is that it attempts to solve a much harder problem - it computes a set of trajectories that constitute witnesses or concrete counterexamples, whereas in vacuity detection using BMC only one such trajectory is produced - a satisfying assignment to the SAT formula. This is in analogy to using STE versus

using BMC for model checking - STE finds the set of all counterexamples for $A \Longrightarrow C$, while BMC finds only one counterexample. However, the advantage of Algorithm 2 is that it exploits the abstraction in STE, whereas vacuity detection using BMC runs on the concrete model.

In [29], vacuity detection for SAT-based STE is presented as well.

6.3. Preprocessing for Vacuity Detection

There are some cases in which even if there exist constraints in A on internal nodes, vacuity detection can be avoided by a preliminary analysis based on the following observation: hidden vacuity may only occur if for some assignment ϕ to V, an internal node (n,t) is constrained by A to either 0, or 1, but its value as calculated according to the values of its source nodes is X. We call such a node (n,t) a *problematic node*. For example, in Example 1 from Section 3, the value of (N3,0) as calculated according to its source nodes is X, and it is constrained by A to 1.

In order to avoid unnecessary vacuity detection, we suggest to detect all problematic nodes as follows. Let $int(A)$ denote all internal nodes (n,t) on which there exists a constraint in A. Let $h^1_{n,t}$ and $h^0_{n,t}$ denote the dual rail representation of the node (n,t) in σ^A. Let $m^1_{n,t}$ and $m^0_{n,t}$ denote the dual rail representation of the value of (n,t) as calculated according to the values of its source nodes in π^A. Then the Boolean expression $\bigvee_{(n,t)\in int(A)}((h^0_{n,t} \vee h^1_{n,t}) \wedge \neg m^1_{n,t} \wedge \neg m^0_{n,t})$ represents all assignments to V for which there exists a problematic node (n,t). If this Boolean expression is identical to 0, then no problematic nodes exist and vacuity detection is unnecessary.

7. Experimental Results

We implemented our automatic refinement algorithm **AutoSTE** on top of STE in Intel's FORTE environment [27]. **AutoSTE** receives a circuit M and an STE assertion $A \Longrightarrow C$. When $[\pi^A \models C] = X$, it chooses a refinement goal $(root, tt)$ out of the undecided nodes, as described in Section 5. Next, it computes the set of relevant inputs (in,t). The Heuristics described in Section 5 are applied in order to choose a subset of those inputs. In our experimental results we restrict the number of refined candidates in each iteration to 1. A is changed as described in Section 5 and STE is rerun on the new assertion.

We ran **AutoSTE** on two different circuits, which are challenging for Model Checking: the Content Addressable Memory (CAM) from Intel's GSTE tutorial, and IBM's Calculator 2 design [31]. The latter has a complex specification.Therefore, it constitutes a good example for the benefit the user can gain from automatic refinement in STE. All runs were performed on a 3.2 GHz Pentium 4 computer with 4 GB memory.

A detailed description of the experiments can be found in [29].

8. Conclusions and Future Work

This work describes a first attempt at automatic refinement of STE assertions. We have developed an automatic refinement technique which is based on heuristics. The refined assertion preserves the semantics of the original assertion. We have implemented our

automatic refinement in the framework of Forte, and ran it on two nontrivial circuits of dissimilar functionality. The experimental results show success in automatic verification of several nontrivial assertions.

Another important contribution of our work is identifying that STE results may hide vacuity. This possibility was never raised before. We formally defined STE vacuity and proposed two methods for vacuity detection.

Additional work is needed in order to further evaluate the suggested automatic refinement on industrial-size examples of different functionality. Such an evaluation is very likely to result in new heuristics. A preliminary work has recently been done for STE in [12] and for GSTE in [11].

We would also like to implement our suggested vacuity detection algorithms and compare their performance. In addition, we would like to develop an automatic refinement techniques to SAT based STE [33,24,17], and integrate SAT based refinement techniques [22,10].

References

[1] S. Adams, M. Bjork, T. Melham, and C. Seger. Automatic abstraction in symbolic trajectory evaluation. In *8th International Conference on Formal methods in Computer-Aided Design (FMCAD'07)*, Austin, Texas, November 2007.

[2] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, Boulder, CO, USA, July 2003. Springer.

[3] Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *CAV'02: Proceedings of Conference on Computer-Aided Verification*, 2002.

[4] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 596–602. ACM Press, 1994.

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV'97: Proceedings of Conference on Computer-Aided Verification*, 1997.

[6] Armin Biere, Allesandro Cimatti, Edmond M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99: Conference on tools and algorithms for the construction and analysis of systems*, 1999.

[7] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*, pages 274–287, 1999.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986.

[9] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. Regular vacuity. In *13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, Lecture Notes in Computer Science, Saarbrucken, Germany, October 2005. Springer-Verlag.

[10] Pankaj Chauhan, Edmond M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *FMCAD'02: Proceedings of the Forth International Conference on Formal Methods in Computer-Aided Design*, 2002.

[11] Y. Chen, Y. He, F. Xie, and J. Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *8th International Conference on Formal methods in Computer-Aided Design (FMCAD'07)*, Austin, Texas, November 2007.

[12] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, LNCS, Budapest, March-April 2008. Springer.

[13] Ching-Tsun Chou. The mathematical foundation of symbolic trajectory evaluation. In *CAV'99: Proceedings of Conference on Computer-Aided Verification*, 1999.

[14] Edmond M. Clarke, Orna Grumberg, S. Jha, Y. Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV'00: Proceedings of Conference on Computer-Aided Verification*, 2000.

[15] Edmond M. Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *CAV'02: Proceedings of Conference on Computer-Aided Verification*, 2002.

[16] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-valued circuit SAT for STE with automatic refinement. In *Fifth International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *LNCS*, Tokyo, Japan, October 2007.

[17] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-Valued Circuit SAT for STE with Automatic Refinement . In *ATVA '07*, 2007.

[18] Scott Hazelhurst and Carl-Johan H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic journal of IGPL*, 7(3), 1999.

[19] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *CHARME'99: Conference on Correct Hardware Design and Verification Methods*, pages 82–96, 1999.

[20] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4(2), 2003.

[21] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[22] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS'03: Conference on tools and algorithms for the construction and analysis of systems*, 2003.

[23] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (FOCS'77)*, 1977.

[24] Jan-Willem Roorda and Koen Claessen. A new SAT-based algorithm for symbolic trajectory evaluation. In *CHARME'05: Proceedings of Correct Hardware Design and Verification Methods*, 2005.

[25] Tom Schubert. High level formal verification of next-generation microprocessors. In *DAC'03: Proceedings of the 40th conference on Design automation*.

[26] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.

[27] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom F. Melham, Mark Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.

[28] Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In *CAV'03: Proceedings of Conference on Computer-Aided Verification*, 2003.

[29] Rachel Tzoref. Automated refinement and vacuity detection for Symbolic Trajectory Evaluation. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2006.

[30] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for Symbolic Trajectory Evaluation. In *18th International Conference on Computer Aided Verification (CAV'06)*, LNCS 4144, Seattle, August 2006.

[31] Bruce Wile, Wolfgang Roesner, and John Goss. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan-Kaufmann, 2005.

[32] James C. Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2001.

[33] Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *DCC*, 2004.

[34] Jin Yang and Amit Goel. GSTE through a case study. In *ICCAD*, 2002.

[35] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In *FMCAD'02: Proceedings of the Forth International Conference on Formal Methods in Computer-Aided Design*, 2002.

[36] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3), 2003.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Automated and Interactive Theorem Proving

John HARRISON
johnh@ichips.intel.com
Intel Corporation JF1-13, 2111 NE 25th Avenue, Hillsboro OR 97124, USA

Abstract. The idea of mechanizing reasoning is an old dream that can be traced at least back to Leibniz. Since about 1950, there has been considerable research on having computers perform logical reasoning, either completely autonomously (automated theorem proving) or in cooperation with a person (interactive theorem proving). Both approaches have achieved notable successes. For example, several open mathematical problems such as the Robbins Conjecture have been settled by automated theorem provers, while interactive provers have been applied to formalization of non-trivial mathematics and the verification of complex computer systems. However, it can be difficult for a newcomer to gain perspective on the field, since it has already fragmented into various special subdisciplines. The aim of these lectures will be to give a broad overview that tries to establish some such perspective. I will cover a range of topics from Boolean satisfiability checking (SAT), several approaches to first-order automated theorem proving, special methods for equations, decision procedures for important special theories, and interactive proof. I will not say much in detail about applications, but will give some suitable references for those interested.

1. Introduction and propositional logic
2. First order logic
3. Algebraic and arithmetical theories
4. Interactive theorem proving
5. Proof-producing decision procedures

Implementations of many algorithms discussed, in (I hope) a fairly simple and pedagogical style, can be found on my Web page in a comprehensive package of logical code:

`http://www.cl.cam.ac.uk/~jrh13/atp/index.html`

0. Preamble

Our main goal is machine assistance with logical decision problems roughly of the form: given a formula p in some formal logical system, is p logically valid / consistent / satisfiable / true? There are two different aspects:

- Theoretical: What is possible in principle?
- Practical: What is possible/useful to use?

The theoretical questions actually lie at the root of much of modern theoretical computer science, at least in its historical development. For example, the development of models of computation such as Turing machines was explicitly aimed at giving a negative answer to the *Entscheidungsproblem* (decision problem) for first order logic. And the famous P = NP question was first framed as a question about satisfiability in propositional logic. On the practical side, machine-assisted theorem proving is important in program verification, as well as various artificial intelligence systems, and sometimes even compilers.

We will mainly focus on subsets of classical first-order logic, since most of the important techniques can be illustrated in this context. But mechanization of proof in other logics such as intutionistic logic, higher-order logic and temporal logic is also important. Many of these use refinements or extensions of methods we will consider here, and you will get more information from other lecturers. In particular, many fragments of temporal logic can be decided automatically in a practical fashion (Clarke, Grumberg, and Peled 1999).

The gap between theory and practice can be quite wide, and the tractability of a problem can depend on the point of view. For example, the decision procedure propositional logic might be regarded as trivial (because it's decidable), or intractable (because it's NP-complete). In practice, it seems to be challenging, but current implementations can handle some surprisingly big problems. And of course, the tractability of a problem depends on how much CPU time you're willing to devote to it; what is tractable for a sophisticated attempt to solve an open problem in mathematics might be intractable for a small prover inside a program verifier.

For the purposes of these lectures, the main prerequisite is just the basic formal syntax of first-order logic and a few items of terminology. You might want to just glance quickly through this and come back to it later when needed. But do try to grasp the important distinction between *validity* in all interpretations and *truth* in a particular one.

0.1. Syntax

Here is a rough translation table from natural language to formal logic. For example, $p \wedge q \Rightarrow p \vee q$ means 'if p and q are true, then either p or q is true'. We assume that 'or' is interpreted inclusively, i.e. $p \vee q$ means 'p or q or both'.

English	Formal logic	Other
false	$\bot$	F
true	$\top$	T
not p	$\neg p$	$-p$, $\sim p$, $\overline{p}$
p and q	$p \wedge q$	$p\&q$, $p \cdot q$
p or q	$p \vee q$	$p \mid q$, p *or* q
p implies q	$p \Rightarrow q$	$p \rightarrow q$, $p \supset q$
p iff q	$p \Leftrightarrow q$	$p \equiv q$, $p \sim q$
For all x, p	$\forall x.\, p$	$(\forall x)(p)$
There exists x such that p	$\exists x.\, p$	$(\exists x)(p)$

Although this is useful as a general guide, we sometimes need to be quite precise about the formal language. We first have a class of *terms*, which are built up from variables (like 'x') and constants (like '1') by applying functions (like '$+$'). For formal simplicity, we can imagine constants as functions that take no arguments, and so regard terms as generated by the following BNF grammar:

$$\begin{aligned} term &\longrightarrow variable \\ &\mid\ function(term,\ldots,term) \end{aligned}$$

We will freely use conventional mathematical notation, writing $x+1$ rather than $+(x,1())$ for example, but we should think of the underlying abstract syntax tree as what we're really manipulating. Likewise we have a BNF grammar for formulas:

$$\begin{aligned} formula &\longrightarrow relation(term,\ldots,term) \\ &\mid\ \bot \\ &\mid\ \top \\ &\mid\ \neg formula \\ &\mid\ formula \wedge formula \\ &\mid\ formula \vee formula \\ &\mid\ formula \Rightarrow formula \\ &\mid\ formula \Leftrightarrow formula \\ &\mid\ \forall variable.\, formula \\ &\mid\ \exists variable.\, formula \end{aligned}$$

The most basic *atomic* formulas of the form $relation(term,\ldots,term)$ are formed by applying a *relation* or *predicate* symbol to terms, for example the '$\leq$' operator to two numbers. From these basic formulas, more complex ones are built up by applying the logical *connectives* like '$\wedge$' and *quantifiers* like '$\forall x.$'.

0.2. Semantics

Formulas of first order logic mean nothing in isolation, but only with respect to an *interpretation* or *model* that tells us how to interpret the constants, functions and relations, and a *valuation* that tells us how to interpret the variables. Formally speaking, an interpretation M consists of:

- A set D (the *domain*), normally assumed nonempty.
- For each n-ary function symbol f, a function $f_M : D^n \to D$.
- For each n-ary relation symbol f, a function $R_M : D^n \to \text{bool}$.

while a valuation is simply a function $v : V \to D$ assigning to each variable an element of the domain. Now, given an interpretation M and a valuation v, each formula p maps to a well-defined truth value, which can be found by interpreting all the constants, functions and relations and applying the valuation to all variables. We will give a more formal description below, but it might be more enlightening just to see a few examples. Take a language with two constants 0 and 1, one binary function '$\cdot$' and one binary relation '$=$', and consider the formula:

$$\forall x.\, \neg(x = 0) \Rightarrow \exists y.\, x \cdot y = 1$$

One natural way to interpret the formulas is with the domain D being $\mathbb{R}$, the set of real numbers, and the constants and functions interpreted in the obvious way. In that case, the formula says that every nonzero real number x has a multiplicative inverse y. It is therefore true in that interpretation. However, it's important to remember that there are many other possible interpretations of the symbols, and these may or may not make the formula come out true. For example, let the domain D be the set $\{0, 1, 2\}$, interpret 0 and 1 in the natural way and '$\cdot$' as multiplication modulo 3:

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

A little calculation shows that the formula also holds in this interpretation, even though it's very different from the original 'intended' model. Yet the slight change of taking as the domain $D = \{0, 1, 2, 3\}$ with multiplication modulo 4 makes the formula false, because the element 2 doesn't have an inverse:

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
2	0	3	2	1

A formula is (logically) *valid* if it is true in *all* interpretations, whatever their (nonempty) domain and however the functions and relations are interpreted. Dually, we say that a formula is *satisfiable* if it has *some interpretation* in which it holds for all valuations. It's important to observe that, in general, logical validity is a much stronger requirement than validity in some particular intended interpretation. Thus, the problems of deciding whether a formula holds in all interpretations and is true in some particular interpretation are, in general, quite different, and one may be much easier or harder than the other.

Since equality is such a central relation in mathematics, we often want to restrict ourselves to *normal* interpretations where the equality symbol is interpreted by equality in the interpretation. One can show that a formula p is satisfiable in a normal interpretation if the formula together with additional 'equality axioms' is satisfiable in any interpretation at all. These equality axioms consist of an assertion that equality is an equivalence relation, i.e. is reflexive, symmetric and transitive:

$$\forall x.\, x = x$$

$$\forall x\, y.\, x = y \Leftrightarrow y = x$$

$$\forall x\, y\, z.\, x = y \wedge y = z \Rightarrow x = z$$

as well as assertions of *congruence* for each n-ary function f in p:

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\, x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

and similarly for each n-ary relation R in p:

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\, x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow R(x_1, \ldots, x_n) \Rightarrow R(y_1, \ldots, y_n)$$

0.3. Explicit implementation

To make the above definitions completely concrete and explicit, here is an implementation in Objective CAML. We start with a few auxiliary functions to perform 'set' operations on lists:

```
open List;;
let insert x l = if mem x l then l else x::l;;

let union l1 l2 = fold_right insert l1 l2;;
let unions l = fold_right union l [];;
let intersect l1 l2 = filter (fun x -> mem x l2) l1;;
let subtract l1 l2 = filter (fun x -> not (mem x l2)) l1;;
let subset l1 l2 = for_all (fun t -> mem t l2) l1;;
let set_eq l1 l2 = subset l1 l2 & subset l2 l1;;
```

and then define the syntax of terms and formulas:

```
type term = Var of string
          | Fn of string * term list;;

type formula = False
             | True
             | Atom of string * term list
             | Not of formula
             | And of formula * formula
             | Or of formula * formula
             | Imp of formula * formula
             | Iff of formula * formula
             | Forall of string * formula
             | Exists of string * formula;;
```

For some purposes, it's important to know the *free variables* $FV(p)$ of a formula p. This can be defined recursively as follows; note that variables are 'bound' by outer quantifiers. Intuitively speaking, it's only the valuation of the free variables that affects the truth-value assigned to a formula, so a *sentence* (= formula with no free variables) is true or false in any particular interpretation, without reference to the valuation.

```
let rec fvt tm =
  match tm with
    Var x -> [x]
  | Fn(f,args) -> unions (map fvt args);;

let rec fv fm =
  match fm with
    False -> []
  | True -> []
  | Atom(p,args) -> unions (map fvt args)
  | Not(p) -> fv p
  | And(p,q) -> union (fv p) (fv q)
  | Or(p,q) -> union (fv p) (fv q)
  | Imp(p,q) -> union (fv p) (fv q)
  | Iff(p,q) -> union (fv p) (fv q)
  | Forall(x,p) -> subtract (fv p) [x]
  | Exists(x,p) -> subtract (fv p) [x];;
```

We now define what an interpretation is. Note that to implement this in a programming language, we're forced to use a finite domain, which we represent as a list. However, the definition of validity below extends mathematically to an arbitrary domain:

```
type ('a)interpretation =
  Interp of ('a)list *
            (string -> ('a)list -> 'a) *
            (string -> ('a)list -> bool);;

let domain(Interp(d,funs,preds)) = d
and func(Interp(d,funs,preds)) = funs
and relation(Interp(d,funs,preds)) = preds;;
```

and here is that definition:

```
let (|->) x a f = fun y -> if y = x then a else f(y);;

let rec termval md v tm =
  match tm with
    Var(x) -> v x
  | Fn(f,args) -> func(md) f (map (termval md v) args);;

let rec holds md v fm =
  match fm with
    False -> false
  | True -> true
  | Atom(r,args) -> relation(md) r (map (termval md v) args)
  | Not(p) -> not(holds md v p)
  | And(p,q) -> (holds md v p) & (holds md v q)
  | Or(p,q) -> (holds md v p) or (holds md v q)
  | Imp(p,q) -> not(holds md v p) or (holds md v q)
  | Iff(p,q) -> (holds md v p = holds md v q)
  | Forall(x,p) -> for_all (fun a -> holds md ((x |-> a) v) p) (domain md)
  | Exists(x,p) -> exists (fun a -> holds md ((x |-> a) v) p) (domain md);;
```

We can even apply it to the various interpretations of our example formula:

```
let rec (--) m n = if n < m then [] else m::(m+1 -- n);;

let mod_interp n =
  let fns f args =
    match (f,args) with
      ("0",[]) -> 0
    | ("1",[]) -> 1 mod n
    | ("*",[x;y]) -> (x * y) mod n
    | _ -> failwith "uninterpreted function"
  and prs p args =
    match (p,args) with
      ("=",[x;y]) -> x = y
    | _ -> failwith "uninterpreted relation" in
  Interp(0 -- (n - 1),fns,prs);;

let p = Forall("x",Imp(Not(Atom("=",[Var "x"; Fn("0",[])])),
                       Exists("y",Atom("=",[Fn("*",[Var "x"; Var "y"]);
                                            Fn("1",[])]))));;
```

for example:

```
# holds (mod_interp 3) (fun x -> failwith "") p;;
- : bool = true
# holds (mod_interp 4) (fun x -> failwith "") p;;
- : bool = false
```

1. Propositional logic

In propositional logic, one only has nullary relation symbols and no quantifiers. A valid propositional formula is often called a *tautology*. At first sight this might seem a somewhat boring subset, but many problems of practical interest can be expressed in it, as we note below.

1.1. Decidability

The problems of propositional validity (= tautology) and satisfiability testing are both decidable. Moreover, they are interreducible, since p is valid precisely if $\neg p$ is *not* satisfiable, though in computer science at least it's traditional to emphasize the satisfiability problem 'SAT'. Indeed, it's not hard to see that the problems are decidable. The truth of a propositional formula is determined by the assignments of truth-values to its relation symbols. A formula only involves finitely many of these, and so the number of possible interpretations is finite and we can try them all. It's so easy that we can present explicit code. First we have a function to return all the relations in a formula:

```
let rec preds fm =
  match fm with
    True | False -> []
  | Atom(a,[]) -> [a]
  | Not p -> preds p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (preds p) (preds q)
  | _ -> failwith "preds: non-propositional formula";;
```

and to convert a function from names to booleans into a proper interpretation:

```
let interpretation v =
  Interp([false;true],
         (fun f a -> failwith "no function symbols allowed"),
         (fun p a -> if a = [] then v(p) else failwith "non-nullary"));;
```

Now the following auxiliary function tries a formula `p` on all valuations that can be constructed from the initial one `v` by assigning to the atoms in `ats`:

```
let rec alltrue ats v p =
  match ats with
    a::oas -> alltrue oas ((a |-> true) v) p & alltrue oas ((a |-> false) v) p
  | [] -> holds (interpretation v) (fun x -> failwith "") p;;
```

and so we can get functions for tautology and satisfiability checking, e.g.

```
let tautology p = alltrue (preds p) (fun _ -> failwith "") p;;
let satisfiable p = not(tautology(Not p));;
```

For instance, here we try $p \Rightarrow p$, $p \Rightarrow q$ and $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$:

```
# tautology(Imp(Atom("p",[]),Atom("p",[])));;
- : bool = true
# tautology(Imp(Atom("p",[]),Atom("q",[])));;
- : bool = false
# tautology(Imp(Imp(Atom("p",[]),Atom("q",[])),
                Imp(Not(Atom("q",[])),Not(Atom("p",[])))));;
- : bool = true
```

From a theoretical point of view then, SAT is easy. But the situation looks very different if we think about the computational complexity. The above algorithm essentially takes 2^n steps if there are n atomic propositions. This is hardly practical even for a hundred atomic propositions, let alone thousands or millions. And in fact, no subexponential

algorithm for SAT is known. On the contrary, the extensive theory of NP-completeness (Cook 1971) implies that *if* there is a polynomial-time algorithm for SAT, then there are also polynomial-time algorithms for many apparently very difficult combinatorial problems, which fuels the belief that the existence of such an algorithm is unlikely. Still, there are algorithms that often perform quite well on many real problems. These still usually involve true/false case-splitting of variables, but in conjunction with more intelligent simplification.

1.2. The Davis-Putnam method

Most high-performance SAT checkers are based on the venerable Davis-Putnam algorithm (Davis and Putnam 1960), or more accurately on the 'DPLL' algorithm, a later improvement (Davis, Logemann, and Loveland 1962).

The starting-point is to put the formula to be tested for satisfiability in 'conjunctive normal form'. A formula is said to be in conjunctive normal form (CNF) when it is an 'and of ors', i.e. of the form:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

with each C_i in turn of the form:

$$l_{i1} \vee l_{i2} \vee \cdots \vee l_{im_i}$$

and all the l_{ij}'s *literals*, i.e. primitive propositions or their negations. The individual conjuncts C_i of a CNF form are often called *clauses*. We usually consider these as sets, since both conjunction and disjunction are associative, commutative and idempotent, so it makes sense to talk of $\bot$ as the empty clause. If C_i consists of one literal, it is called a *unit clause*. Dually, disjunctive normal form (DNF) reverses the role of the 'and's and 'or's. These special forms are analogous to 'fully factorized' and 'fully expanded' in ordinary algebra — think of $(x+1)(x+2)(x+3)$ as CNF and $x^3+6x^2+11x+6$ as DNF. Again by analogy with algebra, we can always translate a formula into CNF by repeatedly rewriting with equivalences like:

$$\begin{aligned}
\neg(\neg p) &\Leftrightarrow p \\
\neg(p \wedge q) &\Leftrightarrow \neg p \vee \neg q \\
\neg(p \vee q) &\Leftrightarrow \neg p \wedge \neg q \\
p \vee (q \wedge r) &\Leftrightarrow (p \vee q) \wedge (p \vee r) \\
(p \wedge q) \vee r &\Leftrightarrow (p \vee r) \wedge (q \vee r)
\end{aligned}$$

However, this in itself can cause the formula to blow up exponentially before we even get to the main algorithm, which is hardly a good start. One can do better by introducing new variables to denote subformulas, and putting the resulting list of equivalences into CNF — so-called *definitional CNF*. It's not hard to see that this preserves satisfiability. For example, we start with the formula:

$$(p \vee (q \wedge \neg r)) \wedge s$$

introduce new variables for subformulas:

$$\begin{array}{l}(p_1 \Leftrightarrow q \wedge \neg r) \wedge \\ (p_2 \Leftrightarrow p \vee p_1) \wedge \\ (p_3 \Leftrightarrow p_2 \wedge s) \wedge \\ p_3\end{array}$$

then transform to CNF:

$$\begin{array}{l}(\neg p_1 \vee q) \wedge (\neg p_1 \vee \neg r) \wedge (p_1 \vee \neg q \vee r) \wedge \\ (\neg p_2 \vee p \vee p_1) \wedge (p_2 \vee \neg p) \wedge (p_2 \vee \neg p_1) \wedge \\ (\neg p_3 \vee p_2) \wedge (\neg p_3 \vee s) \wedge (p_3 \vee \neg p_2 \vee \neg s) \wedge \\ p_3\end{array}$$

The DPLL algorithm is based on the following satisfiability-preserving transformations:

I The 1-literal rule: if a unit clause p appears, remove $\neg p$ from other clauses and remove all clauses including p.

II The affirmative-negative rule: if p occurs *only* negated, or *only* unnegated, delete all clauses involving p.

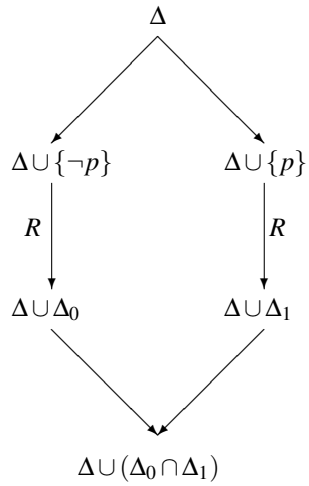
III Case-splitting: consider the two separate problems by adding p and $\neg p$ as new unit clauses.

If you get the empty set of clauses, the formula is satisfiable; if you get an empty *clause*, it is unsatisfiable. Since the first two rules make the problem simpler, one only applies the case-splitting rule when no other progress is possible. In the worst case, many case-splits are necessary and we get exponential behaviour. But in practice it works quite well.

1.3. Industrial-strength SAT checkers

The above simple-minded sketch of the DPLL algorithm leaves plenty of room for improvement. The choice of case-splitting variable is often critical, the formulas can be represented in a way that allows for efficient implementation, and the kind of backtracking that arises from case splits can be made more efficient via 'intelligent backjumping' and 'conflict clauses'. Two highly efficient DPLL-based theorem provers are Chaff (Moskewicz, Madigan, Zhao, Zhang, and Malik 2001) and BerkMin (Goldberg and Novikov 2002).

Another interesting technique that is used in the tools from Prover Technology (`www.prover.com`), as well as the experimental system Heerhugo (Groote 2000), is Stålmarck's dilemma rule (Stålmarck and Säflund 1990). This involves using case-splits in a non-nested fashion, accumulating common information from both sides of a case split and feeding it back:



In some cases, this works out much better than the usual DPLL algorithm. For a nice introduction, see Sheeran and Stålmarck (2000). Note that this method is covered by patents (Stålmarck 1994).

1.4. Applications of propositional logic

There is a close correspondence between propositional logic and digital circuit design. At a particular time-step, we can regard each internal or external wire in a (binary) digital computer as having a Boolean value, 'false' for 0 and 'true' for 1, and think of each circuit element as a Boolean function, operating on the values on its input wire(s) to produce a value at its output wire. The most basic building-blocks of computers used by digital designers, so-called *logic gates*, correspond closely to the usual logical connectives. For example an 'AND gate' is a circuit element with two inputs and one output whose output wire will be high (true) precisely if both the input wires are high, and so it corresponds exactly in behaviour to the 'and' ('$\wedge$') connective. Similarly a 'NOT gate' (or *inverter*) has one input wire and one output wire, which is high when the input is low and low when the input is high; hence it corresponds to the 'not' connective ('$\neg$'). Thus, there is a close correspondence between digital circuits and formulas which can be sloganized as follows:

Digital design	Propositional Logic
circuit	formula
logic gate	propositional connective
input wire	atom
internal wire	subexpression
voltage level	truth value

An important issue in circuit design is proving that two circuits have the same function, i.e. give identical results on all inputs. This arises, for instance, if a designer makes some special optimizations to a circuit and wants to check that they are "safe". Using the above correspondences, we can translate such problems into checking that a number of propositional formulas $P_n \Leftrightarrow P'_n$ are tautologies. Slightly more elaborate problems in circuit design (e.g. ignoring certain 'don't care' possibilities) can also be translated to tautology-checking. Thus, efficient methods for tautology checking directly yield useful tools for hardware verification.

Many other finite arithmetical and combinatorial problems can also be encoded as problems of propositional validity or satisfiability. Notably, one can express the assertion that a particular number is prime as a propositional formula, essentially by encoding a multiplier circuit and claiming that a certain combination of outputs can only occur in degenerate cases. For instance, the following formula asserts that 5 is prime:

$$
\begin{array}{l}
(out_0 \Leftrightarrow x_0 \wedge y_0) \wedge \\
(out_1 \Leftrightarrow (x_0 \wedge y_1 \Leftrightarrow \neg(x_1 \wedge y_0))) \wedge \\
(v_{22} \Leftrightarrow x_0 \wedge y_1 \wedge x_1 \wedge y_0) \wedge \\
(u_{20} \Leftrightarrow (x_1 \wedge y_1 \Leftrightarrow \neg v_{22})) \wedge \\
(u_{21} \Leftrightarrow x_1 \wedge y_1 \wedge v_{22}) \wedge \\
(out_2 \Leftrightarrow u_{20}) \wedge \\
(out_3 \Leftrightarrow u_{21}) \wedge \\
out_0 \wedge \neg out_1 \wedge out_2 \wedge \neg out_3 \\
\Rightarrow \bot
\end{array}
$$

Proving such a formula to be a tautology verifies that the number is prime, while proving it to be satisfiable indicates that the number is composite. Moreover, a satisfying valuation can be mapped into factors.

Although the above is hardly competitive with more direct means of factoring and primality testing, the situation with some combinatorial problems is better. The cornerstone of the elaborate theory of NP-completeness is exactly the huge collection of combinatorial problems that can all be reduced to each other and to propositional satisfiability. Recently it's become increasingly clear that this is useful not just as a theoretical reduction but as a practical approach. Surprisingly, many combinatorial problems are solved better by translating to SAT than by customized algorithms! This is no doubt a tribute to the enormous engineering effort that has gone into SAT solvers. Thus, we might consider SAT, the satisfiability problem, as a kind of machine code into which other combinatorial problems can be 'compiled'.

2. First-order logic

In contrast to propositional logic, many interesting questions about first order and higher order logic are undecidable even in principle, let alone in practice. Church (1936) and Turing (1936) showed that even pure logical validity in first order logic is undecidable, introducing in the process many of the basic ideas of computability theory.

On the other hand, it is not too hard to see that logical validity is *semidecidable* — this is certainly a direct consequence of completeness theorems for proof systems in first order logic (Gödel 1930), and was arguably implicit in work by Skolem (1922). This

means that we can at least program a computer to enumerate all valid first order formulas. One simple approach is based on the following logical principle, due to Skolem and Gödel but usually mis-named "Herbrand's theorem":

> Let $\forall x_1, \ldots, x_n. P[x_1, \ldots, x_n]$ be a first order formula with only the indicated universal quantifiers (i.e. the body $P[x_1, \ldots, x_n]$ is quantifier-free). Then the formula is satisfiable iff the infinite set of 'ground instances' $p[t_1^i, \ldots, t_n^i]$ that arise by replacing the variables by arbitrary variable-free terms made up from functions and constants in the original formula is *propositionally* satisfiable.

We can get the original formula into the special form required by some simple normal form transformations, introducing Skolem functions to replace existentially quantified variables. By the compactness theorem for propositional logic, if the infinite set of instances is unsatisfiable, then so will be some finite subset. In principle we can enumerate *all* possible sets, one by one, until we find one that is not propositionally satisfiable. (If the formula is satisfiable, we will never discover it by this means. By undecidability, we know this is unavoidable.) A precise description of this procedure is tedious, but a simple example may help. Suppose we want to prove that the following is valid. This is often referred to as the 'drinker's principle', because you can think of it as asserting that there is some person x such that if x drinks, so does everyone.

$$\exists x. \forall y. D(x) \Rightarrow D(y)$$

We start by negating the formula. To prove that the original is valid, we need to prove that this is unsatisfiable:

$$\neg(\exists x. \forall y. D(x) \Rightarrow D(y))$$

We then make some transformations to a logical equivalent so that it is in 'prenex form' with all quantifiers at the front.

$$\forall x. \exists y. D(x) \wedge \neg D(y)$$

We then introduce a Skolem function f for the existentially quantified variable y:

$$\forall x. D(x) \wedge \neg D(f(x))$$

We now consider the *Herbrand* universe, the set of all terms built up from constants and functions in the original formula. Since here we have no nullary constants, we need to add one c to get started (this effectively builds in the assumption that all interpretations have a non-empty domain). The Herbrand universe then becomes $\{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$. By Herbrand's theorem, we need to test all sets of ground instances for propositional satisfiability. Let us enumerate them in increasing size. The first one is:

$$D(c) \wedge \neg D(f(c))$$

This is not propositionally unsatisfiable, so we consider the next:

$$(D(c) \wedge \neg D(f(c))) \wedge (D(f(c)) \wedge \neg D(f(f(c)))$$

Now this is propositionally unsatisfiable, so we terminate with success.

2.1. Unification-based methods

The above idea (Robinson 1957) led directly some early computer implementations, e.g. by Gilmore (1960). Gilmore tested for propositional satisfiability by transforming the successively larger sets to disjunctive normal form. A more efficient approach is to use the Davis-Putnam algorithm — it was in this context that it was originally introduced (Davis and Putnam 1960). However, as Davis (1983) admits in retrospect:

> ... effectively eliminating the truth-functional satisfiability obstacle only uncovered the deeper problem of the combinatorial explosion inherent in unstructured search through the Herbrand universe ...

The next major step forward in theorem proving was a more intelligent means of choosing substitution instances, to pick out the small set of relevant instances instead of blindly trying all possibilities. The first hint of this idea appears in Prawitz, Prawitz, and Voghera (1960), and it was systematically developed by Robinson (1965), who gave an effective syntactic procedure called *unification* for deciding on appropriate instantiations to make terms match up correctly.

There are many unification-based theorem proving algorithms. Probably the best-known is *resolution*, in which context Robinson (1965) introduced full unification to automated theorem proving. Another important method quite close to resolution and developed independently at about the same time is the inverse method (Maslov 1964; Lifschitz 1986). Other popular algorithms include tableaux (Prawitz, Prawitz, and Voghera 1960), model elimination (Loveland 1968; Loveland 1978) and the connection method (Kowalski 1975; Bibel and Schreiber 1975; Andrews 1976). Crudely speaking:

- Tableaux = Gilmore procedure + unification
- Resolution = Davis-Putnam procedure + unification

Tableaux and resolution can be considered as classic representatives of 'top-down' and 'bottom-up' methods respectively. Roughly speaking, in top-down methods one starts from a goal and works backwards, while in bottom-up methods one starts from the assumptions and works forwards. This has significant implications for the very nature of unifiable variables, since in bottom-up methods they are local (implicitly universally quantified) whereas in top-down methods they are global, correlated in different portions of the proof tree. This is probably the most useful way of classifying the various first-order search procedures and has a significant impact on the problems where they perform well.

2.2. Decidable problems

Although first order validity is undecidable in general, there are special classes of formulas for which it is decidable, e.g.

- AE formulas, which involve no function symbols and when placed in prenex form have all the universal quantifiers before the existential ones.

- Monadic formulas, involving no function symbols and only monadic (unary) relation symbols.
- Purely universal formulas

The decidability of AE formulas is quite easy to see, because no function symbols are there to start with, and because of the special quantifier nesting, none are introduced in Skolemization. Therefore the Herbrand universe is finite and the enumeration of ground instances cannot go on forever. The decidability of the monadic class can be proved in various ways, e.g. by transforming into AE form by pushing quantifiers inwards ('miniscoping'). Although neither of these classes is particularly useful in practice, it's worth noting that the monadic formulas subsume traditional Aristotelian syllogisms, at least on a straightforward interpretation of what they are supposed to mean. For example

If all M are P, and all S are M, then all S are P

can be expressed using monadic relations as follows:

$$(\forall x.\, M(x) \Rightarrow P(x)) \wedge (\forall x.\, S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\, S(x) \Rightarrow P(x))$$

For purely universal formulas, we can use *congruence closure* (Nelson and Oppen 1980; Shostak 1978; Downey, Sethi, and Tarjan 1980). This allows us to prove that one equation follows from others, e.g. that

$$\forall x.\, f(f(f(x)) = x \wedge f(f(f(f(f(x))))) = x \Rightarrow f(x) = x$$

As the name implies, congruence closure involves deducing all equalities between subterms that follow from the asserted ones by using equivalence and congruence properties of equality. In our case, for example, the first equation $f(f(f(x)) = x$ implies $f(f(f(f(x))) = f(x)$ and hence $f(f(f(f(f(x))))) = f(f(x))$, and then symmetry and transitivity with the second equation imply $f(f(x)) = x$, and so on. It straightforwardly extends to deciding the entire universal theory by refutation followed by DNF transformation and congruence closure on the equations in the disjuncts, seeing whether any negated equations in the same disjunct are implied.

An alternative approach is to reduce the problem to SAT by introducing a propositional variable for each equation between subterms, adding constraints on these variables to reflect congruence properties. This has the possibly significant advantage that no potentially explosive DNF transformation need be done, and exploits the power of SAT solvers. For example if $E_{m,n}$ (for $0 \leq m,n \leq 5$) represents the equation $f^m(x) = f^n(x)$, we want to deduce $E_{3,0} \wedge E_{5,0} \Rightarrow E_{1,0}$ assuming the equality properties $\bigwedge_n E_{n,n}$ (reflexivity), $\bigwedge_{m,n} E_{m,n} \Rightarrow E_{n,m}$ (symmetry), $\bigwedge_{m,n,p} E_{m,n} \wedge E_{n,p} \Rightarrow E_{m,p}$ (transitivity) and $\bigwedge_{m,n} E_{m,n} \Rightarrow E_{m+1,n+1}$ (congruence).

2.3. Theories

Also interesting in practice are situations where, rather than absolute logical validity, we want to know whether statements follow from some well-accepted set of mathematical axioms, or are true in some particular interpretation like the real numbers $\mathbb{R}$. We generalize the notion of logical validity, and say that p is a logical consequence of axioms A,

written $A \models p$, if for any valuation, every interpretation that makes all the formulas in A true also makes p true. (This looks straightforward but unfortunately there is some inconsistency in standard texts. For example the above definition is found in Enderton (1972), whereas in Mendelson (1987) the definition has the quantification over valuations per formula: every interpretation in which each formula of A is true in all valuations makes p true in all valuations. Fortunately, in most cases all the formulas in A are sentences, formulas with no free variables, and then the two definitions coincide.) Ordinary validity of p is the special cases $\emptyset \models p$, usually written just $\models p$.

By a *theory*, we mean a set of formulas closed under first-order validity, or in other words, the set of logical consequences of a set of axioms. The smallest theory is the set of consequences of the empty set of axioms, i.e. the set of logically valid formulas. Note also that for any particular interpretation, the set of formulas true in that interpretation is also a theory. Some particularly important characteristics of a theory are:

- Whether it is *consistent*, meaning that we never have both $T \models p$ and $T \models \neg p$. (Equivalently, that we do not have $T \models \bot$, or that *some* formula does not follow from T.)
- Whether it is *complete*, meaning that for any sentence p, either $T \models p$ or $T \models \neg p$.
- Whether it is *decidable*, meaning that there is an algorithm that takes as input a formula p and decides whether $T \models p$.

Note that since we have a semidecision procedure for first-order validity, any complete theory based on a finite (or even semicomputable, with a slightly more careful analysis) set of axioms is automatically decidable: just search in parallel for proofs of $A \Rightarrow p$ and $A \Rightarrow \neg p$.

3. Arithmetical theories

First order formulas built up from equations and inequalities and interpreted over common number systems are often decidable. A common way of proving this is *quantifier elimination*.

3.1. Quantifier elimination

We say that a theory T in a first-order language L *admits quantifier elimination* if for each formula p of L, there is a quantifier-free formula q such that $T \models p \Leftrightarrow q$. (We assume that the equivalent formula contains no new free variables.) For example, the well-known criterion for a quadratic equation to have a (real) root can be considered as an example of quantifier elimination in a suitable theory T of reals:

$$T \models (\exists x.\ ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geq 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$$

If a theory admits quantifier elimination, then in particular any closed formula (one with no free variables, such as $\forall x.\ \exists y.\ x < y$) has a T-equivalent that is ground, i.e. contains no variables at all. In many cases of interest, we can quite trivially decide whether a ground formula is true or false, since it just amounts to evaluating a Boolean combination of arithmetic operations applied to constants, e.g. $2 < 3 \Rightarrow 4^2 + 5 < 23$. (One interesting exception is the theory of algebraically closed fields of unspecified characteristic,

where quantifiers can be eliminated but the ground formulas cannot in general be evaluated without knowledge about the characteristic.) Consequently quantifier elimination in such cases yields a decision procedure, and also shows that such a theory T is complete, i.e. every closed formula can be proved or refuted from T. For a good discussion of quantifier elimination and many explicit examples, see Kreisel and Krivine (1971). One of the simplest examples is the theory of 'dense linear (total) orders without end points' is based on a language containing the binary relation '$<$' as well as equality, but no function symbols. It can be axiomatized by the following set of sentences:

$$\begin{array}{l}
\forall x\, y.\, x = y \vee x < y \vee y < x \\
\forall x\, y\, z.\, x < y \wedge y < z \Rightarrow x < z \\
\forall x.\, x \not< x \\
\forall x\, y.\, x < y \Rightarrow \exists z.\, x < z \wedge z < y \\
\forall x.\, \exists y.\, x < y \\
\forall x.\, \exists y.\, y < x
\end{array}$$

3.2. Presburger arithmetic

One of the earliest theories shown to have quantifier elimination is linear arithmetic over the natural numbers or integers, as first shown by Presburger (1930). Linear arithmetic means that we are allows the usual equality and inequality relations, constants and addition, but no multiplication, except by constants. In fact, to get quantifier elimination we need to add infinitely many divisibility relations D_k for all integers $k \geq 2$. This doesn't affect decidability because those new relations are decidable for particular numbers.

Presburger's original algorithm is fairly straightforward, and follows the classic quantifier elimination pattern of dealing with the special case of an existentially quantified conjunction of literals. (We can always put the formula into disjunctive normal form and distribute existential quantifiers over the disjuncts, then start from the innermost quantifier.) For an in-depth discussion of Presburger's original procedure, the reader can consult Enderton (1972) and Smoryński (1980), or indeed the original article, which is quite readable — Stansifer (1984) gives an annotated English translation. A somewhat more efficient algorithm more suited to computer implementation is given by Cooper (1972).

3.3. Complex numbers

Over the complex numbers, we can also allow multiplication and still retain quantifier elimination. Here is a sketch of a naive algorithm for complex quantifier elimination. By the usual quantifier elimination pre-canonization, it suffices to be able to eliminate a single existential quantifier from a conjunction of positive and negative equations:

$$\exists x.\, p_1(x) = 0 \wedge \cdots \wedge p_n(x) = 0 \wedge q_1(x) \neq 0 \wedge \cdots q_m(x) \neq 0$$

We'll sketch now how this can be reduced to the $m \leq 1$ and $n \leq 1$ case. To reduce n we can use one equation to reduce the powers of variables in the others by elimination, e.g.

$$\begin{aligned} 2x^2+5x+3=0 \wedge x^2-1=0 &\Leftrightarrow 5x+5=0 \wedge x^2-1=0 \\ &\Leftrightarrow 5x+5=0 \wedge 0=0 \\ &\Leftrightarrow 5x+5=0 \end{aligned}$$

To reduce m, we may simply multiply all the $q_i(x)$ together since $q_i(x) \neq 0 \wedge q_{i+1}(x) \neq 0 \Leftrightarrow q_i(x) \cdot q_{i+1}(x) \neq 0$. Now, the problem that remains is:

$$\exists x.\, p(x) = 0 \wedge q(x) \neq 0$$

or equivalently $\neg(\forall x.\, p(x) = 0 \Rightarrow q(x) = 0)$. Consider the core formula:

$$\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$$

Assume that neither $p(x)$ nor $q(x)$ is the zero polynomial. Since we are working in an algebraically closed field, we know that the polynomials $p(x)$ and $q(x)$ split into linear factors whatever they may be:

$$\begin{aligned} p(x) &= (x-a_1) \cdot (x-a_2) \cdot \cdots \cdot (x-a_n) \\ q(x) &= (x-b_1) \cdot (x-b_2) \cdot \cdots \cdot (x-b_m) \end{aligned}$$

Now $p(x) = 0$ is equivalent to $\bigvee_{1 \leq i \leq n} x = a_i$ and $q(x) = 0$ is equivalent to $\bigvee_{1 \leq j \leq m} x = b_j$. Thus, the formula $\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$ says precisely that

$$\forall x.\, \bigvee_{1 \leq i \leq n} x = a_i \Rightarrow \bigvee_{1 \leq j \leq m} x = b_j$$

or in other words, all the a_i appear among the b_j. However, since there are just n linear factors in the antecedent, a given factor $(x-a_i)$ cannot occur more than n times and thus the polynomial divisibility relation $p(x)|q(x)^n$ holds. Conversely, if this divisibility relation holds for $n \neq 0$, then clearly $\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$ holds. Thus, the key quantified formula can be reduced to a polynomial divisibility relation, and it's not difficult to express this as a quantifier-free formula in the coefficients, thus eliminating the quantification over x.

3.4. Real algebra

In the case of the real numbers, one can again use addition and multiplication arbitrarily and it is decidable whether the formula holds in $\mathbb{R}$. A simple (valid) example is a case of the Cauchy-Schwartz inequality:

$$\forall x_1\, x_2\, y_1\, y_2.\, (x_1 \cdot y_1 + x_2 \cdot y_2)^2 \leq (x_1^2 + x_2^2) \cdot (y_1^2 + y_2^2)$$

This decidability result is originally due to Tarski (1951), though Tarski's method has non-elementary complexity and has apparently never been implemented. Perhaps the most efficient general algorithm currently known, and the first actually to be implemented on a computer, is the Cylindrical Algebraic Decomposition (CAD) method introduced by Collins (1976). (For related work around the same time see Łojasiewicz (1964) and the

method, briefly described by Rabin (1991), developed in Monk's Berkeley PhD thesis.) A simpler method, based on ideas by Cohen, was developed by Hörmander (1983) — see also Bochnak, Coste, and Roy (1998) and Gårding (1997) — and it is this algorithm that we will sketch.

Consider first the problem of eliminating the quantifier from $\exists x.\, P[x]$, where $P[x]$ is a Boolean combination of univariate polynomials (in x). Suppose the polynomials involved in the body are $p_1(x),\ldots,p_n(x)$. The key to the algorithm is to obtain a *sign matrix* for the set of polynomials. This is a division of the real line into a (possibly empty) ordered sequence of m points $x_1 < x_2 < \cdots < x_m$ representing precisely the zeros of the polynomials, with the rows of the matrix representing, in alternating fashion, the points themselves and the intervals between adjacent pairs and the two intervals at the ends:

$$(-\infty,x_1),x_1,(x_1,x_2),x_2,\ldots,x_{m-1},(x_{m-1},x_m),x_m,(x_m,+\infty)$$

and columns representing the polynomials $p_1(x),\ldots,p_n(x)$, with the matrix entries giving the signs, either positive ($+$), negative ($-$) or zero (0), of each polynomial p_i at the points and on the intervals. For example, for the collection of polynomials:

$$p_1(x) = x^2 - 3x + 2$$
$$p_2(x) = 2x - 3$$

the sign matrix looks like this:

Point/Interval	p_1	p_2
$(-\infty,x_1)$	$+$	$-$
x_1	0	$-$
(x_1,x_2)	$-$	$-$
x_2	$-$	0
(x_2,x_3)	$-$	$+$
x_3	0	$+$
$(x_3,+\infty)$	$+$	$+$

Note that x_1 and x_3 represent the roots 1 and 2 of $p_1(x)$ while x_2 represents 1.5, the root of $p_2(x)$. However the sign matrix contains no numerical information about the location of the points x_i, merely specifying the order of the roots of the various polynomials and what signs they take there and on the intervening intervals. It is easy to see that the sign matrix for a set of univariate polynomials $p_1(x),\ldots,p_n(x)$ is sufficient to answer any question of the form $\exists x.\, P[x]$ where the body $P[x]$ is quantifier-free and all atoms are of the form $p_i(x) \bowtie_i 0$ for any of the relations $=, <, >, \leq, \geq$ or their negations. We simply need to check each row of the matrix (point or interval) and see if one of them makes each atomic subformula true or false; the formula as a whole can then simply be "evaluated" by recursion.

In order to perform general quantifier elimination, we simply apply this basic operation to all the innermost quantified subformulas first (we can consider a universally

quantified formula $\forall x.\, P[x]$ as $\neg(\exists x.\, \neg P[x])$ and eliminate from $\exists x.\, \neg P[x]$). This can then be iterated until all quantifiers are eliminated. The only difficulty is that the coefficients of a polynomial may now contain other variables as parameters. But we can quite easily handle these by performing case-splits over the signs of coefficients and using pseudo-division of polynomials instead of division as we present below.

So the key step is finding the sign matrix, and for this the following simple observation is key. To find the sign matrix for

$$p, p_1, \ldots, p_n$$

it suffices to find one for the set of polynomials

$$p', p_1, \ldots, p_n, q_0, q_1, \ldots, q_n$$

where p', which we will sometimes write p_0 for regularity's sake, is the derivative of p, and q_i is the remainder on dividing p by p_i. For suppose we have a sign matrix for the second set of polynomials. We can proceed as follows.

First, we split the sign matrix into two equally-sized parts, one for the $p', p_1, \ldots, p_n$ and one for the $q_0, q_1, \ldots, q_n$, but for now keeping all the points in each matrix, even if the corresponding set of polynomials has no zeros there. We can now infer the sign of $p(x_i)$ for each point x_i that is a zero of one of the polynomials $p', p_1, \ldots, p_n$, as follows. Since q_k is the remainder of p after division by p_k, $p(x) = s_k(x)p_k(x) + q_k(x)$ for some $s_k(x)$. Therefore, since $p_k(x_i) = 0$ we have $p(x_i) = q_k(x_i)$ and so we can derive the sign of p at x_i from that of the corresponding q_k.

Now we can throw away the second sign matrix, giving signs for the $q_0, \ldots, q_n$, and retain the (partial) matrix for $p, p', p_1, \ldots, p_n$. We next 'condense' this matrix to remove points that are not zeros of one of the $p', p_1, \ldots, p_n$, but only of one of the q_i. The signs of the $p', p_1, \ldots, p_n$ in an interval from which some other points have been removed can be read off from any of the subintervals in the original subdivision — they cannot change because there are no zeros for the relevant polynomials there.

Now we have a sign matrix with correct signs at all the points, but undetermined signs for p on the intervals, and the possibility that there may be additional zeros of p inside these intervals. But note that since there are certainly no zeros of p' inside the intervals, there can be at most one additional root of p in each interval. Whether there is one can be inferred, for an internal interval (x_i, x_{i+1}), by seeing whether the signs of $p(x_i)$ and $p(x_{i+1})$, determined in the previous step, are both nonzero and are different. If not, we can take the sign on the interval from whichever sign of $p(x_i)$ and $p(x_{i+1})$ is nonzero (we cannot have them both zero, since then there would have to be a zero of p' in between). Otherwise we insert a new point y between x_i and x_{i+1} which is a zero (only) of p, and infer the signs on the new subintervals (x_i, y) and (y, x_{i+1}) from the signs at the endpoints. Other polynomials have the same signs on (x_i, y), y and (y, x_{i+1}) that had been inferred for the original interval (x_i, x_{i+1}). For external intervals, we can use the same reasoning if we temporarily introduce new points $-\infty$ and $+\infty$ and infer the sign of $p(-\infty)$ by flipping the sign of p' on the lowest interval $(-\infty, x_1)$ and the sign of $p(+\infty)$ by copying the sign of p' on the highest interval $(x_n, +\infty)$.

3.5. Word problems

Suppose K is a class of structures, e.g. all groups. The *word problem* for K asks whether a set E of equations between constants implies another such equation $s = t$ in all algebraic structures of class K. More precisely, we may wish to distinguish:

- The uniform word problem for K: deciding given any E and $s = t$ whether $E \models_M s = t$ for all interpretations M in K.
- The word problem for K, E: with E fixed, deciding given any $s = t$ whether $E \models_M s = t$ for all interpretations M in K.
- The free word problem for K: deciding given any $s = t$ whether $\models_M s = t$ for all interpretations M in K.

As a consequence of general algebraic results (e.g. every integral domain has a field of fractions, every field has an algebraic closure), there is a close relationship between word problems and results for particular interpretations. For example, for any universal formula in the language of rings, such as a word problem implication $\bigwedge_i s_i = t_i \Rightarrow s = t$, the following are equivalent, and hence we can solve it using complex quantifier elimination:

- It holds in all integral domains of characteristic 0
- It holds in all fields of characteristic 0
- It holds in all algebraically closed fields of characteristic 0
- It holds in any given algebraically closed field of characteristic 0
- It holds in $\mathbb{C}$

There is also a close relationship between word problems and ideal membership questions (Scarpellini 1969; Simmons 1970), sketched in a later section. These ideal membership questions can be solved using efficient methods like Gröbner bases (Buchberger 1965; Cox, Little, and O'Shea 1992).

3.6. Practical decision procedures

In many 'practical' applications of decision procedures, a straightforward implementation of one of the above quantifier elimination procedures may be a poor fit, in both a positive and negative sense:

- Fully general quantifier elimination is not needed
- The decidable theory must be combined with others

For example, since most program verification involves reasoning about integer inequalities (array indices etc.), one might think that an implementation of Presburger arithmetic is appropriate. But in practice, most of the queries (a) are entirely universally quantified, and (b) do not rely on subtle divisibility properties. A much faster algorithm can be entirely adequate. In some cases the problems are even more radically limited. For example, Ball, Cook, Lahriri, and Rajamani (2004) report that the majority of their integer inequality problems fall into a very class with at most two variables per inequality and coefficients of only ± 1, for which a much more efficient decision procedure is available (Harvey and Stuckey 1997). Of course there are exceptions, with some applications such as indexed predicate abstraction (Lahiri and Bryant 2004) demanding more general quantifier prefixes.

It's quite common in program verification to need a combination of a decidable theory, or indeed more than one, with uninterpreted function symbols. For example, in typical correctness theorems for loops, one can end up with problems like the following (the antecedent comes from a combination of the loop invariant and the loop termination condition):

$$x - 1 < n \wedge \neg(x < n) \Rightarrow a[x] = a[n]$$

In pure Presburger arithmetic one can deduce $x - 1 < n \wedge \neg(x < n) \Rightarrow x = n$, but one needs a methodology for making the further trivial deduction $x = n \Rightarrow a[x] = a[n]$. For non-trivial quantifier prefixes, this problem rapidly becomes undecidable, but for purely universal formulas like the above, there are well-established methods. The Nelson and Oppen (1979) approach is the most general. It exploits the fact that the only communication between component procedures need be equations and negated equations ($s = t$ and $s \neq t$), by virtue of a result in logic known as the *Craig interpolation theorem*. An alternative, which can be viewed as an optimization of Nelson-Oppen for some common cases, is due to Shostak (1984). It has taken a remarkably long time to reach a rigorous understanding of Shostak's method; indeed Reuß and Shankar (2001) showed that Shostak's original algorithm and all the then known later refinements were in fact incomplete and potentially nonterminating!

4. Interactive theorem proving

Even though first order validity is semi-decidable, it is seldom practical to solve interesting problems using unification-based approaches to pure logic. Nor is it the case that practical problems often fit conveniently into one of the standard decidable subsets. The best we can hope for in most cases is that the human will have to guide the proof process, but the machine may be able to relieve the tedium by filling in gaps, while always ensuring that no mistakes are made. This kind of application was already envisaged by Wang (Wang 1960)

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous than *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

The first notable interactive provers were the SAM (semi-automated mathematics) series. In 1966, the fifth in the series, SAM V, was used to construct a proof of a hitherto unproven conjecture in lattice theory (Bumcrot 1965). This was indubitably a success for the semi-automated approach because the computer automatically proved a result now called "SAM's Lemma" and the mathematician recognized that it easily yielded a proof of Bumcrot's conjecture.

Not long after the SAM project, the AUTOMATH (de Bruijn 1970; de Bruijn 1980) and Mizar (Trybulec 1978; Trybulec and Blair 1985) proof checking systems appeared, and each of them in its way has been profoundly influential. Although we will refer to these systems as 'interactive', we use this merely as an antonym of 'automatic'. In fact,

both AUTOMATH and Mizar were oriented around batch usage. However, the files that they process consist of a *proof*, or a proof sketch, which they *check* the correctness of, rather than a statement for which they attempt to find a proof automatically.

Mizar has been used to proof-check a very large body of mathematics, spanning pure set theory, algebra, analysis, topology, category theory and various unusual applications like mathematical puzzles and computer models. The body of mathematics formally built up in Mizar, known as the 'Mizar Mathematical Library' (MML), seems unrivalled in any other theorem proving system. The 'articles' (proof texts) submitted to the MML are automatically abstracted into human-readable form and published in the *Journal of Formalized Mathematics*, which is devoted entirely to Mizar formalizations.[1]

4.1. LCF — a programmable proof checker

The ideal proof checker should be *programmable*, i.e. users should be able to extend the built-in automation as much as desired. There's no particular difficulty in allowing this. Provided the basic mechanisms of the theorem prover are straightforward and well-documented and the source code is made available, there's no reason why a user shouldn't extend or modify it. However, the difficulty comes if we want to restrict the user to extensions that are logically sound — as presumably we might well wish to, since unsoundness renders questionable the whole idea of machine-checking of supposedly more fallible human proofs. Even fairly simple automated theorem proving programs are often subtler than they appear, and the difficulties of integrating a large body of special proof methods into a powerful interactive system without compromising soundness is not trivial.

One influential solution to this difficulty was introduced in the Edinburgh LCF project led by Robin Milner (Gordon, Milner, and Wadsworth 1979). Although this was for an obscure 'logic of computable functions' (hence the name LCF), the key idea, as Gordon (Gordon 1982) emphasizes, is equally applicable to more orthodox logics supporting conventional mathematics, and subsequently many programmable proof checkers were designed using the same principles, such as Coq,[2] HOL (Gordon and Melham 1993), Isabelle (Paulson 1994) and Nuprl (Constable 1986).

The key LCF idea is to use a special type (say `thm`) of proven theorems in the implementation language, so that anything of type `thm` must by construction have been *proved* rather than simply asserted. (In practice, the implementation language is usually a version of ML, which was specially designed for this purpose in the LCF project.) This is enforced by making `thm` an *abstract type* whose only constructors correspond to approved inference rules. But the user is given full access to the implementation language and can put the primitive rules together in more complicated ways using arbitrary programming. Because of the abstract type, any result of type `thm`, however it was arrived at, must ultimately have been produced by correct application of the primitive rules. Yet the means for arriving at it may be complex. We will consider how to render some decision procedures in proof-producing style in the next section.

[1] Available on the Web via `http://www.mizar.org/JFM`.

[2] See the Coq Web page `http://pauillac.inria.fr/coq`.

4.2. An LCF kernel for first-order logic

To explain the LCF idea in more concrete terms, we will show a complete LCF-style kernel for first order logic with equality, implemented in Objective CAML, starting with the basic syntax for first-order logic defined at the beginning. Before proceeding, we define some OCaml functions for useful syntax operations: constructing an equation, checking whether a term occurs in another, and checking whether a term occurs free in a formula. Note that we want to avoid the `fv` function earlier to emphasize that we aren't even relying on set operations.

```
let mk_eq s t = Atom("=",[s;t]);;

let rec occurs_in s t =
  s = t or
  match t with
    Var y -> false
  | Fn(f,args) -> exists (occurs_in s) args;;

let rec free_in t fm =
  match fm with
    False -> false
  | True -> false
  | Atom(p,args) -> exists (occurs_in t) args
  | Not(p) -> free_in t p
  | And(p,q) -> free_in t p or free_in t q
  | Or(p,q) -> free_in t p or free_in t q
  | Imp(p,q) -> free_in t p or free_in t q
  | Iff(p,q) -> free_in t p or free_in t q
  | Forall(y,p) -> not (occurs_in (Var y) t) & free_in t p
  | Exists(y,p) -> not (occurs_in (Var y) t) & free_in t p;;
```

There are many complete proof systems for first order logic. We will adopt a Hilbert-style proof system close to one first suggested by Tarski (1965), and subsequently presented in a textbook (Monk 1976). The idea is to avoid defining relatively tricky syntactic operations like substitution. We first define the signature for the OCaml abstract datatype of theorems:

```
module type Proofsystem =
   sig type thm
       val axiom_addimp : formula -> formula -> thm
       val axiom_distribimp :
            formula -> formula -> formula -> thm
       val axiom_doubleneg : formula -> thm
       val axiom_allimp : string -> formula -> formula -> thm
       val axiom_impall : string -> formula -> thm
       val axiom_existseq : string -> term -> thm
       val axiom_eqrefl : term -> thm
       val axiom_funcong : string -> term list -> term list -> thm
       val axiom_predcong : string -> term list -> term list -> thm
       val axiom_iffimp1 : formula -> formula -> thm
       val axiom_iffimp2 : formula -> formula -> thm
       val axiom_impiff : formula -> formula -> thm
       val axiom_true : thm
       val axiom_not : formula -> thm
```

```
    val axiom_or : formula -> formula -> thm
    val axiom_and : formula -> formula -> thm
    val axiom_exists : string -> formula -> thm
    val modusponens : thm -> thm -> thm
    val gen : string -> thm -> thm
    val concl : thm -> formula
 end;;
```

and then the actual implementation of the primitive inference rules. For example, `modusponens` is the traditional *modus ponens* inference rule allowing us to pass from two theorems of the form $\vdash p \Rightarrow q$ and $\vdash p$ to another one $\vdash q$:

$$\frac{\vdash p \Rightarrow q \quad \vdash p}{\vdash q}$$

In the usual LCF style, this becomes a function taking two arguments of type `thm` and producing another. In fact, most of these inference rules have no theorems as input, and can thus be considered as axiom schemes. For example, `axiom_addimp` creates theorems of the form $\vdash p \Rightarrow (q \Rightarrow p)$ and `axiom_existseq` creates those of the form $\exists x.\, x = t$ provided x does not appear in the term t:

```
module Proven : Proofsystem =
  struct type thm = formula
        let axiom_addimp p q = Imp(p,Imp(q,p))
        let axiom_distribimp p q r = Imp(Imp(p,Imp(q,r)),Imp(Imp(p,q),Imp(p,r)))
        let axiom_doubleneg p = Imp(Imp(Imp(p,False),False),p)
        let axiom_allimp x p q = Imp(Forall(x,Imp(p,q)),Imp(Forall(x,p),Forall(x,q)))
        let axiom_impall x p =
          if not (free_in (Var x) p) then Imp(p,Forall(x,p))
          else failwith "axiom_impall"
        let axiom_existseq x t =
          if not (occurs_in (Var x) t) then Exists(x,mk_eq (Var x) t)
          else failwith "axiom_existseq"
        let axiom_eqrefl t = mk_eq t t
        let axiom_funcong f lefts rights =
           fold_right2 (fun s t p -> Imp(mk_eq s t,p))
                       lefts rights (mk_eq (Fn(f,lefts)) (Fn(f,rights)))
        let axiom_predcong p lefts rights =
           fold_right2 (fun s t p -> Imp(mk_eq s t,p))
                       lefts rights (Imp(Atom(p,lefts),Atom(p,rights)))
        let axiom_iffimp1 p q = Imp(Iff(p,q),Imp(p,q))
        let axiom_iffimp2 p q = Imp(Iff(p,q),Imp(q,p))
        let axiom_impiff p q = Imp(Imp(p,q),Imp(Imp(q,p),Iff(p,q)))
        let axiom_true = Iff(True,Imp(False,False))
        let axiom_not p = Iff(Not p,Imp(p,False))
        let axiom_or p q = Iff(Or(p,q),Not(And(Not(p),Not(q))))
        let axiom_and p q = Iff(And(p,q),Imp(Imp(p,Imp(q,False)),False))
        let axiom_exists x p = Iff(Exists(x,p),Not(Forall(x,Not p)))
        let modusponens pq p =
          match pq with Imp(p',q) when p = p' -> q
                      | _ -> failwith "modusponens"
        let gen x p = Forall(x,p)
        let concl c = c
  end;;
```

Although simple, these rules are in fact complete for first-order logic with equality. At first they are tedious to use, but using the LCF technique we can build up a set of derived rules. The following derives $p \Rightarrow p$:

```
let imp_refl p = modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
                                          (axiom_addimp p (Imp(p,p))))
                             (axiom_addimp p p);;
```

Before long, we can reach the stage of automatic derived rules that, for example, prove propositional tautologies automatically, perform Knuth-Bendix completion, and prove first order formulas by standard proof search and translation into primitive inferences.

4.3. Proof style

One feature of the LCF style is that proofs (being programs) tend to be highly *procedural*, in contrast to the more declarative proofs supported by Mizar — for more on the contrast see Harrison (1996b). This can have important disadvantages in terms of readability and maintainability. In particular, it is difficult to understand the formal proof scripts in isolation; they need to be run in the theorem prover to understand what the intermediate states are. Nevertheless as pointed out in (Harrison 1996a) it is possible to implement more declarative styles of proof on top of LCF cores. For more recent experiments with Mizar-like declarative proof styles see Syme (1997), Wenzel (1999), Zammit (1999) and Wiedijk (2001).

4.4. A panorama of interactive theorem provers

There are numerous interactive theorem provers in the world. Wiedijk (2006) gives an instructive survey of some of the main interactive theorem provers (including a few such as Otter that might be considered automatic but which can be used in a more interactive style) giving the highlights of each one and showing proofs of the irrationality of $\sqrt{2}$ in each. Here is a quick summary of each one considered there:

- HOL — Seminal LCF-style prover for classical simply typed higher-order logic with several versions.
- Mizar — Pioneering system for formalizing mathematics, originating the declarative style of proof.
- PVS — Prover designed for applications with an expressive classical type theory and powerful automation.
- Coq — LCF-like prover for constructive Calculus of Constructions with reflective programming language.
- Otter/IVY — Powerful automated theorem prover for pure first-order logic plus a proof checker.
- Isabelle/Isar — Generic prover in LCF style with a newer declarative proof style influenced by Mizar.
- Alfa/Agda — Prover for constructive type theory integrated with dependently typed programming language.
- ACL2 — Highly automated prover for first-order number theory without explicit quantifiers, able to do induction proofs itself.

- PhoX — prover for higher-order logic designed to be relatively simple to use in comparison with Coq, HOL etc.
- IMPS — Interactive prover for an expressive logic supporting partially defined functions.
- Metamath — Fast proof checker for an exceptionally simple axiomatization of standard ZF set theory.
- Theorema — Ambitious integrated framework for theorem proving and computer algebra built inside Mathematica.
- Lego — Well-established framework for proof in constructive type theory, with a similar logic to Coq.
- Nuprl — LCF-style prover with powerful graphical interface for Martin-Löf type theory extended with new constructs.
- Omega — Unified combination in modular style of several theorem-proving techniques including proof planning.
- B prover — Prover for first-order set theory designed to support verification and refinement of programs.
- Minlog — Prover for minimal logic supporting practical extraction of programs from proofs.

5. Proof-producing decision procedures

Suppose we want to have the power of standard decision procedures such as quantifier elimination for real algebra, but we are determined to produce a *proof*. Most obviously, this might be because we really distrust complicated code, and want to have a theorem prover in the LCF style. There are other reasons too for insisting on a proof. For example, the idea behind proof-carrying code (Necula 1997) is that code will be accompanied by a proof of certain key properties, which can be checked at the point of use. Unless one expects the runtime environment to implement a variety of quite complicated decision procedures, it's preferable to keep this proof restricted to a limited repertoire of steps. We can still use arbitrary decision procedures, provided they can record a simple proof.

Assuming then that we do want to produce a proof, how much difference does this make to the implementation of decision procedures? Essentially we can divide decision procedures into two kinds: those that admit a nice compact proof, and those that (apparently) do not. Different techniques are appropriate in order to generate proofs in the two cases.

5.1. Separate certification

An interesting characteristic of decision problems generally, not just those for logical questions, is whether they admit some sort of 'certificate' which can be used to check the correctness of the answer relatively simply and easily. Of course, 'simple' and 'easy' are used vaguely here, but there is one well-known instance where there is a nice rigorous general theory, namely the class of NP problems. A decision problem is in the class NP if there is a *certificate* that can be used to check the correctness of a positive ('yes') answer in a time polynomial in the size of the input instance. Dually, a problem is in the class co-NP if there is a similar certificate for *negative* answers. For example, the problem of

deciding if a number is prime is clearly in co-NP, because if a number isn't prime, a suitable certificate is the factors, which can be checked in polynomial time. For instance, to verify that the number

$$349052951084765094914784961990389813341776463849338784399082057 7\times$$
$$327691329932667095499619881908344614131776429679929425397982885 33$$

is not prime, one simply needs to multiply the two factors and check that you get the number in question. But note that *finding* the certificate can be very difficult. The above problem, for example ('RSA129'), was eventually solved by a cooperative effort of around a thousand users lavishing spare CPU cycles on the task.

Moreover, it turns out that if a number is prime, that admits a polynomial-time checkable certificate too (Pratt 1975), albeit not quite such a simple one as just the factors. Therefore the problem of primality testing is not only in co-NP, but also in NP. Actually, quite recently it has been proved that the problem can be solved directly in polynomial time (Agrawal, Kayal, and Saxena 2002). Still, the above is a good illustration of our theme of separate certifiability.

Turning to logical problems, the obvious starting-point is the SAT problem for propositional logic. As noted, this was the original NP problem: if a formula is satisfiable then a suitable certificate is a satisfying valuation, which can be checked in polynomial time by a recursive pass over the formula. It is not known whether the problem is also in NP, i.e. whether every tautology has some short 'proof' in a rather general sense of proof. This is another well-known open problem in complexity theory, NP = co-NP.

Still, if we consider first-order theorem proving as discussed earlier, it is the case in practice that real problems normally do admit short proofs in one of the usual standard systems (though theoretical counterexamples are known). Thus, a realistic first-order proof is usually dominated by a search through a huge space of possible proofs, but the final proof, when found, is usually short and can be checked quickly. This approach has been used for a long time to incorporate first-order proof methods into LCF-style provers (Kumar, Kropf, and Schneider 1991), and more recently, has even been exploited to plug in 'off-the-shelf' external first-order provers into LCF systems (Hurd 1999). Similarly Harrison and Théry (1998) use the Maple computer algebra system to solve polynomial factorization and transcendental function integration problems. In each case the checking process (respectively multiplying polynomials and taking derivatives) is substantially easier than the process of finding the certificate (in both cases the certificate is just the 'answer').

Another interesting example is the universal theory of the complex numbers with addition and multiplication. We have seen a quantifier elimination procedure for this theory in general, but it doesn't seem to admit efficient certification. Restricting ourselves to universally quantified formulas, however, things are better, thanks to a classic theorem of algebraic geometry, the (weak) *Hilbert Nullstellensatz*:

> The polynomial equations $p_1(x_1,\ldots,x_n) = 0$, $\ldots$, $p_k(x_1,\ldots,x_n) = 0$ in an algebraically closed field have *no* common solution iff there are polynomials $q_1(x_1,\ldots,x_n)$, $\ldots$, $q_k(x_1,\ldots,x_n)$ such that the following polynomial identity holds:

$$q_1(x_1,\ldots,x_n)\cdot p_1(x_1,\ldots,x_n) + \cdots + q_k(x_1,\ldots,x_n)\cdot p_k(x_1,\ldots,x_n) = 1$$

To verify a universal formula, we can negate it and split it into disjunctive normal form, then refute each disjunct, all variables being implicitly existentially quantified. By a trick due to Rabinowitsch, we can replace any negated equations by equations at the cost of introducing more variables, since $p \neq 0$ is equivalent to $\exists x.\ px - 1 = 0$. Now to refute a conjunction of equations

$$p_1(x_1,\ldots,x_n) = 0 \wedge \cdots \wedge p_k(x_1,\ldots,x_n) = 0$$

we just have to come up with the various 'cofactors' $q_i(x_1,\ldots,x_n)$ such that

$$q_1(x_1,\ldots,x_n)\cdot p_1(x_1,\ldots,x_n) + \cdots + q_k(x_1,\ldots,x_n)\cdot p_k(x_1,\ldots,x_n) = 1$$

It is now easy to see that the original set of equations has no solution, because by this identity, the existence of a common zero for the $p_i(x_1,\ldots,x_n)$ would imply $0 = 1$. Note that this reasoning relies only on one direction of the Nullstellensatz (the easy one), but we need to appeal to the other direction to know that such cofactors always exist. The traditional Nullstellensatz proofs are nonconstructive, but there are algorithms for finding the cofactors. Perhaps the simplest and most effective is to take the standard Gröbner basis algorithm (Buchberger 1965) and instrument it with a little additional 'proof recording' (Harrison 2001).

If we turn to the universal theory of the reals, there is again a Nullstellensatz, though a significantly more complicated one. The direct analog of the complex Nullstellensatz involved adding a 'sum of squares':

> The polynomial equations $p_1(x_1,\ldots,x_n) = 0$, ..., $p_k(x_1,\ldots,x_n) = 0$ in an real closed field have *no* common solution iff there are polynomials $q_1(x_1,\ldots,x_n)$, ..., $q_k(x_1,\ldots,x_n)$ and $s_1(x_1,\ldots,x_n)$, ..., $s_m(x_1,\ldots,x_n)$ such that the following polynomial identity holds:
>
> $$q_1(x_1,\ldots,x_n)\cdot p_1(x_1,\ldots,x_n) + \cdots + q_k(x_1,\ldots,x_n)\cdot p_k(x_1,\ldots,x_n) + s_1(x_1,\ldots,x_n)^2 + \cdots + s_m(x_1,\ldots,x_n)^2 = -1$$

Since a sum of squares over the reals is nonnegative, it is similarly easy to get a proof out of that certificate. There are more general forms of the real Nullstellensatz that allow one to refute a collection of strict and non-strict inequalities, equations and inequations. Although theoretically, analogs of the Rabinowitsch trick mean that equations are enough:

$$p > 0 \Leftrightarrow \exists x.\ px^2 - 1 = 0$$
$$p \geq 0 \Leftrightarrow \exists x.\ p - x^2 = 0$$

the general Nullstellensatz is more efficient in practice. For a more detailed study of this topic, and a method of generating the certificates using semidefinite programming, see Parrilo (2003). We will content ourselves with one simple example. Suppose we want to show that if a quadratic equation has a (real) solution, its discriminant is nonnegative:

$$\forall a\ b\ c\ x.\ ax^2 + bx + c = 0 \Rightarrow b^2 - 4ac \geq 0$$

A suitable certificate is the following. Since the first term on the right is a square, and the second is zero by hypothesis, it is clear that the LHS is nonnegative. Almost all the computational cost is in coming up with the appropriate square term and multiple of the input equation to make this identity hold; checking it is then easy.

$$b^2 - 4ac = (2ax + b)^2 - 4a(ax^2 + bx + c)$$

Our theme of checkability has been stressed by a number of researchers, notably Blum (1993). He suggests that in many situations, checking results may be more practical and effective than verifying code. This argument is related to, in some sense a generalization of, arguments by Harrison (1995) in favour of the LCF approach to theorem proving rather than so-called 'reflection'. Mehlhorn et al. (1996) describe the addition of result checking to routines in the LEDA library of C++ routines for computational geometry (e.g. finding convex hulls and Voronoi diagrams).

5.2. Reflection

For some decision problems, no particularly efficient certification method is known — take for example the general first-order theory of reals. In this case, if we want an easily checkable certificate, the only option seems to be to implement the procedure in such a way that it generates a complete 'trace' of logical inferences justifying its every step. The problem is that such a certificate is likely to be very large, making it inefficient to check and possibly even to generate.

An implementation of Hörmander's algorithm that produces a HOL proof as it runs has been written by McLaughlin (McLaughlin and Harrison 2005). It is substantially slower than a standard implementation that does not produce theorems. Nevertheless, a few techniques help to narrow the gap. In particular, by proving suitably general lemmas, one can encode quite 'generic' patterns of transformation, so that many of the steps of the algorithm can be justified just by an instantiation of the lemma to specific variables. This idea has long been used by HOL experts — for an early example see Melham (1989) — under the name 'proforma theorems'.

In an extreme form, one can essentially encode all the data structures inside the logic, and express the steps of the algorithm as equations or implications that can be 'executed' by rewriting or more delicate logical steps. This may involve defining a separate class of syntactic objects inside the logic and defining the semantic map. For example, for Presburger arithmetic in HOL, we have defined a type of restricted first-order terms of arithmetic:

```
let cform_INDUCT,cform_RECURSION = define_type
  "cform = Lt int
         | Gt int
         | Eq int
         | Ne int
         | Divides int int
         | Ndivides int int
         | And cform cform
         | Or cform cform
         | Nox bool";;
```

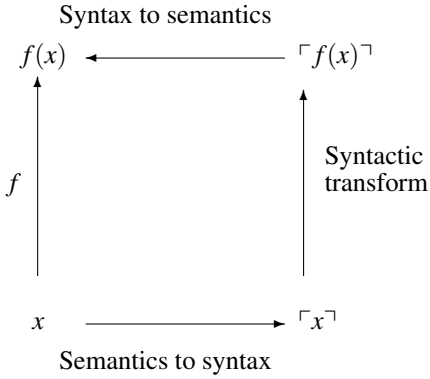
together with the semantic map

```
let interp = new_recursive_definition cform_RECURSION
  `(interp x (Lt e) ⇔ x + e < &0) ∧
   (interp x (Gt e) ⇔ x + e > &0) ∧
   (interp x (Eq e) ⇔ (x + e ⇔&0)) ∧
   (interp x (Ne e) ⇔ ¬(x + e = &0)) ∧
   (interp x (Divides c e) ⇔ c divides (x + e)) ∧
   (interp x (Ndivides c e) ⇔ ¬(c divides (x + e))) ∧
   (interp x (And p q) ⇔ interp x p ∧ interp x q) ∧
   (interp x (Or p q) ⇔ interp x p ∨ interp x q) ∧
   (interp x (Nox P) ⇔ P)`;;
```

Now, the core quantifier elimination transformation can be expressed directly as theorems about the syntax, e.g.

```
|- ∀p d. alldivide d p ∧ &0 < d
         ==> ((∃x. interp x p) ⇔
              ∃j. &1 <= j ∧ j <= d ∧
                  (interp j (minusinf p) ∨
                   ∃b. b IN Bset p ∧ interp (b + j) p))
```

where the various syntactic notions `alldivide`, `minusinf` and `interp` are defined on the syntax. Now, in order to eliminate a quantifier in HOL from an expression $\exists x.\, P[x]$, one first 'rewrites backwards' with the definition of `interp` to map it into a formula in the canonical form $\exists x.\, \texttt{interp}\ x\ p$, appeals to the above general theorem to transform it into a quantifier-free equivalent, then 'rewrites forward' with the definition of `interp` to eliminate the internal syntax:



A similar approach is popular in the Coq theorem prover, where it is commonly called *reflection*. The benefits in Coq are disproportionately large since making the syntactic transformations by 'calculation' is much more efficient than inference in general, which not only checks proofs as it goes, but generates large 'proof objects'.

A further generalization of reflection is to attempt to verify a decision procedure once and for all, so that it can be relied upon without producing a proof as it runs. This is certainly an interesting topic, but takes us too far from our main theme. You should hear more on this topic from other lectures.

References

Agrawal, M., Kayal, N., and Saxena, N. (2002) PRIMES is in P. Available on the Web via `http://www.cse.iitk.ac.in/news/primality.html`.

Alur, R. and Peled, D. A. (eds.) (2004) *Computer Aided Verification, 16th International Conference, CAV 2004*, Volume 3114 of *Lecture Notes in Computer Science*, Boston, MA. Springer-Verlag.

Andrews, P. B. (1976) Theorem proving by matings. *IEEE transactions on Computers*, **25**, 801–807.

Ball, T., Cook, B., Lahriri, S. K., and Rajamani, S. K. (2004) Zapato: Automatic theorem proving for predicate abstraction refinement. See Alur and Peled (2004), pp. 457–461.

Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., and Théry, L. (eds.) (1999) *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Volume 1690 of *Lecture Notes in Computer Science*, Nice, France. Springer-Verlag.

Bibel, W. and Schreiber, J. (1975) Proof search in a Gentzen-like system of first order logic. In Gelenbe, E. and Potier, D. (eds.), *Proceedings of the International Computing Symposium*, pp. 205–212. North-Holland.

Blum, M. (1993) Program result checking: A new approach to making programs more reliable. In Lingas, A., Karlsson, R., and Carlsson, S. (eds.), *Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings*, Volume 700 of *Lecture Notes in Computer Science*, Lund, Sweden, pp. 1–14. Springer-Verlag.

Bochnak, J., Coste, M., and Roy, M.-F. (1998) *Real Algebraic Geometry*, Volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag.

Buchberger, B. (1965) *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. Ph. D. thesis, Mathematisches Institut der Universität Innsbruck. English translation to appear in Journal of Symbolic Computation, 2006.

Bumcrot, R. (1965) On lattice complements. *Proceedings of the Glasgow Mathematical Association*, **7**, 22–23.

Caviness, B. F. and Johnson, J. R. (eds.) (1998) *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation. Springer-Verlag.

Church, A. (1936) An unsolvable problem of elementary number-theory. *American Journal of Mathematics*, **58**, 345–363.

Clarke, E. M., Grumberg, O., and Peled, D. (1999) *Model Checking*. MIT Press.

Collins, G. E. (1976) Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Brakhage, H. (ed.), *Second GI Conference on Automata Theory and Formal Languages*, Volume 33 of *Lecture Notes in Computer Science*, Kaiserslautern, pp. 134–183. Springer-Verlag.

Constable, R. (1986) *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall.

Cook, S. A. (1971) The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pp. 151–158.

Cooper, D. C. (1972) Theorem proving in arithmetic without multiplication. In Melzer, B. and Michie, D. (eds.), *Machine Intelligence 7*, pp. 91–99. Elsevier.

Cox, D., Little, J., and O'Shea, D. (1992) *Ideals, Varieties, and Algorithms*. Springer-Verlag.

Davis, M. (1983) The prehistory and early history of automated deduction. See Siekmann and Wrightson (1983), pp. 1–28.

Davis, M., Logemann, G., and Loveland, D. (1962) A machine program for theorem proving. *Communications of the ACM*, **5**, 394–397.

Davis, M. and Putnam, H. (1960) A computing procedure for quantification theory. *Journal of the ACM*, **7**, 201–215.

de Bruijn, N. G. (1970) The mathematical language AUTOMATH, its usage and some of its extensions. In Laudet, M., Lacombe, D., Nolin, L., and Schützenberger, M. (eds.), *Symposium on Automatic Demonstration*, Volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Springer-Verlag.

de Bruijn, N. G. (1980) A survey of the project AUTOMATH. In Seldin, J. P. and Hindley, J. R. (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pp. 589–606. Academic Press.

Downey, P. J., Sethi, R., and Tarjan, R. (1980) Variations on the common subexpression problem. *Journal of the ACM*, **27**, 758–771.

Enderton, H. B. (1972) *A Mathematical Introduction to Logic*. Academic Press.

Gårding, L. (1997) *Some Points of Analysis and Their History*, Volume 11 of *University Lecture Series*. American Mathematical Society / Higher Education Press.

Gilmore, P. C. (1960) A proof method for quantification theory: Its justification and realization. *IBM Journal of research and development*, **4**, 28–35.

Gödel, K. (1930) Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, **37**, 349–360. English translation 'The completeness of the axioms of the functional calculus of logic' in Heijenoort (1967), pp. 582–591.

Goldberg, E. and Novikov, Y. (2002) BerkMin: a fast and robust Sat-solver. In Kloos, C. D. and Franca, J. D. (eds.), *Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, Paris, France, pp. 142–149. IEEE Computer Society Press.

Gordon, M. J. C. (1982) Representing a logic in the LCF metalanguage. In Néel, D. (ed.), *Tools and notions for program construction: an advanced course*, pp. 163–185. Cambridge University Press.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Groote, J. F. (2000) The propositional formula checker Heerhugo. *Journal of Automated Reasoning*, **24**, 101–125.

Harrison, J. (1995) Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz`.

Harrison, J. (1996a) A Mizar mode for HOL. In Wright, J. v., Grundy, J., and Harrison, J. (eds.), *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, pp. 203–220. Springer-Verlag.

Harrison, J. (1996b) Proof style. In Giménez, E. and Paulin-Mohring, C. (eds.), *Types for Proofs and Programs: International Workshop TYPES'96*, Volume 1512 of *Lecture Notes in Computer Science*, Aussois, France, pp. 154–172. Springer-Verlag.

Harrison, J. (2001) Complex quantifier elimination in HOL. In Boulton, R. J. and Jackson, P. B. (eds.), *TPHOLs 2001: Supplemental Proceedings*, pp. 159–174. Division of Informatics, University of Edinburgh. Published as Informatics Report Series EDI-INF-RR-0046. Available on the Web at `http://www.informatics.ed.ac.uk/publications/report/0046.html`.

Harrison, J. and Théry, L. (1998) A sceptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, **21**, 279–294.

Harvey, W. and Stuckey, P. (1997) A unit two variable per inequality integer constraint solver for constraint logic programming. *Australian Computer Science Communications*, **19**, 102–111.

Heijenoort, J. v. (ed.) (1967) *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*. Harvard University Press.

Hörmander, L. (1983) *The Analysis of Linear Partial Differential Operators II*, Volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag.

Hurd, J. (1999) Integrating Gandalf and HOL. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 311–321.

Kowalski, R. (1975) A proof procedure using connection graphs. *Journal of the ACM*, **22**, 572–595.

Kreisel, G. and Krivine, J.-L. (1971) *Elements of mathematical logic: model theory* (Revised second ed.). Studies in Logic and the Foundations of Mathematics. North-Holland. First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles' published by Dunod, Paris in 1964.

Kumar, R., Kropf, T., and Schneider, K. (1991) Integrating a first-order automatic prover in the HOL environment. In Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J. (eds.), *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, University of California at Davis, Davis CA, USA, pp. 170–176. IEEE Computer Society Press.

Lahiri, S. K. and Bryant, R. E. (2004) Indexed predicate discovery for unbounded system verification. See Alur and Peled (2004), pp. 135–147.

Lifschitz, V. (1986) *Mechanical Theorem Proving in the USSR: the Leningrad School*. Monograph Series on Soviet Union. Delphic Associates, 7700 Leesburg Pike, #250, Falls Church, VA 22043. Phone: (703) 556-0278. See also 'What is the inverse method?' in the Journal of Automated Reasoning, vol. 5, pp. 1–23, 1989.

Łojasiewicz, S. (1964) Triangulations of semi-analytic sets. *Annali della Scuola Normale Superiore di Pisa, ser. 3*, **18**, 449–474.

Loveland, D. W. (1968) Mechanical theorem-proving by model elimination. *Journal of the ACM*, **15**, 236–251.

Loveland, D. W. (1978) *Automated theorem proving: a logical basis*. North-Holland.

Maslov, S. J. (1964) An inverse method of establishing deducibility in classical predicate calculus. *Doklady Akademii Nauk*, **159**, 17–20.

McLaughlin, S. and Harrison, J. (2005) A proof-producing decision procedure for real arithmetic. In Nieuwenhuis, R. (ed.), *CADE-20: 20th International Conference on Automated Deduction, proceedings*, Volume 3632 of *Lecture Notes in Computer Science*, Tallinn, Estonia, pp. 295–314. Springer-Verlag.

Mehlhorn, K. et al. (1996) Checking geometric programs or verification of geometric structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (FCRC'96)*, Philadelphia, pp. 159–165. Association for Computing Machinery.

Melham, T. F. (1989) Automating recursive type definitions in higher order logic. In Birtwistle, G. and Subrahmanyam, P. A. (eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 341–386. Springer-Verlag.

Mendelson, E. (1987) *Introduction to Mathematical Logic* (Third ed.). Mathematics series. Wadsworth and Brooks Cole.

Monk, J. D. (1976) *Mathematical logic*, Volume 37 of *Graduate Texts in Mathematics*. Springer-Verlag.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001) Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 530–535. ACM Press.

Necula, G. C. (1997) Proof-carrying code. In *Conference record of POPL'97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119. ACM Press.

Nelson, G. and Oppen, D. (1980) Fast decision procedures based on congruence closure. *Journal of the ACM*, **27**, 356–364.

Nelson, G. and Oppen, D. C. (1979) Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, **1**, 245–257.

Parrilo, P. A. (2003) Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, **96**, 293–320.

Paulson, L. C. (1994) *Isabelle: a generic theorem prover*, Volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag. With contributions by Tobias Nipkow.

Pratt, V. (1975) Every prime has a succinct certificate. *SIAM Journal of Computing*, **4**, 214–220.

Prawitz, D., Prawitz, H., and Voghera, N. (1960) A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, **7**, 102–128.

Presburger, M. (1930) Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pp. 92–101, 395. Warsaw. Annotated English version by Stansifer (1984).

Rabin, M. O. (1991) Decidable theories. In Barwise, J. and Keisler, H. (eds.), *Handbook of mathematical logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 595–629. North-Holland.

Reuß, H. and Shankar, N. (2001) Deconstructing Shostak. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 19–28. IEEE Computer Society Press.

Robinson, A. (1957) Proving a theorem (as done by man, logician, or machine). In *Summaries of Talks Presented at the Summer Institute for Symbolic Logic*. Second edition published by the Institute for Defense Analysis, 1960. Reprinted in Siekmann and Wrightson (1983), pp. 74–76.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**, 23–41.

Scarpellini, B. (1969) On the metamathematics of rings and integral domains. *Transactions of the American Mathematical Society*, **138**, 71–96.

Sheeran, M. and Stålmarck, G. (2000) A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, **16**, 23–58.

Shostak, R. (1978) An algorithm for reasoning about equality. *Communications of the ACM*, **21**, 356–364.

Shostak, R. (1984) Deciding combinations of theories. *Journal of the ACM*, **31**, 1–12.

Siekmann, J. and Wrightson, G. (eds.) (1983) *Automation of Reasoning — Classical Papers on Computational Logic, Vol. I (1957-1966)*. Springer-Verlag.

Simmons, H. (1970) The solution of a decision problem for several classes of rings. *Pacific Journal of Mathematics*, **34**, 547–557.

Skolem, T. (1922) Einige Bemerkungen zur axiomatischen Begründung der Mengenlehre. In *Matematikerkongress i Helsingfors den 4–7 Juli 1922, Den femte skandinaviska matematikerkongressen, Redogörelse*. Akademiska Bokhandeln, Helsinki. English translation "Some remarks on axiomatized set theory" in Heijenoort (1967), pp. 290–301.

Smoryński, C. (1980) *Logic Number Theory I: An Introduction*. Springer-Verlag.

Stålmarck, G. (1994) System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent number 5,276,897; see also Swedish Patent 467 076.

Stålmarck, G. and Säflund, M. (1990) Modeling and verifying systems and software in propositional logic. In Daniels, B. K. (ed.), *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, Gatwick, UK, pp. 31–36. Pergamon Press.

Stansifer, R. (1984) Presburger's article on integer arithmetic: Remarks and translation. Technical Report CORNELLCS:TR84-639, Cornell University Computer Science Department.

Syme, D. (1997) DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Tarski, A. (1951) *A Decision Method for Elementary Algebra and Geometry*. University of California Press. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J. C. C. McKinsey. Reprinted in Caviness and Johnson (1998), pp. 24–84.

Tarski, A. (1965) A simplified formalization of predicate logic with identity. *Arkhiv für mathematische Logik und Grundlagenforschung*, **7**, 61–79.

Trybulec, A. (1978) The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, **6**, 136–140.

Trybulec, A. and Blair, H. A. (1985) Computer aided reasoning. In Parikh, R. (ed.), *Logics of Programs*, Volume 193 of *Lecture Notes in Computer Science*, Brooklyn, pp. 406–412. Springer-Verlag.

Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, **42**, 230–265.

Wang, H. (1960) Toward mechanical mathematics. *IBM Journal of research and development*, **4**, 2–22.

Wenzel, M. (1999) Isar - a generic interpretive approach to readable formal proof documents. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 167–183.

Wiedijk, F. (2001) Mizar light for HOL Light. In Boulton, R. J. and Jackson, P. B. (eds.), *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, Volume 2152 of *Lecture Notes in Computer Science*, pp. 378–394. Springer-Verlag.

Wiedijk, F. (2006) *The Seventeen Provers of the World*, Volume 3600 of *Lecture Notes in Computer Science*. Springer-Verlag.

Zammit, V. (1999) On the implementation of an extensible declarative proof language. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 185–202.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Correctness of Effect-Based Program Transformations

Martin HOFMANN
Institut für Informatik, Ludwig-Maximilians-Universität, Munich, Germany

Abstract. We consider a type system capable of tracking reading, writing and allocation in a higher-order language with dynamically allocated references.

We give a denotational semantics to this type system which allows us to validate a number of effect-dependent program equivalences in the sense of observational equivalence. An example is the following:

$$x = e; y = e; e'(x, y) \text{ is equivalent to } x = e; e'(x, x)$$

provided that e does not read from memory regions that it writes to and moreover does not allocate memory that is encapsulated in the values of x and y.

Here x can be a higher-order function or a reference or a combination of both.

The two sides of the above equivalence turn out to be related in the denotational semantics which implies that they are observationally equivalent, ie can be replaced by one another in any (well-typed) program.

On the way we learn popular techniques such as parametrised logical relations, regions, admissible relations, etc., which belong to the toolbox of researchers in principles of programming languages.

Keywords. program transformation, denotational semantics, correctness of programs, logical relations.

1. Introduction

Many analyses and logics for imperative programs are concerned with establishing whether particular mutable variables may be read or written by a phrase. For example, the equivalence of while-programs

```
C ; if B then C′ else C′′  =
if B then (C;C′) else (C;C′′)
```

is valid when `B` does not read any variable which `C` might write. Hoare-style programming logics often have rules with side conditions on possibly-read and possibly-written variable sets, and reasoning about concurrent processes is dramatically simplified if one can establish that none of them may write a variable which another may read.

Effect systems are static analyses that compute upper bounds on the possible side-effects of computations. The literature contains many effect systems that analyse which storage cells may be read and which storage cells may be written (as well as many other properties), but few satisfactory accounts of the semantics of this information, or of the

uses to which it may be put. Note that because effect systems *over*-estimate the possible side-effects of expressions, the information they capture is of the form that particular variables will definitely *not* be read or will definitely *not* be written. But what does that mean?

Thinking operationally, it may seem entirely obvious what is meant by saying that a variable X will not be read (written) by a command C, viz. no execution trace of C contains a read (resp. write) operation to X. But such intensional interpretations of program properties are over-restrictive, cannot be interpreted in a standard semantics, do not behave well with respect to program equivalence or contextual reasoning and are hard to maintain during transformations. Thus we seek extensional properties that are more liberal than the intensional ones yet still validate the transformations or reasoning principles we wish to apply.

We begin by defining a simple language with global integer references and describe an effect typing system allowing us to track reading to and writing from individual locations. We then state a list of effect-dependent program equivalences whose correctness with respect to observational equivalence we then embark on proving. To do this, we develop a relational semantics which models effects as sets of relations that are preserved by computations exhibiting that effect. The more side effects the fewer relations are preserved. In particular, if an operation may read location ℓ then only those relations R for which sRs' implies $s(\ell) = s'(\ell)$ can be preserved. If an operation writes ℓ then only those relations R for which sRs' implies $s[\ell{:=}n]Rs'[\ell{:=}n]$ for all n can be preserved. The relational semantics then defines a partial equivalence relation between values of the same given type which is shown to imply observational equivalence and at the same time to include the equational theory generated by our list of effect-dependent program transformations which therefore are valid with respect to observational equivalence.

We then extend this basic framework to encompass dynamically allocated references, recursive definitions, references of structured and even functional types (the latter two not contained in these lecture notes, though). Each of these extensions requires new methods such as domains, partial bijections, Kripke logical relations, which are of independent interest. With dynamically allocated references manifest effects can sometimes be discounted from the analysis on the grounds that they affect only "private" portions of the store, a phenomenon known as *effect masking*. We thus also explain effect masking semantically and show how it helps us to justify more program transformations.

These notes are based on joint work with Lennart Beringer, Nick Benton, and Andrew Kennedy; a large portion of the material is from our joint publications [5,4]; Quasi-PERS in the context of logical relations appear here for the first time.

2. Syntax

Types are given by the following grammar:

$$A, B := \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{ref} \mid A \times B \mid A \to B$$

We assume an infinite supply $\mathbb{L}$ of locations ranged over by ℓ, possibly decorated, and an infinite supply of variables ranged over by x, y, z possibly decorated. In concrete examples we may also use other identifiers for variables.

Terms (e) are given by the grammar:

$$e ::= x \mid n \mid \ell \mid \texttt{true} \mid \texttt{false} \mid e_1\ op\ e_2 \mid () \mid (e_1, e_2) \mid e.1 \mid e.2 \mid e_1\ e_2 \mid$$
$$\lambda x.e \mid \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid\ !e \mid e_1 := e_2$$

Here n ranges over integer constants and op ranges over a suitable set of binary operators including arithmetic operations and comparisons.

The expression $!e$ denotes the value contained in the reference (denoted by) e and $e_1 := e_2$ denotes assignment. The type $\texttt{unit}$ has exactly one element denoted $()$. The type $A \times B$ is the cartesian product of A and B, its elements are pairs (x, y) where $x : A$ and $y : B$. The components of such a pair are accessed with the projections .1 and .2. In examples, we also use products with more than one two factors whose components are then accessed with .1, .2, .3, etc. We use the abbreviation

$$e_1; e_2 \stackrel{def}{=} \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2$$

when x is not free in e_2.

2.1. Example programs

The following example programs illustrate the language concepts; we give them here with their types informally anticipating the typing rules from the next subsection.

$$\mathit{ASSIGNER} \stackrel{def}{=} \lambda x.\ell := x : \texttt{int} \to \texttt{unit}$$

This assigns a given value to the fixed location ℓ.

$$\mathit{COUNTER} \stackrel{def}{=} \lambda x.(\lambda u.x := !x + 1, \lambda u.!x, \lambda u.x := 0) :$$
$$\texttt{ref} \to (\texttt{unit} \to \texttt{unit}) \times (\texttt{unit} \to \texttt{int}) \times (\texttt{unit} \to \texttt{unit})$$

The function *COUNTER* takes a reference as an argument and returns a "counter object" comprising methods for incrementing, getting the current value of, and resetting that reference.

$$\begin{aligned}
\mathit{MEMO} \stackrel{def}{=}\ & \lambda l_1.\lambda l_2.\lambda f. \\
& l_1 := 0; l_2 := f(0); \\
& \lambda x.\texttt{if}\ x = !l_1\ \texttt{then}\ !l_2\ \texttt{else} \\
& \quad \texttt{let}\ u \Leftarrow f\ x\ \texttt{in}\ l_1 := x; l_2 := u; u : \\
& \texttt{ref} \to \texttt{ref} \to (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}
\end{aligned}$$

A memo functional. It takes two references and a function f as arguments. It returns a function f' which does the same as f but is arguably more efficient: f' saves the last argument it has been called with in reference l_1 and puts the corresponding f-value in l_2. Thus, if f' is called several times with the same argument in a row then only the first time a possibly expensive call to f is launched.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash \ell : \mathtt{ref}}$$

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 \mathit{\ op\ } e_2 : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash e.1 : A} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash e.2 : B}$$

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma, x{:}A \vdash e_2 : B}{\Gamma \vdash \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2 : B}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : A} \qquad \frac{\Gamma \vdash e : \mathtt{ref}}{\Gamma \vdash !e : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{ref} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 {:=} e_2 : \mathtt{unit}}$$

Figure 1. Typing rules without effects

2.2. Typing rules

A typing context Γ binds variables to types, we may write it in the form

$$\Gamma := x_1 : A_1, \ldots, x_n : A_n$$

Then $\mathrm{dom}(\Gamma) \stackrel{def}{=} \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = A_i$.

A typing judgement is an assertion of the form $\Gamma \vdash e : A$ where Γ binds the free variables of e. It is inductively defined by the typing rules in Figure 1.

3. Semantics

We do not give evaluation rules for references but instead show how, even in the presence of references, terms and functions can be understood as mathematical functions. This kind of giving semantics is known as denotational semantics. Note that, for the sake of simplicity, our language does not contain recursion, hence all programs terminate. This simplification allows us to use ordinary total functions on sets rather than continuous functions on Scott domains or similar. It is perfectly possible to add terminating loops such as for-loops, which again we refrain from doing, this time only to keep the syntax small.

We model states as functions from locations to integer values and assign to each type A a set $[\![A]\!]$ by the following clauses:

$$\begin{aligned}
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![\mathtt{unit}]\!] &= \{()\} \\
[\![\mathtt{bool}]\!] &= \{\mathtt{true}, \mathtt{false}\} \\
[\![A \times B]\!] &= [\![A]\!]\} \times [\![B]\!] \\
[\![A \to B]\!] &= [\![A]\!] \Rightarrow T([\![B]\!]) \\
T(X) &= \mathbb{S} \Rightarrow \mathbb{S} \times X \\
\mathbb{S} &= \mathbb{L} \Rightarrow \mathbb{Z}
\end{aligned}$$

The last three clauses deserve some explanation: For sets X, Y the set $X \Rightarrow Y$ comprises all functions from X to Y. If there were no references we could simply put $[\![A \to B]\!] =$

$$[\![x]\!]\eta\ s = (s, \eta(x))$$
$$[\![c]\!]\eta\ s = (s, c)$$
$$[\![(e_1, e_2)]\!]\eta\ s = (s_2, (v_1, v_2)) \text{ where } (s_1, v_1) = [\![e_1]\!]\eta\ s, (s_2, v_2) = [\![e_2]\!]\eta\ s_1$$
$$[\![e_1\ e_2]\!]\eta\ s = v_1\ v_2\ s_2 \text{ where } (s_1, v_1) = [\![e_1]\!]\eta\ s, (s_2, v_2) = [\![e_2]\!]\eta\ s_1$$
$$[\![\lambda x.e]\!]\eta\ s = (s, f) \text{ where } f(v) = [\![e]\!]\eta[x \mapsto v]$$
$$[\![\texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2]\!]\eta\ s = [\![e_2]\!]\eta[x \mapsto v_1]\ s_1 \text{ where } (s_1, v_1) = [\![e_1]\!]\eta\ s$$
$$[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\eta\ s = [\![e_2]\!]\eta\ s_1 \text{ when } [\![e_1]\!]\eta\ s = (s_1, \texttt{true})$$
$$[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\eta\ s = [\![e_3]\!]\eta\ s_1 \text{ when } [\![e_1]\!]\eta\ s = (s_1, \texttt{false})$$
$$[\![!e]\!] = s_1(\ell) \text{ where } (s_1, \ell) = [\![e]\!]\eta\ s$$
$$[\![e_1 := e_2]\!]\eta\ s = (s_2[\ell \mapsto v_2], ()) \text{ where } (s_1, \ell) = [\![e_1]\!]\eta\ s, (s_2, v_2) = [\![e_2]\!]\eta\ s_1$$

Figure 2. Selected semantic equations defining runtime behaviour

$[\![A]\!] \Rightarrow [\![B]\!]$. But a term such as $\lambda x.\ell := x$ does not merely return a value but also has a side effect, namely to assign the argument to the reference ℓ. Semantically, this is modelled by the operator $T(-)$, a so-called *monad*. Here $T(X)$ is simply an abbreviation for the set of functions that explain how to get from a given state the new state and the value. Accordingly, elements of $T([\![A]\!])$ will also serve as denotations of (as yet to be evaluated) terms of type A. Variables of type A, on the other hand, will always have values in $[\![A]\!]$ because we assume a call-by-value semantics whereby an expression must be fully evaluated before its value can be bound to a variable. Of course, when binding a function to a variable, side effects may be encapsulated in that function as "latent effects" which only come to bear when the function is evaluated. Accordingly, an element of, say $[\![\texttt{unit} \to \texttt{unit}]\!]$ does in general refer to the state.

Let Γ be a type context. An environment for Γ is a function η that maps each variable $x \in \mathrm{dom}(\Gamma)$ to an element $\eta(x) \in [\![\Gamma(x)]\!]$. If now $\Gamma \vdash e : A$ then we define an element

$$[\![e]\!]\eta \in T([\![A]\!])$$

by the clauses in Figure 2. Note that $[\![e]\!]\eta$ is not an element of $[\![A]\!]$ itself because the evaluation of e may cause side effects and its value may depend on the state. The missing clauses are left as exercises.

A-normal form By introducing additional let-expressions it is possible to transform any term into a term in which the term formers $(-,-), .1, .2$, application, $!(-), -:=-$, are applied to variables only and such that moreover the guard of a case distinction is a variable. For example, the term $g(!x)(!f\ y)$ becomes

$$\begin{array}{l}\texttt{let}\ u \Leftarrow !x\ \texttt{in}\\ \texttt{let}\ l \Leftarrow f\ y\ \texttt{in}\\ \texttt{let}\ v \Leftarrow !l\ \texttt{in let}\ h \Leftarrow g\ u\ \texttt{in}\ h\ v\end{array}$$

$$(\lambda x.e)v \stackrel{sem}{=} e[x := v]$$
$$(v_1, v_2).1 \stackrel{sem}{=} v_1$$
$$(v_1, v_2).2 \stackrel{sem}{=} v_2$$
$$v \stackrel{sem}{=} () \text{ if } v : \texttt{unit}$$
$$v \stackrel{sem}{=} (v.1, v.2) \text{ if } v : A \times B$$
$$v \stackrel{sem}{=} \lambda x.v\ x \text{ if } v : A \to B$$
$$\texttt{if } v \texttt{ then } (\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2) \texttt{ else } e_2 \stackrel{sem}{=} \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2$$
$$(\texttt{let } x \Leftarrow e_1 \texttt{ in } e_2).1 \stackrel{sem}{=} \texttt{let } x \Leftarrow e_1 \texttt{ in } e_2.1$$

Figure 3. Semantically valid equations

This form has been called administrative normal form (ANF, A normal form) in [7]. If we assume that terms are in ANF the semantic equations and also typing rules can be considerably simplified. For example, we have

$$[\![(x, y)]\!]\eta\ s = (s, (\eta(x), \eta(y)))$$
$$[\![x{:=}y]\!]\eta\ s = (s[\eta(x){\mapsto}\eta(y)], ())$$

and so forth. We henceforth assume all terms to be in ANF except in concrete examples.

The ANF also appears in the compilation of functional languages and is closely related to static single assignment (SSA) form known from intermediate language used by compilers.

4. Effect-independent equivalences

If two terms have equal semantics they can be replaced by one another. The equations in Figure 4 hold in this sense where v, v_1, v_2 are *values*, i.e., terms obtained by the following grammar:

$$v ::= x \mid \ell \mid c \mid \lambda x.e \mid (v_1, v_2) \mid v.1 \mid v.2 \mid \texttt{if } v \texttt{ then } v_1 \texttt{ else } v_2$$

If v is a value then $[\![v]\!]\eta\ s = (s, w)$ for some semantic value w depending only on v and η but not on s. There are several more such generally valid equations involving "let" and "if". We remark that it is possible to give a complete set of equations that characterise semantic equality assuming that expressions may have arbitrary side effects [3]. Here, however, we are interested in equivalences that might not hold in general, but do hold under extra assumptions on the kinds of side effects that could possibly happen.

5. Monads and metalanguage

The expressions on the left-hand side of the semantic equations (Fig. 2) look very much like terms of a programming language themselves even though they are meant to be "plain English". One can formalise this and accordingly introduce a *metalanguage* [17] which then allows one to interpret various concrete languages with different features. Indeed, this was one of the initial motivations for the design of the programming language ML (Meta Language) [10].

Such a metalanguage is then a simply-typed lambda calculus with an additional type former T which can be instantiated according to the kinds of side effects offered by the concrete language to be interpreted. This type former T must come equipped with constructs `val` and `let` governed by the following typing rules.

$$\frac{\Gamma \vdash e_1 : T(A_1) \quad \Gamma, x{:}A_1 \vdash e_2 : T(A_2)}{\Gamma \vdash \mathtt{let}\, x \Leftarrow e_1\, \mathtt{in}\, e_2 : A_2} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{val}\, e : T(A)}$$

Thus the definition of a monad comprises the operator T itself as well as "let" and "val", just as a group is not only the underlying set but also the multiplication operation. Again, just as groups the monads admit an equational axiomatisation. Of course, additional type and term formers depending on the particular instance are needed. In our case this is the type `ref` and the operations for reading and writing.

$$\mathtt{read} : \mathtt{ref} \to T(\mathtt{int})$$

$$\mathtt{write} : \mathtt{ref} \to \mathtt{int} \to T(\mathtt{unit})$$

To model exceptions one would use $T(A) = A \overset{+}{\cup} \mathtt{exn}$ where `exn` is a set modelling a basic type of exceptions. One then introduces constants

$$\mathtt{throw} : \mathtt{exn} \to T(A)$$

$$\mathtt{catch} : (\mathtt{exn} \times T(A)) \to T(A)$$

Function types in the metalanguage are always pure thus do a priori not have side effects. The function space in the concrete language is then rendered as $A \to T(B)$.

The pure programming language Haskell allows one to define one's own monads and boasts syntactic abbreviations which create the illusion of working in a side-effecting concrete language where in fact one always works with a pure meta language.

The type structure of the meta language is somewhat richer than that of the concrete language in that the latter has no pendant of types like $\mathtt{int} \to \mathtt{int}$ which denote (in the meta language!) side-effect-free functions. More one this topic can be found in [3].

6. Effects

Our goal is to employ *refined* types to gain information about the nature of side effects possibly occurring during the evaluation of a term *statically* that is to say without knowing the environment in which the term is to be evaluated.

For each location $\ell \in \mathbb{L}$ we introduce the type $\mathtt{ref}_\ell$ which contains precisely that one reference: a *singleton type*. Furthermore, for each $\ell \in \mathbb{L}$ we introduce the two *elementary effects* rd_ℓ (reading) and wr_ℓ (writing). An *effect* is then a set of elementary effects.

The refined types are given by the grammar

$$A, B := \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{ref}_\ell \mid A \times B \mid A \xrightarrow{\varepsilon} B$$

where $\ell \in \mathbb{L}$ and ε is an effect. The *refined typing judgement* takes the form

$$\Gamma \vdash e : A, \varepsilon$$

where now Γ maps variables to refined types. The refined typing judgement says that the evaluation of e in an environment that respects the refined typing in Γ will yield a result in A and moreover exhibit at most the side effects declared in ε. Later on we will provide rigorous definitions of "respects" and "exhibit" allowing us to justify effect-dependent program transformations.

The effect in $A \xrightarrow{\varepsilon} B$ is often called a *latent effect* for it will be brought to bear not immediately but only once a function of that type is being evaluated.

We abbreviate $A \xrightarrow{\emptyset} B$ by $A \to B$ and we abbreviate $\Gamma \vdash e : A, \emptyset$ by $\Gamma \vdash e : A$. Also, we may elide the empty context.

For example, for arbitrary $\ell \in \mathbb{L}$ we have the following refined typings.

$$\vdash \ell{:=}9 : \mathtt{unit}, \{wr_\ell\}$$

$$\vdash \ell{:=}!\ell : \mathtt{unit}, \{rd_\ell, wr_\ell\}$$

$$\vdash \lambda x.\ell{:=}x : \mathtt{int} \xrightarrow{\{wr_\ell\}} \mathtt{unit}$$

$$\vdash \mathit{COUNTER} : \mathtt{ref}_\ell \to (\mathtt{unit} \xrightarrow{\{rd_\ell, wr_\ell\}} \mathtt{unit}) \times (\mathtt{unit} \xrightarrow{\{rd_\ell\}} \mathtt{int}) \times (\mathtt{unit} \xrightarrow{\{wr_\ell\}} \mathtt{unit})$$

$$\vdash \mathit{MEMO} : \mathtt{ref}_{\ell_1} \to \mathtt{ref}_{\ell_2} \to (\mathtt{int} \xrightarrow{\varepsilon_1} \mathtt{int}) \xrightarrow{\varepsilon_2} \mathtt{int} \xrightarrow{\varepsilon_3} \mathtt{int}$$

In the last typing ε_1 is arbitrary, $\varepsilon_2 = \varepsilon_1 \cup \{wr_{\ell_1}, wr_{\ell_2}\}$ and $\varepsilon_3 = \varepsilon_1 \cup \{rd_{\ell_1}, wr_{\ell_1}, rd_{\ell_2}, wr_{\ell_2}\}$. This last example shows that effect annotation can only conservatively approximate the actual behaviour of a program. A computation *MEMO* $\ell_1\ \ell_2\ f\ v$ may or may not exhibit the effect ε_1 depending on whether or not the required f-value can be looked up or not. The second example is also interesting in that, semantically, the term in question is side-effect free, but our type system has no way of discovering this. However, we could use our semantics to justify a stronger type system that would ascribe a pure typing to the term $\ell{:=}!\ell$.

7. Effect system

We now give typing rules that formally define the refined typing judgment. We assume ANF, i.e., the typing rules apply to terms in ANF which will save a considerable amount of repetition. Recall that $\Gamma \vdash A$ abbreviates $\Gamma \vdash A, \emptyset$, etc. Also some trivial rules, e.g., for arithmetic operators are omitted. The rules are in Figure 4. The last three rules in Figure 4 define and use an auxiliary subtyping relation that allows one to weaken the refined typing. so as to obtain common refined typings e.g. in two branches of a conditional. For example, without subtyping we would not be able to assign a type to $\mathtt{if}\ x\ \mathtt{then}\ \lambda y.y\ \mathtt{else}\ \lambda y.!\ell$.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma(x) = \mathtt{ref}_\ell \quad \Gamma(y) = \mathtt{int}}{\Gamma \vdash x{:=}y : \mathtt{unit}, \{wr_\ell\}} \qquad \frac{\Gamma(x) = \mathtt{ref}_\ell}{\Gamma \vdash !x : \mathtt{int}, \{rd_\ell\}}$$

$$\frac{\Gamma \vdash e_1 : A_1, \varepsilon_1 \quad \Gamma, x{:}A_1 \vdash e_2 : A_2, \varepsilon_2}{\Gamma \vdash \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2 : A_2, \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Gamma(x) = A_1 \xrightarrow{\varepsilon} A_2 \quad \Gamma(y) = A_1}{\Gamma \vdash x\ y : A_2, \varepsilon}$$

$$\frac{\Gamma \vdash e_1 : A, \varepsilon_1 \quad \Gamma \vdash e_2 : A, \varepsilon_2 \quad \Gamma(x) = \mathtt{bool}}{\Gamma \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : A, \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Gamma, x{:}A \vdash e : B, \varepsilon}{\Gamma \vdash \lambda x.e : A \xrightarrow{\varepsilon} B, \emptyset}$$

$$\frac{\Gamma \vdash e : A_1, \varepsilon_1 \quad A_1 <: A_2 \quad \varepsilon_1 \subseteq \varepsilon_2}{\Gamma \vdash e : A_2, \varepsilon_2}$$

$$\frac{}{A <: A} \qquad \frac{A_1 <: A_2 \quad A_3 <: A_4 \quad \varepsilon_1 \subseteq \varepsilon_2}{A_2 \xrightarrow{\varepsilon_1} A_3 <: A_1 \xrightarrow{\varepsilon_2} A_4}$$

Figure 4. Typing rules for effect typing

Effect polymorphism and type inference In general, any given term will have infinitely many different types that are mutually incomparable in the subtyping relation. The information about a term that can be gleaned through the typing rules thus comprises an infinite set. In order to analyse programs automatically one will thus use a finitary notation for such infinite families of types. It is common to employ type schemes like in the Hindley-Milner type inference for the simply-typed lambda calculus and used in the ML programming language.

These type schemes then contain type variables, effect variables, and location variables. In addition type schemes may contain constraints stipulating inclusion of certain effects and difference of locations.

The types for the running examples given above can be considered as instances of such type schemes if one interprets the metavariables ε, ℓ, etc. as actual variables.

Just as in the case of ML type inference one will allow schematic types for let-bound variables which can then be instantiated in different ways in the body. Thus, the following term cannot be typed with the rules given above but can be typed with the help of type schemes:

$$\mathtt{let}\ f \Leftarrow \lambda x.x{:=}0\ \mathtt{in}\ (f\ \ell_1); (f\ \ell_2)$$

with ℓ_1 and ℓ_2 two different locations.

Using type schemes one would assign the type $\mathtt{ref}_\ell \xrightarrow{wr_\ell} \mathtt{unit}$ which can be instantiated with $\ell = \ell_1$ and $\ell = \ell_2$. This is particularly important in the case of recursive definitions "let rec" where, incidentally, ML does not allow multiple instantiations in the body. For any given term there then exists a most general schematic type which can be efficiently computed with an appropriate extension of the Hindley-Milner inference algorithm. We will not consider the area of automatic type inference in these notes; for more information see [9,14,24] which are the standard references on effect typing.

8. Effect-based program transformation

We will now employ effect information in order to justify program transformations. Here is a first example of such a program transformation.

Dead code elimination Suppose that a term contains a subterm e which admits the typing $\Gamma \vdash e : \mathtt{unit}, \varepsilon$ in its context where $\{\ell \mid wr_\ell \in \varepsilon\} = \emptyset$ then e can be replaced by $()$, i.e., removed. In what sense is such replacement admissible? In any case the semantics of the two terms are in general not equal. We shall answer that question in the next section.

To facilitate the formulation of further equivalences we introduce the following notations:

$$\mathrm{rds}(\varepsilon) = \{\ell \mid rd_\ell \in \varepsilon\}$$
$$\mathrm{wrs}(\varepsilon) = \{\ell \mid wr_\ell \in \varepsilon\}$$

Here are further transformations:

Code motion: Suppose that two terms e_1, e_2 admit the following typings in their context: $\Gamma \vdash e_1 : A, \varepsilon_1$ and $\Gamma \vdash e_2 : A, \varepsilon_2$ where $\mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset$, $\mathrm{wrs}(\varepsilon_1) \cap \mathrm{rds}(\varepsilon_2) = \emptyset$, and $\mathrm{wrs}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset$. Then the term (e_1, e_2) (i.e., $\mathtt{let}\ x_1 \Leftarrow e_1\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow e_2\ \mathtt{in}\ (x_1, x_2)$ in ANF) may be replaced with $\mathtt{let}\ x_2 \Leftarrow e_2\,\mathtt{in}\mathtt{let}\ x_1 \Leftarrow e_1\ \mathtt{in}\ (x_1, x_2)$. Along with the general rule that semantically equal terms may be replaced by one another this means that the order of evaluation of e_1 and e_2 may be exchanged.

Duplicated code Suppose that the term e can be typed as $\Gamma \vdash e : A, \varepsilon$ in its context where $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$. Then (e, e) may be replaced with $\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ (x, x)$. This then means that the duplicate execution of e can be contracted to a single one.

Pure Lambda Hoist Suppose that a term e can be typed as $\Gamma \vdash e : A$ in its context; thus, e is *pure* (free of side effects). Then the following two terms can be replaced by one another:

$$\lambda x.\mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ e'(x, y)$$
$$\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \lambda y.e'(x, y)$$

This means that side-effect-free computations that do not depend on formal parameters can be extracted ("hoisted") from a function (method) body and evaluated once and for all in advance.

We note that in general the typing assumptions made in the context are crucial for the required typings to hold. Here is an example of this situation:

$$f : \mathtt{unit} \xrightarrow{\varepsilon} A \vdash f(\ell{:=}1) : A, \varepsilon \cup \{wr_\ell\}$$

Thus *Duplicated Computation* applies to the term $f(\ell{:=}1)$ provided that $\mathrm{rds}(\varepsilon) \cap (\mathrm{wrs}(\varepsilon) \cup \{\ell\}) = \emptyset$. Thus, the effect typing allows one to add information about side effects to the specification of a function (method), here f, and to use that information at the call sites of the function for the purposes of program transformation.

Note that the semantics of $\mathtt{let}\ x \Leftarrow f(\ell{:=}1)\ \mathtt{in}\ (x, x)$ and $(f(\ell{:=}1), f(\ell{:=}1))$ are different because the semantics does not model effect annotations. Refining the semantics so that it does allow to capture such information is the main goal of this work as are extensions to a richer language of course.

Before, however, refining the semantics, we must make it clear in what sense we want program equivalences to hold.

9. Observational equivalence

We choose to justify program equivalences as observational equivalence, i.e., we will show that two terms deemed equivalent can be replaced by one another within any closed term of basic type. Formally,

Definition 9.1. Let v_1 and v_2 be closed and pure terms of some refined type A, i.e.,

$$\vdash v_1 : A$$
$$\vdash v_2 : A$$

Then v_1 and v_2 are *observationally equivalent at type A*, written,

$$\models v_1 \equiv v_2 : A$$

if for all closed and pure terms $v : A \xrightarrow{\varepsilon} \texttt{bool}$ with ε arbitrary one has that whenever

$$[\![v\ v_1]\!](s_0) = (s_1, b_1)$$
$$[\![v\ v_2]\!](s_0) = (s_2, b_2)$$

then $b_1 = b_2$. Here s_0 is the initial state defined for example by $s_0(\ell) = 0$ for all ℓ.

The term v thus represents the observation made about v_1 and v_2. The state reached after the observation is discarded, only the boolean result is retained.

However, in our situation the following is the case: if $\models v_1 = v_2 : A$ and $\vdash v : A \xrightarrow{\varepsilon} \texttt{bool}$ and

$$[\![v\ v_1]\!](s_0) = (s_1, b_1)$$
$$[\![v\ v_2]\!](s_0) = (s_2, b_2)$$

then $s_1 = s_2$ follows, too. To see this, fix $\ell \in \mathbb{L}$ and suppose that $s_1(\ell) = c$. Define another observation $v' := \lambda x.v\ x; (!\ell{=}{=}c)$ where == is an equality test. From the observational equivalence of v_1 and v_2 applied to v' it can then be concluded that $s_2(\ell) = c$, too. Note that this would not be so if local references not visible from the outside are admitted.

By a similar argument it follows that observational equivalence would stay the same if observations of integer type or multiple observations were allowed.

Definition 9.2. Two terms e_1, e_2, where $x_1{:}A_1, \ldots, x_n{:}A_n \vdash e_1, e_2 : A, \varepsilon$ are observationally equivalent at A, ε if $\lambda x_1 \ldots \lambda x_n.\lambda y.e_i$ for $i = 1, 2$ are observationally equivalent at $A_1 \to \cdots \to A_n \to \texttt{unit} \xrightarrow{\varepsilon} A$.

Notice that observational equivalence is always type-dependent.

Proposition 9.1. *If two terms have equal semantics then they are observationally equivalent at any type. The converse is in general not true.*

This is because the definition of observational equivalence refers to terms only via their semantics.

10. Relational semantics

Before we embark on the formal definition let us informally motivate the concept of modelling effects as sets of relations to be preserved. Assume that we have only two locations X and Y, i.e. $\mathbb{L} = \{X, Y\}$ and let $c : \mathbb{S} \to \mathbb{S}$ be (a denotation of) a command. How can we formalise that c

- may read X but not Y
- may write Y but not X

Note that c, being a denotation, we do not have access to its execution trace. One obvious attempt at giving such a formalisation would be to require that there exists a function $f : \mathbb{Z} \to \mathbb{Z}$ such that

$$c(s).X = s.X$$

$$c(s).Y = s.Y \vee c(s).Y = f(s.X)$$

and the decision in the disjunction only depends on $s.X$

In order to formalise the latter restriction we might additionally require the existence of a function $g : \mathbb{Z} \to \{0, 1\}$ such that

$$c(s).Y = g(s.X) \cdot s.Y + (1 - g(s.X)) \cdot f(s.X)$$

Let us temporaily say that c has property *Direct* if such functions f, g exist.

Alternatively, we can formalise the intended effect property by requiring that for all relations $R \subseteq \mathbb{S} \times \mathbb{S}$ *compatible with these effects* we have

$$\forall s, s'.sRs' \Rightarrow c(s)Rc(s')$$

where R is "compatible with these effects" if

- whenever sRs' then $s.X = s'.X$ (because we may read X)
- whenever sRs' and $v \in \mathbb{Z}$ then $s[Y \mapsto v]Rs'[Y \mapsto v]$ (because we may write Y)

Let us temporarily say that command c has property *Relational* if this is the case. One can now show that the two properties *Direct* and *Relational* are equivalent. The property *Direct* has the advantage of being (perhaps) more intuitive; the formulation *Relational*, on the other hand, eases compositional reasoning; e.g. it is straightforward that if c satisfies *Relational* so does the composition $c; c$. Furthermore, *Relational* is easy to generalise to more complicated sets of effects: A command may exhibit some effect ε if it preserves all store relations that are compatible with that effect; the more effects are exhibited the fewer relations are to be preserved.

The relational semantics of effects has the additional advantage that it generalises to a symmetric, transitive relation in the case where commands return values with nontrivial observational equivalence.

We will now formalise this notion and extend it to all types. This will result in a refined relational semantics that distinguishes refined types with equal underlying type and in particular validates the above program equivalences. Of course, this semantics should not be coarser than observational equivalence; as we shall see this will be a consequence of the relational semantics being compositionally defined and nontrivial (not all values receive the same denotation).

Definition 10.1. A binary relation R on a set A is a *partial equivalence relation* (PER) if R is symmetric and transitive. The *support* of R is the set $\mathrm{supp}(R) = \{x \mid xRx\}$. The restriction of R to $\mathrm{supp}(R)$ is an equivalence relation. Thus, a PER corresponds to a partition of a subset of A into classes.

For each refined type A we define the underlying simple type $|A|$ in the obvious way by removing all effect information. This notation also applies to typing contexts and judgements.

It is our goal to define a PER $[\![A]\!]$ on $[\![|A|]\!]$ for each type A such that the support $\mathrm{supp}([\![A]\!])$ singles out those elements of $[\![|A|]\!]$ that respect the effect information contained in A. On this support, the equivalence relation $[\![A]\!]$ should refine observational equivalence and identify at least those elements whose equivalence can be deduced from the equational theory generated by our program equivalences and congruence rules.

As already mentioned in the introduction we describe effects of computations by sets of state relations to be preserved.

Definition 10.2. For each effect ε we define a set of relations $\mathcal{R}_\varepsilon$ on states as follows:

$$\begin{aligned}
R \in \mathcal{R}_\emptyset &\iff R \subseteq \mathbb{S} \times \mathbb{S} \\
R \in \mathcal{R}_{\varepsilon_1 \cup \varepsilon_2} &\iff R \in \mathcal{R}_{\varepsilon_i} \text{ for } i = 1, 2 \\
R \in \mathcal{R}_{rd_\ell} &\iff \forall s\ s'.sRs' \Rightarrow s.\ell = s'.\ell \\
R \in \mathcal{R}_{wr_\ell} &\iff \forall s\ s'\ v.sRs' \Rightarrow s[\ell \mapsto v]Rs'[\ell \mapsto v]
\end{aligned}$$

Definition 10.3 (relational semantics)**.** The definition of the relational semantics is then given as follows.

$$\begin{aligned}
(v, v') \in [\![A]\!] &\iff v = v' \text{ and } A \in \{\mathtt{bool}, \mathtt{unit}, \mathtt{int}\} \\
(v, v') \in [\![\mathtt{ref}_\ell]\!] &\iff v = v' = \ell \\
((v_1, v_2), (v_1', v_2')) \in [\![A_1 \times A_2]\!] &\iff (v_i, v_i') \in [\![A_i]\!] \text{ for } i = 1, 2 \\
(f, f') \in [\![A \xrightarrow{\varepsilon} B]\!] &\iff \forall (v, v') \in [\![A]\!].(f\ v, f'\ v') \in T_\varepsilon([\![B]\!]) \\
(f, f') \in T_\varepsilon(A) &\iff \forall R \in \mathcal{R}_\varepsilon.\forall s, s'.sRs' \Rightarrow s_1 R s_1' \wedge vAv' \\
&\qquad \text{where } f(s) = (s_1, v), f'(s') = (s_1', v')
\end{aligned}$$

Consider that $f \in T([\![\mathtt{int}]\!])$ and that $(f, f) \in T_{\{rd_\ell\}}([\![\mathtt{int}]\!])$. Let $R \in \mathcal{R}_{\{rd_\ell\}}$ be given by

$$sRs' \iff s(\ell) = s'(\ell)$$

We then find that if sRs' and $f(s) = (s_1, v), f(s') = (s_1', v')$ then in particular $v = v'$ so the result depends only on ℓ. Now fix s and consider the relation $R' \in \mathcal{R}_{\{rd_\ell\}}$ given by

$$s'R's'' \iff s' = s'' = s$$

Thus, if $f(s) = s_1, v$ then $s_1 R s_1$ so $s_1 = s$, i.e., f did indeed not write.

By experimenting a bit more one sees that the relational semantics does indeed capture our intuitions about reading and writing. For example, if $(f, f) \in T_{\{rd_\ell, wr_{\ell'}\}}$ then we can show by similar arguments that f will modify at most the ℓ' component of the store and if it does so then to a value that depends only on ℓ, etc.

We can now state and prove the fundamental lemma which essentially asserts the soundness of our effect typing rules with respect to our relational semantics. More precisely, it says that the effect information obtained syntactically does indeed adequately describe the semantic behaviour of the term. The proof of the fundamental lemma is by induction on typing derivations and does not present any surprises.

Theorem 10.1 (fundamental lemma)**.** *Suppose that* $\Gamma \vdash e : A, \varepsilon$ *and* $(\eta(x), \eta'(x)) \in [\![\Gamma(x)]\!]$ *for all* $x \in \mathrm{dom}(\Gamma)$. *Then* $([\![e]\!]\eta, [\![e]\!]\eta') \in T_\varepsilon([\![A]\!])$.

Theorem 10.2 (observational equivalence)**.** *Let* e, e' *be terms with* $\Gamma \vdash e : A, \varepsilon$ *and* $\Gamma \vdash e' : A, \varepsilon$. *For all* η, η' *with* $(\eta(x), \eta'(x)) : [\![\Gamma(x)]\!]$ *for* $x \in \mathrm{dom}(\Gamma)$ *suppose that* $([\![e]\!]\eta, [\![e]\!]\eta') \in T_\varepsilon([\![A]\!])$.

This is proved by applying the fundamental lemma to the observation and using the fact that the relational semantics at base types is equality.

Theorem 10.3 (Program equivalences)**.** *The abovementioned program equivalence "dead code" is semantically valid in the following sense: If* $\Gamma \vdash e : \mathtt{unit}, \varepsilon$ *and* $\mathrm{wrs}(\varepsilon) = \emptyset$ *and* $(\eta(x), \eta'(x)) : [\![\Gamma(x)]\!]$ *for all* $x \in \mathrm{dom}(\Gamma)$ *then* $([\![e]\!]\eta, [\![()]\!]\eta') \in T_\varepsilon([\![\mathtt{unit}]\!])$. *Analogous statements hold for the other equivalences , "code motion", "duplicated code", "pure lambda hoist".*

More generally: if the equivalence of two terms e, e' where $\Gamma \vdash e : A, \varepsilon$ and $\Gamma \vdash e' : A, \varepsilon$ is derivable using equational reasoning (with reflexivity and congruence rules restricted to well-typed terms) from the four program equivalences, and universally valid semantic equivalences (terms with equal denotational semantics ($[\![-]\!]$) are equivalent) then for η, η' as above it holds that $([\![e]\!]\eta, [\![e']\!]\eta') \in T_\varepsilon([\![A]\!])$.

11. Dynamic allocation

We add a new basic type `ref` and a new term former $\mathtt{ref}(-)$ such that $\mathtt{ref}(e) : \mathtt{ref}$ when $e : \mathtt{int}$. The idea is that $\mathtt{ref}(e)$ generates a new reference initialised with the value of the variable e.

This constructs permits more elegant versions of our example programs:

$$\begin{aligned}
\mathit{COUNTER} &\stackrel{def}{=} \mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ (\lambda u.x{:=}!x+1, \lambda u.!x, \lambda u.x{:=}0) : \\
&\quad (\mathtt{unit} \to \mathtt{unit}) \times (\mathtt{unit} \to \mathtt{int}) \times (\mathtt{unit} \to \mathtt{unit}) \\
\mathit{MEMO} &\stackrel{def}{=} \mathtt{let}\ l_1 \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \mathtt{let}\ l_2 \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \lambda f. \\
&\quad l_1{:=}0; l_2{:=}0; \\
&\quad \lambda x.\mathtt{if}\ x = !l_1\ \mathtt{then}\ !l_1\ \mathtt{else} \\
&\qquad \mathtt{let}\ u \Leftarrow f\ x\ \mathtt{in}\ l_1{:=}x; l_2{:=}u; u : \\
&\quad (\mathtt{int} \to \mathtt{int}) \to \mathtt{int} \to \mathtt{int}
\end{aligned}$$

In order to model this semantically, we assume a set $\mathbb{S}$ of states endowed with the following operations: There is a constant $\emptyset \in \mathbb{S}$, the empty state. If $s \in \mathbb{S}$ then $\mathrm{dom}(s) \subseteq \mathbb{L}$ and if $\ell \in \mathrm{dom}(s)$ then $s.\ell \in \mathbb{Z}$ is a value; if $v \in \mathbb{Z}, \ell \in \mathrm{dom}(s)$ then $s[\ell{\mapsto}v] \in \mathbb{S}$; finally $new(s, v)$ yields a pair (ℓ, s') where $\ell \in \mathbb{L}$ and $s' \in \mathbb{S}$. These operations are subject to the following axioms:

$$\begin{aligned}
&\mathrm{dom}(\emptyset) = \emptyset \\
&\mathrm{dom}(s[\ell{\mapsto}v]) = \mathrm{dom}(s) \\
&(s[\ell{\mapsto}v]).\ell' = \textit{if}\ \ell = \ell'\ \textit{then}\ v\ \textit{else}\ s.\ell' \\
&new(s, v) = (\ell, s') \Rightarrow \mathrm{dom}(s') = \mathrm{dom}(s) \cup \{\ell\} \wedge \\
&\qquad \ell \notin \mathrm{dom}(s) \wedge s'.\ell = v
\end{aligned}$$

This abstract datatype can be implemented in a number of ways, e.g., as finite maps. We do not want to commit ourselves to any particular implementation, in particular, we do not make the perhaps plausible assumption that the newly allocated reference ℓ_0 when $new(s, v) = (s', \ell_0)$ depends only on $\mathrm{dom}(s)$.

Assuming ANF we then put

$$\begin{aligned}
[\![!x]\!]\eta\ s &= s.\eta(x) \\
[\![x{:=}y]\!]\eta\ s &= s[\eta(x){\mapsto}\eta(y)] \\
[\![\mathtt{ref}(x)]\!]\eta\ s &= new(s, \eta(x))
\end{aligned}$$

The other semantic equations remain unchanged.

12. Refined typing with regions

We will now extend refined typing to dynamic allocation. To that end we partition the allocated memory area into disjoint regions. These regions are not reflected physically at runtime; they merely play a role in the type system. Thus, the denotational semantics is not affected by the regions in any way.

We assume an infinite supply *Regs* of region (identifiers) and define refined types as follows:

$$A, B := \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{ref}_{\mathtt{r}} \mid A \times B \mid A \xrightarrow{\varepsilon} B$$

where an effect ε is a subset of the set of *elementary effects*

$$\{rd_{\mathsf{r}}, wr_{\mathsf{r}}, al_{\mathsf{r}} \mid \mathsf{r} \in Regs\}$$

Thus, accesses to individual references are approximated by accesses to regions. Furthermore, a new elementary effect al_{r} signalling an allocation within region r is introduced. The refined type $\mathtt{ref}_{\mathsf{r}}$ represents the set of locations within region r.

The typing rules are analogous to the global case. For example, we can deduce:

$$\mathit{COUNTER} : (\mathtt{unit} \overset{\{rd_{\mathsf{r}}, wr_{\mathsf{r}}\}}{\longrightarrow} \mathtt{unit}) \times (\mathtt{unit} \overset{\{rd_{\mathsf{r}}\}}{\longrightarrow} \mathtt{int}) \times (\mathtt{unit} \overset{\{wr_{\mathsf{r}}\}}{\longrightarrow} \mathtt{unit})$$

where r is an arbitrary region. We give explicitly the rules for allocation and for writing:

$$\frac{\Gamma(x) = \mathtt{int}}{\Gamma \vdash \mathtt{ref}(x) : \mathtt{ref}_{\mathsf{r}} : \{al_{\mathsf{r}}\}} \qquad \frac{\Gamma(x) = \mathtt{ref}_{\mathsf{r}} \quad \Gamma(y) = \mathtt{int}}{\Gamma \vdash x{:=}y : \mathtt{unit}, \{wr_{\mathsf{r}}\}}$$

Note that the choice of region r in the rule for allocation is arbitrary. Of course, when typing a program, we will try to use as many regions as possible so as to maximise the applicability of program transformations which will for example require that simultaneous write accesses happen in different regions.

With integer references only it happens rarely that we are not able to spend a different region on every single allocation made. Once we have structured data like lists and recursion, even primitive recursion, it will no longer be possible to do so.

An altogether new feature is the following *masking rule*

$$\frac{\Gamma \vdash e : A, \varepsilon \quad \mathsf{r} \text{ does not occur in } \Gamma \text{ or } \tau}{\Gamma \vdash e : A, \varepsilon \setminus \{wr_{\mathsf{r}}, rd_{\mathsf{r}}, al_{\mathsf{r}}\}}$$

This rule allows one to delete effects concerning a region r that is mentioned neither in the types of the free variables nor in the type of the result. If, for instance, e is a closed expression of type $\mathtt{int}$ which internally uses one or more instances of *COUNTER*, then e can a priori be typed as

$$\vdash e : \mathtt{int}, \{rd_{\mathsf{r}}, wr_{\mathsf{r}}, al_{\mathsf{r}}\}$$

with r an arbitrary region. The masking rule then allows us to derive the pure typing $\vdash e : \mathtt{int}$.

We will now extend the relational semantics to this situation and in particular show that effect-dependent program equivalences continue to hold in the presence of masking. Thus, the above term e could be evaluated several times instead of once, etc. without altering the semantic meaning in the sense of observational equivalence.

We remark that this region-based type system was originally introduced by Tofte and Talpin in order to enable block-structured memory management without garbage collection. The idea is to type a program (automatically) in the region type system with aggressive use of masking. At runtime masked regions are then deallocated (freed) after the masked expression has been evaluated. We remark that unlike in our application this does require a certain amount of runtime support in the form of a table that records which physical locations belong to which region. In our example the *COUNTER* objects would be deallocated once the result has been computed.

Tofte and Talpin employ this kind of memory management in their ML-Kit compiler. More recently, region-based memory management has appeared (for obvious reasons) in Real-Time Java (www.rtsj.org).

12.1. Formal preliminaries

Definition 12.1 (Quasi PER)**.** A binary relation R on a set A is a *Quasi PER* (QPER) if whenever xRy and zRy and zRw, then xRw, too. In other words, $RR^{-1}R = R$.

QPERs have been introduced as "difunctional relations" in a 1940s paper and appear from time to time in the literature. The terminology QPER is nonstandard.

If R is a QPER define $R_1 = RR^{-1}$ and $R_2 = R^{-1}R$. Both R_1 and R_2 are PERs and R defines a bijection between their respective sets of equivalence classes.

Definition 12.2. If R is any binary relation on a set A we denote $\mathit{QPER}(R)$ the least QPER containing R.

The following characterisation of $\mathit{QPER}(R)$ is interesting but will not be needed in the sequel.

Proposition 12.1. *Let R be a binary relation on a set A. One has*

$$(x, x') \in \mathit{QPER}(R) \iff$$
$$\forall k, k' : A \Rightarrow \{0,1\}.(\forall y, y'.yRy' \Rightarrow ky = k'y') \Rightarrow kx = k'x'$$

In the previous sections we approximated contextual equivalence by a partial equivalence relation $[\![A]\!]$ for each refined type A. In the presence of dynamic allocation such a simple-minded setup will no longer work and we move to families of relations (known as "Kripke logical relation") indexed by state layouts which we refer to as *parameters*. Parameters introduce (a) a 'representation independence' for state, capturing the fact that behaviour is invariant under permutation of locations; and (b) a distinction between observable and non-observable locations, as expressed syntactically by the masking rule.

Furthermore, the relations $[\![A]\!]_\varphi$ for φ a parameter will not be PERs but only QPERs.

Lemma 12.2. *Let R, S be binary relations on sets A, B, let Q be a QPER on C and $f, f' : A \times B \to C$ be functions. If*

$$\forall a, a', b, b'.aRa' \wedge bSb' \Rightarrow f(a,b)Qf'(a',b')$$

then

$$\forall a, a', b, b'.a\mathit{QPER}(R)a' \wedge b\mathit{QPER}(S)b' \Rightarrow f(a,b)Qf'(a',b')$$

Proof. Fix a, a' such that aRa' and form $U = \{(b, b') \mid f(a,b)Qf'(a',b')\}$.

The assumption yields $S \subseteq U$ so, since U is a QPER (!), we obtain

$$\forall a, a', b, b'.aRa' \wedge b\mathit{QPER}(S)b' \Rightarrow f(a,b)Qf'(a',b')$$

The claim now follows using the QPER $\{(a, a') \mid \forall b, b'.b\mathit{QPER}(S)b' \Rightarrow f(a,b)Qf'(a',b')\}$. □

12.2. Partial bijections

The relational interpretation of types will depend on parameters that approximate the current store layout. The central ingredient of parameters are partial bijections which we now define. They describe the locations which belong to a given region in two computations that are deemed equivalent.

Definition 12.3 (partial bijection)**.** A *partial bijection* is a triple (L, L', f) where L, L' are finite subsets of $\mathbb{L}$ and $f \subseteq L \times L'$ such that $(l_1, l_1') \in f$ and $(l_2, l_2') \in f$ imply $l_1 = l_2 \Leftrightarrow l_1' = l_2'$.

If $t = (L, L', f)$ is a partial bijection, we write $\mathrm{dom}(t) = L, \mathrm{dom}'(t) = L'$ and refer to f simply by t itself. We let (ℓ, ℓ') denote the partial bijection $(\{\ell\}, \{\ell'\}, \{(\ell, \ell')\})$, and let $\emptyset$ denote the empty partial bijection.

Two partial bijections t_1, t_2 are *disjoint* if $\mathrm{dom}(t_1) \cap \mathrm{dom}(t_2) = \emptyset$ and $\mathrm{dom}'(t_1) \cap \mathrm{dom}'(t_2) = \emptyset$. In this case, we write $t_1 \otimes t_2$ for the partial bijection given by

$$\begin{aligned} t_1 \otimes t_2 = (&\mathrm{dom}(t_1) \cup \mathrm{dom}(t_2), \\ &\mathrm{dom}'(t_1) \cup \mathrm{dom}'(t_2), \\ &t_1 \cup t_2) \end{aligned}$$

Partial bijections are ordered as follows: $t' \geq t$ if and only if $t' = t \otimes t''$ for some (uniquely determined) t''.

12.3. Parameters

When modelling global store we approximated observational equivalence by a partial equivalence relation $[\![A]\!]$ for each refined type A. In the presence of dynamic allocation such a simple-minded setup will no longer work and we move to families of relations indexed by store layouts which we refer to as *parameters*. Parameters introduce (a) a 'representation independence' for state, capturing the fact that behaviour is invariant under permutation of locations; and (b) a distinction between observable and non-observable locations, as expressed syntactically by the masking rule. Aspect (a) of parameters is expressed by assigning to each region identifier a partial bijection between locations in the store.

For aspect (b) of parameters we introduce a special symbol $\tau \notin \mathit{Regs}$ to represent the part of the store arising from regions "masked out" by the masking rule. Commands must not alter this portion of the store at all. We will thus sometimes refer to τ as the *silent* region. This intended meaning will become clear subsequently; for now, τ is just a symbol.

We are now ready to give a formal definition of parameters.

Definition 12.4 (parameter)**.** A *parameter* φ is a function that assigns to every region r (including the silent region) a partial bijection $\varphi(\mathsf{r})$ such that

- distinct regions map to disjoint partial bijections.
- $\mathrm{dom}(\varphi) = \bigcup_{\mathsf{r} \in \mathit{Regs} \cup \{\tau\}} \mathrm{dom}(\varphi(\mathsf{r}))$ and $\mathrm{dom}'(\varphi) = \bigcup_{\mathsf{r} \in \mathit{Regs} \cup \{\tau\}} \mathrm{dom}'(\varphi(\mathsf{r}))$ are both finite sets so that in fact $\varphi(\mathsf{r}) = \emptyset$ for all but finitely many regions r.

If φ and φ' are parameters such that

$$\mathrm{dom}(\varphi) \cap \mathrm{dom}(\varphi') = \mathrm{dom}'(\varphi) \cap \mathrm{dom}'(\varphi') = \emptyset$$

then φ, φ' are called *disjoint* and we write $\varphi \otimes \varphi'$ for the obvious juxtaposition of φ and φ' given by $(\varphi \otimes \varphi')(\mathsf{r}) = \varphi(\mathsf{r}) \otimes \varphi'(\mathsf{r})$.

Whenever we write $\varphi \otimes \psi$ then φ and ψ are presumed to be disjoint from each other so, a statement like $\exists \psi. \ldots \varphi \otimes \psi \ldots$ is understood as "there exists ψ disjoint from φ such that $\ldots \varphi \otimes \psi \ldots$".

The set of parameters is partially ordered by $\varphi' \geq \varphi \iff \varphi' = \varphi \otimes \psi$ for some necessarily unique ψ.

If t is a partial bijection then $[\mathsf{r} \mapsto t]$ is the parameter such that $\varphi(\mathsf{r}) = t, \varphi(\mathsf{r}') = \emptyset$ when $\mathsf{r}' \neq \mathsf{r}$.

Thus, if $\ell \notin \mathrm{dom}(\psi)$ and $\ell' \notin \mathrm{dom}'(\psi)$ then we can form $\psi \otimes [\mathsf{r} \mapsto (\ell, \ell')]$ to add the link (ℓ, ℓ') to r in ψ. Similarly, if $\mathsf{r} \notin \mathrm{dom}(\varphi)$, we can form $\varphi \otimes [\mathsf{r} \mapsto \emptyset]$ to initialise a new region r with $\emptyset$.

We let $\varphi - \mathsf{r}$ denote the parameter defined by

$$(\varphi - \mathsf{r})(\mathsf{r}') = \varphi(\mathsf{r}') \text{ when } \mathsf{r}' \neq \mathsf{r}$$
$$(\varphi - \mathsf{r})(\tau) = \varphi(\tau) \otimes (\mathrm{dom}(\varphi(\mathsf{r})), \mathrm{dom}'(\varphi(\mathsf{r})), \emptyset)$$

Definition 12.5 (state relations)**.** If L, L' are sets of locations, a *state relation* on L, L' is defined as a nonempty relation $R \subseteq \mathbb{S} \times \mathbb{S}$ such that whenever $(s, s') \in R$ and $s \sim_L s_1$ and $s' \sim_{L'} s_1'$ then $(s_1, s_1') \in R$, too. We write $\mathrm{StRel}(L, L')$ for the set of all state relations on L, L'.

Given such a relation, we now formalize what it means to 'respect' an effect ε under some parameter φ.

Definition 12.6 (relations and effects)**.** Let R be a state relation on $\mathrm{dom}(\varphi)$, $\mathrm{dom}'(\varphi)$. We say that R *respects* ε *at* φ if it is preserved by all commands that exhibit only ε on the state layout delineated by φ. Formally, we define:

- R respects $\{rd_{\mathsf{r}}\}$ at φ if $(s, s') \in R$ implies $s.\ell = s'.\ell'$ for all $(\ell, \ell') \in \varphi(\mathsf{r})$;
- R respects $\{wr_{\mathsf{r}}\}$ at φ if for all $(s, s') \in R$ and for all $(\ell, \ell') \in \varphi(\mathsf{r})$ and $v \in \mathbb{Z}$, we have $(s[\ell \mapsto v], s'[\ell' \mapsto v]) \in R$;
- R respects $\{al_{\mathsf{r}}\}$ always.

We then define the set $\mathcal{R}_\varepsilon(\varphi)$ of all store relations that respect ε at φ as follows:

$$\mathcal{R}_\varepsilon(\varphi) = \{R \in \mathrm{StRel}(\mathrm{dom}(\varphi), \mathrm{dom}'(\varphi)) \mid \forall e \in \varepsilon, R \text{ resp. } e \text{ at } \varphi\}.$$

Unfortunately, we cannot track the allocation effect with relations; this will be done separately in the definition of the monad.

Finally, we introduce two additional bits of notation. If $s, s' \in \mathbb{S}$ we define

$$s, s' \models \varphi \iff \mathrm{dom}(s) = \mathrm{dom}(\varphi) \wedge \mathrm{dom}(s') = \mathrm{dom}'(\varphi)$$

We also define the following:

$$s \sim_\varphi s' \iff \forall \mathsf{r} \in \mathit{Regs}.\, \forall (\ell, \ell') \in \varphi(\mathsf{r}).\ s.\ell = s'.\ell'$$

13. Logical Relation

This section defines the relational semantics of refined types.

Definition 13.1 (logical relation)**.** Let A be a refined type and φ be a parameter. We define a QPER $[\![A]\!]_\varphi$ on $[\![|A|]\!]$ by the following clauses.

$$\begin{aligned}
[\![A]\!]_\varphi &\equiv \{(v,v) \mid v \in [\![|A|]\!]\} \text{ when } A \in \{\mathtt{int}, \mathtt{bool}, \mathtt{unit}\} \\
[\![\mathtt{ref}_\mathsf{r}]\!]_\varphi &\equiv \varphi(\mathsf{r}) \\
[\![A \times B]\!]_\varphi &\equiv [\![A]\!]_\varphi \times [\![B]\!]_\varphi \\
[\![A \xrightarrow{\varepsilon} B]\!]_\varphi &\equiv \{(f,f') \mid \forall\, \varphi' \geq \varphi. \forall (x,x') \in [\![A]\!]_{\varphi'}. \\
&\qquad\qquad (f(x), f'(x')) \in (T_\varepsilon [\![B]\!])_{\varphi'}\} \\
(T_\varepsilon Q)_\varphi &\equiv \mathit{QPER}(\{(f,f') \mid s, s' \models \varphi \Rightarrow \\
&\qquad \forall\, R \in \mathcal{R}_\varepsilon(\varphi). s\ R\ s' \Rightarrow s_1\ R\ s_1' \ \wedge \\
&\qquad \exists \psi. (\psi(\mathsf{r}) \neq \emptyset \Rightarrow \mathsf{r} \in \mathrm{als}(\varepsilon)) \ \wedge s_1, s_1' \models \varphi \otimes \psi \wedge \\
&\qquad s_1 \sim_\psi s_1' \wedge (v,v') \in Q_{\varphi \otimes \psi} \\
&\qquad \text{where } (s_1, v) {=} f\ s \text{ and } (s_1', v') {=} f'\ s'\})
\end{aligned}$$

We define $[\![\Theta]\!]_\varphi$ by $[\![\Theta]\!]_\varphi \equiv \{(\gamma,\gamma') \mid \forall\, i.\ (\gamma(x_i), \gamma'(x_i)) \in [\![A_i]\!]_\varphi\}$ where $\Theta = x_1 : A_1, \ldots, x_n : A_n$.

The definition of the logical relation on computation types deserves some explanation. First, it says that the store behaviour of two related computations must respect all relations that are compatible with the declared effect. Since these relations are completely unconstrained on the silent region τ, this implies in particular that the silent region may neither be read nor modified. The existential quantifier asserts a (disjoint) extension ψ of the current parameter φ which is to hold all newly allocated references. The result values (v, v') are then required to be related with respect to the extended parameter $\varphi \otimes \psi$. Note that if v and v' contain newly allocated references then $(v, v') \in [\![B]\!]_\varphi$ will in general not hold.

The semantics of value types is monotonic with respect to the ordering on parameters.

Lemma 13.1 (Monotonicity)**.** *If $\varphi' \geq \varphi$ then $[\![A]\!]_{\varphi'} \supseteq [\![A]\!]_\varphi$.*

Lemma 13.2 (QPER)**.** *For each φ the relation $[\![A]\!]_\varphi$ is a QPER.*

Lemma 13.3 (masking)**.** *Suppose that r does not occur anywhere in A. Then $[\![A]\!]_\varphi = [\![A]\!]_{\varphi - \mathsf{r}}$.*

Lemma 13.4 (extension)**.**

$$\mathcal{R}_{\varepsilon - \mathsf{r}}(\varphi) = \mathcal{R}_\varepsilon(\varphi \otimes [\mathsf{r} \mapsto \emptyset])$$

The following establishes semantic soundness for our subtyping relation.

Lemma 13.5 (Soundness of subtyping)**.** *If $A_1 \leq A_2$ then for all φ one has $[\![A_1]\!]_\varphi \subseteq [\![A_2]\!]_\varphi$.*

We now have the following 'fundamental theorem' of logical relations, which states that terms are related to themselves.

Theorem 13.6 (Fundamental Theorem)**.** *If $\Gamma \vdash e : A, \varepsilon$ and $(\gamma, \gamma') \in [\![\Gamma]\!]_\varphi$, then*

$$([\![e]\!]\gamma, [\![e]\!]\gamma') \in T_\varepsilon([\![A]\!])_\varphi.$$

Proof. The proof is by induction on typing derivations; thanks to Lemma 12.2 we can do as if the *QPER*$(-)$ closures were absent in applications of the induction hypothesis. We now give the cases for the typing rule for "`let` " and for the masking rule.

Case "`let` ": Here e is $\texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2$ and we have $\Gamma \vdash e_1 : A_1, \varepsilon_1$ and $\Gamma, x{:}A_1 \vdash e_2 : A : \varepsilon_2$, so $e = \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2$ and $\varepsilon = \varepsilon_1 \cup \varepsilon_2$.

Let $(\gamma, \gamma') \in [\![\Gamma]\!]_\varphi$ and $R \in \mathcal{R}_\varepsilon(\varphi)$ and sRs' and $s, s' \models \varphi$ and $(\check{s}_1, \check{v}) = [\![e_1]\!]\gamma\ s$ and $(\check{s}'_1, \check{v}') = [\![e_1]\!]\gamma'\ s'$. By the induction hypothesis applied to e_1 and the aforementioned use of Lemma 12.2 we have $s_1 R s'_1$ and $\check{s}_1, \check{s}'_1 \models \varphi \otimes \check{\psi}$ and $(\check{v}_1, \check{v}'_1) \in [\![A_1]\!]_{\varphi\otimes\check{\psi}}$.

By monotonicity we have $(\gamma, \gamma') \in [\![\Gamma]\!]_{\varphi\otimes\check{\psi}}$ and so $(\gamma[x{\mapsto}\check{v}], \gamma'[x{\mapsto}\check{v}']) \in [\![\Gamma, x{:}A_1]\!]_{\varphi\otimes\check{\psi}}$. We conclude with the induction hypothesis applied to e_2.

Case "masking rule": Suppose $\Gamma \vdash e : A, \varepsilon$ and $\mathsf{r} \notin \Gamma, A$. Suppose $(\gamma, \gamma') \in [\![\Gamma]\!]_{\varphi_0}$. Put $f = [\![e]\!]\gamma, f' = [\![e]\!]\gamma', \varphi = \varphi_0 - \mathsf{r}$. By the masking lemma we have $(\gamma, \gamma') \in [\![\Gamma]\!]_\varphi$.

Let us apply IH(e) to $\varphi \otimes [\mathsf{r}{\mapsto}\emptyset]$. We obtain $(f, f') \in T_\varepsilon([\![A]\!])_{\varphi\otimes[\mathsf{r}\mapsto\emptyset]}$.

We should now prove $(f, f') \in T_{\varepsilon-\mathsf{r}}([\![A]\!])_\varphi$ whence $(f, f') \in T_{\varepsilon-\mathsf{r}}([\![A]\!])_{\varphi_0}$ by masking lemma again.

So assume $s, s' \models \varphi$ and $R \in \mathcal{R}_{\varepsilon-\mathsf{r}}(\varphi)$ and sRs'. By the extension lemma we have $R \in \mathcal{R}_\varepsilon(\varphi \otimes [\mathsf{r}{\mapsto}\emptyset])$ so the induction hypothesis gives $s_1 R s'_1$ and ψ such that $s_1, s'_1 \models \varphi \otimes [\mathsf{r}{\mapsto}\emptyset] \otimes \psi$ and $(v, v') \in [\![A]\!]_{\varphi\otimes[\mathsf{r}\mapsto\emptyset]\otimes\psi}$ where $f\ s = (s_1, v)$ and $f'\ s' = (s'_1, v')$.

The masking lemma gives $(v, v') \in [\![A]\!]_{\varphi\otimes(\psi-\mathsf{r})}$ and we are done. □

14. Applications

We introduce the notation

$$s \sim_{\mathrm{rds}_\varphi(\varepsilon)} s' \iff \forall \mathsf{r}{\in}\mathrm{rds}(\varepsilon).\forall(\ell, \ell'){\in}\varphi(\mathsf{r}).s.\ell = s'.\ell'$$

It expresses that s and s' agree on those locations that are read given effect ε.

We also define

$$\begin{aligned} nwrs_\varphi(\varepsilon) &= \mathrm{dom}(\varphi) \setminus \textstyle\bigcup_{\mathsf{r}\in\mathrm{wrs}(\varepsilon)} \mathrm{dom}(\varphi(\mathsf{r})) \\ nwrs'_\varphi(\varepsilon) &= \mathrm{dom}'(\varphi) \setminus \textstyle\bigcup_{\mathsf{r}\in\mathrm{wrs}(\varepsilon)} \mathrm{dom}'(\varphi(\mathsf{r})) \end{aligned}$$

Thus $nwrs_\varphi(\varepsilon)$ and $nwrs'_\varphi(\varepsilon)$ comprise the locations on the left (resp. right) side that are not written to, given effect ε. This includes the locations in the silent region.

Lemma 14.1. *Suppose* $\Gamma \vdash e : T_\varepsilon A$ *and* $(\gamma, \gamma') \in [\![\Gamma]\!]_\varphi$ *and* $s_0, s_0' \models \varphi$ *and* $[\![e]\!]\gamma s_0 = (s_1, x)$ *and* $[\![e]\!]\gamma' s_0' = (s_1', x')$.

If $s_0 \sim_{\mathrm{rds}_\varphi(\varepsilon)} s_0'$ *then there exists* ψ *with* $\psi(\mathsf{r}) \neq \emptyset \Rightarrow \mathsf{r} \in \mathrm{als}\varepsilon$ *and disjoint from* s_0, s_0' *such that*

1. $s_1, s_1' \models \varphi \otimes \psi$ *and* $(x, x') \in [\![A]\!]_{\varphi\otimes\psi}$ *and* $s_1 \sim_\psi s_1'$.
2. $s_0 \sim_{nwrs_\varphi(\varepsilon)} s_1$ *and* $s_0' \sim_{nwrs'_\varphi(\varepsilon)} s_1'$.
3. *For each* $(\ell, \ell') \in \varphi(\mathsf{r})$ *where* $\mathsf{r} \in$ *Regs we have either*
   - $s_0.\ell = s_1.\ell$ *and* $s_0'.\ell' = s_1'.\ell'$ *(unchanged) or*
   - $s_1.\ell = s_1'.\ell'$ *(identically written).*
4. *Suppose that* $\ell \in \mathrm{dom}(\varphi)$ *but there is no* ℓ', r *such that* $(\ell, \ell') \in \varphi(\mathsf{r})$. *Then* $s_0.\ell = s_1.\ell$. *A symmetric statement holds for* s_0', s_1'.

Notice that Part 2 asserts in particular that the contents of the silent region do not change from s_0 to s_1.

Definition 14.1 (semantic equality)**.** Suppose that $\Gamma \vdash e_i : A, \varepsilon$ for $i = 1, 2$. We write $\Gamma \models e_1 = e_2 : A, \varepsilon$ to mean that for all φ and $(\gamma, \gamma') \in [\![\Theta]\!]_\varphi$ one has $([\![e_1]\!]\gamma, [\![e_2]\!]\gamma') \in [\![A]\!]_\varphi$.

Proposition 14.2. *If* $\Gamma \models e_1 = e_2 : A, \varepsilon$ *then* e_1 *and* e_2 *are observationally equivalent.*

Proposition 14.3. *If* $\Gamma \models e_1 = e_2 : A, \varepsilon$ *and* $\Gamma \models e_2 = e_3 : A, \varepsilon$ *then* $\Gamma \models e_1 = e_3 : A, \varepsilon$.

Proof. Suppose that $(\gamma, \gamma') \in [\![\Gamma]\!]\varphi$. We should prove $([\![e_1]\!]\gamma, [\![e_3]\!]\gamma') \in T_\varepsilon([\![A]\!])_\varphi$.

We have $([\![e_i]\!]\gamma, [\![e_i]\!]\gamma') \in T_\varepsilon([\![A]\!])_\varphi$ for $i = 1, 2, 3$ by the Fundamental Lemma. The assumption gives $([\![e_i]\!]\gamma, [\![e_{i+1}]\!]\gamma') \in T_\varepsilon([\![A]\!])_\varphi$ for $i = 1, 2$. We conclude by QPER-ness. □

Similarly, we can show that semantic equality is symmetric and transitive on well-typed terms and that it is a congruence with respect to all term formers.

We can now state our program equivalences in the form of semantic equalities.

Proposition 14.4 (duplicated computation)**.** *Suppose that* $\Gamma \vdash e : A, \varepsilon$ *and suppose that* $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \mathrm{als}(\varepsilon) = \emptyset$. *Thus,* e *reads and writes on disjoint portions of the store and makes no allocations except possibly in the silent region. Then* $\Gamma \models e_1 = e_2 : A, \varepsilon$ *where*

$$e_1 := \texttt{let}\ x \Leftarrow e\ \texttt{in}\ \texttt{val}\ (x, x)$$
$$e_2 := \texttt{let}\ x \Leftarrow e\ \texttt{in}\ \texttt{let}\ y \Leftarrow e\ \texttt{in}\ \texttt{val}\ (x, y)$$

Analogous statements hold for dead code, pure lambda hoist, commuting computations.

We remark that the program equivalences we get for pure computations are complete in the following sense:

Proposition 14.5. *Let* C *be a cartesian closed category and* T *be a strong monad on* C. *Suppose that in the Kleisli category* C_T *the laws of dead computation, commuting computations, duplicated computations, lambda hoist are valid. Then* C_T *is cartesian closed.*

In particular, the Kleisli category consisting of computations of type $T_\emptyset(A)$ modulo contextual equivalence is cartesian closed.

15. Recursion

In order to model recursion we associate a Scott domain (in fact ω-chain complete partial order with bottom would suffice) with every type and in particular define $[\![A \xrightarrow{\varepsilon} B]\!]$ as the domain of strict, continuous maps from $[\![A]\!]$ to $T[\![B]\!]$ where TX is the domain of strict maps from the flat domain of states to $\mathbb{S} \otimes X$ where $\otimes$ denotes strict product. I.e., a computation either does not terminate (is bottom) or returns a pair consisting of a new state and a (non-bottom) value. Recursive definitions can then be interpreted in the usual way as suprema of Kleene chains.

One must then make sure that all semantic relations $[\![A]\!]_\varphi$ are admissible QPERS, ie contain $(\bot, \bot)$ and are closed under suprema of chains in the sense that if $(v_i, v_i') \in [\![A]\!]_\varphi$ then $(\bigsqcup_i v_i, \bigsqcup_i v_i') \in [\![A]\!]_\varphi$, too. This is achieved by adding appropriate clauses at base types and adapting the closure operator $\mathit{QPER}(R)$ so as to yield the least admissible QPER comprising R.

Unfortunately, the program equivalence "dead code" breaks down in the presence of recursions since the elided code fragment might not terminate. One can fix this by introducing a termination condition as a semantic side condition to be discharged either by semantic reasoning or by some other type system, e.g. by introducing a "nontermination effect". Similar considerations apply to pure lambda hoist.

16. Conclusion

We have given a relational semantics to a region-based effect type system for a higher-order language with dynamically allocated store. The relational semantics is shown sound for contextual equivalence and thus provides a powerful proof principle for the latter. We have used the semantics to establish the soundness of a collection of useful effect-based program transformations. It would probably be very hard to establish these directly from the definition of contextual equivalence and no such proof appears to exist in the literature.

There has been a great deal of previous work on the soundness of region-based memory management and of its close cousin, encapsulated monadic state, as provided by `runST` in Haskell [12]. We mention some particularly relevant references. Banerjee et al. [2] translate the region calculus into a variant of System F and give a denotational model showing that references in masked regions do not affect the rest of the computation. Moggi and Sabry [18] prove syntactic type soundness for encapsulated lazy state. Fluet and Morrisett [8] bring the two lines of work together by giving a type- and behaviour-preserving translation from a variant of the region calculus into an extension of System F with a region-indexed family of monads. Naumann [19] uses simulation relations to capture a notion of observational purity for boolean-valued specifications that allows mutation of encapsulated state.

The general problem of modelling and proving equivalences in languages with dynamically allocated store and higher order features is a difficult one, with a very long

history [25]. The basic techniques we use here, such as partial bijections and parametric logical relations, have been developed and refined over the last 25 years or so [11,15,20,21,22,6]. The focus of much of this previous work has been on showing tricky equivalences between particular pairs of terms, such as the well-known Meyer-Sieber examples [15]. One might expect that equivalences justified by simple program analyses, such as those considered here, would generally be much easier to establish than some of the more contorted examples from the literature. Whilst this is broadly true – our relational reasoning technique is far from complete, yet suffices for establishing the interesting equational consequences of the effect system – completely generic reasoning is surprisingly difficult. When proving concrete equivalences one treats the context generically, but has two particular, literal terms in one's hand, whose denotations one can work with explicitly. In the case of purely type-based equivalences, on the other hand, both the context and the terms are abstract; all one knows are the types, and the semantics of those types has to capture enough information to justify all instances of the transformation.

An alternative approach to proving 'difficult' contextual equivalences is to use techniques based on bisimulation. Originally developed for process calculi by Park and Milner [16], bisimulation was adapted for the untyped lambda calculus by Abramsky [1]. Other researchers, particularly Sumii and Pierce, subsequently developed notions of bisimulation for typed lambda calculi that could deal with the kind of encapsulation (data abstraction) given by existential types [23]. These methods have recently been refined by Koutavas and Wand, and applied to an untyped higher-order language with storage [13] and to object-based calculi. It would be extremeSly worthwhile to investigate whether bisimulation methods can be applied to the typed (and, as discussed above, type-directed) impure equivalences studied here.

Like the characterisation of contextual equivalence with logical relations given by Pitts and Stark, this does not directly solve the problem at hand here. Firstly, we use *typed* contextual equivalence which has fewer observing contexts and is thus coarser than untyped contextual equivalence. Indeed, in [13] an extension to the typed case is left as an open question.

Secondly, as already mentioned, our applications concern *relative* contextual equivalences involving unknown programs. It is not yet clear how the bisimulation method fares with those. It is, however, true that it could be interesting and profitable to base our technical development on bisimulations rather than logical relations.

References

[1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Directions in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1988.

[2] A. Banerjee, N. Heintze, and J. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS)*, 1999.

[3] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[4] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effectbased program transformations with dynamic allocation.

[5] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Reading, writing, and relations: Towards extensional semantics for effect analyses. In *4th Asian Symposium on Programming Languages and Systems (APLAS)*, *LNCS*, 2006.

[6] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *LNCS*, 2005.

[7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*. ACM, 1993.

[8] M. Fluet and G. Morrisett. Monadic regions. *Journal of Functional Programming*, 2006. to appear.

[9] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, August 1986.

[10] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[11] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, 1984.

[12] S. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4), 1995.

[13] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higherorder imperative programs. In *POPL*, pages 141–152, 2006.

[14] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symposium on Principles of Programming Languages (POPL)*, 1988.

[15] A. R. Meyer and K. Sieber. Towards a fully abstract semantics for local variables: Preliminary report. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1988.

[16] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[17] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asiloomar, CA*, pages 14–23, 1989.

[18] E. Moggi and A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, 11(6), 2001.

[19] D. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, To appear.

[20] P. W. OŠHearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[21] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

[22] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.

[23] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *POPL Š05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2005.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), June 1984. Revised from LICS 1992.

[25] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Abstract and Concrete Models for Recursion

Martin HYLAND
DPMMS, CMS, Wilberforce Road, Cambridge CB3 0WB, U.K.

Abstract. We present a view of recursion in terms of traced monoidal categories. We sketch relevant abstract mathematics and give examples of the applicability of this point of view to various aspects of modern computer science.

Keywords. Finite automata, Flow diagrams, Fixed points, Categories, Traces

Introduction

I hope that this account of the material from my lectures at the Marktoberdorf 2007 Summer School is sufficiently self-contained to make it possible for those not present to learn from it. There is certainly little overlap with the notes I produced before the lectures; and the relation to the slides which I used for the lectures and which can be found on the Summer School website is not close either. The earlier material contains some inexact formulations, and I have attempted to make things more precise where I can. It is best to think of what is presented here as another reworking of some basic material.

The idea of recursion is central to programming. For example a subroutine may be called many times in the running of a programme: each time it is called it does what is in some sense the same thing (though hopefully to different initial data). The idea that recursion amounts to repeating the same thing over and over is familiar enough from a range of computing practice. These lectures will introduce an abstract mathematical way to think about this basic phenomenon. The examples will probably be quite familiar, but we shall look at them in a new way.

There are many abstract approaches to recursion. I have chosen to concentrate on one which is both abstract and of considerable generality, but which reflects current concrete practice. The focus will be on the idea of what is called a trace on a symmetric monoidal category. The definition is relatively recent: the original paper, treating a more general situation than we need, is [16].

Ideas drawn from abstract mathematics can seem far from concrete practice, but often the apparent distance is illusory. Typical diagrammatic methods in computing (for example wiring diagrams) reflect free categories with structure, and this makes an immediate connection between the concrete and the abstract. (A precise mathematical treatment of relevant notions of free structure is given in [11] and the treatment is further extended in [12].) For the purpose of this paper one can see the situation as follows. On the one hand the abstract notion of trace introduced to analyse recursion can be given

a concrete diagrammatic representation; on the other many diagrams with feedback are best analysed in terms of a trace

The notion of a traced monoidal category is treated in a computer science context in the book [9] by Hasegawa. That also discusses diagrams for which I have no space here. Unfortunately the book is out of print, but some information can be extracted from the author's home page. Another source of ideas but from a different perspective is [23]. The magnum opus [4] by Bloom and Esik deals in effect with traced monoidal categories with good properties, though from a point of view quite different from mine. The wealth of material is daunting but it is a valuable reference. Generally there is nothing in the literature which provides ideal background for what I say here. I have tried to do without too many prerequisites. However there is no avoiding some category theory, and I have had to restrict myself to the merest sketch. Mac Lane's book [20] remains a standard reference both for the basic theory and for relevant material on monoidal categories (chapter VII). For those with no real idea of category theory the gentler introduction in [1] is recommended. The book [2] by Barr and Wells focuses on the needs of computer scientists. A third edition and much electronic material can be found on the web.

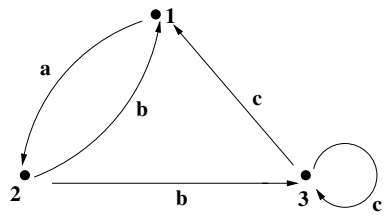
My aim has been to give a treatment of abstract material via examples. So I touch on simple mathematical examples of trace which involve relations, partial functions and permutations. I give a brief indication of how both flow diagrams from basic imperative programming and fixed points in functional programming fit into the view of recursion in terms of traces. But our leading examples will be finite automata and regular languages. This familiar material illustrates very well the flavour of the trace point of view.

There are a range of exercises, and those who hope to learn from this account should regard them as an integral part of the whole. I hope that for the most part they will be accessible to those with little background in abstract mathematics. Their organisation could probably be improved. There seems no point in trying to give an indication of difficulty. Experience at the summer school made it clear that this varied very much according to background, and also (if I may allow myself a tease) according to the standards of proof which people find acceptable.

1. Motivation: Finite Automata and Regular Languages

1.1. An example

Let us start by looking at a typical small finite automaton.



We regard 1 as the initial state and 3 as the terminal state. Then with the usual conventions, the automaton accepts the regular language $a(ba)^*b(c(ab)^*)^*$. This fact is not entirely obvious. It arises from a matrix identity. To be in accord with the usual notation for functions and matrices I write this as

$$\begin{pmatrix} 0 & b & c \\ a & 0 & 0 \\ 0 & b & c \end{pmatrix}^* = \begin{pmatrix} (c^*ba)^* & (c^*ba)^*c^*b & (c^*ba)^*cc^* \\ (ac^*b)^*a & (ac^*b)^* & (ac^*b)^*acc^* \\ ((ba)^*c)^*b(ab)^*a & ((ba)^*c)^*b(ab)^* & ((ba)^*c)^* \end{pmatrix}.$$

The bottom left hand entry, the entry in the $(3,1)$ place, is the reverse of the regular of the regular language accepted. (That we get the reverse just comes from the change of convention.)

What we have here is an example of something familiar to most computer science students: the relation between finite automata and regular languages. Just because of it is so well known[1] it seems best to use this relation as the main focus of these notes. So let us start by reviewing some of this well known material.

1.2. Finite automata

The traditional approach is as follows. One is given a finite alphabet Σ. Then a finite automaton on the alphabet Σ is given by a finite set of states Q and an action $\Sigma \times Q \to Q$. The key issue is what sort of action to take. We shall see that a particularly judicious choice is to take non-deterministic automata and allow internal silent actions or moves: that is, arrows are marked either by a letter of the alphabet Σ or by 1 (or τ in Milner's notation) signifying the silent action.

Definition 1 *A finite non-deterministic automaton on the alphabet Σ is given a finite set of states Q and together with the action which is a relation $(1 + \Sigma) \times Q \to Q$.*

When the action is just a relation $\Sigma \times Q \to Q$, we have the usual notion of non-deterministic automaton. If further such an action is a partial function then the automaton is *deterministic*. The case of automata with silent actions but which are otherwise deterministic is also important. (There is nothing wrong mathematically with the restricted case where the action is total, but total functions generally are difficult from the point of view of recursion.)

Let Σ^* be the set of words in Σ. Mathematically it is the free monoid on Σ (the monoid operation is concatenation) and so one readily extends the action to a monoid action, which is again a relation $\Sigma^* \times Q \to Q$. Given an initial state q_0 and some terminal states t_i we are in the usual situation: we say that a word w is accepted or recognized by the machine if $w.q_0$ is related to some terminal t_i. In this context a set of words is called a language and we consider the languages recognized by finite automata, the *recognizable languages*.

[1] Before the summer school participants are asked to fill in a questionnaire indicating their degree of knowledge of various topics involved in the lectures. This was the one about which most students felt they had good knowledge, and the only one about which all knew something.

1.3. Regular languages

Let Σ be a finite alphabet and Σ^* the collection of finite words from Σ. Alternatively $\Sigma^* = \mathrm{List}(\Sigma)$ is the set of finite lists. We call a subset of Σ^* a language, and now denote such languages by a, b and so on. The collection $P(\Sigma^*)$ of languages has algebraic structure given by the following collection of operations.

- A constant zero 0 which is the empty set of words.
- A binary operation of sum $a + b$ given by the union $a \cup b$.
- A constant, the unit 1, which is the set containing just the empty word.
- A binary operation of multiplication ab given by $\{xy | x \in a \text{ and } y \in b\}$, that is, by elementwise concatenation of words from a and from b
- A unary operation, the star a^* which is the infinite union $1 + a + a^2 + \cdots$, that is, the collection of all finite concatenations of words from a.

We postpone discussion of the properties of these operations to sections 8 and 10.1, but we use them here to define the notion of regular language.

Definition 2 *The family of regular languages or regular events is the least family of languages containing the singleton letters from Σ and closed under the above operations.*

The fact that every regular language is recognizable follows by induction on the basis of the following familiar exercises. (Depending on your background you may or may not be used to using silent actions to help here. If you are not used to this point of view note that the reduction of non-deterministic to deterministic automata using the power set goes through. That is relevant to the exercises below.)

1.4. Kleene's Theorem

In a first year computer science course one is likely to see the following famous result of Stephen Kleene.

Theorem 1 *The languages recognizable by finite automata coincide with the regular languages or regular events.*

One direction of this equivalence, namely that regular languages are recognizable, is in a sense hands on coding[2] and so is generally felt to be the easier. I invite readers to remember their preferred proof in the exercises below. The other direction, that recognizable languages are regular, is usually taken to be the harder direction. In this paper we try to show that the two directions hang together conceptually. To make this clear we need to know some abstract mathematics. In particular we need to know that there are particular traced monoidal categories $\mathbf{Aut}$ of finite automata and $\mathbf{Reg}$ of matrices of regular langauges. Then the fact that recognizable languages are regular arises from the following.

Theorem 2 *The definition of languages by automata is implemented by a traced monoidal functor $\mathbf{Aut} \to \mathbf{Reg}$.*

[2] With our choice of notion of automaton it is very easy. This is not an accident. Think about it!

This would not perhaps be very compelling were it an isolated phenomenon. But it is not. As we say in the Introduction, diagrammatic methods pervade computer science. As well as automata of many kinds, there are circuits, flow diagrams, interactive systems, action structures and even diagrammatic methods in proof theory. One unifying point of view is that to the degree that one can glue diagrams together, one has a categorical composition. Moreover many diagrams permit feedback, giving a form of recursion. One general form of feedback is encapsulated in the idea of a traced monoidal category, and often with suitable modifications (which we illustrate in the case of finite automata) diagrams with feedback can be interpreted as maps in a traced monoidal category. So one can consider traced monoidal categories as a general setting in which to understand feedback. Before giving the mathematical background to this point of view, we briefly consider another example in the next section.

Exercise 1

1. *Show that for any finite set there is a finite automaton which recognizes it.*
2. *Formulate and prove a result to the effect that if there are no loops in an automaton, then the language recognized is finite.*
3. *Show that if a language is recognized by a non-deterministic finite automaton, then there is a deterministic finite automaton which also recognizes it.*
4. *Show that the empty set is a recognizable language and that the recognizable languages are closed under unions.*
5. *Show that the singleton containing the empty word is recognizable, and that the recognizable languages are closed under concatenation: if a and b are recognizable then so is $ab = \{xy | x \in a \text{ and } y \in b\}$.*
6. *Show that if a is recognizable then so is $a^* = \bigcup_{n \geq 0} a^n$ the set of all finite concatenations of words from a.*

2. Motivation: imperative programs and state

2.1. Imperative programming

Kleene's Theorem is concerned with understanding (indeed computing) the result of a certain restricted form of feedback. In this section we consider how a general form of feedback is used to define a general computation processes.

In 1971 when I started as a graduate student in Oxford, my supervisor Robin Gandy taught an undergraduate course in recursion theory using register machines. I think that he had the idea originally from John Shepherdson, but some version had occured to many people independently around 1960. In Gandy's treatment, register machines were officially given by numbered sequences of commands of the forms:

$$x := x + 1 \text{ goto } i\,; \quad \text{if } x = 0 \text{ then goto } i; \text{ else } x := x - 1 \text{ goto } j\,.$$

(Here i and j refer to the next command to be executed.) So computing was explained in terms of a very primitive (one might say basic) kind of imperative programming. But almost immediately Gandy stopped using the official sequence of commands and started using flow diagrams. Here I define what are essentially the flow diagrams for a slight modification of the primitive language for register machines.

We suppose that we are given a finite set of locations, references or registers x, y , z ... . Our flow diagrams will be formed by linking instances of atomic nodes equipped with *input* and *output* ports. Take as the atomic nodes the following:

- for each register x, nodes $x := x + 1$ with one input and one output port;
- for each register x, nodes $x := x - 1$ with one input and one output port;
- for each register x, nodes if $x = 0$ then; else with one input port and two output ports

A flow diagram program is given by a finite collection of instances of such registers together with wirings from instances of output ports to input ports. (Usually it would come with a special start node (instruction) with just one output, and a stop node with just an input., but we would do best in effect to allow a number of start and stop nodes.)

If the flow diagram has the structure of a tree, then intuitively for any starting values in the registers, computation proceeds through the diagram without any return to a node already visited. But cycles in the diagram give feedback loops and so allow real recursion to occur.

2.2. Implementation on states

The standard interpretation of register machines is that they define partial recursive functions. This derives from a reading of machines as operating on states, and as with finite automata, this operation can also be thought of in terms of traced monoidal categories. We give the barest outline of this point of view.

A state is an assignment of natural numbers to registers: supposing there are r registers, we write $S = \mathbb{N}^r$ for the set of states. Let $\mathcal{P}$ be the collection of partial functions $\phi : S \to S$. Take a register machine M with n inputs and m outputs. The operation of the machine gives an $n \times m$ matrix $\Phi_M = (\phi_{ij})$ with entries $\phi_{ij} \in \mathcal{P}$. Here ϕ_{ij} is the partial function on states which arises when we start the machine at unput j and we emerge at output i. It is evident that the matrix has the property that the partial functions in the columns have disjoint domains. It follows that we can consider these matrices as maps in a simple *unique decomposition category*, in the sense of Haghverdi and Scott. (See [7] and [8] for ramifications of the theory of such special traced monoidal categories.) Write $\mathbf{Par}$ for the traced monoidal category just described. Now there is also a traced monoidal category $\mathbf{Flow}$ of flow diagrams, and quite analogous to the situation for finite automata we have the following.

Theorem 3 *The interpretation of register machines as operations on state is implemented by a traced monoidal functor* $\mathbf{Flow} \to \mathbf{Par}$.

A more detailed analysis of flow diagrams in what is essentially the spirit of the above discussion can be found in [21]. There is not space to develop things further. Of course the standard definition of computability by flow diagrama or register machines is well known. So the following exercises are more for contemplation than detailed work.

Exercise 2

1. *Probably you have experience of much more advanced programming, but just for fun write a program in the form of a flow diagram to calculate the Fibonacci sequence. How would you show that it does so? Can you say how many instructions are traversed in the calculation of the nth Fibonacci number?*

2. *Show that in the original programming language with numbered instructions, one can restrict the first (augment the register by one) instruction to the case*

$$i:\ x := x + 1 \ \mathsf{goto}\ i + 1\,.$$

What is one doing in this argument?
3. *Write out a formal explanation of the operation of a flow diagram program as an operation on states?*

3. Background: Elementary Category Theory

3.1. Categories

The definition of a category due to Eilenberg and Mac Lane arose from an attempt to make precise the sense in which the totality of mathematical structures of a given kind, together with the structure-preserving maps between them, can itself be regarded as a mathematical structure in its own right. We refer the reader to [1], to [2] and to [20] for the basic mathematical theory. The account in [2] is particularly directed towards computer science. Here we content ourselves with an informal account.

A category $\mathcal{C}$ consists of a collection $\mathrm{ob}(\mathcal{C})$ of *objects* (here denoted by uppercase letters $U, V, W, X, Y, Z, ...$) and for each pair (X, Y) of objects a collection $\mathcal{C}(X, Y)$ of *morphisms (or arrows or maps) from X to Y* (one writes suggestively $f : X \to Y$ for $f \in \mathcal{C}(X, Y)$); together with

identities $1_X = \mathrm{id}_X \in \mathcal{C}(X, X)$ for each X in $\mathrm{ob}(\mathcal{C})$,
composition $m_{X,Y,Z} = \circ : \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) \to \mathcal{C}(X, Z) \quad (g, f) \to g \circ f = gf,$

satisfying the following axioms:

- if $f \in \mathcal{C}(X, Y)$ then $f \circ 1_X = f$ and $1_Y \circ f = f$;
- $h \circ (g \circ f) = (h \circ g) \circ f$ whenever $f \in \mathcal{C}(X, Y)$, $g \in \mathcal{C}(Y, Z)$ and $h \in \mathcal{C}(Z, W)$.

It is good to have a range of examples in mind. The original motivation comes from large categories. First there are categories of sets: the category $\mathbf{Sets}$ of sets itself is the basic example, but one has also categories which give the Boolean-valued models for set theory, and at yet a further level of generality toposes [15]. Then there are categories of algebras: familiar examples are the categories of groups and of rings; more generally one has the category of T-algebras for a monad (sometimes called a triple) T. Then there are categories of spaces: the most familiar is $\mathbf{Top}$, the category of topological spaces; but there are many other notions of space, for example $[\Delta^{\mathrm{op}}, \mathbf{Sets}]$, the category of simplicial sets (see [22]), and from algebraic geometry, the category of schemes. Finally we mention some categories in computer science: Scott domains are well established and stable domains have now a substantial theory; I mention as well the more recent categories of games.

It is not only the case that collections of mathematical structures form categories, but also the case that many structures which appear in mathematics are themselves categories of some kind. This is a particularly fertile idea, which I learnt early in my career from Bill Lawvere. I do not try to survey the range of special examples, which have emerged

over the years, but give a traditional list of small categories. Preorders are categories with at most one map between any two objects. Monoids are categories with just one object. Groupoids are categories in which all maps are invertible. Finally groups are one object groupoids.

3.2. Free categories

Let $G = (E \overrightarrow{\longrightarrow} V)$ be a directed graph: V is the collection of vertices, E the collection of directed edges and the two maps give source and target. The category $\mathbb{C}_G$ generated by G has as objects the set V of vertices, and as maps $A \to B$ the paths

$$A = A_1 \to A_2 \to A_3 \to \cdots \to A_n = B$$

from A to B in the graph. Identities are the trivial paths, and composition is given by concatenation of paths. This is the simplest example of a free construction in category theory. Free categories with structure play an important role in theoretical computer science, and frequently they can be constructed in just the same hands on fashion.

3.3. Functors and natural transformations

The idea of a functor arose out of the observation that in algebraic topology invariants such as the homology groups are defined in a simple uniform fashion, with the consequence that maps between spaces induce maps between the homology groups *in a natural way*. This amounts to regarding the categories as (large) structures and the constructions as structure preserving maps, and so one arrives at the general notion of a functor. A functor $F : \mathcal{C} \to \mathcal{D}$ assigns to each object $X \in \mathcal{C}$ an object $F(X) \in \mathcal{D}$ and to each map $f : X \to Y$ a map $F(f) : F(X) \to F(Y)$ such that

- $F(1_X) = 1_{F(X)}$
- $F(g \circ f) = F(g) \circ F(f)$ whenever $g \circ f$ is defined.

Clearly for any category $\mathcal{C}$, there is an identity functor $1_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$; it acts as the set-theoretic identity on both objects and maps. Furthermore if $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ are functors there is a composite $G \circ F : \mathcal{C} \to \mathcal{E}$; again this is given by set-theoretic composition on both objects and maps. If we restrict to small categories in the spirit of Lawvere this gives us the (large) *category of all small categories* $\mathbf{Cat}$. This has as objects the small categories $\mathbb{C}$; and as maps, functors $F : \mathbb{C} \to \mathbb{D}$. The identities and composition are as just described.

The idea of natural isomorphisms and more generally of natural transformations was part of category theory from the beginning. Suppose that $F, G : \mathcal{C} \to \mathcal{D}$ are functors. A *natural transformation* $\alpha : F \to G$ consists of a family of maps $\alpha_U : FU \to GU$ indexed over the objects $U \in \mathcal{C}$ such that for all maps $f : U \to V$ in $\mathcal{C}$, the diagram

$$\begin{array}{ccc} FU & \xrightarrow{\alpha_U} & GU \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FV & \xrightarrow{\alpha_V} & GV \end{array}$$

commutes. If all α_U are isomorphism, then α is a *natural isomorphism*.

If $\mathbb{C}$ and $\mathbb{D}$ are small categories, then $[\mathbb{C}, \mathbb{D}]$, with objects the functors from $\mathbb{C}$ to $\mathbb{D}$ and with maps the natural transformations, is itself a small category.

Exercise 3

1. *Show that $\mathbb{C}_G$ is the free category generated by G in the sense that any graph homomorphism from G to the underlying grapg of some category $\mathbb{D}$ extends uniquely to a functor from $\mathbb{C}_G$ to $\mathbb{D}$.*
2. *(a) What is the free category on the unique graph of the form $(0 \overrightarrow{\longrightarrow} 1)$?*
   (b) What is the free category on the unique graph of the form $(0 \overrightarrow{\longrightarrow} 1)$?
   (c) What is the free category on the graph $(1 \overrightarrow{\longrightarrow} 2)$ where the source and target are distinct?
   (d) What is the free category on the graph $(\mathbb{N} \overrightarrow{\longrightarrow} \mathbb{N})$ with source the identity and target the successor?
3. *Show that* Cat *has products. Show further that we have a natural isomorphism*

$$\mathbf{Cat}(\mathbb{C} \times \mathbb{D}, \mathbb{E}) \cong \mathbf{Cat}(\mathbb{C}[\mathbb{D}, \mathbb{E}]),$$

so that $\mathbf{Cat}$ *is a cartesian closed category. (Hence it models the typed lambda calculus, see for example [19].)*

4. Background: Symmetric monoidal categories

4.1. Intuition and examples

We refer to Mac Lane [20] for the notion of a monoidal and of a symmetric monoidal category. Here we give an informal description. A *monoidal category* is a category $\mathcal{A}$ equipped with

- a tensor functor $\otimes : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$
- a choice of object $I \in \mathcal{A}$

making $\mathcal{A}$ a monoid in a suitable up to isomorphism sense. That means that we have natural isomorphisms

$$a_{UVW} : (U \otimes V) \otimes W \longrightarrow U \otimes (V \otimes W),$$

$$l_U : I \otimes U \longrightarrow U \text{ and } r_U : U \otimes I \longrightarrow U,$$

rather than equalities; and for good sense, these natural isomorphisms should satisfy coherence conditions. However there is a precise sense (the Mac Lane Coherence Theorem) in which one can replace such a monoidal category by one in which we do have equality. Such monoidal categories are called *strictly associative* or just *strict*. Most of the examples with which we are concerned are strict or have obvious strict replacements. A monoidal category is *symmetric* if it is equipped with a symmetry that is, a natural isomorphism

$$c_{UV} : U \otimes V \longrightarrow V \otimes U$$

with $c^2 = 1$, again satisfying coherence conditions. The symmetry isomorphism is very seldom an identity.

I first run through some simple general sources of symmetric monoidal categories. These require the knowledge of very elementary category theory.

1. Categories with finite products are symmetric monoidal. A choice of terminal object and of binary products gives the monoidal structure.

2. Dually, categories with finite coproducts are symmetric monoidal. A choice of initial object and of binary coproducts gives the monoidal structure.

3. Categories with biproducts (which we treat briefly in section 8 but see [20] for details) are symmetric monoidal. In this case the monoidal structure is both a product and coproduct.

4. If T is any commutative algebraic theory, then the category of T-algebras automatically has a tensor product.

It is good to have a couple of very specific examples.

5. The category $\mathbf{Rel}$ of (finite) sets and relations has a tensor product given by coproduct (disjoint union) of sets. This is an example of a category with biproducts.

6. The category $\mathbf{Rel}$ of (finite) sets and relations has a tensor product given by product of sets. In $\mathbf{Rel}$ this is neither a categorical product or coproduct. This is an example of a subcategory of a category of algebras for a commutative theory - the theory of complete $\bigvee$-lattices.

4.2. The free symmetric monoidal category

We describe the free symmetric monoidal category on an object. (Technically we are considering the strictly associative version of this notion.) The category $\mathbf{Perm}$ (for permutations) has as objects the natural numbers 0, 1, ... The maps from n to n are the elements of the symmetric group S_n with their usual composition; there are no maps n to m when $n \neq m$. The tensor product is given by $n + m$ on objects with the obvious extension to maps. The braid relations

$$s_i.s_{i+1}.s_i = s_{i+1}.s_i.s_{i+1} \text{ for } 1 \leq i \leq n-2; \qquad s_i.s_j = s_j.s_i \text{ for } |i-j| > 1,$$

where s_i is the transposition $(i\, i+1)$, enforce the coherence of the obvious symmetry $c_{n,m} \in S_{n+m} = \mathbf{Perm}(n+m, n+m)$.

The monoidal categories are a rich area of study. There are many more identifications of free such structures relevant to computer science. An early and important one is the (augmented) simplicial category originally from topology: the characterization by generators and relations in [22] comes from a characterization as a free monoidal category. There is also a wealth of material relating monoidal and symmetric monoidal categories to higher dimensional category theory.

4.3. The category of automata

We want to explain how to give a category $\mathbf{Aut}$ whose maps correspond to finite automata (on a fixed language Σ). As in the case of the free symmetric monoidal category **Perm**, we take the objects of **Aut** to be the natural numbers. As maps from n to m it is natural to take finite automata with n distinct (numbered) input states and m distinct (distinct) output states. (However we do not ask the inputs to be distinct from the outputs.) We can almost do what we want, but in fact we need to take the automata modulo contraction of insignificant silent actions. We quickly explain this notion. Let us say that a state is a *sink* if no actions lead from it and a *source* if no actions lead too it. A silent action is *insignificant* if it leads from a state which becomes a sink when it is deleted to a state which becomes a source when it is deleted.

The key point is that in $\mathbf{Aut}$, the composition of $A : n \to m$ and $B : m \to p$ is the automaton obtained from A and B taken disjointly by adding a silent action from each output of A to the corresponding input of B. To have identities for this composition, we need to contract some silent actions[3] and these actions are insignificant. (Then it does not matter whether we give identities $n \to n$ by drawing n silent actions or simply by identifying input and output nodes with no actions.)

The category $\mathbf{Aut}$ has a very easy tensor product. On objects we take addition as before and on maps the disjoint union of finite automata. Symmetries are represented by trivial automata in the same style as identities.

Exercise 4

1. *Suppose that $\mathcal{A}$ is a symmetric monoidal category. Note that any $\mathcal{A}(A, A)$ is a monoid under composition. Show that the monoid $\mathcal{A}(I, I)$ is commutative.*
2. *Suppose that $\mathcal{A}$ is a category with (chosen) finite products. Show how to define the structure a, l and r of a monoidal category on $\mathcal{A}$. Look up the axioms for a monoidal category and show that they hold.*
3. *The example $\mathbf{Perm}$ above is the free symmeric monoidal category generated by an object. What does this mean? What is needed to prove it?*
4. *Look up the coherence diagrams for a symmetry and show that they hold for the evident symmetry on $\mathbf{Perm}$.*
5. *The objects in the free monoidal category with a particular kind of tensor product generated by a single object can be taken to be $0, 1, 2, \cdots$ as was the case for $\mathbf{Perm}$. (The object n corresponds to the n-fold tensor product of the generating object.)*
   (a) What is the free category with coproducts on an object? So what is the free category with products on an object?
   (b) What is the free symmetric monoidal category in which the unit I is initial? So what is the free symmetric monoidal category in which the unit I is terminal?
6. *(i) What becomes of our category $\mathbf{Aut}$ if we take $\Sigma = \emptyset$, the empty set?*
   (ii) What becomes of our category $\mathbf{Aut}$ if we take $\Sigma = 1$, a one element set?
7. *With a little manipulation you should be able to use the flow diagrams from Section 2.1 to give a category $\mathbf{Flow}$ along the following lines. Again take the objects*

[3] An alternative formulation is that we do not allow dangling silent actions, that is, unque silent action form an input or to an output. That gives a slightly different category. I am not sure yet which I prefer.

to be the natural numbers. We will need some dummy flow diagrams (wirings with no nodes) to represent identities. Then the basic idea is that maps from n to m are given by a flow diagram with a map from $\{1, \cdots, n\}$ to some inputs and a map from some outputs to $\{1, \cdots, m\}$. Tensor product on objects is addition as before and on maps one takes the disjoint union of flow diagrams. Represent identities and symmetries by trivial automata.

5. Special case: permutations

This section is a warm-up for the general notion of trace. We look at an instance where the computational force seems pretty trivial. (Though in fact this example is the basis for an analysis of the proof theory of multiplicative Linear Logic [6], which forms part of the celebrated Geometry of Interaction perspective.)

We consider the category $\mathbf{Perm}$ whose nonempty sets of maps are $\mathbf{Perm}(n, n) = S_n$, the finite symmetric groups for $n = 0,\ 1,\ 2\ \cdots$.

For $\sigma \in S_{n+m}$ we define the trace $\mathsf{tr}_m(\sigma) \in S_n$ of σ as follows. First we define a subsidiary function $\sigma_m : n + m \to n$ recursively by

$$\sigma_m(i) = \begin{cases} \sigma(i) & \text{if } \sigma(i) \in n, \\ \sigma_m(\sigma(i)) & \text{otherwise.} \end{cases}$$

Then we set $\mathsf{tr}_m(\sigma)(i) = \sigma_m(i)$ for $1 \leq i \leq n$; that is $\mathsf{tr}_m(\sigma)$ is the restriction of the function σ_m to n.

Exercise 5

1. *Justify the recursive definition of σ_m. On what is it a recursion?*
2. *Prove that $\mathsf{tr}_m(\sigma)$ is as required a permutation.*
3. *Show that for $\sigma \in S_k$, $\mathsf{tr}_m(\sigma)$ can be defined for $0 \leq m \leq k$ inductively in m by the formulae*

$$\mathsf{tr}_0(\sigma) = \sigma\,; \quad \mathsf{tr}_{m+1}(\sigma) = \mathsf{tr}_m(\mathsf{tr}_1(\sigma))$$

4. *How does taking the trace of a permutation affect the decomposition into cycles? How does it affect the parity of the permutation?*
5. *Suppose that $\sigma \in S_n$ and $\tau \in S_m$. We define $\sigma + \tau \in S_{n+m}$ by*

$$(\sigma + \tau)(i) = \begin{cases} \sigma(i) & \text{if } 1 \leq i \leq n, \\ \tau(i - n) + n & \text{if } n1 \leq i \leq n + m. \end{cases}$$

   What is $\mathsf{tr}_m(\sigma + \tau)$?
6. *Let $\sigma \in S_{n+m}$ and $\tau \in S_m$. Show that $\mathsf{tr}_m((1 + \tau)\sigma) = \mathsf{tr}_m(\sigma(1 + \tau))$ where $1 \in S_n$ is the identity.*
7. *Let $\gamma \in S_{2n}$ be the product of the disjoint transpositions $(i\ n + i)$ for $1 \leq i \leq n$. What is $\mathsf{tr}_n(\gamma)$.*

6. Traced monoidal categories

6.1. The definition of trace

Here I define the basic notion in terms of which we consider recursion, that of traced monoidal category. We do not need the subtleties of the braided case explained in the basic reference [16]. So for us a *traced monoidal category* is a symmetric monoidal category equipped with a trace operation

$$\frac{f : A \otimes U \to B \otimes U}{\mathsf{tr}_U(f) : A \to B}$$

satisfying elementary properties of feedback. These are as follows.

- (Domain Naturality)
  For $f : A \otimes U \to B \otimes U$ and $g : C \to A$ we have

  $$\mathsf{tr}_U(f(g \otimes \mathrm{id}_U)) = \mathsf{tr}_U(f)g\,.$$

- (Codomain Naturality)
  For $f : A \otimes U \to B \otimes U$ and $h : B \to D$ we have

  $$\mathsf{tr}_U((h \otimes \mathrm{id}_U)f) = h\mathsf{tr}_U(f)\,.$$

- (Trace Naturality)
  For $f : A \otimes U \to B \otimes V$ and $k : V \to U$ we have

  $$\mathsf{tr}_U((\mathrm{id} \otimes k)f) = \mathsf{tr}_V(f(\mathrm{id} \otimes k))$$

- (Action)
  For $f : A \otimes I = A \to B \otimes I = B$,

  $$\mathsf{tr}_I(f) = f$$

  and for $f : A \otimes U \otimes V \to B \otimes U \otimes V$,

  $$\mathsf{tr}_V(\mathsf{tr}_U(f)) = \mathsf{tr}_{U \otimes V}(f)$$

- (Independence)
  For $f : A \otimes U \to B \otimes U$ and $g : C \to D$

  $$\mathsf{tr}_U(g \otimes f) = g \otimes \mathsf{tr}_U(f)$$

- (Symmetry)
  For $c_{U,U}$ the symmetry on U

  $$\mathsf{tr}_U(c_{U,U}) = \mathrm{id}_U\,.$$

I have followed my own private preferences in changing the names of some of these axioms. I regard the first three conditions as all instances of naturality. My Traced Naturality is often called Dinaturality. What I call Action is usually and oddly called Vanishing. My Independence is otherwise Superposing. The Symmetry Axiom is usually called Yanking which has at least a good diagrammatic sense. A useful perspective on the axioms is provided by Hasegawa [9]. He also gives diagrams (without the braidings in [16]) without which the axioms are hard to understand. I drew pictures in the lectures, and the slides can be found on the conference website, but I do not have enough space here.

6.2. The free compact closed category

It is a commonplace amongst workers in Linear Logic that traced monoidal categories provide a backdrop to Girard's Geometry of Interaction. This rests on a construction which was the main result of the original paper [16].

If $\mathbf{C}$ is a traced monoidal category, then its integral completion $\mathrm{Int}(\mathbf{C})$ is defined as follows.

- The objects of $\mathrm{Int}(\mathbf{C})$ are pairs (A_0, A_1) of objects of $\mathbf{C}$.
- Maps $(A_0, A_1) \to (B_0, B_1)$ in $\mathrm{Int}(\mathbf{C})$ are maps $A_0 \otimes B_1 \to B_0 \otimes A_1$ of $\mathbf{C}$.
- Composition of $f : (A_0, A_1) \to (B_0, B_1)$ and $g : (B_0, B_1) \to (C_0, C_1)$ is given by taking the trace $\mathsf{tr}_(\sigma); f \otimes g; \tau)$ of the composite of $f \otimes g$ with the obvious symmetries

$$A_0 \otimes C_1 \otimes B_0 \otimes B_1 \xrightarrow{\sigma} A_0 \otimes B_1 \otimes B_0 \otimes C_1 \,,$$

and

$$B_0 \otimes A_1 \otimes C_0 \otimes B_1 \xrightarrow{\tau} C_0 \otimes A_1 \otimes B_0 \otimes B_1 \,.$$

- Identities $(A_0, A_1) \to (A_0, A_1)$ are given by the identity $A_0 \otimes A_1 \to A_0 \otimes A_1$.

To understand the basic result, you need to know that a compact closed category is a symmetric monoidal category in which all objects have duals. Then the following is proved in [16].

Theorem 4 *Suppose that* $\mathbb{C}$ *is a traced monoidal category. Then* $\mathrm{Int}(\mathbb{C})$ *is a compact closed category. Moreover* Int *extends to a* 2*-functor left biadjoint to the forgetful* 2*-functor from compact closed categories to traced monoidal categories.*

In the sense of this theorem $\mathrm{Int}(\mathbb{C})$ is the free compact closed category generated by the traced monoidal category $\mathbb{C}$.

Exercise 6

1. *Show that what we defined in Section 5 is a trace on the category* $\mathbf{Perm}$*. (We already checked some axioms.)*
2. *Show that the category* $\mathbf{Aut}$ *whose maps are finite automata has a trace.*
3. *Show that the category* $\mathbf{Flow}$ *whose maps are flow diagram programs has a trace. (This assumes that you have completed an earlier exercise.)*
4. *Does the category* $\mathbf{Sets}$ *have a trace? (If you have trouble consider Section 7.)*

5. *(i) Show that the monoidal category of finite sets and relations with $+$ as tensor product has a trace. Is it unique? (If you have trouble consider Section 9.)*
   (ii) Show that the monoidal category of finite sets and relations with $\times$ as tensor product has a trace. Is it unique?
6. *(i) Does the free category with products generated by an object have a trace? If it does is it unique?*
   (ii) Does the free symmetric monoidal category with I terminal generated by an object have a trace? If it does is it unique?
7. *A trace is said to be uniform just when it satisfies the following condition. Whenever*

$$\begin{array}{ccc} A \otimes X & \xrightarrow{f} & B \otimes X \\ \downarrow{\scriptstyle A \otimes h} & & \downarrow{\scriptstyle B \otimes h} \\ A \otimes Y & \xrightarrow[g]{} & B \otimes Y \end{array}$$

   commutes, then $\mathsf{tr}_X(f) = \mathsf{tr}_Y(g)$. *Can you find an example of a uniform trace?*
8. *Show that any compact closed category is equipped with a trace, and prove the above theorem.*
9. *Show that the trace on a compact closed category is essentially unique. (Which earlier question does this answer?)*

7. Traced monoidal categories with products

The notion of a category with products is easy and accessible and we do not give details here. The interesting feature for us is that a general form of functional programming is based on the idea of the (least) fixed point of functionals. In this section we connect that idea with the idea of a trace.

7.1. Traces and fixed points

We consider the special case of a symmetric monoidal category where the tensor product is a categorical product. Write $\Delta = \Delta_A : A \to A \times A$ for the standard diagonal map.

Suppose first that in such a category we have a trace operation

$$\frac{f : A \times U \to B \times U}{\mathsf{tr}_U(f) : A \to B}$$

We derive from it an operation

$$\frac{f : A \times B \to B}{\mathsf{fix}_B f : A \to B}$$

by setting

$$\mathsf{fix}_B f = \mathsf{tr}_B(\Delta_B.f)\,.$$

Proposition 1 *The operation* fix *is a natural parametrised fixed point operation. That is it satisfies the following.*

- *(Fixed Point Property)*
  For $f : A \times B \to B$,

$$f.(A \times \mathsf{fix}_B f).\Delta_A = \mathsf{fix}_B f$$

- *(Naturality)*
  If $f : A \times B \to B$ *and* $g : C \to A$ *then*

$$\mathsf{fix}_B(f(g \times B)) = (\mathsf{fix}_B f)g\,.$$

- *(Second Naturality)*
  If $f : A \times B \to C$ *and* $g : C \to B$ *then*

$$\mathsf{fix}_B gf = g.\mathsf{fix}_C f(A \times g)\,.$$

- *(Diagonal)*
  For $f : A \times B \times B \to B$

$$\mathsf{fix}_B(\mathsf{fix}_B f) = \mathsf{fix}_{B\times B}(\Delta_B.f)\,.$$

When dealing with fixed points the variable-free categorical notation becomes intolerable. When $f : A \times B \to B$ we show the variables by writing $f(a, b)$. And then $\mathsf{fix}_B f : A \to B$ can be written $\mu b.f(a, b)$. This notation presupposes the simple form of Naturality. With it the other equations (Fixed Point Property, Second Naturality, Diagonal) in the last proposition become the following.

$$f(a, \mu b.f(a, b)) = \mu b.f(a, b))$$

$$\mu b.g(f(a, b)) = g(\mu c.f(a, g(c)))$$

$$\mu b_1.\mu b_2 f(a, b_1, b_2) = \mu b.f(a, b, b)$$

The proposition above has a converse. Suppose first that in a category with finite products we have a natural parametrised fixed point operation

$$\frac{f : A \times B \to B}{\mathsf{fix}_B f : A \to B}\,.$$

We derive from it an operation

$$\frac{f : A \times U \to B \times U}{\mathsf{tr}_U(f) : A \to B}$$

as follows. We write $f = (f_1, f_2)$ where $f_1 : A \times U \to B$ and $f_2 : A \times U \to U$: take $\mathsf{fix}_U(f_2) : A \to U$ and set

$$\mathsf{tr}_U(f) = (A \times \mathsf{fix}_U(f_2)).\Delta_A \,.$$

Proposition 2 *The operation* $\mathsf{tr}()$ *just defined is a trace on a category with products.*

The passage from trace to fixed point and back are inverse to one another. This general fact was established independently by the author and Masahito Hasegawa (see [9]) but equivalent observations in a different conceptual framework were made earlier by the authors of [4] and [23].

Theorem 5 *There is a bijection between traces and natural parametrised fixed point operators on a category with products.*

7.2. Functional programming

In the simplest view of functional programming we define partial functions $\phi : \mathbb{N}^k \to \mathbb{N}$. Write P_k for the poset of such functions under extension. It is a Scott in fact algebraic domain with the compact elements being the finite functions. Now in the category with objects products of the P_k and with Scott continuous maps we can take least fixed points. We check that this is a natural parametrized fixed point operator.

Suppose that $f : A \times B \to B$. For $a \in A$ define $f_a : B \to B$ by $f_a(b) = f(a, b)$. Then

$$\mu b.f(a, b) = \bigvee_n f_a^n(\bot)$$

The naturality in A is evident, and we check the other axioms. First $\mu b.f(a, b)$ is a fixed point as by continuity we have

$$f(\mu b.f(a, b)) = f_a(\bigvee_n f_a^n(\bot)) = \bigvee_n f_a^{n+1}(\bot) = \mu b.f(a, b)\,.$$

The Second Naturality equation follows by similar considerations. We have

$$g(\mu c.f(a, g(c)) = g(\bigvee_n (f_a g)^n(\bot)) = \bigvee_n g(f_a g)^n(\bot) = \bigvee_n (g f_a)^n(g\bot)$$

But then

$$\mu b.g(f(a, b)) = \bigvee_n (g f_a)^n(\bot) \leq \bigvee_n (g f_a)^n(g\bot) \leq \bigvee_n (g f_a)^{n+1}(\bot) = \mu b.g(f(a, b))$$

shows that $\mu b.g(f(a, b)) = g(\mu c.f(a, g(c))$.

Finally we wish to show the Diagonal Property. First let $\hat{b} = \mu b.f(a, b, b)$; then $\hat{b} = \mu b.f(a, \hat{b}, \hat{b})$, and is least with this property. Suppose that $b_1 \leq \hat{b}$. Then $b_2 \leq \hat{b}$, implies $f(a, b_1, b_2) \leq \hat{b}$. Arguing inductively we have $f_{a,b_1}^n(\bot) \leq \hat{b}$ for all n and so

taking sups, $\mu b_2.f(a, b_1, b_2) \leq \hat{b}$. This shows that $b_1 \leq \hat{b}$ implies $\mu b_2.f(a, b_1, b_2) \leq \hat{b}$. Repeating the inductive argument we deduce that

$$\mu b_1 \mu b_2.f(a, b_1, b_2) \leq \mu b.f(a, b, b)\,.$$

Now let $\hat{b} = \mu b_1 \mu b_2.f(a, b_1, b_2)$, so that $\mu b_2.f(a, \hat{b}, b_2) = \hat{b}$ and so $f(a, \hat{b}, \hat{b}) = \hat{b}$. Suppose that $b \leq \hat{b}$. Then $f(a, b, b) \leq f(a, \hat{b}, \hat{b}) = \hat{b}$. Again arguing inductively we deduce that the iterates approximating $\mu b.f(a, b, b)$ are all less than or equal to $\hat{b}$. This gives an inequality the other way round, and we deduce that

$$\mu b_1 \mu b_2.f(a, b_1, b_2) = \mu b.f(a, b, b)\,.$$

Exercise 7

1. *Establish the Bekic condition for a natural parametrised fixed point operator. If $f : A \times B \times C \to B \times C$ then we can compute the double fixed point*

   $$\mathsf{fix}_{B\times C}(f) : A \to B \times C$$

   as follows. We write $f = (f_1, f_2)$ where $f_1 : A \times B \times C \to B$ and $f_2 : A \times B \times C \to C$. Then $\mu(b, c).f(a, b, c)$ is the pair

   $$\big(\mu b.f_1(a, b, \mu c.f_2(a, b, c)), \mu c.f_2(a, \mu b.f_1(a, b, \mu c.f_2(a, b, c)), c)\big)\,.$$

2. *Prove the two propositions above in whatever notation you prefer. (I think it is much easier to manipulate the diagrams: you could refer to the slides on the Marktoberdorf Summer School website.)*
3. *Prove the theorem above. (Again you may find it easier with diagrams.)*
4. *Here is an alternative approach to defining a trace in a category with finite products and a natural parametrised fixed point operation. Given $f : A\times U \to B\times U$, we make use of two instances of the first projection $\mathsf{fst}_{A,B} : A \times B \to A$ and $\mathsf{fst}_{B,U} : B \times U \to B$; and we set*

   $$\mathsf{tr}_U(f) = \mathsf{fst}_{B,U}.\mathsf{fix}_{B\times U}(f(\mathsf{fst}_{A,B} \times U)\,.$$

   Is this a (the same) trace as defined earlier?
5. *Let $\mathcal{C}$ be the category of complete lattices and order-preserving maps. It is a category with evident products. Show that any $h : B \to B$ in $\mathcal{C}$ has a least fixed point $\mu b.h(b)$. Show that the operation taking $f : A \times B \to B$ to $\mu b.f : A \to B$ is a natural parametrised fixed point operator.*
6. *Show that a symmetric monoidal category may admit more than one notion of trace.*

8. Categories with biproducts

We consider the special case where the tensor product in a traced monoidal category is a biproduct. This situation is discussed in detail in [20], but for completeness we give a sketch here.

The first important fact, which we invite the reader to establish in the exercises below, is that a category $\mathcal{C}$ with biproducts is *enriched* in commutative monoids. For the general theory of enriched categories one should consult [17]. The concrete content of the enrichment is that each hom-set $\mathcal{C}(A, B)$ is equipped with the structure of a commutative monoid (which we write additively) and composition is bilinear in that structure. It follows that for each object A its endomorphisms $\mathrm{End}_{\mathcal{C}}(A) = \mathcal{C}(A, A)$ has the structure of what is now called (following Bill Lawvere and Steve Schanuel) a rig, that is to say a (noncommutative) ring without negatives. We make the definition explicit. A *rig* is a set R equipped with

- $0 \in R$ and $(- + -) : R \times R \to R$
- $1 \in R$ and $(-.-) : R \times R \to R$

satisfying the familiar rules

- 0 and $+$ make R an (additive) commutative monoid,
- 1 and $.$ make R a (multiplicative) monoid,
- the multiplicative structure distributes over the additive.

This analysis has a kind of converse. Let R be a rig. Then there is a category $\mathbf{Mat}(R)$ with objects the natural numbers and with maps from n to m being $n \times m$ matrices with entries in R.

Theorem 6 $\mathbf{Mat}(R)$ *is a category with biproducts.*

Exercise 8

1. *Suppose that $\mathcal{A}$ is a category with biproducts.*
   (i) Show that for any objects $A, B \in \mathcal{A}$ the hom-set $\mathcal{A}(A, B)$ has the structure of a commutative monoid.
   (ii) Show that a map $A \oplus B \to C \oplus D$ can be represented by a matrix with entries from $\mathcal{A}(A, C)$, $\mathcal{A}(B, C)$, $\mathcal{A}(A, D)$ and $\mathcal{A}(B, D)$. Show further that composition of such maps is by matrix multplication.
2. *(i) Show that composition in $\mathcal{A}$ is a bilinear map of commutative monoids.*
   (ii) Deduce that for any object $A \in \mathcal{A}$, $\mathrm{End}_{\mathcal{A}}(A) = \mathcal{A}(A, A)$ is a rig.
3. *Prove that as claimed above $\mathbf{Mat}(R)$ is a category with biproducts.*
4. *What is the free category with biproducts generated by an object. (It suffices to identify the free rig on no generators. Why?)*

9. Traced Categories with biproducts

In this section we explain what it is to equip a category with biproducts with a trace in terms of rigs. Here we concentrate on the one object case, which is the only case considered in the main reference [4].

9.1. Conway rigs

We recall the notion originally studied briefly by Conway [5]) and discussed also in [4] where it is called a Conway Algebra; but as there is other algebraic structure also associated with the fertile mind of John Horton Conway we rename the structure.

Definition 3 *A* Conway Rig *is a rig A equipped with a unary operation*

$$(-)^* : A \longrightarrow B \,;\, a \to a^*$$

satisfying the two equations

$$(ab)^* = 1 + a(ba)^*b \qquad \text{and} \qquad (a+b)^* = (a^*b)^*a^* \,.$$

Theorem 7 *In a traced monoidal category* $\mathbf{C}$ *where the tensor product is a biproduct, each* $\mathrm{End}_{\mathbf{C}}(A)$ *is a Conway Rig, the operation* $(-)^*$ *being given by*

$$a^* = \mathsf{tr}(\begin{pmatrix} 0 & 1 \\ 1 & a \end{pmatrix}) \,.$$

where we interpret the matrix as a map $\begin{pmatrix} 0 & 1 \\ 1 & a \end{pmatrix} : A \oplus A \to A \oplus A$ *in the obvious way.*

For a converse we restrict ourselves here to the case of a category with biproducts generated by a single object U. (The more general case is just a bit more fiddly.) Then it suffices to require that $\mathrm{End}_{\mathbf{C}}(U)$ be a Conway Rig. The point is this, though there is much checking to do. One takes the trace of a map $A \oplus C \longrightarrow B \oplus C$ given by the matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $a \in \mathcal{C}(A,B)$, $b \in \mathcal{C}(C,B)$, $c \in \mathcal{C}(A,C)$ $d \in \mathcal{C}(C,C)$ using the natural formula

$$\mathsf{tr}(\begin{pmatrix} a & b \\ c & d \end{pmatrix}) = a + bd^*c$$

Note that setting $C = U$ and using the operation $(-)^*$ on $\mathrm{End}_{\mathbf{C}}(U)$ in the formula enables us inductively to define the trace in all cases.

Theorem 8 *Suppose that $\mathcal{C}$ is a category with biproducts generated by an object U. Then traces on $\mathcal{C}$ correspond exactly to choices of Conway Rig structures on* $\mathrm{End}_{\mathbf{C}}(U)$.

Exercise 9

1. *Show that in a traced category with biproducts we must have*

$$\mathsf{tr}(\begin{pmatrix} 1 & a \\ 1 & a \end{pmatrix}) = a^* \,.$$

2. *Show that in a traced category with biproducts we must have*

$$\mathsf{tr}(\begin{pmatrix} 1 & a+b \\ 1 & a+b \end{pmatrix}) = (b^*a)^*b^* \,.$$

3. *Construct a proof of Theorem 7 above.*
4. *Show that if R is a Conway Algebra, then so is $M_n(A)$ the $n \times n$ matrices with entries in R.*
5. *Construct a proof of Theorem 8 above.*

10. Regular Languages and Finite Automata: reprise

10.1. The category of regular languages

The most familiar Conway Rig is that of regular languages or events. Let Σ be a finite alphabet and Σ^* the collection of finite words from Σ. Alternatively $\Sigma^* = \mathrm{List}(\Sigma)$ is the set of finite lists. The collection $P(\Sigma^*)$ of subsets of Σ^* has the structure of a Conway Rig where

- the zero 0 is the empty set of words;
- the sum $a + b$ is given by union $a \cup b$;
- the unit 1 is the set containing just the empty word;
- the multiplication ab is given by $\{xy | x \in a \text{ and } y \in b\}$, that is, by elementwise concatenation of words from a and from b;
- the star a^* is $1 + a + a^2 + \cdots$, that is, the collection of all finite concatenations of words from a.

The substructure generated by the singleton languages whose only words are the letters from Σ has as elements exactly the regular languages: it gives us the Conway Rig Reg of regular languages. By Section 9 there is a traced monoidal category $\mathbf{Reg} = \mathbf{Mat}(Reg)$ with objects $0, 1, 2, \cdots$ in the usual way, and where the maps from n to m are given by $m \times n$-matrices whose entries are regular languages.

10.2. Definition by finite automata

Recall the category $\mathbf{Aut}$ with objects also the natural numbers and with maps given by automata. Note that an automaton can be presented as a matrix with entries very simple elements of Reg. (The presentation of the category $\mathbf{Aut}$ in this fashion can be thought of along the lines of the Girard's Geometry of Interaction, but that takes us too far afield.) Consider an automaton $A \in \mathbf{Aut}(n, m)$. A is itself a $k \times k$ matrix where as we have things $k \geq n, m$. We interpret A as follows. We compute the $k \times k$ matrix A^*, and then restrict to the n input columns and m output rows. This gives us an $n \times m$ matrix with elements from Reg, that is a map in $\mathbf{Reg}(n, m)$. This provides us with evident data for a functor $M : \mathbf{Aut} \to \mathbf{Reg}$. And the root of Kleene's Theorem is that this all works.

Theorem 9 *$M : \mathbb{A} \to \mathrm{Reg}$ is a strong monoidal functor which preserves trace.*

Exercise 10

1. *Show that both $P(\Sigma^*)$ and Reg are Conway algebras.*
2. *Show that in $P(\Sigma^*)$ and so in Reg the following natural equations hold*

$$(a^*)^* = a^* = (a^n)^*(1 + a + \cdots + a^{n-1})$$

3. *When I took Part III of the Mathematical Tripos at Cambridge I had the good fortune to take a course by John Conway based on sections from his book [5]. He gave the following equation*

$$(a + b)^* = ((a + b)(b + (ab^*)^3)^*(1 + (a + b))(1 + ab^* + (ab^*)^2 + (ab^*)^3)$$

as an example of something valid in regular events but not provable from the above axioms for what we now call Conway Rigs and the two equations above. Check its validity. (Where did it come from?)

4. *Show that M in the theorem is indeed a strong monoidal functor and that it does preserve trace. Is there a connection between these two facts?*

11. Concluding section: Free Traced Category with Biproducts

By Section 9 to identify the free traced monoidal category with biproducts on an object U it suffices to identify the free Conway Rig on no generators. The category in question is then given by the matrices construction $\mathbf{Mat}$. Fortunately the free Conway was analyzed years ago by Conway himself, though his analysis is not widely known. In [5] Conway effectively identifies the elements of the free Conway rig on no generators: at least he gives the distinct elements. They are those in the set

$$\{n \mid n \geq 0\} \cup \{n(1^*)^m \mid n, m \geq 1\} \cup \{1^{**}\}.$$

The last set of exercises gives an indication of why this is and touches on related matters. Some of the algebraic manipulation is hard. I remark however that at the Summer School, John Harrison showed me that even the hardest, which had originally taken me a couple of days to discover, was readily found by the Prover9 system of William McCune. So in extremis download it and play!

Exercise 11

1. *(i) Show that* $1 + (1^*)^n = (1^*)^n$.
   (ii) Show that $(1^*)^n + 1^{**} = 1^{**} + 1^{**} = 1^{**}$.
   (iii) Show that $n.1^{**} = 1^*.1^{**} = 1^{**}.1^{**} = 1^{**}$.
   (iv) Show that $1^{***} = 2^* = 1^{**}$.
2. *Using the above equations and developing anything further you need, show that any element in the free Conway rig on no generators is equivalent to one of those given by Conway.*
3. *What is the algebraic structure on the elements of the free Conway Rig. (That is, determine the addition, multiplication and star tables.)*
4. *Show that the elements given by Conway are all distinct.*
5. *I have an interest in understanding classical proofs. here is a calculation in the free Conway rig coming from [14]. Compute the trace in the last four arguments of the matrix*

$$\begin{pmatrix} 0&0&1&1&1&1 \\ 0&0&0&0&1&0 \\ 1&0&0&0&1&0 \\ 0&1&0&0&1&1 \\ 1&0&1&0&0&0 \\ 0&0&2&1&0&0 \end{pmatrix}.$$

Fortunately the cited paper is full of typos and the answer given there is not correct. So there is no point in cheating.

6. *Prove that the following is true in any Conway Algebra.*

$$a^{****} = a^{***}.$$

References

[1] S. Awodey. *Category Theory*. Oxford Logic Guides **49**, Clarendon Press, Oxford, 2006.

[2] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[3] N. Benton and M. Hyland. Traced Premonoidal Categories. *Informatique Théorique et Applications* 37, (2003), 273-299.

[4] S. L. Bloom and Z. Esik. *Iteration Theories.* Springer-Verlag, 1993.

[5] J. H. Conway. *Regular algebra and finite machines.* Chapman and Hall, 1971.

[6] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* **50**, (1987), 1-102.

[7] E. Haghverdi.Unique decomposition categories, Geometry of Interaction and Combinatory Logic. *Mathematical structures in Computer Science* **10**, (2000), 205-231.

[8] E. Haghverdi and P. Scott. A categorical model for the geometry of interaction. *Theoretical Computer Science* **350**, (2006), 252-274.

[9] M. Hasegawa. *Models of sharing graphs. (A categorical semantics for* Let *and* Letrec.*)* Distinguished Dissertation in Computer Science, Springer-Verlag, 1999.

[10] J. M. E. Hyland. Proof Theory in the Abstract. *Annals of Pure and Applied Logic* **114** (2002), 43-78.

[11] M. Hyland and J. Power. Symmetric monoidal sketches. *Proceedings of PPDP 2000*, ACM Press (2000), 280-288.

[12] M. Hyland and J. Power. Symmetric monoidal sketches and categories of wiring diagrams. In *Proceedings of CMCIM 2003*, Electronic Notes in Theoretical Computer Science 100, (2004), 31-46.

[13] M. Hyland and A. Schalk. Glueing and Orthogonality for Models of Linear Logic. *Theoretical Computer Science* **294** (2003) 183-231.

[14] M. Hyland. Abstract Interpretation of Proofs: Classical Propositional Calculus. In *Computer Science Logic (CSL 2004), eds. J. Marcinkowski and A. Tarlecki*, Lecture Notes in Computer Science 3210, (2004), 6-21.

[15] P.T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium, Volumes 1 and 2*. Oxford Logic Guides **43** and **44**. Clarendon Press, Oxford, 2002.

[16] A. Joyal and R. Street and D. Verity. Traced monoidal categories. *Math. Proc Camb Phil. Soc.* **119** (1996), 425-446.

[17] G. M. Kelly. *Basic Concepts of Enriched Category Theory.* LMS Lecture Note Series **64**, Cambridge University Press (1982).

[18] G. M. Kelly and M. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra* **19** (1980), 193-213.

[19] J. Lambek and P.J.Scott. *Introduction to higher order categorical logic.* Cambridge Studies in Advanced Mathematics **7**, Cambridge University Press (1986).

[20] S. Mac Lane. *Categories for the working mathematician.* Graduate Texts in Mathematics **5**, Springer (1971).

[21] U. Martin, E. A. Mathiesen and P. Oliva. Hoare Logic in the abstract. *Proceeding of CSL 2006*, Lecture Notes in Computer science 4207, 501-515, Springer, 2006.

[22] J. P. May. *Simplicial objects in algebraic topology.* Van Nostrand, Princeton. 1967.

[23] G. Stefanescu. *Network Algebra.* Discrete Mathematics and Computer Science Series, Springer-Verlag, 2000.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Secrecy Analysis in Protocol Composition Logic

Arnab ROY [a], Anupam DATTA [b], Ante DEREK [c], John C. MITCHELL [a], and Jean-Pierre SEIFERT [d]

[a] *Stanford University, USA*
[b] *Carnegie Mellon University, USA*
[c] *Google Corporation, USA*
[d] *University of Innsbruck, Austria*

Abstract. We present formal proof rules for inductive reasoning about the way that data transmitted on the network remains secret from a malicious attacker. Extending a compositional protocol logic with an induction rule for secrecy, we prove soundness for a conventional symbolic protocol execution model, adapt and extend previous composition theorems, and illustrate the logic by proving properties of two key agreement protocols. The first example is a variant of the Needham-Schroeder protocol that illustrates the ability to reason about temporary secrets. The second example is Kerberos V5. The modular nature of the secrecy and authentication proofs for Kerberos make it possible to reuse proofs about the basic version of the protocol for the PKINIT version that uses public-key infrastructure instead of shared secret keys in the initial steps.

Keywords. Security protocol analysis, Logic, Secrecy

1. Introduction

Two important security properties for key exchange and related protocols are authentication and secrecy. Intuitively, authentication holds between two parties if each is assured that the other has participated in the same session of the same protocol. A secrecy property asserts that some data that is used in the protocol is not revealed to others. If a protocol generates a fresh value, called a *nonce*, and sends it in an encrypted message, then under ordinary circumstances the nonce remains secret in the sense that only agents that have the decryption key can obtain the nonce. However, many protocols have steps that receive a message encrypted with one key, and send some of its parts out encrypted with a different key. Since network protocols are executed asynchronously by independent agents, some potentially malicious, it is non-trivial to prove that even after arbitrarily many steps of independent protocol sessions, secrets remain inaccessible to an attacker.

Our general approach involves showing that every protocol agent that receives data protected by one of a chosen set of encryption keys only sends sensitive data out under encryption by another key in the set. This reduces a potentially complicated proof about arbitrary runs involving arbitrarily many agents and a malicious attacker to a case-by-case analysis of how each protocol step might save and send data. We formalize this

form of inductive reasoning about secrecy in a set of new axioms and inference rules that are added to Protocol Composition Logic (PCL) [14,8,9,10,11], prove soundness of the system over a conventional symbolic protocol execution model, and illustrate its use with two protocol examples. The extended logic may be used to prove authentication or secrecy, independently and in situations where one property may depend upon the other. Among other challenges, the inductive secrecy rule presented here is carefully designed to be sound for reasoning about arbitrarily many simultaneous protocols sessions, and powerful enough to prove meaningful properties about complex protocols used in practice. While the underlying principles are similar to the "rank function method" [20] and work using the strand space execution model [21], our system provides precise formal proof rules that are amenable to automation. In addition, casting secrecy induction in the framework of Protocol Composition Logic avoids limitations of some forms of rank function arguments and eliminates the need to reason explicitly about possible actions of a malicious attacker. From a broader point of view, we hope that our formal logic will help clearly identify the vocabulary, concepts, and forms of reasoning that are most effective for proving security properties of large-scale practical protocols.

Our first protocol example is a variant of the Needham-Schroeder protocol, used in [16] to illustrate a limitation of the original rank function method and motivate an extension for reasoning about temporary secrets. The straightforward formal proof in section 4 therefore shows that our method does not suffer from the limitations identified in [16]. Intuitively, the advantage of our setting lies in the way that modal formulas of PCL state properties about specific points in protocol execution, rather than only properties that must be true at all points in all runs.

Our second protocol example is Kerberos V5 [17], which is widely used for authenticated client-server interaction in local area networks. The basic protocol has three sections, each involving an exchange between the client and a different service. We develop a formal proof that is modular, with the proof for each section assuming a precondition and establishing a postcondition that implies the precondition of the following section. One advantage of this modular structure is illustrated by our proof for the PKINIT [7] version that uses public-key infrastructure instead of shared secret keys in the initial steps. Since only the first section of PKINIT is different, the proofs for the second and third sections of the protocol remain unchanged. While lengthy machine-checked proofs of Kerberos were previously given [3], and non-formal mathematical proofs have been developed for other abstractions of Kerberos [5], this is the first concise formal logic proof of secrecy and authentication for Kerberos and PKINIT.

Compositional secrecy proofs are made possible by the composition theorems developed in this paper. While these theorems resemble composition theorems for the simpler proof system presented in earlier work [10,15], adapting that approach for reasoning about secrecy requires new insights. For example, while proving that a protocol step does not violate secrecy, it is sometimes necessary to use information from earlier steps. This history information, which was not necessary in our earlier proofs of authentication properties, appears as preconditions in the secrecy induction of the sequential and staged composition theorems.

The rest of the paper is organized as follows. Some background on PCL is given in section 2, followed by the secrecy-related axioms and proof rules in section 3. The first protocol example is presented in section 4. Composition theorems are developed in

section 5, and applied in the proofs for Kerberos in section 6. Related work is summarized in section 7 with conclusions in section 8.

2. Background

Protocol Composition Logic (PCL) is developed in [14,8,9,10], with [11] providing a relatively succinct overview of the most current form. A simple "protocol programming language" is used to represent a *protocol* by a set of *roles*, such as "Initiator", "Responder" or "Server", each specifying a sequence of actions to be executed by an honest participant. Protocol actions include nonce generation, encryption, decryption and communication steps (sending and receiving). A principal can execute one or more copies of each role, concurrently. We use the word *thread* to refer to a principal executing a particular instance of a role. A *thread* X is identified with a pair $(\hat{X}, \eta)$, where $\hat{X}$ is a principal and η is a unique session id. A *run* is a record of all actions executed by honest principals and the attacker during concurrent execution of one or more instances of the protocol. Table 1 describes the syntax of the fragment of the logic that we will need in this paper. Protocol proofs usually use modal formulas of the form $\psi[P]_X\varphi$. The informal reading of the modal formula is that if X starts from a state in which ψ holds, and executes the program P, then in the resulting state the security property φ is guaranteed to hold irrespective of the actions of an attacker and other honest principals. Many protocol properties are naturally expressible in this form.

The formulas of the logic are interpreted over protocol runs containing actions of honest parties executing roles of the protocol and a *Dolev-Yao* attacker (whose possible actions are define by a set of symbolic computation rules). We say that protocol $\mathcal{Q}$ satisfies formula ϕ, denoted $\mathcal{Q} \vDash \phi$, if in all runs R of $\mathcal{Q}$ the formula ϕ holds, *i.e.,* $\mathcal{Q}, R \vDash \phi$. For each run, satisfaction of a formula is defined inductively. For example, $\mathsf{Send}(X, t)$ holds in a run where the thread X has sent the term t. For every protocol action, there is a corresponding action predicate which asserts that the action has occurred in the run. Action predicates are useful for capturing authentication properties of protocols since they can be used to assert which principals sent and received certain messages. $\mathsf{Encrypt}(X, t)$ means that X computes the encrypted term t, while $\mathsf{New}(X, n)$ means X generates fresh nonce n. $\mathsf{Honest}(\hat{X})$ means that $\hat{X}$ acts honestly, *i.e.,* the actions of every thread of $\hat{X}$ precisely follow some role of the protocol. $\mathsf{Start}(X)$ means that the thread X did not execute any actions in the past. $\mathsf{Has}(X, t)$ means X possesses term t. This is "possess" in the symbolic sense of computing the term t using Dolev-Yao rules, *e.g.* receiving it in the clear or receiving it under encryption where the decryption key is known.

To illustrate the terminology used in this section we describe the formalization of Kerberos V5, which is a protocol used to establish mutual authentication and a shared session key between a client and an application server [17]. It involves trusted principals known as the Kerberos Authentication Server (KAS) and the Ticket Granting Server (TGS). There are pre-shared long-term keys between the client and the KAS, the KAS and the TGS, and the TGS and the application server. Typically, the KAS shares long-term keys with a number of clients and the TGS with a number of application servers. However, there is no pre-shared long term secret between a given client and an application server. Kerberos establishes mutual authentication and a shared session key between the client and the application server using the chain of trust leading from the client to the KAS and the TGS to the application server.

Action formulas

$\mathsf{a} ::= \mathsf{Start}(X) \mid \mathsf{Send}(X,t) \mid \mathsf{Receive}(X,t) \mid \mathsf{New}(X,t) \mid \mathsf{SymEnc}(X,t,k) \mid \mathsf{PkEnc}(X,t,k) \mid$
$\mathsf{SymDec}(X,t,k) \mid \mathsf{PkDec}(X,t,k) \mid \mathsf{Sign}(X,t,k) \mid \mathsf{Verify}(X,t,k) \mid \mathsf{Hash}(X,t,k)$

Formulas

$\phi ::= \mathsf{a} \mid \mathsf{Has}(X,t) \mid \mathsf{Honest}(\hat{X}) \mid \phi \wedge \phi \mid \neg\phi \mid \exists V.\ \phi$

Modal form

$\Psi ::= \phi\ [Actions]_X\ \phi$

Table 1. Syntax of the logic

Kerberos has four roles, one for each kind of participant - **Client**, **KAS**, **TGS** and **Server**. The long-term shared symmetric keys are written here in the form $k^{type}_{X,Y}$ where X and Y are the principals sharing the key. The $type$ appearing in the superscript indicates the relationship between X and Y in the transactions involving the use of the key. There are three $type$s required in Kerberos: $c \rightarrow k$ indicates that X is acting as a client and Y is acting as a KAS, $t \rightarrow k$ is for TGS and KAS and $s \rightarrow t$ is for application server and TGS. Kerberos runs in three stages with the client role participating in all three. The description of the roles below is based on the A level formalization of Kerberos V5 in [5].

In the first stage, the client thread (C) generates a nonce (n_1) and sends it to the KAS $(\hat{K})$ along with the identities of the TGS $(\hat{T})$ and itself. The KAS generates a new nonce ($AKey$ - Authentication Key) to be used as a session key between the client and the TGS. It then sends this key along with some other fields to the client encrypted under two different keys - one it shares with the client $(k^{c\rightarrow k}_{C,K})$ and one it shares with the TGS $(k^{t\rightarrow k}_{T,K})$. The message portion encrypted with $k^{t\rightarrow k}_{T,K}$ is called the *ticket granting ticket* (tgt). The client extracts $AKey$ by decrypting the component encrypted with $k^{c\rightarrow k}_{C,K}$ and using a `match` actions to separate $AKey$ from the nonce and $\hat{T}$.

In the second stage, the client generates another nonce, encrypts its identity with the session key established in stage one and sends it to the TGS along with the ticket granting ticket and the nonce. The TGS decrypts tgt with the key it shares with KAS and extracts the session key. It then uses the session key to decrypt the client's encryption and matches this with the identity of the client. The TGS then generates a new nonce to be used as a session key between the client and the application server. It then sends this key along with some other fields to the client encrypted under two different keys - the session key derived in the first stage and one it shares with the aplication server. The encryption with later key is called the service ticket (st). The client extracts this new session key by decrypting the component encrypted with the previous session key.

In the third stage, the client encrypts its identity and a timestamp with $SKey$ and sends it to the application server along with the service ticket. The server decrypts st and extracts the $SKey$. It then uses the session key to decrypt the client's encryption, matches the first component of the decryption with the identity of the client and extracts the timestamp. It then encrypts the timestamp with the session key and sends it back to the client. The client decrypts the message and matches it against the timestamp it used. The control flow of Kerberos exhibits a staged architecture where once one stage has been completed successfully, the subsequent stages can be performed multiple times or aborted and started over if an error occurs.

$\mathbf{Client} = (C, \hat{K}, \hat{T}, \hat{S}, t)$ [	$\mathbf{KAS} = (K)$ [
new n_1;	receive $\hat{C}.\hat{T}.n_1$;
send $\hat{C}.\hat{T}.n_1$;	new $AKey$;
	$tgt :=$ symenc $AKey.\hat{C}, k^{t\rightarrow k}_{T,K}$;
receive $\hat{C}.tgt.enc_{kc}$;	$enc_{kc} :=$ symenc $AKey.n_1.\hat{T}, k^{c\rightarrow k}_{C,K}$;
$text_{kc} :=$ symdec $enc_{kc}, k^{c\rightarrow k}_{C,K}$;	send $\hat{C}.tgt.enc_{kc}$;
match $text_{kc}$ as $AKey.n_1.\hat{T}$;	]$_K$
	$\mathbf{TGS} = (T, \hat{K})$ [
$\cdots$ *stage boundary* $\cdots$	receive $tgt.enc_{ct}.\hat{C}.\hat{S}.n_2$;
	$text_{tgt} :=$ symdec $tgt, k^{t\rightarrow k}_{T,K}$;
new n_2;	match $text_{tgt}$ as $AKey.\hat{C}$;
$enc_{ct} :=$ symenc $\hat{C}, AKey$;	$text_{ct} :=$ symdec $enc_{ct}, AKey$;
send $tgt.enc_{ct}.\hat{C}.\hat{S}, n_2$;	match $text_{ct}$ as $\hat{C}$;
	new $SKey$;
receive $\hat{C}.st.enc_{tc}$;	$st :=$ symenc $SKey.\hat{C}, k^{s\rightarrow t}_{S,T}$;
$text_{tc} :=$ symdec $enc_{tc}, AKey$;	$enc_{tc} :=$ symenc $SKey.n_2.\hat{S}, AKey$;
match $text_{tc}$ as $SKey.n_2.\hat{S}$;	send $\hat{C}.st.enc_{tc}$;
	]$_T$
$\cdots$ *stage boundary* $\cdots$	$\mathbf{Server} = (S, \hat{T})$ [
	receive $st.enc_{cs}$;
$enc_{cs} :=$ symenc $\hat{C}.t, SKey$;	$text_{st} :=$ symdec $st, k^{s\rightarrow t}_{S,T}$;
send $st.enc_{cs}$;	match $text_{st}$ as $SKey.\hat{C}$;
	$text_{cs} :=$ symdec $enc_{cs}, SKey$;
receive enc_{sc};	match $text_{cs}$ as $\hat{C}.t$;
$text_{sc} :=$ symdec $enc_{sc}, SKey$;	$enc_{sc} :=$ symenc $t, SKey$;
match $text_{sc}$ as t;	send enc_{sc};
]$_C$	]$_S$

Table 2. Formal description of Kerberos V5, with $\cdots$ *stage boundary* $\cdots$ comments.

3. Proof System for Secrecy Analysis

In this section, we extend PCL with new axioms and rules for establishing secrecy. Secrecy properties are formalized using the $\mathsf{Has}(X, s)$ predicate, which is used to express that honest principal $\hat{X}$ has the information needed to compute the secret s. In a typical two party protocol, $\hat{X}$ is one of two honest agents and s is a nonce generated by one of them. As an intermediate step, we establish that all occurrences of the secret on

the network are protected by keys. This property can be proved by induction over possible actions by honest principals, showing that no action leaks the secret if it was not compromised already.

We introduce the predicate $\mathsf{SafeMsg}(M, s, \mathcal{K})$ to assert that every occurrence of s in message M is protected by a key in the set $\mathcal{K}$. Technically speaking, there is an $(n+2)$-ary predicate $\mathsf{SafeMsg}^n(M, s, \mathcal{K})$ for each $n > 0$, allowing the elements of set $\mathcal{K}$ to be listed as arguments. However, we suppress this syntactic detail in this paper. The semantic interpretation of this predicate is defined by induction on the structure of messages. It is actually independent of the protocol and the run.

Definition 1 (SafeMsg) *Given a run R of a protocol $\mathcal{Q}$, we say $\mathcal{Q}, R \vDash \mathsf{SafeMsg}(M, s, \mathcal{K})$ if there exists an i such that $\mathsf{SafeMsg}_i(M, s, \mathcal{K})$ where $\mathsf{SafeMsg}_i$ is defined by induction on i as follows:*

$\mathsf{SafeMsg}_0(M, s, \mathcal{K})$	*if M is an atomic term different from s*
$\mathsf{SafeMsg}_0(HASH(M), s, \mathcal{K})$	*for any M*
$\mathsf{SafeMsg}_{i+1}(M_0.M_1, s, \mathcal{K})$	*if* $\mathsf{SafeMsg}_i(M_0, s, \mathcal{K})$ *and* $\mathsf{SafeMsg}_i(M_1, s, \mathcal{K})$
$\mathsf{SafeMsg}_{i+1}(E_{sym}[k](M), s, \mathcal{K})$	*if* $\mathsf{SafeMsg}_i(M, s, \mathcal{K})$ *or* $k \in \mathcal{K}$
$\mathsf{SafeMsg}_{i+i}(E_{pk}[k](M), s, \mathcal{K})$	*if* $\mathsf{SafeMsg}_i(M, s, \mathcal{K})$ *or* $\bar{k} \in \mathcal{K}$

The axioms **SAF0** to **SAF5** below parallel the semantic clauses and follow immediately from them. Equivalences follow as the term algebra is free.

SAF0 $\neg\mathsf{SafeMsg}(s, s, \mathcal{K}) \wedge \mathsf{SafeMsg}(x, s, \mathcal{K})$,
where x is an atomic term different from s

SAF1 $\mathsf{SafeMsg}(M_0.M_1, s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M_0, s, \mathcal{K}) \wedge \mathsf{SafeMsg}(M_1, s, \mathcal{K})$

SAF2 $\mathsf{SafeMsg}(E_{sym}[k](M), s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M, s, \mathcal{K}) \vee k \in \mathcal{K}$

SAF3 $\mathsf{SafeMsg}(E_{pk}[k](M), s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M, s, \mathcal{K}) \vee \bar{k} \in \mathcal{K}$

SAF4 $\mathsf{SafeMsg}(HASH(M), s, \mathcal{K})$

The formula $\mathsf{SendsSafeMsg}(X, s, \mathcal{K})$ states that all messages sent by thread X are "safe" while $\mathsf{SafeNet}(s, \mathcal{K})$ asserts the same property for all threads. These predicates are definable in the logic as $\mathsf{SendsSafeMsg}(X, s, \mathcal{K}) \equiv \forall M.(\mathsf{Send}(X, M) \supset \mathsf{SafeMsg}(M, s, \mathcal{K}))$ and $\mathsf{SafeNet}(s, \mathcal{K}) \equiv \forall X.\ \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$.

In secrecy proofs, we will explicitly assume that the thread generating the secret and all threads with access to a relevant key belong to honest principals. This is semantically necessary since a dishonest principal may reveal its key, destroying secrecy of any data encrypted with it. These honesty assumptions are expressed by the formulas KeyHonest and OrigHonest respectively. KOHonest is the conjunction of the two.

- $\mathsf{KeyHonest}(\mathcal{K}) \equiv \forall X.\ \forall k \in \mathcal{K}.\ (\mathsf{Has}(X, k) \supset \mathsf{Honest}(\hat{X}))$
- $\mathsf{OrigHonest}(s) \equiv \forall X.\ (\mathsf{New}(X, s) \supset \mathsf{Honest}(\hat{X}))$.
- $\mathsf{KOHonest}(s, \mathcal{K}) \equiv \mathsf{KeyHonest}(\mathcal{K}) \wedge \mathsf{OrigHonest}(s)$

We now have the necessary technical machinery to state the induction rule. At a high-level, the **NET** rule states that if each "possible protocol step" P locally sends out safe messages, assuming all messages in the network were safe prior to that step,

then all messages on the network are safe. A possible protocol step P is drawn from the set BS of all basic sequences of roles of the protocol. The basic sequences of a role arise from any partition of the actions in the role into subsequences, provided that if any subsequence contains a `receive` action, then this is the first action of the basic sequence.

$$\textbf{NET} \quad \forall \rho \in \mathcal{Q}. \forall P \in BS(\rho).$$

$$\frac{\mathsf{SafeNet}(s,\mathcal{K})\ [P]_X\ \mathsf{Honest}(\hat{X}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(X,s,\mathcal{K})}{\mathcal{Q} \vdash \mathsf{KOHonest}(s,\mathcal{K}) \wedge \Phi \supset \mathsf{SafeNet}(s,\mathcal{K})}\ (*)$$

The side condition $(*)$ is: $[P]_A$ does not capture free variables in Φ and $\mathcal{K}$ and the variable s. Φ should be prefix closed (explained in Section 3). The **NET** rule is written as a rule scheme, in a somewhat unusual form. When applied to a specific protocol $\mathcal{Q}$, there is one formula in the antecedent of the applicable rule instance for each role $\rho \in \mathcal{Q}$ and for each basic sequence $P \in BS(\rho)$; see [11].

The axioms **NET0** to **NET3** below are used to establish the antecedent of the **NET** rule. Many practical security protocols consist of steps that each receive a message, perform some operations, and then send a resulting message. The proof strategy in such cases is to use **NET1** to reason that messages received from a safe network are safe and then use this information and the **SAF** axioms to prove that the output message is also safe.

NET0 $\mathsf{SafeNet}(s,\mathcal{K})\ [\,]_X\ \mathsf{SendsSafeMsg}(X,s,\mathcal{K})$

NET1 $\mathsf{SafeNet}(s,\mathcal{K})\ [\texttt{receive}\ M]_X\ \mathsf{SafeMsg}(M,s,\mathcal{K})$

NET2 $\mathsf{SendsSafeMsg}(X,s,\mathcal{K})\ [\mathtt{a}]_X\ \mathsf{SendsSafeMsg}(X,s,\mathcal{K})$, where a is not a send.

NET3 $\mathsf{SendsSafeMsg}(X,s,\mathcal{K})\ [\texttt{send}\ M]_X\ \mathsf{SafeMsg}(M,s,\mathcal{K}) \supset \mathsf{SendsSafeMsg}(X,s,\mathcal{K})$

Finally, **POS** and **POSL** are used to infer secrecy properties expressed using the Has predicate. The axiom **POS** states that if we have a safe network with respect to s and key-set $\mathcal{K}$ then the only principals who can possess an unsafe message are the generator of s or possessor of a key in $\mathcal{K}$. The **POSL** rule lets a thread use a similar reasoning locally.

POS $\mathsf{SafeNet}(s,\mathcal{K}) \wedge \mathsf{Has}(X,M) \wedge \neg\mathsf{SafeMsg}(M,s,\mathcal{K})$

$$\supset \exists k \in \mathcal{K}.\ \mathsf{Has}(X,k) \vee \mathsf{New}(X,s)$$

POSL $$\frac{\psi \wedge \mathsf{SafeNet}(s,\mathcal{K})\ [S]_X\ \mathsf{SendsSafeMsg}(X,s,\mathcal{K}) \wedge \mathsf{Has}(Y,M) \wedge \neg\mathsf{SafeMsg}(M,s,\mathcal{K})}{\psi \wedge \mathsf{SafeNet}(s,\mathcal{K})\ [S]_X\ \exists k \in \mathcal{K}.\ \mathsf{Has}(Y,k) \vee \mathsf{New}(Y,s)},$$

where S is any basic sequence of actions.

Following are useful theorems which follow easily from the axioms.

SREC $\mathsf{SafeNet}(s,\mathcal{K}) \wedge \mathsf{Receive}(X,M) \supset \mathsf{SafeMsg}(M,s,\mathcal{K})$

SSND $\mathsf{SafeNet}(s,\mathcal{K}) \wedge \mathsf{Send}(X,M) \supset \mathsf{SafeMsg}(M,s,\mathcal{K})$

The collection of new axioms and rules are summarized in Appendix B. We write $\Gamma \vdash \gamma$ if γ is provable from the formulas in Γ and any axiom or inference rule of the proof system,

except the honesty rule **HON** from previous formulations of PCL (see Appendix A) and the secrecy rule **NET**. We write $\mathcal{Q} \vdash \gamma$ if γ is provable from the axioms and inference rules of the proof system including the rules **HON** and **NET** for protocol $\mathcal{Q}$.

In the following theorem and proof, the closure $\tilde{\mathcal{M}}$ of a set $\mathcal{M}$ of messages is the least set containing $\mathcal{M}$ and closed under pairing, unpairing, encryption with any public key or symmetric key, decryption with a private key or a symmetric key not in $\mathcal{K}$, and hashing.

Theorem 1 *If $\mathcal{M}$ is a set of messages, all safe with respect to secret s and key-set $\mathcal{K}$ then the closure $\tilde{\mathcal{M}}$ contains only safe messages.*

Proof. Since $\tilde{\mathcal{M}}$ is the minimal set satisfying the given conditions, any element $m \in \tilde{\mathcal{M}}$ can be constructed from elements in $\mathcal{M}$ using a finite sequence of the operations enumerated. From the semantics of $\mathsf{SafeMsg}$ it is easily seen that all the operations preserve safeness. Hence by induction, all the elements of $\tilde{\mathcal{M}}$ will be safe. □

Lemma 1 *If a thread X possesses an unsafe message with respect to secret s and key-set $\mathcal{K}$ then either X received an unsafe message earlier, or X generated s, or X possesses a key in $\mathcal{K}$.*

Proof. Suppose thread X does not satisfy any of the conditions enumerated. Then all the messages it initially knows and has received are safe messages. Since it does not have a key in $\mathcal{K}$, the list of operations in theorem 1 enumerates a superset of all the operations it can do on this initial safe set (in the Dolev-Yao model). Hence, by theorem 1, X cannot compute any unsafe message. So it cannot possess an unsafe message – a contradiction. □

Theorem 2 (Soundness) *If $\mathcal{Q} \vdash \gamma$, then $\mathcal{Q} \vDash \gamma$. Furthermore, if $\Gamma \vdash \gamma$, then $\Gamma \vDash \gamma$.*

Proof. Soundness for this proof system is proved by induction on the length of proofs of the axioms and rules. The most interesting cases are sketched below, after the following definition.

A *prefix closed* formula Φ is a formula such that if a run R of a protocol $\mathcal{Q}$ satisfies Φ then any prefix of R also satisfies Φ. For example, the formula $\neg\mathsf{Send}(X, t)$ is prefix closed. This is because if in any run R, thread X has not sent the term t, it cannot have sent t in any prefix of R. In general, the negation of any action formula is prefix closed. Another example is $\forall X.\ \mathsf{New}(X, s) \supset \hat{X} = \hat{A}$ because this can be re-written as $\forall X.\ \neg\mathsf{New}(X, s) \vee \hat{X} = \hat{A}$ which is a disjunction of the negation of an action formula and an equality constraint.

NET : Consider a run R of protocol $\mathcal{Q}$ such that the consequent of **NET** is false. We will show that the antecedent is false too. We have $\mathcal{Q}, R \vDash \mathsf{KOHonest}(s, \mathcal{K}) \wedge \Phi$, but $\mathcal{Q}, R \nvDash \mathsf{SafeNet}(s, \mathcal{K})$. This implies that $\mathcal{Q}, R \vDash \exists m, X.\ \mathsf{Send}(X, m) \wedge \neg\mathsf{SafeMsg}(m, s, \mathcal{K})$. Note that there must be a first instance when an unsafe message is sent out - let $\tilde{m}$ be the first such message. Hence, we can split R into $R_0.R_1.R_2$ such that $\mathcal{Q}, R_0 \vDash \mathsf{SafeNet}(s, \mathcal{K})$ and $R_1 = \langle X\ \texttt{sends}\ \tilde{m}; Y\ \texttt{receives}\ \tilde{m}\rangle$, for some Y.

More formally, let us have:

1. $\mathcal{Q}, R \vDash \mathsf{KOHonest}(s, \mathcal{K}) \wedge \Phi$
2. $\mathcal{Q}, R \nvDash \mathsf{SafeNet}(s, \mathcal{K})$

Condition 2 implies that $\mathcal{Q}, R \vDash \exists m, X.\ \mathsf{Send}(X, m) \wedge \neg\mathsf{SafeMsg}(m, s, \mathcal{K})$. Note that there must be a first instance when an unsafe message is sent out - let $\tilde{m}$ be the first such message. Hence, we can split R into $R_0.R_1.R_2$ such that:

- $\mathcal{Q}, R_0 \vDash \mathsf{SafeNet}(s, \mathcal{K})$
- $R_1 = \langle([\texttt{receive}\ x; S']_Y \mid [\texttt{send}\ \tilde{m}; T']_X \longrightarrow [S'(\tilde{m}/x)]_Y \mid [T']_X)\rangle$

Since this is the first send of an unsafe message, therefore X could not have received an unsafe message earlier. Therefore, by the lemma, either X generated s or, X has a key in $\mathcal{K}$. In both cases, $\mathsf{KOHonest}(s, \mathcal{K})$ implies $\mathsf{Honest}(\hat{X})$. Therefore the fragment $[\texttt{send}\ \tilde{m}]_X$ must be part of a sequence of actions $[P]_X$ such that P is a basic sequence of one of the roles in $\mathcal{Q}$. That is, $R = R'_0.R'_1.R'_2$ such that R'_0 is a prefix of R_0, P matches $R'_1|_X$ with substituition σ and R'_2 is the rest of R. So we have:

- P matches $R'_1|_X$ with substituition σ
- $\mathcal{Q}, R'_0 \vDash \mathsf{SafeNet}(s, \mathcal{K})$
- $\mathcal{Q}, R'_0.R'_1 \vDash \mathsf{Honest}(\hat{X}) \wedge \Phi$, since Φ is prefix closed.
- $\mathcal{Q}, R'_0.R'_1 \nvDash \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$

Hence, we have: $\mathcal{Q}, R \nvDash \mathsf{SafeNet}(s, \mathcal{K})[P]_X \mathsf{Honest}(\hat{X}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$, thus violating the premise.

POS : $\mathsf{SafeNet}(s, \mathcal{K})$ implies no thread sent out an unsafe message in the run. Hence no thread received an unsafe message. Therefore, by lemma 1, any thread X possessing an unsafe message must have either generated s or possesses a key in $\mathcal{K}$.

POSL : The premise of the rule informally states that starting from a "safe" network and additional constraints ψ thread X concludes that some thread Y possesses an unsafe message M in all possible runs of any protocol. Specifically this should be true for a run where thread X executes the basic sequence $[S]_X$ uninterspersed with the actions of any other thread except the receipt of messages sent by X. Now the premise implies that X only sends safe messages - also since S is a basic sequence, the only message that X can receive in $[S]_X$ will be only at its beginning, which, due to the starting "safe" network precondition will be a safe message. Hence we can conclude that thread Y possessed an unsafe message before X started executing $[S]_X$ *i.e.*, when $\mathsf{SafeNet}(s, \mathcal{K})$ was true. Therefore using axiom **POS** we derive that thread Y either generated s or possesses a key in $\mathcal{K}$, which establises the conclusion of **POSL**.

Formally, assume that the following formula is valid:

$$\mathcal{P} : \psi \wedge \mathsf{SafeNet}(s, \mathcal{K})[S]_X \mathsf{SendsSafeMsg}(X, s, \mathcal{K}) \wedge \mathsf{Has}(Y, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K})$$

Consider R, an arbitrary run of the protocol $\mathcal{Q}$ such that $R = R_0.R_1.R_2$ and the following conditions hold:

1. S matches $R_1|_X$ with substituition σ.
2. $\mathcal{Q}, R_0 \models \sigma(\psi \wedge \mathsf{SafeNet}(s, \mathcal{K}))$

Therefore, from the validity of $\mathcal{P}$ we have:

$$\mathcal{Q}, R_0.R_1 \models \sigma(\mathsf{SendsSafeMsg}(X, s, \mathcal{K}) \wedge \mathsf{Has}(Y, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K}))$$

Now, we construct a run $R_1' \cong \sigma S$, that is, R_1' has only actions of the thread X (any send/receive by X is with a buffer chord). Since the conditions 1 and 2 still hold for the run $R' = R_0.R_1'.R_2$, we have:

$$\mathcal{Q}, R_0.R_1' \models \sigma(\mathsf{SendsSafeMsg}(X, s, \mathcal{K}) \wedge \mathsf{Has}(Y, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K}))$$

We have two cases here: $Y = X$ or, $Y \neq X$. In the first case, since $\mathcal{Q}, R_0 \models \sigma\mathsf{SafeNet}(s, \mathcal{K})$ and $[S]_X$ can receive at most once - just after R_0, therefore, if thread X possesses an unsafe message then $\sigma(\exists k \in \mathcal{K}.\ \mathsf{Has}(X, k) \vee \mathsf{New}(X, s))$ - and this fact cannot be altered by further actions of X.

In the second case, we observe that R_1' does not contain the action of any thread other than X, excepting receipt of the messages sent by X, which are safe anyway. Therefore, $\mathcal{Q}, R_0 \models \sigma(\mathsf{Has}(Y, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K}))$. From this, condition 2 and **POS** we have: $\mathcal{Q}, R_0 \models \sigma(\exists k \in \mathcal{K}.\ \mathsf{Has}(Y, k) \vee \mathsf{New}(Y, s))$. Again, further actions by any thread after R_0 cannot alter this fact. Therefore, $\mathcal{Q}, R_0.R_1 \models \sigma(\exists k \in \mathcal{K}.\mathsf{Has}(Y, k) \vee \mathsf{New}(Y, s))$.

Hence, for all runs R the following formula holds:

$$\mathcal{Q}, R \models \psi \wedge \mathsf{SafeNet}(s, \mathcal{K})\ [S]_X\ \exists k \in \mathcal{K}.\ \mathsf{Has}(Y, k) \vee \mathsf{New}(Y, s)$$

□

4. Analysis of a variant of NSL

In this section we use the proof system developed in section 3 to prove a secrecy property of a simple variant $NSLVAR$ of the Needham-Schroeder-Lowe protocol, proposed in [16], in which parties A and B use an authenticated temporary secret n_a to establish a secret key k that is in turn used to protect the actual message m. The main difference from the original NSL protocol is that the initiator's nonce is leaked in the final message. Reasoning from A's point of view, nonce n_a should be secret between A and B at the point of the run in the protocol where A is just about to send the last message. This protocol was originally used to demonstrate a limitation of the original rank function method in reasoning about temporary secrets. Modal formulas in PCL allow us to naturally express and prove properties that hold at intermediate points of a protocol execution.

Formally, $NSLVAR$ is a protocol defined by roles $\{\mathbf{Init}, \mathbf{Resp}\}$, with the roles, written using the protocol program notation, given in Table 3.

Theorem 3 *Let* $\tilde{\mathbf{Init}}$ *denote the initial segment of the initiator's role ending just before the last send action. The nonce* n_a *is a shared secret between* A *and* B *in every state of the protocol where* A *has executed* $\tilde{\mathbf{Init}}$ *and no further actions, as long as both* $\hat{A}$ *and* $\hat{B}$ *are honest. Formally,*

$$NSLVAR \vdash [\tilde{\mathbf{Init}}]_A\ \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B}) \supset (\mathsf{Has}(X, n_a) \supset \hat{X} = \hat{A} \vee \hat{X} = \hat{B})$$

$\mathbf{Init} = (A, \hat{B}, m)$ [	$\mathbf{Resp} = (B)$ [
new n_a;	receive enc_{r1};
$enc_{r1} :=$ pkenc $\hat{A}.n_a, \hat{B}$;	$text_{r1} :=$ pkdec $enc_{r1}, \hat{B}$;
send enc_{r1};	match $text_{r1}$ as $\hat{A}.n_a$;
	new k;
receive enc_i;	$enc_i :=$ pkenc $n_a.\hat{B}.k, \hat{A}$;
$text_i :=$ pkdec $enc_i, \hat{A}$;	send enc_i;
match $text_i$ as $n_a.\hat{B}.k$;	
$enc_{r2} :=$ symenc m, k;	
	receive $enc_{r2}.n_a$;
send $enc_{r2}.n_a$;	$m :=$ symdec enc_{r2}, k;
$]_A$	$]_B$

Table 3. Formal description of *NSLVAR*

Proof Sketch. To prove the secrecy property, we start off by proving an authentication property $[\tilde{\mathbf{Init}}]_A \, \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B}) \supset \Phi$, where Φ is the conjunction of the following formulas:

$$\Phi_1 : \forall X, \hat{Y}.\, \mathsf{New}(X, n_a) \wedge \mathsf{PkEnc}(X, \hat{X}.n_a, \hat{Y}) \supset \hat{Y} = \hat{B}$$

$$\Phi_2 : \forall X, \hat{Y}, n.\, \mathsf{New}(X, n_a) \supset \neg \mathsf{PkEnc}(X, n.\hat{X}.n_a, \hat{Y})$$

$$\Phi_3 : \forall X, e.\, \mathsf{New}(X, n_a) \supset \neg \mathsf{Send}(X, e.n_a)$$

$$\Phi_4 : \mathsf{Honest}(\hat{X}) \wedge \mathsf{Send}(X, e.n) \supset \mathsf{New}(X, n)$$

$$\Phi_5 : \mathsf{Honest}(\hat{X}) \wedge \mathsf{PkEnc}(X, \hat{X'}.n, \hat{Y}) \supset \hat{X'} = \hat{X}$$

Informally, Φ_1 and Φ_2 hold because from the thread A's point of view it is known that it itself generated the nonce n_a and did not send it out encrypted with any other principal's public key except $\hat{B}$'s and that too in a specific format described by the protocol. Φ_3 holds because we are considering a state in the protocol execution where A has not yet sent the last message - sending of the last message will make $\mathsf{Send}(A, e.n_a)$ true with $e = E_{sym}[k](m)$. These intuitive explanations can be formalized using a previously developed fragment of PCL but we will omit those steps in this paper. Φ_4 and Φ_5 follow from a straightforward use of the honesty rule.

In the next step we prove the antecedents of the **NET** rule. We take $\mathcal{K} = \{\bar{k_A}, \bar{k_B}\}$ where the bar indicates private key which makes $\mathsf{KeyHon}(\mathcal{K}) \equiv \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B})$. In addition, since thread A generates n_a, therefore $\mathsf{KOHonest}(n_a, \mathcal{K}) \equiv \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B})$. We show that all basic sequence of the protocol send "safe" messages, assuming that formula Φ holds and that the predicate SafeNet holds at the beginning of that basic sequence. Formally, for every basic sequence $\mathbf{P} \in$

	$[\tilde{\mathbf{Init}}]_A\ \mathsf{New}(A, n_a)$	(1)
$(-1), \mathbf{N1}$	$[\tilde{\mathbf{Init}}]_A\ \mathsf{New}(X, n_a) \supset X = A$	(2)
	$\mathsf{Start}(A)[]_A\ \neg\mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \vee \hat{Y} = \hat{B}$	(3)
	$\neg\mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \vee \hat{Y} = \hat{B}\ [\texttt{new}\ \ n_a;]_A\ \neg\mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \vee \hat{Y} = \hat{B}$	(4)
	$\top\ [enc_{r1} := \texttt{pkenc}\ \ \hat{A}.n_a, \hat{B};]_A\ \mathsf{PkEnc}(A, \hat{A}.n_a, \hat{B})$	(5)
	$\neg\mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \vee \hat{Y} = \hat{B}\ [\texttt{send}\ \ enc_{r1};$ $\texttt{receive}\ \ enc_i;$ $text_i := \texttt{pkdec}\ \ enc_i, \hat{A};$ $\texttt{match}\ \ text_i\ \texttt{as}\ \ n_a.\hat{B}.k;$ $enc_{r2} := \texttt{symenc}\ \ m, k;]_A\ \neg\mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \vee \hat{Y} = \hat{B}$	(6)
	$[\tilde{\mathbf{Init}}]_A\ \mathsf{PkEnc}(A, \hat{A}.n_a, \hat{Y}) \supset \hat{Y} = \hat{B}$	(7)
(-1)	$[\tilde{\mathbf{Init}}]_A\ \Phi_1$	(8)

Table 4. Formal proof of $[\tilde{\mathbf{Init}}]_A\ \Phi_1$

$\{\mathbf{Init}_1, \mathbf{Init}_2, \mathbf{Resp}_1, \mathbf{Resp}_2\}$ we prove that:

$$\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{P}]_{A'}\ \mathsf{Honest}(\hat{A'}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(A', n_a, \mathcal{K})$$

The formal proof is done in Appendix C. The variables used in the basic sequence we are inducting over are consistently primed so that we do not capture variables in Φ, n_a or $\mathcal{K}$. Finally, we use the $\mathbf{NET}$ rule and $\mathbf{POS}$ axiom to show that n_a is a shared secret between A and B at a state where A has just finished executing $\tilde{\mathbf{Init}}$. $\square$

5. Compositional Reasoning for Secrecy

In this section, we present composition theorems that allow secrecy proofs of compound protocols to be built up from proofs of their parts. An application of this method to the Kerberos protocol is given in the next section. We consider three kinds of composition operations on protocols—*parallel*, *sequential*, and *staged*—as in our earlier work [10,15]. However, adapting that approach for reasoning about secrecy requires new insights. One central concept in our compositional proof methods is the notion of an *invariant*. An invariant for a protocol is a logical formula that characterizes the environment in which it retains its security properties. While in previous work we had one rule for establishing invariants (the $\mathbf{HON}$ rule [10]), reasoning about secrecy requires, in addition, the $\mathbf{NET}$ rule introduced in this paper. A second point of difference arises from the fact that reasoning about secrecy requires a certain degree of global knowledge.

Specifically, while proving that a protocol step does not violate secrecy, it is sometimes necessary to use information from earlier steps. In the technical presentation, this history information shows up as preconditions in the secrecy induction of the sequential and staged composition theorems.

Definition 2 (Parallel Composition) *The parallel composition $\mathcal{Q}_1 \mid \mathcal{Q}_2$ of protocols $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is the union of the sets of roles of $\mathcal{Q}_1$ and $\mathcal{Q}_2$.*

The parallel composition operation allows modelling agents who simultaneously engage in sessions of multiple protocols. The parallel composition theorem provides a method for ensuring that security properties established independently for the constituent protocols are still preserved in such a situation.

Theorem 4 (Parallel Composition) *If $\mathcal{Q}_1 \vdash \Gamma$ and $\Gamma \vdash \Psi$ and $\mathcal{Q}_2 \vdash \Gamma$ then $\mathcal{Q}_1 \mid \mathcal{Q}_2 \vdash \Psi$, where Γ denotes the set of invariants used in the proof of Ψ.*

One way to understand the parallel composition theorem is to visualize the proof tree for Ψ for protocol $\mathcal{Q}_1$ in red and green colors. The steps which use the invariant rules are colored red and correspond to the part $\mathcal{Q}_1 \vdash \Gamma$, while all other proof steps are colored green and correspond to the part $\Gamma \vdash \Psi$. While composing protocols, all green steps are obviously preserved since they involve proof rules which hold for all protocols. The red steps could possibly be violated because of $\mathcal{Q}_2$. For example, one invariant may state that honest principals only sign messages of a certain form, while $\mathcal{Q}_2$ may allow agents to sign other forms of messages. The condition $\mathcal{Q}_2 \vdash \Gamma$ ensures that this is not the case, i.e., the red steps still apply for the composed protocol.

Definition 3 (Sequential Composition) *A protocol $\mathcal{Q}$ is a sequential composition of two protocols $\mathcal{Q}_1$ and $\mathcal{Q}_2$, if each role of $\mathcal{Q}$ is obtained by the sequential composition of a role of $\mathcal{Q}_1$ with a role of $\mathcal{Q}_2$.*

In practice, key exchange is usually followed by a secure message transmission protocol which uses the resulting shared key to protect data. Sequential composition is used to model such compound protocols. Formally, the composed role $P_1; P_2$ is obtained by concatenating the actions of P_1 and P_2 with the output parameters of P_1 substituted for the input parameters of P_2 (cf. [10]).

Theorem 5 (Sequential Composition) *If $\mathcal{Q}$ is a sequential composition of protocols $\mathcal{Q}_1$ and $\mathcal{Q}_2$ then we can conclude $\mathcal{Q} \vdash \mathsf{KOHonest}(s, \mathcal{K}) \wedge \Phi \supset \mathsf{SafeNet}(s, \mathcal{K})$ if the following conditions hold for all $P_1; P_2$ in $\mathcal{Q}$, where $P_1 \in \mathcal{Q}_1$ and $P_2 \in \mathcal{Q}_2$:*

1. *(Secrecy induction)*
   - $\forall i. \forall S \in BS(P_i).\ \theta_{P_i} \wedge \mathsf{SafeNet}(s, \mathcal{K})\ [S]_X\ \mathsf{Honest}(\hat{X}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$
2. *(Precondition induction)*
   - $\mathcal{Q}_1 \mid \mathcal{Q}_2 \vdash \mathsf{Start}(X) \supset \theta_{P_1}$ *and* $\mathcal{Q}_1 \mid \mathcal{Q}_2 \vdash \theta_{P_1}[P_1]_X\ \theta_{P_2}$
   - $\forall i. \forall S \in BS(P_i).\ \theta_{P_i}[S]_X\ \theta_{P_i}$.

The final conclusion of the theorem is a statement that secrecy of s is preserved in the composed protocol. The secrecy induction is very similar to the **NET** rule. It states that all basic sequences of the two roles only send out safe messages. This step is

compositional since the condition is proved independently for steps of the two protocols. One point of difference from the **NET** rule is the additional precondition θ_{P_i}. This formula usually carries some information about the history of the execution, which helps in deciding what messages are safe for A to send out. For example, if θ_{P_i} says that A received some message m, then it is easy to establish that m is a safe message for A to send out again. The precondition induction proves that the θ_{P_i}'s hold at each point where they are assumed in the secrecy induction. The first bullet states the base case of the induction: θ_{P_1} holds at the beginning of the execution and θ_{P_2} holds when P_1 completes. The second bullet states that the basic sequences of P_1 and P_2 preserve their respective preconditions.

Definition 4 (Staged Composition) *A protocol $\mathcal{Q}$ is a staged composition of protocols $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n$ if each role of $\mathcal{Q}$ is of the form $RComp(\langle R_1, R_2, \ldots, R_n\rangle)$, where R_i is a role of protocol $\mathcal{Q}_i$.*

Consider the representation of sequential composition of n protocols as a directed graph with edges from $\mathcal{Q}_i$ to $\mathcal{Q}_{i+1}$. The staged composition operation extends sequential composition by allowing self loops and arbitrary backward arcs in this chain. This control flow structure is common in practice, e.g., Kerberos [17], IEEE 802.11i [1], and IKEv2 [6]. A role in this composition, denoted $RComp(\langle ...\rangle)$ corresponds to a possible execution path in the control flow graph by a single thread (cf. [15]). Note that the roles are built up from a finite number of basic sequences of the component protocol roles.

Theorem 6 (Staged Composition) *If $\mathcal{Q}$ is a staged composition of protocols $\mathcal{Q}_1$, $\mathcal{Q}_2$, $\cdots$, $\mathcal{Q}_n$ then we can conclude $\mathcal{Q} \vdash \mathsf{KOHonest}(s, \mathcal{K}) \wedge \Phi \supset \mathsf{SafeNet}(s, \mathcal{K})$ if for all $RComp(\langle P_1, P_2, \cdots, P_n\rangle) \in \mathcal{Q}$:*

1. *(Secrecy induction)*
   - $\forall i. \forall S \in BS(P_i).\ \theta_{P_i} \wedge \mathsf{SafeNet}(s, \mathcal{K})\ [S]_X\ \mathsf{Honest}(\hat{X}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$
2. *(Precondition induction)*
   - $\mathcal{Q}_1 \mid \mathcal{Q}_2 \cdots \mid \mathcal{Q}_n \vdash \mathsf{Start}(X) \supset \theta_{P_1}$ *and* $\mathcal{Q}_1 \mid \mathcal{Q}_2 \cdots \mid \mathcal{Q}_n \vdash \forall i.\ \theta_{P_i}[P_i]_X\ \theta_{P_{i+1}}$
   - $\forall i. \forall S \in \bigcup_{j \geq i} BS(P_j).\ \theta_{P_i}[S]_X\ \theta_{P_i}$.

The secrecy induction for staged composition is the same as for sequential composition. However, the precondition induction requires additional conditions to account for the control flows corresponding to backward arcs in the graph. The technical distinction surfaces in the second bullet of the precondition induction. It states that precondition θ_{P_i} should also be preserved by basic sequences of all higher numbered components, i.e., components from which there could be backward arcs to the beginning of P_i.

6. Analysis of Kerberos V5

In this section we analyze Kerberos V5, which was described in section 2. The security properties of Kerberos that we prove are listed in table 5. We abbreviate the honesty assumptions by defining $\mathsf{Hon}(\hat{X_1}, \cdots, \hat{X_n}) \equiv \mathsf{Honest}(\hat{X_1}) \wedge \cdots \mathsf{Honest}(\hat{X_n})$. The security objectives are of two types: authentication and secrecy. The authentication objectives take the form that a message of a certain format was indeed sent by some thread

$$SEC_{akey} : \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset (\mathsf{Has}(X, AKey) \supset \hat{X} \in \{\hat{C}, \hat{K}, \hat{T}\})$$

$$SEC_{skey} : \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}, \hat{S}) \supset (\mathsf{Has}(X, SKey) \supset \hat{X} \in \{\hat{C}, \hat{K}, \hat{T}, \hat{S}\})$$

$$AUTH_{kas} : \exists \eta.\ \mathsf{Send}((\hat{K}, \eta), \hat{C}.E_{sym}[k^{t\rightarrow k}_{T,K}](AKey.\hat{C}).E_{sym}[k^{c\rightarrow k}_{C,K}](AKey.n_1.\hat{T}))$$

$$AUTH_{tgs} : \exists \eta.\ \mathsf{Send}((\hat{T}, \eta), \hat{C}.E_{sym}[k^{s\rightarrow t}_{S,T}](SKey.\hat{C}).E_{sym}[AKey](SKey.n_2.\hat{S}))$$

$$SEC^{client}_{akey} : [\mathbf{Client}]_C\ SEC_{akey} \qquad AUTH^{client}_{kas} : [\mathbf{Client}]_C\ \mathsf{Hon}(\hat{C}, \hat{K}) \supset AUTH_{kas}$$

$$SEC^{kas}_{akey} : [\mathbf{KAS}]_K\ SEC_{akey} \qquad AUTH^{tgs}_{kas} : [\mathbf{TGS}]_T\ \mathsf{Hon}(\hat{T}, \hat{K}) \supset \exists n_1.\ AUTH_{kas}$$

$$SEC^{tgs}_{akey} : [\mathbf{TGS}]_T\ SEC_{akey}$$

$$AUTH^{client}_{tgs} : [\mathbf{Client}]_C\ \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset AUTH_{tgs}$$

$$SEC^{client}_{skey} : [\mathbf{Client}]_C\ SEC_{skey} \qquad AUTH^{server}_{tgs} : [\mathbf{Server}]_S\ \mathsf{Hon}(\hat{S}, \hat{T})$$

$$SEC^{tgs}_{skey} : [\mathbf{TGS}]_T\ SEC_{skey} \qquad \supset \exists n_2, AKey.\ AUTH_{tgs}$$

Table 5. Kerberos Security Properties

of the expected principal. The secrecy objectives take the form that a putative secret is known only to certain principals. For example, $AUTH^{client}_{kas}$ states that when the thread C finishes executing the **Client** role, some thread of $\hat{K}$ (the KAS) indeed sent the expected message; SEC^{client}_{akey} states that the authorization key is secret after execution of the **Client** role by C; the other security properties are analogous.

Theorem 7 (KAS Authentication) *On execution of the* **Client** *role by a principal it is guaranteed that the intended KAS indeed sent expected response assuming that the both the client and the KAS are honest. Similar result holds for a principal executing the* **TGS** *role. Formally, KERBEROS* $\vdash AUTH^{client}_{kas}, AUTH^{tgs}_{kas}$

Proof Sketch. In the course of executing the **Client** role, principal $\hat{C}$ receives a message containing the encrypted term $E_{sym}[k^{c\rightarrow k}_{C,K}](AKey.n_1.\hat{T})$. Using axiom **ENC4**, we derive that this message was encrypted by one of the owners of $k^{c\rightarrow k}_{C,K}$, which is either $\hat{C}$ or $\hat{K}$. Then, by using the rule **HON** we establish that no thread of $\hat{C}$ does this unless $\hat{C} = \hat{K}$, and so this must be some thread of $\hat{K}$. Once again we use the **HON** rule to reason that if an honest thread encrypts a message of this form then it also sends out a message of the form described in $AUTH_{kas}$. The proof of $AUTH^{tgs}_{kas}$ is along identical lines. In Appendix D.2, we first give a *template* proof for the underlying reasoning and then instantiate it for both $AUTH^{client}_{kas}$ and $AUTH^{tgs}_{kas}$. □

Theorem 8 (Authentication Key Secrecy) *On execution of the* **Client** *role by a principal, secrecy of the Authentication Key is preserved assuming that the client, the KAS and the TGS are all honest. Similar results hold for principals executing the* **KAS** *and* **TGS** *roles. Formally, KERBEROS* $\vdash SEC^{client}_{akey}, SEC^{kas}_{akey}, SEC^{tgs}_{akey}$

Proof Sketch. In Appendix D.3 we formally prove the secrecy of the session key $AKey$ with respect to the key-set $\mathcal{K} = \{k^{c\rightarrow k}_{C,K}, k^{t\rightarrow k}_{T,K}\}$. The proof is modular and broadly, there are two stages to the proof:

1. In the first stage we assume certain conditions, denoted Φ, and the honesty of principals $\hat{C}, \hat{K}$ and $\hat{T}$ and prove that this implies $\mathsf{SafeNet}(AKey, \mathcal{K})$. The proof of this part uses the Staged Composition Theorem. The components of this proof are:
   - secrecy induction - we will describe this shortly.
   - precondition induction - in case of *KERBEROS* most basic sequences do not need any precondition to facilitate the secrecy induction. For two of the basic sequences in the Client program, the preconditions are simply of the form that a certain message was received. Since receiving a message is a monotonic property, that is - once it is true it is always true thereafter - the precondition induction goes through simply.
2. In the second stage we prove that execution of the $\mathbf{Client}, \mathbf{KAS}$ or the $\mathbf{TGS}$ roles discharge the assumptions Φ. These proofs are derived from the authentication properties $AUTH_{kas}^{client}, AUTH_{kas}^{tgs}$. Now we combine the two derivations, use the $\mathbf{POS}$ axiom and conclude $SEC_{akey}^{client}, SEC_{akey}^{kas}$ and SEC_{akey}^{tgs}.

As the form of the secrecy induction suggests, we do an induction over all the basic sequences of *KERBEROS*. Broadly, the induction uses a combination of the following types of reasoning:

- The secrecy axioms enumerated in the proof system section. The structure of Kerberos suggests that in many of the basic sequences the messages being sent out are functions of messages received. A key strategy here is to use $\mathbf{NET1}$ and the safe network hypothesis to derive that the message received is safe and then proceed to prove that the messages being sent out are also safe. Consider as an example the sequence of actions by an application server thread $[\mathbf{Server}]_{S'}$: S' receives a message $E_{sym}[SKey'](\hat{C'}.t')$ and sends out a message $E_{sym}[SKey'](t')$. It is provable, just by using the $\mathbf{SAF}$ axioms that the later message is safe if the former message is safe.

- Derivations from Φ: The structure of Φ is dictated by the structure of the basic sequences we are inducing over. A practical proof strategy is starting the induction without figuring out a Φ at the outset and construct parts of the Φ as we do induction over an individual basic sequence. In case of *KERBEROS*, these parts are formulae that state that the generating thread of the putative secret $AKey$ did not perform certain types of action on $AKey$ or did it in a restricted form. The motivation for this structure of the Φ parts is that many of the basic sequences generate new nonces and send them out unprotected or protected under a set of keys different from $\mathcal{K}$. The Φ parts tell us that this is not the way the secret in consideration was sent out. For example consider one of the parts $\Phi_1 : \forall X, M.\ \mathsf{New}(X, AKey) \supset \neg(\mathsf{Send}(X, M) \wedge \mathsf{ContainsOpen}(M, AKey))$ - this tells us that the generator of $AKey$ did not send it out unprotected in the open.

- Derivations from the θ's, that is, the preconditions. These are conditions which are true at the beginning of the basic sequence we are inducing over with respect to the staged control flow that *KERBEROS* exhibits. As before, a practical proof strategy is to find out what precondition we need for the secrecy induction and do the precondition induction part afterwards. Consider for example the end of the first stage of the client thread $[\mathbf{Client}]_{C'}$. We know that at the beginning of the second stage the following formula always holds - $\theta : \mathsf{Receive}(\hat{C'}, tgt'.E_{sym}[k_{C',K'}^{c \rightarrow k}](AKey'.n_1'.\hat{T'}))$. The reason this information is necessary is that the second stage sends out tgt' in the open - in order to reason that this send is safe, given the safe network hypothesis at the beginning of the

second stage, we use the precondition and the theorem **SREC** to derive that tgt' was safe to begin with. □

Theorem 9 (TGS Authentication) *On execution of the* **Client** *role by a principal it is guaranteed that the intended TGS indeed sent the expected response assuming that the client, the KAS and the TGS are all honest. Similar result holds for a principal executing the* **Server** *role. Formally, KERBEROS* $\vdash AUTH_{tgs}^{client}, AUTH_{tgs}^{server}$

Proof Sketch. The proof of $AUTH_{tgs}^{server}$ can be instantiated from the *template* proof for theorem 7 and is formally done in Appendix D.2. The proof of $AUTH_{tgs}^{client}$ uses the secrecy property SEC_{akey}^{client} established in theorem 8 and is formally done in Appendix D.4. At a high level, the client reasons that since $AKey$ is known only to $\hat{C}$, $\hat{K}$ and $\hat{T}$, the term $E_{sym}[AKey](SKey.n_2.\hat{S})$ - which it receives during the protocol execution - could only have been computed by one of them. Some non-trivial technical effort is required to prove that this encryption was indeed done by a thread of $\hat{T}$ and not by any thread of $\hat{C}$ or $\hat{K}$, which could have been the case if *e.g.*, there existed a reflection attack. After showing that it was indeed a thread of $\hat{T}$ who encrypted the term, we use the honesty rule to show that it indeed sent the expected response to C's message. □

Theorem 10 (Service Key Secrecy) *On execution of the* **Client** *role by a principal, secrecy of the Service Key is preserved assuming that the client, the KAS, the TGS and the application server are all honest. Similar result holds for a principal executing the* **TGS** *role. Formally, KERBEROS* $\vdash SEC_{skey}^{client}, SEC_{skey}^{tgs}$

Proof Sketch. The idea here is that the Service Key $SKey$ is protected by the key-set $\{k_{S,T}^{s \to t}, AKey\}$. The proof of this theorem being very similar to the proof of theorem 8 is omitted from this paper. □

Kerberos with PKINIT

We prove theorems for Kerberos with PKINIT [22] that are analogous to theorems 7-10 and are listed in Table 6. The proofs are omitted due to space constraints. In the first stage of Kerberos with PKINIT, the KAS establishes the authorization key encrypted with a symmetric key which in turn is sent to the client encrypted with its public key. For client $\hat{C}$ and KAS $\hat{K}$ let us denote this symmetric key by $k_{C,K}^{pkinit}$. Since the structure of the rest of the protocol remains the same with respect to the level of formalization in this paper [7], we can take advantage of the PCL proofs for the symmetric key version. In particular, the proofs for the properties of Kerberos with PKINIT analogous to $AUTH_{kas}^{tgs}, AUTH_{tgs}^{client}$ and $AUTH_{tgs}^{server}$ are identical in structure to the symmetric key version. The proof of the property corresponding to $AUTH_{kas}^{client}$ is different because of the differing message formats in the first stage. There is an additional step of proving the secrecy of $k_{C,K}^{pkinit}$, after which the secrecy proofs of $AKey$ and $SKey$ are reused with only the induction over the first stage of the client and the KAS being redone.

7. Related Work

Some secrecy proofs using the CSP [20] or strand space [21] protocol execution model use inductive arguments that are similar to the form of inductive reasoning codified in

$$SEC_k : \mathsf{Hon}(\hat{C}, \hat{K}) \supset (\mathsf{GoodKeyAgainst}(X, k) \vee \hat{X} \in \{\hat{C}, \hat{K}\})$$

$$SEC_{akey} : \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset (\mathsf{GoodKeyAgainst}(X, AKey) \vee \hat{X} \in \{\hat{C}, \hat{K}, \hat{T}\})$$

$$SEC_{skey} : \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}, \hat{S}) \supset (\mathsf{GoodKeyAgainst}(X, SKey) \vee \hat{X} \in \{\hat{C}, \hat{K}, \hat{T}, \hat{S}\})$$

$$AUTH_{kas} : \exists \eta.\ \mathsf{Send}((\hat{K}, \eta), E_{pk}[pk_C](Cert_K.SIG[sk_K](k.ck)).$$
$$\hat{C}.E_{sym}[k^{t \rightarrow k}_{T,K}](AKey.\hat{C}).E_{sym}[k](AKey.n_1.t_K.\hat{T}))$$

$$AUTH_{tgs} : \exists \eta.\ \mathsf{Send}((\hat{T}, \eta), \hat{C}.E_{sym}[k^{s \rightarrow t}_{S,T}](SKey.\hat{C}).E_{sym}[AKey](SKey.n_2.\hat{S}))$$

$$SEC_k^{client} : [\mathbf{Client}]_C\ SEC_k \qquad SEC_k^{kas} : [\mathbf{KAS}]_K\ SEC_k$$

$$SEC_{akey}^{client} : [\mathbf{Client}]_C\ SEC_{akey} \qquad AUTH_{kas}^{client} : [\mathbf{Client}]_C\ \mathsf{Hon}(\hat{C}, \hat{K}) \supset AUTH_{kas}$$

$$SEC_{akey}^{kas} : [\mathbf{KAS}]_K\ SEC_{akey} \qquad AUTH_{kas}^{tgs} : [\mathbf{TGS}]_T\ \mathsf{Hon}(\hat{T}, \hat{K})$$

$$SEC_{akey}^{tgs} : [\mathbf{TGS}]_T\ SEC_{akey} \qquad \supset \exists n_1, k, ck, t_K.\ AUTH_{kas}$$

$$AUTH_{tgs}^{client} : [\mathbf{Client}]_C\ \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset AUTH_{tgs}$$

$$SEC_{skey}^{client} : [\mathbf{Client}]_C\ SEC_{skey} \qquad AUTH_{tgs}^{server} : [\mathbf{Server}]_S\ \mathsf{Hon}(\hat{S}, \hat{T})$$

$$SEC_{skey}^{tgs} : [\mathbf{TGS}]_T\ SEC_{skey} \qquad \supset \exists n_2, AKey.\ AUTH_{tgs}$$

Table 6. PKINIT Security Properties

our formal system. For example, within CSP, properties of messages that may appear on the network have been identified by defining a *rank function* [20,16], with an inductive proof used to show that rank is preserved by the attacker actions and all honest parties. In comparison, arguments in our formal logic use a conjunction involving the SafeNet predicate and protocol specific properties Φ in our inductive hypotheses. These two formulas together characterize the set of possible messages appearing on the network and can be viewed as a symbolic definition of a rank function. We believe that our method is as powerful as the rank function method for any property expressible in our logic. However, it is difficult to prove a precise connection without first casting the rank function method in a formal setting that relies on a specific class of message predicates.

One drawback of the rank functions approach is that the induction is performed by "global" reasoning – trying to capture all possible properties of the system at once. This makes the method less applicable since it cannot handle protocols which deal with temporary secrets or use authentication to ensure secrecy properties. Although some of these issues can be resolved by extensions of the rank function method [13,12], we expect that the tools available in PCL are more general and may be better suited for application to real-world protocols.

Our composition theorems allow us to use a divide-and-conquer approach for complex protocols with different parts serving different purposes. By varying the preconditions of the secrecy induction in the staged composition theorem, we are essentially modifying the rank function as we shift our attention from one protocol stage to the other.

Because of its widespread deployment and relative complexity, Kerberos has been the subject of several logical studies. Bella and Paulson use automated theorem proving techniques to reason explicitly about properties of Kerberos that hold in all traces containing actions of honest parties and a malicious attacker [3]. Our high-level axiomatic proofs are significantly more concise since we do not require explicit reasoning about attacker actions. Another line of work uses a multiset rewriting model [4,2] to develop proofs in the symbolic and computational model. However, proofs in these papers use unformalized (though rigorous) mathematical arguments and are not modular.

8. Conclusion

We present formal axioms and proof rules for inductive reasoning about secrecy and prove soundness of this system over a conventional symbolic model of protocol execution. The proof system uses a *safe message* predicate to express that any secret conveyed by the message is protected by a key from a chosen list. This predicate allows us to define two additional concepts: a principal *sends safe messages* if every message it sends is safe, and the *network is safe* if every message sent by every principal is safe.

Our main inductive rule for secrecy, **NET**, states that if every honest principal preserves safety of the network, then the network is safe, assuming that only honest principals have access to keys in the chosen list. The remainder of the system makes it possible to discharge assumptions used in the proof, and prove (when appropriate) that only honest principals have the chosen keys. While it might initially seem that network safety depends on the actions of malicious agents, a fundamental advantage of Protocol Composition Logic is that proofs only involve induction over protocol steps executed by honest parties.

We illustrate the expressiveness of the logic presented in this paper by proving properties of two protocols, a variant of the Needham-Schroeder protocol that illustrates the ability to reason about temporary secrets, and Kerberos. The modular nature of the secrecy and authentication proofs for Kerberos makes it possible to reuse proofs about the basic version of the protocol for the PKINIT version that uses public-key infrastructure instead of shared secret keys in the initial steps. Compositional secrecy proofs are made possible by the composition theorems developed in section 5 of this paper.

We have also developed a proof system for secrecy analysis that is sound over a "computational" protocol execution model which involves probabilistic polynomial-time computation [19]. The proofs of Kerberos security properties in the computationally sound logic turn out to be syntactically analogous to the symbolic version described in this paper. However, the proofs for NSL and variants are not entirely analogous to the symbolic versions. Specifically, these proofs involve axioms capturing some subtle ways in which cryptographic reduction proofs work which do not seem to have a direct correspondence with the symbolic way of interpreting the cryptographic primitives.

References

[1] IEEE P802.11i/D10.0. Medium Access Control (MAC) security enhancements, amendment 6 to IEEE Standard for local and metropolitan area networks part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications., April 2004.

[2] M. Backes, I. Cervesato, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Cryptographically sound security proofs for basic and public-key kerberos. In *Proceedings of 11th European Symposium on Research in Computer Security*, 2006. To appear.

[3] G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proceedings of the 5th European Symposium on Research in Computer Security*, pages 361–375, Louvain-la-Neuve, Belgium, Sept. 1998. Springer-Verlag LNCS 1485.

[4] F. Butler, I. Cervesato, A. D. Jaggard, and A. Scedrov. A Formal Analysis of Some Properties of Kerberos 5 Using MSR. In *Fifteenth Computer Security Foundations Workshop — CSFW-15*, pages 175–190, Cape Breton, NS, Canada, 24–26 June 2002. IEEE Computer Society Press.

[5] F. Butler, I. Cervesato, A. D. Jaggard, and A. Scedrov. Verifying confidentiality and authentication in kerberos 5. In *ISSS*, pages 1–24, 2003.

[6] E. C. Kaufman. Internet Key Exchange (IKEv2) Protocol, 2005. RFC.

[7] I. Cervesato, A. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key kerberos. Technical report.

[8] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, pages 109–125. IEEE, 2003.

[9] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition. In *Proceedings of 19th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 83. Electronic Notes in Theoretical Computer Science, 2004.

[10] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13:423–482, 2005.

[11] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.

[12] R. Delicata and S. Schneider. Temporal rank functions for forward secrecy. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*, pages 126–139. IEEE Computer Society, 2005.

[13] R. Delicata and S. A. Schneider. Towards the rank function verification of protocols that use temporary secrets. In *Proceedings of the Workshop on Issues in the Theory of Security: WITS '04*, 2004.

[14] N. Durgin, J. C. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 241–255. IEEE, 2001.

[15] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of ieee 802.11i and tls. In *ACM Conference on Computer and Communications Security*, pages 2–15, 2005.

[16] J. Heather. Strand spaces and rank functions: More than distant cousins. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 104, 2002.

[17] J. Kohl and B. Neuman. The kerberos network authentication service, 1991. RFC.

[18] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[19] A. Roy, A. Datta, A. Derek, and J. Mitchell. Inductive proofs of computational secrecy. In *Proc. 12th European Symposium On Research In Computer Security*, 2007.

[20] S. Schneider. Verifying authentication protocols with csp. *IEEE Transactions on Software Engineering*, pages 741–58, 1998.

[21] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1), 1999.

[22] L. Zhu and B. Tung. Public key cryptography for initial authentication in kerberos, 2006. Internet Draft.

A. Protocol Logic

A.1. Axioms and Inference Rules

A representative fragment of the axioms and inference rules in the proof system are collected in Table 7. For expositional convenience, we divide the axioms into four groups.

The axioms about protocol actions state properties that hold in the state reached by executing one of the actions in a state in which formula ϕ holds. Note that the a in axiom **AA1** is any one of the actions and a is the corresponding predicate in the logic. Axiom **N1** states that two different threads cannot generate the same nonce while axiom **AN2** states that if a thread generates a nonce and does nothing else, only that thread possesses the nonce.

The possession axioms reflect a fragment of Dolev-Yao rules for constructing or decomposing messages while the encryption axioms symbolically model encryption. The generic rules are used for manipulating modal formulas.

A.2. The Honesty Rule

The honesty rule is essentially an invariance rule for proving properties of all roles of a protocol. It is similar to the basic invariance rule of LTL [18]. The honesty rule is used to combine facts about one role with inferred actions of other roles.

For example, suppose Alice receives a response from a message sent to Bob. Alice may wish to use properties of Bob's role to reason about how Bob generated his reply. In order to do so, Alice may assume that Bob is honest and derive consequences from this assumption. Since honesty, by definition in this framework, means "following one or more roles of the protocol," honest principals must satisfy every property that is a provable invariant of the protocol roles. Using the notation just introduced, the honesty rule may be written as follows.

$$\frac{[\,]_X\ \phi \quad \forall \rho \in \mathcal{Q}. \forall P \epsilon BS(\rho).\ \phi\ [P]_X\ \phi}{\mathcal{Q} \vdash \mathsf{Honest}(\hat{X}) \supset \phi}\ \mathbf{HON} \qquad \begin{array}{l}\text{no free variable}\\ \text{in } \phi \text{ except } X\\ \text{bound in } [P]_X\end{array}$$

In words, if ϕ holds at the beginning of every role of $\mathcal{Q}$ and is preserved by all its basic sequences, then every honest principal executing protocol $\mathcal{Q}$ must satisfy ϕ. The side condition prevents free variables in the conclusion $\mathsf{Honest}(\hat{X}) \supset \phi$ from becoming bound in any hypothesis. Intuitively, since ϕ holds in the initial state and is preserved by all basic sequences, it holds at all pausing states of any run.

Axioms for protocol actions

AA1 $\phi[a]_X\ \mathsf{a}$

AA2 $\mathsf{Start}(X)[\,]_X\ \neg\mathsf{a}(\mathsf{X})$

AA3 $\neg\mathsf{Send}(X,t)[b]_X\neg\mathsf{Send}(X,t)$ if $\sigma\mathsf{Send}(X,t) \neq \sigma\mathsf{b}$ for all substitutions σ

AN2 $\phi[\mathtt{new}\ \ x]_X\ \mathsf{Has}(Y,x) \supset (Y = X)$

ARP $\mathsf{Receive}(X,p(x))[\mathtt{match}\ \ q(x)\ \mathtt{as}\ \ q(t)]_X\ \mathsf{Receive}(X,p(t))$

P1 $\mathsf{Persist}(X,t)[a]_X\ \mathsf{Persist}(X,t)$, for $\mathsf{Persist} \in \{\mathsf{Has},\mathsf{Send},\mathsf{Receive}\}$

N1 $\mathsf{New}(X,n) \wedge \mathsf{New}(Y,n) \supset X = Y$

Possession Axioms

ORIG $\mathsf{New}(X,x) \supset \mathsf{Has}(X,x)$ **TUP** $\mathsf{Has}(X,x) \wedge \mathsf{Has}(X,y) \supset \mathsf{Has}(X,x.y)$

REC $\mathsf{Receive}(X,x) \supset \mathsf{Has}(X,x)$ **PROJ** $\mathsf{Has}(X,x.y) \supset \mathsf{Has}(X,x) \wedge \mathsf{Has}(X,y)$

Encryption Axioms

Let $\mathsf{Enc} \in \{\mathsf{SymEnc},\mathsf{PkEnc}\}, \mathsf{Dec} \in \{\mathsf{SymDec},\mathsf{PkDec}\}$ in the following:

ENC0 $[m' := \mathtt{enc}\ \ m,k;]_X\ \mathsf{Enc}(X,m,k)$

ENC1 $\mathsf{Start}(X)\ [\,]_X\ \neg\mathsf{Enc}(X,m,k)$

ENC2 $\pi(X,m,k)\ [\mathsf{a}]_X\ \pi(X,m,k)$, for $\pi \in \{\mathsf{Enc},\neg\mathsf{Enc}\}$

where, either $\mathsf{a} \neq \mathtt{enc}\ \ \cdots$ or, $\mathsf{a} = (p := \mathtt{enc}\ \ k', q)$, such that $(q,k') \neq (m,k)$

ENC3 $\mathsf{Enc}(X,m,k) \supset \mathsf{Has}(X,k) \wedge \mathsf{Has}(X,m)$

ENC4 $\mathsf{SymDec}(X,E[k](m),k) \supset \exists Y.\ \mathsf{SymEnc}(Y,m,k)$

PENC4 $\mathsf{PkDec}(X,E[k](m),\bar{k}) \supset \exists Y.\ \mathsf{PkEnc}(Y,m,k)$

Generic Rules

$$\frac{\theta[P]_X\phi \quad \theta[P]_X\psi}{\theta[P]_X\phi \wedge \psi}\ \mathbf{G1} \qquad \frac{\theta' \supset \theta \quad \theta[P]_X\phi \quad \phi \supset \phi'}{\theta'[P]_X\phi'}\ \mathbf{G2} \qquad \frac{\phi}{\theta[P]_X\phi}\ \mathbf{G3}$$

Table 7. Fragment of the Proof System

B. New Definitions, Axioms and Rules for Secrecy

$$\mathsf{SendsSafeMsg}(X,s,\mathcal{K}) \equiv \forall M.\ (\mathsf{Send}(X,M) \supset \mathsf{SafeMsg}(M,s,\mathcal{K}))$$

$$\mathsf{SafeNet}(s,\mathcal{K}) \equiv \forall X.\ \mathsf{SendsSafeMsg}(X,s,\mathcal{K})$$

$$\mathsf{KeyHonest}(\mathcal{K}) \equiv \forall X.\ \forall k \in \mathcal{K}.\ (\mathsf{Has}(X,k) \supset \mathsf{Honest}(\hat{X}))$$

$$\mathsf{OrigHonest}(s) \equiv \forall X.\ (\mathsf{New}(X,s) \supset \mathsf{Honest}(\hat{X}))$$

$$\mathsf{KOHonest}(s,\mathcal{K}) \equiv \mathsf{KeyHonest}(\mathcal{K}) \wedge \mathsf{OrigHonest}(s)$$

SAF0 $\neg\mathsf{SafeMsg}(s, s, \mathcal{K}) \wedge \mathsf{SafeMsg}(x, s, \mathcal{K})$,

where x is an atomic term different from s

SAF1 $\mathsf{SafeMsg}(M_0.M_1, s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M_0, s, \mathcal{K}) \wedge \mathsf{SafeMsg}(M_1, s, \mathcal{K})$

SAF2 $\mathsf{SafeMsg}(E_{sym}[k](M), s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M, s, \mathcal{K}) \vee k \in \mathcal{K}$

SAF3 $\mathsf{SafeMsg}(E_{pk}[k](M), s, \mathcal{K}) \equiv \mathsf{SafeMsg}(M, s, \mathcal{K}) \vee \bar{k} \in \mathcal{K}$

SAF4 $\mathsf{SafeMsg}(HASH(M), s, \mathcal{K})$

NET $\forall \rho \in \mathcal{Q}. \forall P \in BS(\rho).$

$$\frac{\mathsf{SafeNet}(s, \mathcal{K})\ [P]_X\ \mathsf{Honest}(\hat{X}) \wedge \Phi \supset \mathsf{SendsSafeMsg}(X, s, \mathcal{K})}{\mathcal{Q} \vdash \mathsf{KOHonest}(s, \mathcal{K}) \wedge \Phi \supset \mathsf{SafeNet}(s, \mathcal{K})}\ (*)$$

$(*)$: $[P]_A$ does not capture free variables in Φ, $\mathcal{K}$, s, and Φ is prefix closed.

NET0 $\mathsf{SafeNet}(s, \mathcal{K})\ [\,]_X\ \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$

NET1 $\mathsf{SafeNet}(s, \mathcal{K})\ [\texttt{receive}\ \ M]_X\ \mathsf{SafeMsg}(M, s, \mathcal{K})$

NET2 $\mathsf{SendsSafeMsg}(X, s, \mathcal{K})\ [\mathtt{a}]_X\ \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$, where $\mathtt{a}$ is not a send.

NET3 $\mathsf{SendsSafeMsg}(X, s, \mathcal{K})\ [\texttt{send}\ \ M]_X\ \mathsf{SafeMsg}(M, s, \mathcal{K}) \supset \mathsf{SendsSafeMsg}(X, s, \mathcal{K})$

POS $\mathsf{SafeNet}(s, \mathcal{K}) \wedge \mathsf{Has}(X, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K})$

$\supset \exists k \in \mathcal{K}.\ \mathsf{Has}(X, k) \vee \mathsf{New}(X, s)$

POSL $$\frac{\psi \wedge \mathsf{SafeNet}(s, \mathcal{K})\ [S]_X\ \mathsf{SendsSafeMsg}(X, s, \mathcal{K}) \wedge \mathsf{Has}(Y, M) \wedge \neg\mathsf{SafeMsg}(M, s, \mathcal{K})}{\psi \wedge \mathsf{SafeNet}(s, \mathcal{K})\ [S]_X\ \exists k \in \mathcal{K}.\ \mathsf{Has}(Y, k) \vee \mathsf{New}(Y, s)},$$

where S is any basic sequence of actions.

SREC $\mathsf{SafeNet}(s, \mathcal{K}) \wedge \mathsf{Receive}(X, M) \supset \mathsf{SafeMsg}(M, s, \mathcal{K})$

SSND $\mathsf{SafeNet}(s, \mathcal{K}) \wedge \mathsf{Send}(X, M) \supset \mathsf{SafeMsg}(M, s, \mathcal{K})$

C. PCL Proof of NSL Variant Secrecy

As in the theorem, $\tilde{\mathbf{Init}}$ is the initial segment of the $\mathbf{Init}$ role excluding the last send action. To prove the secrecy property, we start off by proving an authentication property $[\tilde{\mathbf{Init}}]_A\ \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B})\ \supset\ \Phi$, where Φ is the conjunction of the following formulas:

$$\Phi_1 : \forall X, \hat{Y}.\ \mathsf{New}(X, n_a) \wedge \mathsf{Send}(X, E_{pk}[k_Y](\hat{X}.n_a)) \supset \hat{Y} = \hat{B}$$

$$\Phi_2 : \forall X, \hat{Y}, n.\ \mathsf{New}(X, n_a) \supset \neg\mathsf{Send}(X, E_{pk}[k_Y](n.\hat{X}.n_a))$$

$$\Phi_3 : \forall X, e.\ \mathsf{New}(X, n_a) \supset \neg\mathsf{Send}(X, e.n_a)$$

$$\Phi_4 : \mathsf{Honest}(\hat{X}) \wedge \mathsf{Send}(X, E_{sym}[k_0](m_0).n) \supset \mathsf{New}(X, n)$$

$$\Phi_5 : \mathsf{Honest}(\hat{X}) \wedge \mathsf{PkEnc}(X, \hat{X}'.n, \hat{Y}) \supset \hat{X}' = \hat{X}$$

In the next step we prove the antecedents of the $\mathbf{NET}$ rule. We take $\mathcal{K} = \{\bar{k_A}, \bar{k_B}\}$ where the bar indicates private key which makes $\mathsf{KeyHon}(\mathcal{K}) \equiv \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B})$. In addition, since thread A generates n_a, therefore $\mathsf{KOHonest}(n_a, \mathcal{K}) \equiv \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B})$. We show that all basic sequence of the protocol send "safe" messages, assuming that formula Φ holds and that the predicate SafeNet holds at the beginning of that basic sequence. Formally, for every basic sequence $\mathbf{P} \in \{\mathbf{Init}_1, \mathbf{Init}_2, \mathbf{Resp}_1, \mathbf{Resp}_2\}$ we prove that:

$$\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{P}]_{A'}\ \mathsf{Honest}(\hat{A}') \wedge \Phi \supset \mathsf{SendsSafeMsg}(A', n_a, \mathcal{K})$$

The variables used in the basic sequence we are inducting over are consistently primed so that we do not capture variables in Φ, n_a or $\mathcal{K}$. Finally, we use the $\mathbf{NET}$ rule and $\mathbf{POS}$ axiom to show that n_a is a shared secret between A and B at a state where A has just finished executing $\tilde{\mathbf{Init}}$.

Let, $[\mathbf{Init}_1]_{A'}$: $[\mathtt{new}\ \ n'_a;$
$\quad enc'_{r1} := \mathtt{pkenc}\ \ \hat{A}'.n'_a, \hat{B}';$
$\quad \mathtt{send}\ \ enc'_{r1};]_{A'}$

Case 1 : $n'_a \neq n_a$ (1)

(1) $[\mathbf{Init}_1]_{A'}\ \mathsf{SafeMsg}(E_{pk}[k_{B'}](\hat{A}'.n'_a), n_a, \mathcal{K})$ (2)

(2), $\mathbf{NET}*$ $\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{Init}_1]_{A'}\ \mathsf{SendsSafeMsg}(A', n_a, \mathcal{K})$ (3)

Case 2 : $n'_a = n_a$ (4)

$[\mathbf{Init}_1]_{A'}\ \mathsf{New}(A', n_a) \wedge \mathsf{Send}(A', E_{pk}[k_{B'}](\hat{A}'.n_a))$ (5)

Φ_1 $[\mathbf{Init}_1]_{A'}\ \hat{B}' = \hat{B}$ (6)

(6) $[\mathbf{Init}_1]_{A'}\ \mathsf{SafeMsg}(E_{pk}[k_{B'}](\hat{A}'.n'_a), n_a, \mathcal{K})$ (7)

(7), $\mathbf{NET}*$ $\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{Init}_1]_{A'}\ \mathsf{SendsSafeMsg}(A', n_a, \mathcal{K})$ (8)

Let, $[\mathbf{Init}_2]_{A'}$: $[\texttt{receive}\ enc'_i;$

$text'_i := \texttt{pkdec}\ enc'_i, \hat{A}';$

$\texttt{match}\ text'_i\ \texttt{as}\ n'_a.\hat{B}'.k';$

$enc'_{r2} := \texttt{symenc}\ m', k';$

$\texttt{send}\ enc'_{r2}.n'_a;]_{A'}$

$$[\mathbf{Init}_2]_{A'}\ \mathsf{Send}(A', E_{sym}[k'](m').n'_a) \tag{9}$$

$$\Phi_4 \quad \mathsf{Honest}(\hat{X}) \wedge \mathsf{Send}(X, E_{sym}[k_0](m_0).n) \supset \mathsf{New}(X, n) \tag{10}$$

$$(9), (10) \quad [\mathbf{Init}_2]_{A'}\ \mathsf{New}(A', n'_a) \wedge \mathsf{Send}(A', E_{sym}[k'](m').n'_a) \tag{11}$$

$$\Phi_3, (11) \quad [\mathbf{Init}_2]_{A'}\ n'_a \neq n_a \tag{12}$$

$$\mathbf{SAF0}, (12) \quad [\mathbf{Init}_2]_{A'}\ \mathsf{SafeMsg}(n'_a, n_a, \mathcal{K}) \tag{13}$$

$$\mathbf{SAF0} \quad [\mathbf{Init}_2]_{A'}\ \mathsf{SafeMsg}(m', n_a, \mathcal{K}) \tag{14}$$

$$\mathbf{SAF}*, (13), (14) \quad [\mathbf{Init}_2]_{A'}\ \mathsf{SafeMsg}(E_{sym}[k'](m').n'_a, n_a, \mathcal{K}) \tag{15}$$

$$(15) \quad \mathsf{SafeNet}(n_a, \mathcal{K})\ [\mathbf{Init}_2]_{A'}\ \mathsf{SendsSafeMsg}(A', n_a, \mathcal{K}) \tag{16}$$

Let, $[\mathbf{Resp}_1]_{B'}$: $[\texttt{receive}\ enc'_{r1};$

$text'_{r1} := \texttt{pkdec}\ enc'_{r1}, \hat{B}';$

$\texttt{match}\ text'_{r1}\ \texttt{as}\ \hat{A}'.n'_a;$

$\texttt{new}\ k';$

$enc'_i := \texttt{pkenc}\ n'_a.\hat{B}'.k', \hat{A}';$

$\texttt{send}\ enc'_i;]_{B'}$

$$[\mathbf{Resp}_1]_{B'}\ \mathsf{New}(B', k') \wedge \mathsf{Send}(B', E_{pk}[k_{A'}](n'_a.\hat{B}'.k')) \tag{17}$$

$$\Phi_2, (17) \quad [\mathbf{Resp}_1]_{B'}\ k' \neq n_a \tag{18}$$

$$\text{Case 1} : \mathsf{SafeMsg}(n'_a, n_a, \mathcal{K}) \tag{19}$$

$$\mathbf{SAF}*, (18) \quad \mathsf{SafeMsg}(E_{pk}[k_{A'}](n'_a.\hat{B}'.k'), n_a, \mathcal{K}) \tag{20}$$

$$\mathbf{NET}*, (20) \quad \mathsf{SafeNet}(n_a, \mathcal{K})\ [\mathbf{Resp}_1]_{B'}\ \mathsf{SendsSafeMsg}(B', n_a, \mathcal{K}) \tag{21}$$

Case 2 : $\neg\mathsf{SafeMsg}(n'_a, n_a, \mathcal{K})$ (22)

ENC4 $[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\exists X.\ \mathsf{PkEnc}(X, \hat{A}'.n'_a, \hat{B}')$ (23)

Inst $X \mapsto X_0$ $[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\mathsf{PkEnc}(X_0, \hat{A}'.n'_a, \hat{B}')$ (24)

(24) $[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\mathsf{Has}(X_0, n'_a)$ (25)

NET$*$, (25) $\mathsf{SafeNet}(n_a, \mathcal{K})$

$[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\mathsf{SendsSafeMsg}(B', n_a, \mathcal{K}) \wedge \mathsf{Has}(X_0, n'_a) \wedge \neg\mathsf{SafeMsg}(n'_a, n_a, \mathcal{K})$ (26)

POSL, (26) $\mathsf{SafeNet}(n_a, \mathcal{K})$

$[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\exists k \in \mathcal{K}.\ \mathsf{Has}(X_0, k) \vee \mathsf{New}(X_0, n_a)$ (27)

(27) $\hat{X_0} = \hat{A} \vee \hat{X_0} = \hat{B}$ (28)

(28) $\mathsf{Honest}(\hat{X_0})$ (29)

Φ_5 $\mathsf{Honest}(\hat{X}) \wedge \mathsf{PkEnc}(X, \hat{X}'.n, \hat{Y}) \supset \hat{X}' = \hat{X}$ (30)

(24), (28), $\mathsf{SafeNet}(n_a, \mathcal{K})$

(29), (30) $[\texttt{receive}\ enc'_{r1}; \texttt{match}\ enc'_{r1}\ \texttt{as}\ E_{pk}[k_{B'}](\hat{A}'.n'_a);]_{B'}$

$\hat{A}' = \hat{A} \vee \hat{A}' = \hat{B}$ (31)

SAF3, (31) $\mathsf{SafeNet}(n_a, \mathcal{K})\ [\mathbf{Resp}_1]_{B'}\ \mathsf{SafeMsg}(E_{pk}[k_{A'}](n'_a.\hat{B}'.k'), n_a, \mathcal{K})$ (32)

(32) $\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{Resp}_1]_{B'}\mathsf{SendsSafeMsg}(B', n_a, \mathcal{K})$ (33)

Let, $[\mathbf{Resp}_2]_{B'}$: $[\texttt{receive}\ enc'_{r2}.n'_a;$

$m' := \texttt{symdec}\ enc'_{r2}, k';]_{B'}$

NET$*$ $\mathsf{SafeNet}(n_a, \mathcal{K})[\mathbf{Resp}_2]_{B'}\mathsf{SendsSafeMsg}(B', n_a, \mathcal{K})$ (34)

NET $[\tilde{\mathbf{Init}}]_A\ \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B}) \supset \mathsf{SafeNet}(n_a, \mathcal{K})$ (35)

POS, (35) $[\tilde{\mathbf{Init}}]_A\ \mathsf{Honest}(\hat{A}) \wedge \mathsf{Honest}(\hat{B}) \supset (\mathsf{Has}(X, n_a) \supset \hat{X} = \hat{A} \vee \hat{X} = \hat{B})$ (36)

D. Proof of Kerberos Security Properties

D.1. Environmental Assumptions

Long term symmetric keys possessed by pairs of honest principals are possessed by only themselves.

$$\Gamma_0 : \forall X, Y, Z, type.\ \mathsf{Hon}(\hat{X}, \hat{Y}) \wedge \mathsf{Has}(Z, k^{type}_{X,Y}) \supset (\hat{Z} = \hat{X} \vee \hat{Z} = \hat{Y})$$

D.2. Proofs of $AUTH^{client}_{kas}$, $AUTH^{tgs}_{kas}$ and $AUTH^{server}_{tgs}$

Below we give a *template* proof of $[\mathbf{Role}]_X\ \mathsf{Hon}(\hat{X}, \hat{Y}) \supset \exists \eta.\ \mathsf{SymEnc}((\hat{Y}, \eta), M_1, k^{type}_{X,Y})$, where $\mathbf{Role}$ receives the message $M_0.E_{sym}[k^{type}_{X,Y}](M_1).M_2$.

Reference to equations by negative numbers is relative to the current equation - *e.g.*, (-1) refers to the last equation. Reference by positive number indicates the actual number of the equation.

$$[\mathbf{Role}]_X\ \mathsf{SymDec}(X, E_{sym}[k^{type}_{X,Y}](M_1), k^{type}_{X,Y}) \tag{1}$$

$$\mathsf{Hon}(\hat{X}, \hat{Y}), \Gamma_0 \quad [\mathbf{Role}]_X\ \exists \eta.\ \mathsf{SymEnc}((\hat{X}, \eta), M_1, k^{type}_{X,Y})$$

$$\mathbf{ENC4}, (-1) \qquad \vee\ \exists \eta.\ \mathsf{SymEnc}((\hat{Y}, \eta), M_1, k^{type}_{X,Y}) \tag{2}$$

$$\text{Case 1} : \hat{X} = \hat{Y} \tag{3}$$

$$(-2, -1) \quad [\mathbf{Role}]_X\ \exists \eta.\ \mathsf{SymEnc}((\hat{Y}, \eta), M_1, k^{type}_{X,Y}) \tag{4}$$

$$\text{Case 2} : \hat{X} \neq \hat{Y} \tag{5}$$

$$\mathbf{HON} \quad \mathsf{Honest}(\hat{X_0}) \wedge \hat{X_0} \neq \hat{Y_0} \supset \forall M.\ \neg \mathsf{SymEnc}(X_0, M, k^{type}_{X_0,Y_0}) \tag{6}$$

$$\mathsf{Hon}(\hat{X}), (-1) \quad [\mathbf{Role}]_X\ \neg \exists \eta.\ \mathsf{SymEnc}((\hat{X}, \eta), M_1, k^{type}_{X,Y}) \tag{7}$$

$$(-6, -1) \quad [\mathbf{Role}]_X\ \exists \eta.\ \mathsf{SymEnc}((\hat{Y}, \eta), M_1, k^{type}_{X,Y}) \tag{8}$$

Instantiating for $AUTH^{client}_{kas}$:

$$[\mathbf{Client}]_C \; \exists \eta.\; \mathsf{SymEnc}((\hat{C}, \eta), AKey.n_1.\hat{T}, k^{c \rightarrow k}_{C,K}) \tag{9}$$

$$\mathbf{HON} \quad \mathsf{Honest}(\hat{X}) \wedge \mathsf{SymEnc}(X, Key.n.\hat{T}_0, k^{c \rightarrow k}_{C_0,X}) \supset \mathsf{Send}(X, \hat{C}_0.E_{sym}[k^{t \rightarrow k}_{T_0,X}](Key.\hat{C}_0).E_{sym}[k^{c \rightarrow k}_{C_0,X}](Key.n.\hat{T}_0)) \tag{10}$$

$$\mathsf{Hon}(\hat{K}), \; (-2,-1) \quad [\mathbf{Client}]_C \; \exists \eta.\; \mathsf{Send}((\hat{K}, \eta), \hat{C}.E_{sym}[k^{t \rightarrow k}_{T,K}](AKey.\hat{C}).\; E_{sym}[k^{c \rightarrow k}_{C,K}](AKey.n_1.\hat{T})) \tag{11}$$

$$(-1) \quad AUTH^{client}_{kas} \tag{12}$$

Instantiating for $AUTH^{tgs}_{kas}$:

$$[\mathbf{TGS}]_T \; \exists \eta.\; \mathsf{SymEnc}((\hat{K}, \eta), AKey.\hat{C}, k^{t \rightarrow k}_{T,K}) \tag{13}$$

$$\mathbf{HON} \quad \mathsf{Honest}(\hat{X}) \wedge \mathsf{SymEnc}(X, Key.\hat{C}_0, k^{t \rightarrow k}_{Y,X}) \supset \exists n.\; \mathsf{Send}(X, \hat{C}_0.E_{sym}[k^{t \rightarrow k}_{Y,X}](Key.\hat{C}_0).E_{sym}[k^{c \rightarrow k}_{C_0,X}](Key.n.\hat{Y})) \tag{14}$$

$$\mathsf{Hon}(\hat{K}), \; (-2,-1) \quad [\mathbf{TGS}]_T \; \exists \eta, n.\; \mathsf{Send}((\hat{K}, \eta), \hat{C}.E_{sym}[k^{t \rightarrow k}_{T,K}](AKey.\hat{C}).\; E_{sym}[k^{c \rightarrow k}_{C,K}](AKey.n_1.\hat{T})) \tag{15}$$

$$(-1) \quad AUTH^{tgs}_{kas} \tag{16}$$

Instantiating for $AUTH^{server}_{tgs}$:

$$[\mathbf{Server}]_S \; \exists \eta.\; \mathsf{SymEnc}((\hat{T}, \eta), E_{sym}[k^{s \rightarrow t}_{S,T}](SKey.\hat{C})) \tag{17}$$

$$\mathbf{HON} \quad \mathsf{Honest}(\hat{X}) \wedge \mathsf{SymEnc}(X, Key.\hat{C}_0, k^{s \rightarrow t}_{Y,X}) \supset \exists n, Key'.\; \mathsf{Send}(X, \hat{C}_0.E_{sym}[k^{s \rightarrow t}_{Y,X}](Key.\hat{C}_0).E_{sym}[Key'](Key.n.\hat{Y})) \tag{18}$$

$$\mathsf{Hon}(\hat{T}), \; (-2,-1) \quad [\mathbf{Server}]_S \; \exists \eta, n, Key'.\; \mathsf{Send}((\hat{T}, \eta), \hat{C}.E_{sym}[k^{s \rightarrow t}_{S,T}](SKey.\hat{C}).\; E_{sym}[Key'](SKey.n.\hat{S})) \tag{19}$$

$$(-1) \quad AUTH^{server}_{tgs} \tag{20}$$

D.3. Proof of $SEC^{client}_{akey}, SEC^{kas}_{akey}, SEC^{tgs}_{akey}$

In this section we formally prove the secrecy of the session key $AKey$ with respect to the key-set $\mathcal{K} = \{k^{c\rightarrow k}_{C,K}, k^{t\rightarrow k}_{T,K}\}$.

The assumed condition Φ is the conjunction of the following formulas where the predicate $\mathsf{ContainsOpen}(m, a)$ asserts that a can be obtained from m by a series of unpairings only - no decryption required.

$$\Phi_1 : \forall X, M.\, \mathsf{New}(X, AKey) \supset \neg(\mathsf{Send}(X, M) \wedge \mathsf{ContainsOpen}(M, AKey))$$

$$\Phi_2 : \forall X, \hat{C}_0, \hat{K}_0, \hat{T}_0, n.\, \mathsf{New}(X, AKey) \wedge \mathsf{SymEnc}(X, AKey.n.\hat{T}_0, k^{c\rightarrow k}_{\hat{C}_0,K_0})$$
$$\supset \hat{X} = \hat{K} \wedge \hat{C}_0 = \hat{C} \wedge \hat{T}_0 = \hat{T}$$

$$\Phi_3 : \forall X, \hat{S}_0, \hat{C}_0.\, \mathsf{New}(X, AKey) \supset \neg\mathsf{SymEnc}(X, AKey.\hat{C}_0, k^{s\rightarrow t}_{\hat{S}_0,X})$$

Observe that Φ is prefix closed. The only principals having access to a key in $\mathcal{K}$ are $\hat{C}, \hat{K}$ and $\hat{T}$. In addition, Φ_2 assumes that some thread of $\mathcal{K}$ generated $AKey$. Therefore, we have $\mathsf{KOHonest}(AKey, \mathcal{K}) \equiv \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T})$. As the form of the secrecy induction suggests, we do an induction over all the basic sequences of *KERBEROS*.

Let, $[\mathbf{Client}_1]_{C'} : [\texttt{new}\ n'_1; \texttt{send}\ \hat{C}'.\hat{T}'.n'_1;]_{C'}$

$$[\mathbf{Client}_1]_{C'}\ \mathsf{New}(C', n'_1) \wedge \mathsf{Send}(C', \hat{C}'.\hat{T}'.n'_1) \quad (1)$$

$\Phi_1, (1)$ $\quad [\mathbf{Client}_1]_{C'}\ n'_1 \neq AKey \quad (2)$

(2) $\quad [\mathbf{Client}_1]_{C'}\ \mathsf{SafeMsg}(\hat{C}'.\hat{T}'.n'_1, AKey, \mathcal{K}) \quad (3)$

$\mathbf{NET2}, (3)$ $\quad \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_1]_{C'}\ \mathsf{SendsSafeMsg}(C', AKey, \mathcal{K}) \quad (4)$

Let, $[\mathbf{Client}_2]_{C'} : [\texttt{receive}\ \hat{C}'.tgt'.enc'_{kc};$
$text'_{kc} := \texttt{symdec}\ enc'_{kc}, k^{c\rightarrow k}_{C',K'};$
$\texttt{match}\ text'_{kc}\ \texttt{as}\ AKey'.n'_1.\hat{T}';]_{C'}$

$$\mathbf{NET}* \quad \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_2]_{C'}\ \mathsf{SendsSafeMsg}(C', AKey, \mathcal{K}) \quad (5)$$

Precondition θ_3 : $\mathsf{Receive}(C', \hat{C}'.tgt'.E_{sym}[k^{c\rightarrow k}_{C',K'}](AKey'.n'_1.\hat{T}'))$

Let, $[\mathbf{Client}_3]_{C'}$: $[\texttt{new}\ \ n'_2;$

$enc'_{ct} := \texttt{symenc}\ \ \hat{C}', AKey';$

$\texttt{send}\ \ tgt'.enc'_{ct}.\hat{C}'.\hat{S}', n'_2;]_{C'}$

SREC $\mathsf{SafeNet}(AKey, \mathcal{K}) \wedge \theta_3 \supset$

$\mathsf{SafeMsg}(\hat{C}'.tgt'.E_{sym}[k^{c\rightarrow k}_{C',K'}](AKey'.n'_1.\hat{T}'), AKey, \mathcal{K})$ (6)

SAF1 $\mathsf{SafeMsg}(\hat{C}'.tgt'.E_{sym}[k^{c\rightarrow k}_{C',K'}](AKey'.n'_1.\hat{T}'), AKey, \mathcal{K}) \supset$

$\mathsf{SafeMsg}(tgt', AKey, \mathcal{K})$ (7)

(7) $\theta_3 \wedge \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_3]_{C'}\ \mathsf{SafeMsg}(tgt', AKey, \mathcal{K})$ (8)

$[\mathbf{Client}_3]_{C'}\ \mathsf{New}(C', n'_2) \wedge \mathsf{Send}(C', tgt'.E_{sym}[AKey'](\hat{C}').\hat{C}'.\hat{S}'.n'_2)$ (9)

Φ_1, (9) $[\mathbf{Client}_3]_{C'}\ n'_2 \neq AKey$ (10)

(8), (10) $\theta_3 \wedge \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_3]_{C'}$

$\mathsf{SafeMsg}(tgt', AKey, \mathcal{K}) \wedge \mathsf{SafeMsg}(n'_2, AKey, \mathcal{K})$ (11)

(11) $\theta_3 \wedge \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_3]_{C'}$

$\mathsf{SafeMsg}(tgt'.E_{sym}[AKey'](\hat{C}').\hat{C}'.\hat{S}'.n'_2, AKey, \mathcal{K})$ (12)

NET$*$, (12) $\theta_3 \wedge \mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Client}_3]_{C'}\ \mathsf{SendsSafeMsg}(C', AKey, \mathcal{K})$ (13)

$\cdots$ proof for following BS similar to (5) $\cdots$

$\mathsf{SafeNet}(AKey, \mathcal{K})\ [\texttt{receive}\ \ \hat{C}'.st'.enc'_{tc};$

$text'_{tc} := \texttt{symdec}\ \ enc'_{tc}, AKey';$

$\texttt{match}\ \ text'_{tc}\ \texttt{as}\ \ SKey'.n'_2.\hat{S}';]_{C'}$

$\mathsf{SendsSafeMsg}(C', AKey, \mathcal{K})$ (14)

Precondition θ_5 : $\mathsf{Receive}(C', \hat{C}'.st'.E_{sym}[AKey'](SKey'.n'_2.\hat{S}'))$

$\cdots$ proof for following BS similar to (13) $\cdots$

$\theta_5 \wedge \mathsf{SafeNet}(AKey, \mathcal{K})\ [enc'_{cs} := \texttt{symenc}\ \ \hat{C}'.t', SKey';$

$\texttt{send}\ \ st'.enc'_{cs};]_{C'}$

$\mathsf{SendsSafeMsg}(C', AKey, \mathcal{K})$ (15)

$\cdots$ proof for following BS similar to (5) $\cdots$

$\mathsf{SafeNet}(AKey, \mathcal{K})$ [receive enc'_{sc};

$text'_{sc} :=$ symdec $enc'_{sc}, SKey'$;

match $text'_{sc}$ as t';$]_{C'}$

$\mathsf{SendsSafeMsg}(C', AKey, \mathcal{K})$ (16)

Let, $[\mathbf{KAS}]_{K'}$: [receive $\hat{C}'.\hat{T}'.n'_1$;

new $AKey'$;

$tgt' :=$ symenc $AKey'.\hat{C}', k^{t\rightarrow k}_{T',K'}$;

$enc'_{kc} :=$ symenc $AKey'.n'_1.\hat{T}', k^{c\rightarrow k}_{C',K'}$;

send $\hat{C}'.tgt'.enc'_{kc}$;$]_{K'}$

Case 1 : $AKey' = AKey$

$[\mathbf{KAS}]_{K'}\mathsf{New}(K', AKey) \wedge \mathsf{SymEnc}(K', AKey.n'_1.\hat{T}', k^{c\rightarrow k}_{C',K'})$ (17)

Φ_2, (17) $[\mathbf{KAS}]_{K'}\hat{C}' = \hat{C} \wedge \hat{K}' = \hat{K} \wedge \hat{T}' = \hat{T}$ (18)

(18) $[\mathbf{KAS}]_{K'}k^{c\rightarrow k}_{C',K'} \in \mathcal{K} \wedge k^{t\rightarrow k}_{T',K'} \in \mathcal{K}$ (19)

$\mathbf{SAF}*$, (19) $\mathsf{SafeNet}(AKey, \mathcal{K})$ $[\mathbf{KAS}]_{K'}$ $\mathsf{SafeMsg}($

$\hat{C}'.E_{sym}[k^{t\rightarrow k}_{T',K'}](AKey'.\hat{C}').E_{sym}[k^{c\rightarrow k}_{C',K'}](AKey'.n'_1.\hat{T}'),$

$AKey, \mathcal{K})$ (20)

Case 2 : $AKey' \neq AKey$

NET1 $\mathsf{SafeNet}(AKey, \mathcal{K})$ [receive $\hat{C}'.\hat{T}'.n'_1$;$]_{K'}$ $\mathsf{SafeMsg}(\hat{C}'.\hat{T}'.n'_1, AKey, \mathcal{K})$ (21)

(21) $\mathsf{SafeNet}(AKey, \mathcal{K})$ [receive $\hat{C}'.\hat{T}'.n'_1$;$]_{K'}$ $\mathsf{SafeMsg}(n'_1, AKey, \mathcal{K})$ (22)

$\mathbf{SAF}*$, (22) $\mathsf{SafeNet}(AKey, \mathcal{K})$ $[\mathbf{KAS}]_{K'}$ $\mathsf{SafeMsg}($

$\hat{C}'.E_{sym}[k^{t\rightarrow k}_{T',K'}](AKey'.\hat{C}').E_{sym}[k^{c\rightarrow k}_{C',K'}](AKey'.n'_1.\hat{T}')),$

$AKey, \mathcal{K})$ (23)

(20), (23), $\mathbf{NET}*$ $\mathsf{SafeNet}(AKey, \mathcal{K})$ $[\mathbf{KAS}]_{K'}$ $\mathsf{SendsSafeMsg}(K', AKey, \mathcal{K})$ (24)

Let, $[\mathbf{TGS}]_{T'}$: $[\texttt{receive}\ tgt'.enc'_{ct}.\hat{C'}.\hat{S'}.n'_2;$

$text'_{tgt} := \texttt{symdec}\ tgt', k^{t\rightarrow k}_{T,K};$

$\texttt{match}\ text'_{tgt}\ \texttt{as}\ AKey'.\hat{C'};$

$text'_{ct} := \texttt{symdec}\ enc'_{ct}, AKey';$

$\texttt{match}\ text'_{ct}\ \texttt{as}\ \hat{C'};$

$\texttt{new}\ SKey';$

$st' := \texttt{symenc}\ SKey'.\hat{C'}, k^{s\rightarrow t}_{S,T};$

$enc'_{tc} := \texttt{symenc}\ SKey'.n'_2.\hat{S'}, AKey';$

$\texttt{send}\ \hat{C'}.st'.enc'_{tc};]_{T'}$

NET1, SAF1 $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\texttt{receive}\ enc'_{ct1}.enc'_{ct2}.\hat{C'}.\hat{S'}.n'_2;]_{K'}$

$\mathsf{SafeMsg}(n'_2, AKey, \mathcal{K})$ (25)

$[\mathbf{TGS}]_{T'}\ \mathsf{New}(T', SKey') \wedge \mathsf{SymEnc}(T', SKey'.\hat{C'}, k^{s\rightarrow t}_{S',T'})$ (26)

Φ_3, (26) $[\mathbf{TGS}]_{T'}\ SKey' \neq AKey$ (27)

(25), (27), **SAF**$*$ $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{TGS}]_{T'}\ \mathsf{SafeMsg}($

$\hat{C'}.E_{sym}[k^{s\rightarrow t}_{S',T'}](SKey'.\hat{C'}).E_{sym}[AKey'](SKey'.n'_2.\hat{S'}), AKey, \mathcal{K})$ (28)

NET$*$, (28) $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{TGS}]_{T'}\ \mathsf{SendsSafeMsg}(T', AKey, \mathcal{K})$ (29)

Let, $[\mathbf{Server}]_{S'}$: $[\texttt{receive}\ st'.enc'_{cs};$

$text'_{st} := \texttt{symdec}\ st', k^{s\rightarrow t}_{S,T};$

$\texttt{match}\ text'_{st}\ \texttt{as}\ SKey'.\hat{C'};$

$text'_{cs} := \texttt{symdec}\ enc'_{cs}, SKey';$

$\texttt{match}\ text'_{cs}\ \texttt{as}\ \hat{C'}.t';$

$enc'_{sc} := \texttt{symenc}\ t', SKey';$

$\texttt{send}\ enc'_{sc};]_{S'}$

NET1, SAF0 $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Server}]_{S'}\ \mathsf{SafeMsg}(E_{sym}[SKey'](\hat{C'}.t'), AKey, \mathcal{K})$ (30)

SAF$*$, (30) $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Server}]_{S'}\ \mathsf{SafeMsg}(t', AKey, \mathcal{K}) \vee SKey' \in \mathcal{K}$ (31)

SAF1, (31) $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Server}]_{S'}\ \mathsf{SafeMsg}(E_{sym}[SKey'](t'), AKey, \mathcal{K})$ (32)

NET2 $\mathsf{SafeNet}(AKey, \mathcal{K})\ [\mathbf{Server}]_{S'}\ \mathsf{SendsSafeMsg}(S', AKey, \mathcal{K})$ (33)

$$\text{Theorem 6} \quad \Phi \wedge \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset \mathsf{SafeNet}(AKey, \mathcal{K}) \tag{34}$$

$$\mathbf{POS}, (34) \quad \Phi \wedge \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset (\mathsf{Has}(X, AKey) \supset (\hat{X} = \hat{C} \vee \hat{X} = \hat{K} \vee \hat{X} = \hat{T})) \tag{35}$$

Based on $AUTH_{kas}^{client}$, the actions in $[\mathbf{KAS}]_K$, $AUTH_{tgs}^{client}$ and a few additional steps, we can infer that:

$$\textit{KERBEROS} \vdash [\mathbf{Client}]_C \; \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset \Phi$$

$$\textit{KERBEROS} \vdash [\mathbf{KAS}]_K \; \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset \Phi$$

$$\textit{KERBEROS} \vdash [\mathbf{TGS}]_T \; \mathsf{Hon}(\hat{C}, \hat{K}, \hat{T}) \supset \Phi$$

Combining these with the secrecy derivation (35) we have:

$$\textit{KERBEROS} \vdash SEC_{akey}^{client}, SEC_{akey}^{kas}, SEC_{akey}^{tgs}$$

D.4. Proof of $AUTH_{tgs}^{client}$

This proof uses the secrecy property SEC_{akey}^{client} which established the secrecy of $AKey$ among $\hat{C}, \hat{K}$ and $\hat{T}$ assuming their honesty. Again, reference to equations by negative numbers is relative to the current equation - *e.g.*, (-1) refers to the last equation. Reference by positive number indicates the actual number of the equation.

$$[\mathbf{Client}]_C\ \mathsf{SymDec}(C, E_{sym}[AKey](SKey.n_2.\hat{S}), AKey) \quad (1)$$

(-1) $$[\mathbf{Client}]_C\ \exists X.\ \mathsf{SymEnc}(X, SKey.n_2.\hat{S}, AKey) \quad (2)$$

Inst $X \mapsto X_0, (-1)$ $$[\mathbf{Client}]_C\ \mathsf{SymEnc}(X_0, SKey.n_2.\hat{S}, AKey) \quad (3)$$

$\mathbf{ENC3}, (-1)$ $$[\mathbf{Client}]_C\ \mathsf{Has}(X_0, AKey) \quad (4)$$

$SEC_{AKey}^{client}, (-1)$ $$\hat{X_0} = \hat{C} \wedge \hat{X_0} = \hat{K} \wedge \hat{X_0} = \hat{T} \quad (5)$$

$\mathbf{HON}$ $$\mathsf{Honest}(\hat{X}) \wedge \mathsf{SymEnc}(X, Key'.n.\hat{S_0}, Key) \wedge Key \neq k_{Z,X}^{c \rightarrow k} \supset \exists \hat{K_0}, \hat{C_0}.\ \mathsf{SymDec}(X, E_{sym}[k_{X,K_0}^{t \rightarrow k}](Key.\hat{C_0})) \wedge \mathsf{Send}(X, \hat{C_0}.E_{sym}[k_{S_0,X}^{s \rightarrow t}](Key'.\hat{C_0}).E_{sym}[Key](Key'.n.\hat{S_0})) \quad (6)$$

Inst, $(-4, -1)$ $$[\mathbf{Client}]_C\ \mathsf{SymDec}(X_0, E_{sym}[k_{X_0,K_0}^{t \rightarrow k}](AKey.\hat{C_0})) \wedge \mathsf{Send}(X_0, \hat{C_0}.E_{sym}[k_{S,X_0}^{s \rightarrow t}](SKey.\hat{C_0}).E_{sym}[AKey](SKey.n_2.\hat{S})) \quad (7)$$

(-1) $$[\mathbf{Client}]_C\ \exists Y.\ \mathsf{SymEnc}(Y, AKey.\hat{C_0}, k_{X_0,K_0}^{t \rightarrow k}) \quad (8)$$

Inst $Y \mapsto Y_0, (-1)$ $$[\mathbf{Client}]_C\ \mathsf{SymEnc}(Y_0, AKey.\hat{C_0}, k_{X_0,K_0}^{t \rightarrow k}) \quad (9)$$

$\mathbf{ENC3}, (-1)$ $$[\mathbf{Client}]_C\ \mathsf{Has}(Y_0, AKey) \quad (10)$$

$SEC_{AKey}^{client}, (-1)$ $$\mathsf{Honest}(\hat{Y_0}) \quad (11)$$

$\mathbf{HON}$ $$\mathsf{Honest}(\hat{X}) \wedge \mathsf{SymEnc}(Y, Key.\hat{W}, k_{X,Z}^{t \rightarrow k}) \supset \mathsf{New}(X, Key) \quad (12)$$

$(-4, -1)$ $$[\mathbf{Client}]_C\ \mathsf{New}(Y_0, AKey) \quad (13)$$

$AUTH_{kas}^{client}$ $$\mathsf{New}(X, AKey) \wedge \mathsf{SymEnc}(X, AKey.\hat{W}, k_{Y,Z}^{t \rightarrow k}) \supset \hat{Y} = \hat{T} \wedge \hat{Z} = \hat{K} \wedge \hat{W} = \hat{C} \quad (14)$$

$(9, -2, -1)$ $$\hat{X_0} = \hat{T} \wedge \hat{K_0} = \hat{K} \wedge \hat{C_0} = \hat{C} \quad (15)$$

$(7, -1)$ $$[\mathbf{Client}]_C\ \exists \eta.\ \mathsf{Send}((\hat{T}, \eta), \hat{C}.E_{sym}[k_{S,T}^{s \rightarrow t}](SKey.\hat{C}).E_{sym}[AKey](SKey.n_2.\hat{S})) \quad (16)$$

(-1) $$AUTH_{tgs}^{client} \quad (17)$$

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

The Engineering Challenges of Trustworthy Computing

Greg MORRISETT
School of Engineering and Applied Sciences, Harvard University

Abstract. This article provides motivation, background, and references to a handful of topics in language-based security. Specifically, the notes describe three techniques that have been proposed by researchers to address low-level errors in production code. The techniques include software-based fault isolation, type and proof systems for assembly code, and type-safety mechanisms for C code.

Introduction

The software that makes up our computing and communications infrastructure is full of bugs. Some of these bugs are benign, but many times, they can lead to a critical failure or security hole. At best, these bugs can cause a program to crash. At worst, they can allow an attacker to gain complete control of a service or machine.

The classic example of a security-relevant bug is the buffer overrun [1], where a programmer assumes that external input will always fit into an array of some particular, fixed size. A malicious user will craft an input that is larger than the array, and if the code fails to do the proper checks, then the input will overwrite the portion of memory next to the array. When the array is allocated on the control-stack, this allows the attacker to overwrite the values of local variables as well as meta-data, such as the return address of the current procedure. Thus, the attacker has a way to change where the program will "return" when the procedure completes. A very clever attacker will include a program fragment as part of the input and cause control to transfer to this newly injected code. In this fashion, the attacker can cause a server to execute arbitrary code and potentially take over the operation of a machine.

Buffer overruns are not new: Back in 1988, Robert Morris, Jr. exploited a buffer overrun in the `finger` daemon to launch the original Internet worm. Since then, literally thousands of viruses and worms have exploited buffer overruns to gain control of machines. Indeed, at one point, over 50% of the security-relevant bugs in operating systems reported to the Computer Emergency Response Team involved buffer overruns. Today, major software vendors know about this particular vulnerability and have deployed a number of tools to try to find, detect, or stop buffer overrun based attacks. Yet these attacks continue to be successful. For instance, in spite of a very concerted effort by Microsoft to stamp out buffer overruns, a few months after the Vista operating system was released, an exploitable buffer overrun was found in the code that controls animated cursors.

Why are buffer overruns such a problem? In part, this is due to the fact that most of our systems software, including operating systems, network stacks, file systems, databases, web servers and browsers, etc. are coded in low-level, error-prone languages such as C and C++. By default, these languages fail to enforce basic constraints on the integrity of the abstractions provided by the language. In particular, these langauges do not check that a given array index is in bounds and for some reason, programmers just don't seem to be good at realizing when they need to insert the checks themselves.

Of course, the lack of array-bounds-checking is not the only bug that hackers have successfully exploited in C/C++ code. Other examples include format string mismatches, failure to check for error codes upon return, memory leaks, and race conditions. Some of these problems are mitigated by using a *type-safe* language such as Java or C#. In particular, type-safe languages provide basic integrity guarantees for objects, as well as basic integrity guarantees for control-flow. These guarantees are achieved through a combination of static and dynamic checks, as well as run-time services such as garbage collection and stack-inspection.

But type-safe languages do not solve all of the reliability and security problems we face. For example, most type-safe languages check array indices at run-time and signal an error by throwing an exception. If this exception is not caught, then the program will still crash. Thus, while an attacker cannot successfully inject code into the server through an input bug, they can still cause a denial of service.

Another critical issue is that high-level, type-safe languages such as Java and C# depend upon tools such as type-checkers and just-in-time compilers, as well as run-time services such as garbage collection to enforce the type-safety guarantee. Yet those tools and services, more often than not, are coded in C. Indeed, Sun's Java SDK includes over 700,000 lines of C code, and thus it is difficult to claim that Java is really "type-safe".

Yet another key engineering issue is that it is prohibitively expensive to take existing C/C++ code and rewrite it in a safe language. A system such as Windows Vista consists of roughly 50-70 million lines of C code, most of which was inherited from Windows XP. Rewriting the code in any language is likely to introduce new bugs and break compatability with existing applications and tools.

Even where it may be cost-effective to re-implement a service, there is a problem that today's type-safe languages do not always provide the degree of control needed for a given application. Indeed, one of the goals of a high-level language is to abstract from the hardware resources in order to provide portability. Yet, in some settings, such as device drivers or embedded systems, we need direct access and control over machine resources. One of the reasons that C has remained popular for these settings is that it strikes a relatively good balance between portability and control.

Finally, type-safety alone does not guarantee that programs are safe from failures or attacks. There are many instances of bugs that occur at the level of libraries or abstractions above the level of a given language. For example, an SQL injection attack is not all that different from a buffer overrun: The programmer fails to check that input from an untrusted source respects some crucial property. Yet SQL injection attacks can happen just easily in a type-safe language as in C.

Of course, there is no way to guarantee the absence of all bugs in a program. But we can do a much better job of carefully categorizing common bugs and failures, and designing both languages and tools to catch or (better yet) prevent these problems during development. There are many challenges involved in this enterprise, from developing ap-

propriate securty policies, to engineering practical solutions that are applicable to current systems.

In what follows, we will discuss a handful of representative techniques that have emerged from the programming language and compiler communities. Some, such as Software-Based Fault Isolation (SFI), are extremely practical and can be applied to just about any existing software, yet the security guarantees provided by the mechanism are relatively weak. Other mechansisms, such as Proof-Carrying Code, can enforce arbitary security policies in principle, but depend upon a radical change in the way we develop software.

1. Software-Based Isolation

One of the key reasons that operating systems crash is due to bugs in third-party device drivers. Good security design suggests that drivers and other services should be placed in their own address space so that they are isolated from the kernel. Then a bug in a driver, such as a buffer overrun, will not be able to corrupt the state of the kernel. Perhaps the device will stop functioning, but at least the kernel can continue to make progress, and ideally take some corrective actions (e.g., re-initializing the driver).

Similarly, one of the many reasons a web server crashes is due to CGI scripts or servlets that malfunction and corrupt the state of the server. Again, good security design suggests that these services be run in a separate address space in order to isolate failures.

So why are drivers and other kernel modules run in the same domain as the kernel? Why are CGI scripts and servlets run within the same domain as a web-server? The answer is performance. If we put a driver in a separate address space, then we must cross a domain boundary (i.e., perform a context switch) each time the kernel and driver must communicate. Furthermore, data must be copied to and from the kernel, and of course, many DMA devices do not support virtualized access. For a video or even high-speed network driver, these overheads can be prohibitive. Indeed, the drivers are often carefully tuned to avoid any copies at all.

Similarly, for a web-server, the cost of forking an entire process just to perform a simple script action can cause a machine to thrash under heavy load. In both cases, the kernel and the web-server, it has become necessary to run extensions in the context of a service purely for performance reasons. Yet, those extensions, more often than not, have bugs that can lead to failures or security holes.

Thus, a central challenge for security researchers is to provide some form of domain isolation without the overheads of traditional operating system processes, including the costs of starting a new process, crossing process boundaries, and copying data to and from processes.

One way to achieve some degree of isolation is to rely upon a type-safe language, such as Java. The type-safety guarantee ought to ensure that server state is appropriately protected from buggy extensions, as long as the state is appropriately encapsulated in interfaces. However, using a high-level language like Java can come with its own overheads (e.g., pauses due to garbage collection) that may make this choice unattractive. Furthermore, Java does not provide the low-level control over data layout and hardware resources needed to write device drivers. Especially in the context of kernels and embedded systems, we need an isolation solution that can work for essentially arbitary machine code.

A naive approach is to have the server *interpret* any extension code. Then, within the interpreter, the server can check to ensure that the extension is only reading and writing addresses which it should be granted access, and jumping only to appropriate locations within the kernel. Of course, interpreting machine code adds a tremendous time overhead (roughly 10-100x) so this approach is not really suitable.

1.1. Berkeley SFI

There is an approach suggested by Wahbe et al. [17] called *software-based fault isolation* or SFI that can provide isolation for (almost) arbitrary machine code and with relatively low overhead. The basic idea is to read in the code and insert additional checks each time an address is read or written to make sure the effective address lies within the (logical) domain of the extension. In other words, we re-write the machine code so that it becomes "self-checking" with respect to the memory isolation property. This is an instance of a more general notion of an in-lined reference monitor [13] which in principle, can enforce arbitrary safety policies, not just memory isolation. Alternatively, we can think of SFI as what you get when you *partially evaluate* the machine-code interpreter that has been augmented with additional checks.

The central challenge with SFI is that it is not sufficient to insert checks upon reads and writes; we must also adjust jumps so that they take into account the inserted instructions. For jumps with statically known destinations, doing this adjustment is not difficult. For computed jumps (jumps through registers), the situation is a bit more tricky. For first-order procedural code (i.e., C code that does not use function pointers), computed jumps are used for two purposes: (1) when a procedure returns to its caller, it jumps to a supplied return address, and (2) some switch statements are compiled into a computed jump to a destination loaded from an array of addresses—a "jump-table".

On most architectures, we do not have to worry about adjusting return addresses because when the procedure is called, the return address is constructed relative to the point of the call. For switch statements, the rewriter must somehow be able to identify jump tables and adjust the locations. Similarly, for higher-order code involving function pointers, closures, or object vtables, the rewriter needs to know what are the possible entry points so that adjustments can be made. All of these issues are simplified if the rewriter works at the assembly level, instead of the machine code level, where addressing is made explicit through the use of symbolic labels.

Nevertheless, just because the compiler adjusts jump targets at compile time, there is no guarantee that a bug in the extension will not cause a jump target to become corrupted. For example, if an extension has a buffer overrun, then the return address of a function might be replaced with a new value that causes control to transfer past a check. Thus, just as we must check that all reads and writes are to appropriate locations, we must also check that all jumps are to code destinations within the logical address space of the extension. Furthermore, if we wish to stop code injection attacks, then we should not allow the code of the extension to be overwritten.

The overhead of doing a check on every read, write, and computed jump can still be considerable. If we only check writes and jumps, the overheads can be reduced considerably while still maintaining server integrity, but at the price of some confidentiality. Finally, instead of checking that each address we write or jump to is in the logical domain of the extension, we can use some simple tricks to force this property without using

conditionals. In particular, if we align the code and data segments of the extension to a power-of-two address, then we can simply use some bit-masking operations on an address to force it to lie within the appropriate segment. The advantage is that bit-masking operations are usually much cheaper than conditionals on a pipelined architecture. The disadvantage is when an ill-behaved extension attempts to write to an address outside of its domain, it will simply be forced to clobber some arbitrary address in its domain instead of halting with an error message. This style of isolation, where the a faulty program does not necessarily halt, but rather, is forced to execute something else that satisfies the security policy is known as "sandboxing".

1.2. SFI on CISC Machines

The original work on SFI was designed for a RISC architecture (the MIPS) that had a relatively large number of registers. The large number of registers made it easy to dedicate a few registers to the sandboxing task (one for writing, one for jumping, and a couple of scratch registers for computing effective addresses.) In particular, the rewriter kept an invariant that at each program point, the dedicated write and jump registers always held values that lay within the data and code segment ranges respectively. Thus, even if the code manages to jump past a bit-masking operation, it will still only be able to write a value in the data segment of the extension, and only jump to an address in the code segment of the extension. With all of these tricks in place, the overhead of the checks could be reduced to a few percent over the unchecked code.

Unfortunately, many of the tricks used in the original SFI work do not apply to CISC architectures such as the Intel x86. For one thing, the x86 only has a small number of registers, so we cannot afford to dedicate many registers to the sandboxing task. Another issue is that instructions on the x86 are of many different sizes, and it is possible to get different instruction "parses" out of a code segment depending upon the address with which you start. Any sandboxing technique must make sure that, in the presence of arbitrary jumps into the code segment, no checks or invariants will be broken.

To address these concerns, McCamant and Morrisett [8] suggested a solution wherein jumps were constrained via bit-masking to sixteen-byte aligned addresses. The rewriter would then ensure that, a sequence of instructions would never span more than sixteen bytes by inserting appropriate no-ops. This ensured that there was at most one "parse" to the instruction sequence. In addition, sixteen bytes was big enough that the sandboxing instructions for a write or jump could be packed into the same "atomic" sequence. Since jumps were forced via masking to sixteen-byte aligned addresses, this ensured that the masking operations could not be bypassed. In turn, this made it possible to avoid the use of dedicated registers for writing and jumping. The overheads for the padding and checks lead to an average overhead of about 20%, which is relatively small when compared to other mechanisms.

Another difference between the original SFI work and that of McCamant and Morrisett is that the latter included a separate, small checker that could be run to verify that the rewritten code respected the invariants necessary to ensure memory and control-flow isolation. Thus, the rewriter (or compiler) did not have to be trusted. Rather, only the invariant checker needed to be correct. Finally, McCamant modeled a subset of the x86 using the ALCL2 proof development environment, and formally (i.e., mechanically) proved that the invariants enforced by the checker were sufficient to ensure the desired security policy.

It is important to note that the SFI sandboxing policy provides only relatively weak guarantees, and like operating system processes, the abstractions may not be a good match to the application at hand. For example, SFI is not a good solution when the server makes a call into an extension and assumes that the extension will always return within a reasonable time. As another example, though a buffer overrun in an extension will not lead to a code injection attack, it can still lead to a "jump-to-libc" attack where the extension code transfers control to some existing library routine within the extension's code space.

1.3. CFI and XFI

Abadi et al. extended the ideas behind SFI to address the "jump-to-libc" concern and provide a much better security policy thatn basic SFI [4]. In particular, they formulated a notion of *control-flow integrity* and a practical and efficient technique for achieving it. In their setting, they assume that extensions come with a non-deterministic finite automata (NFA) where the states represent instructions and the edges represent potential control-flow transfers. The SFI control-flow policy is a special case where every instruction is the potential target of each computed jump. In practice, a compiler can produce a much more refined NFA that reasonably constrains the execution paths of the program. In particular, within a procedure, the NFA will correspond to the control-flow graph of the procedure.

Across procedures, the NFA can only capture the fact that control might flow into the procedure and then back out to any one of the call sites. For example, if a procedure `foo` is called by both `bar` and `baz`, then the NFA might specify that upon return from `foo`, control can only transfer to either the call site of `foo` within `bar` or within `baz`. This is enough to prevent a "jump-to-libc" attack from being successful, for a change to the return address will not cause control to transfer to an arbitrary procedure.

The CFI policy is enforced by inserting a bit pattern before each potential jump target. A unique bit pattern is chosen for the set of potential targets for a given jump. In the example above, the return sites within both `bar` and `baz` would share a bit-pattern since either one could be the target of `foo`'s return. In addition, code is inserted before each computed jump to check that the intended destination respects the NFA policy by examining the bit pattern right before the destination. Assuming that the code segment cannot be mutated, and that the bit patterns are chosen at random, then with high probability, an atacker cannot cause control to be transferred outside the set of paths represented by the NFA.

A compelling aspect of the CFI work is that the authors constructed a (paper) model for a toy assembly language and proved that the approach ensured control-flow integrity under an extremely strong attacker model. In particular, they modeled the attacker as performing arbitrary (possibly concurrent) changes to the data store in memory. Today, it seems feasible to construct an accurate model and proof for a realistic architecture (e.g., the x86) within a mechanized proof development environment such as Coq, NuPRL, Isabelle/HOL or ACL2.

In later work, Erlingsson et al. extended the basic CFI framework and integrated finer grain protection mechanisms for data access and control [7]. They are able to provide this finer policy in part because they can amortize the cost of access checks across basic blocks and larger units by leveraging the enforced control-flow graph. However, unlike SFI, this extended CFI framework (known as XFI), demands relatively close co-

operation with the compiler in the sense that it assumes a particular calling convention and treatment of the control stack. While this assumption meets most systems code written in C, there are still some routines (e.g., `longjmp`) that are not compatible with the underlying ideas. Furthermore, when the data are organized into relatively large, coarse-grained objects, XFI (and SFI) have small overheads (on the order of 0-10%). But for fine-grained objects (e.g., linked lists), the overheads become much greater (e.g., 60% or more).

In summary the idea of rewriting code to enforce a particular isolation policy is a powerful one: We can enforce both memory and control-flow isolation, and even stronger policies such as CFI's with relatively low overhead on extremely low-level code (essentially assembly language). On the other hand, the security guarantees, while important, are still relatively weak. Furthermore, these techniques do not work as well when we have fine-grained policies dictating access control to data.

2. TAL and PCC

Strongly-typed programming languages, such as ML, Scheme, and Java provide a strong form of memory isolation and control-flow integrity. For example, in Java, reads and writes to memory must be done via object or array references. Such a reference provides a limited capability for accessing memory according to the reference type. Similarly, the destination of a "computed jump" is entirely limited by the type system in the sense that we can only pass control to a procedure or method when it is "of the right type".

But as argued in the introduction, high-level languages are unsuitable for certain programming tasks where control is needed over machine resources, data layout, and control. Furthermore, the run-time systems of today's high-level languages, which includes support for services such as garbage collection, are typically written in low-level languages such as C. Thus, a central research question over the past ten years has been how to adapt the ideas behind type and proof systems for high-level languages to the setting of low-level (i.e., machine) code.

Restricted versions of lambda calculi, such as *continuation-passing style* (CPS), correspond quite closely to the low-level intermediate languages used by modern compilers (e.g., static-single assignment or SSA.) Thus, a good starting point is to study and adapt type systems for these restricted lambda calculi to the setting of machine code. This was the approach suggested by Morrisett et al. [10] where they showed how to systematically compile a simple subset of an ML-like language to MIPS assembly code in a type-preserving fashion. With this approach, they justified a type system for the MIPS assembly language (TAL) that was based upon a straightforward adaptation of typed CPS.

At about the same time that TAL was developed, Necula and Lee proposed the idea of *proof-carrying code* (PCC) [11]. They observed that in principle, one could impose an arbitrary security policy on untrusted machine code by simply requiring that the code come equipped with an explicit, machine-checkable proof that the code would respect the policy when executed. In the original work, they developed a simple packet filter program (to be executed in the context of a kernel) and constructed a proof that the resulting code respected a simple memory and control-flow isolation property. The proof was based upon an axiomatic treatment of the machine code instructions. Utilizing the

Curry-Howard isomorphism, the proof was represented as a dependently-typed LF term, and thus proof checking could be reduced to type-checking.

Of course, to make PCC scalable, we need some largely automated way to construct proofs that code respects a given policy. If we limit the policy to some form of type-safety, in the style of TAL, then the proofs can be built automatically via a type-preserving compiler. This was the approach that Necula and Lee (among others) took in their Special-J compiler [5].

In many respects, the logic-based formulation of PCC was much better than the original formulation of TAL. In particular, certain conditions in TAL, such as checking that an array index was in bounds, weren't easily captured by a conventional type system. Later versions of TAL included limited support for dependent types [16,6] to address this problem, but nevertheless, the approach is less general than PCC since many provably "safe" instruction sequences cannot be validated by the type system.

On the other hand, there are subtleties in providing a *modular* axiomatic semantics and proof system for machine code, in large part because of the issue of computed jumps. Indeed, a modular and relatively complete treatment of axiomatic semantics in the presence of higher-order functions and state is a central research topic these days. Furthermore, there are a number of interesting semantic issues when one attempts to give a logical interpretation of conventional types at the machine level. In particular, the technical details involving state and first-class code pointers are amazingly tricky to get right. Many of these issues were addressed by Shao in his work on verified assembly language [14] as well as Appel's work on so-called "foundational" PCC [2].

A key open question for researchers is how to really tap the potential of proof-carrying code. If we move beyond simple isolation or type-safety policies, how do we get proofs? Presumably, given a high-level language with a suitable program logic or advanced type system that can capture relevant aspects of policies, a "proof-preserving" compiler can take over and yield a proof that the resulting machine code is respects the given policy.

3. Towards Safer C/C++ Code

The C and C++ programming languages are absolutely horrible when it comes to security. But, as argued in the introduction, they are nonetheless the languages of choice when it comes to building systems software. And porting existing applications and services from C/C++ to a new language is often prohibitively expensive.

Recently, software companies have started using static analysis tools to try to find and detect security-relevant bugs in C and C++ code. For example, Microsoft uses a tool called Prefast to search for common coding problems including buffer overruns, failure to check return codes, etc. Because Microsoft has been hurt so badly from security-related errors, developers are now required to run Prefast on code before it can be checked in to a repository. Companies such as Coverity and Fortify sell tools similar to Prefast that also look for common bugs in C and C++ code. These tools are used by third parties such as banks, investment houses, and the military where security issues are a prime concern.

A key advantage of all of these tools is that they work on existing C/C++ code without modification, and unlike conventional testing, can cover all of the paths in the code. However, none of these tools offers a guarantee that they will enforce some isolation pol-

icy such as memory-safety much less type-safety. Indeed, the cursor animation overflow bug in Vista passed through Prefast without warning.

They key reason why these tools are unsound is due to the lack of structure and enforced abstraction in C code. Consequently, any sound static analysis for arbitary C code must be extremely precise and essentially reason at the level of the machine in order to avoid generating false positives (i.e., signalling a warning when there is no problem.) False positives are a big concern for these tools because developers are unwilling to wade through thousands of potential bug reports just to find the one or two that can actually be exploited by a hacker. And of course, to achieve high precision, one needs sophisticated, whole-program analysis in order to account for the contexts in which a given procedure might run. For instance, a procedure such as `memcpy` that copies the contents of one buffer to another needs to know the sizes of the buffers relative to one another in order to determine if there is a potential overflow. In turn, this demands reasoning about `memcpy` in a context-sensitive fashion (i.e., at each call site.)

Unfortunately, it is difficult to scale precise, whole-program, context and path-sensitive analyses to the multi-million line applications that Microsoft develops. Thus, their Prefast tool uses a modular dataflow analysis that works in a context-insensitive and largely path-insensitive fashion. To be sound, such an analysis would signal far too many false positives to be effective. Consequently, t the analysis makes optimistic assumptions regarding inputs to procedures which may in fact turn out to be invalid. To help mitigate this problem, Microsoft has forced developers to annotate procedure interfaces with a limited form of pre- and post-conditions that capture some of the assumptions.

In my opinion, these bug-finding tools are extremely effective in spite of the fact that they are unsound. They strike an engineering compromise between *usability* and soundness. Nevertheless, one can expect that hackers will simply adjust their tactics to look for code that violates the optimistic assumptions made by the tools when searching for bugs.

3.1. Other Tricks for Partially Securing C

Microsoft utilizes a few other tricks to try to minimize the potential damage that an attacker can do when a bug slips through. This kind of "defense in depth" is another key security principle that can help to minimize the damage an attacker might otherwise achieve. For example, they have modified the compiler so that it inserts a random value, called a "cookie", between all stack-allocated character buffers and the return address. Before returning from a procedure, the code checks to make sure the cookie has not been overwritten. Thus, a buffer overrun on a stack-allocated character buffer will be trapped, assuming the attacker cannot guess the value of the cookie. This approach is quite similar to StackGuard [3] which has been available for some time.

Other tricks that Microsoft uses to mitigate attacks include address space randomization (ASR) and on newer x86 implementations that support it, no-execute stack segments. The no-execute stack segment, which is enforced by the virtual memory management unit (MMU), ensures that control cannot be transferred to an address within the segment of memory that holds the control stack. Thus, a buffer overrun that attempts to inject code by overwriting a stack buffer will be trapped by the MMU. ASR helps to stop jump-to-libc attacks by modifying the loader to place libraries in random locations in memory. Thus, an attacker must guess where to jump to when they overflow a buffer.

While all of these mechanisms help to stop potential attacks, they are nonetheless incomplete, even for buffer overruns. For example, if input overruns a heap-allocated buffer, then an attacker can still launch a code-injection attack[1]. Because only newer machines support hardware-based no-execute stack segments, and because Microsoft only inserts cookies for procedures that manipulate character buffers, even simple stack-based attacks are still easily possible. And finally, because they only use a small number of different locations for loading libraries, a worm with a jump-to-libc attack will succeed a significant number of times.

Lest you think that I am picking on Microsoft, I note that their practices are actually well beyond those used by most of the industry. In short, industry still only has stop-gap measures that plug some of the holes, and you can bet that attackers will quickly shift to exploit those holes that remain.

3.2. Going All the Way: Ccured and Cyclone

At least two research projects, Ccured [12] and Cyclone [9], have tried to provide a strong and sound isolation guarantee for C code. To ensure soundness, both systems rely upon inserted checks and meta-data. In particular, both systems have the notion of a "fat pointer" which, unlike a normal C pointer, includes extra information that makes it possible to check whether an offset, relative to that pointer, lies within the boundary of an object.

The emphasis in the Ccured system is on making it easy to port legacy C code to the new system. In particular, Ccured requires minimal changes and annotation from the programmer. It performs a whole-program, type-based analysis to determine whether or not given operations will be safe. If so, then checks can be omitted. Otherwise, not only must the compiler insert a check (e.g., for an array bounds), but it must also constrain the pointer used to contain appropriate meta-data so that these properties can be checked at run-time. In practice, the compiler is able to eliminate almost all run-time checks and results in code that has relatively low overheads, yet offers a much stronger memory isolation guarantee and control-flow isolation guarantee than SFI or even XFI.

Ccured does require a few changes to the C code. For instance, since the sizes of objects change, explicit allocations via `malloc` must be adjusted. As another example, to avoid an unsoundness with `union` values, the programmer must re-work the code to avoid using the `union` (e.g., by using a struct instead.)

Because the meta-data inserted by the compiler renders changes to data representations, care must be taken when crossing from "cured" code yinto native C code. In particular, the meta-data must be stripped off on input and somehow added back upon return, or else the data must be marshaled across the interface, not unlike foreign function interfaces for other high-level languages. The addition of meta-data can also make it difficult to interface to memory-mapped devices, making the approach somewhat difficult to use in the context of a driver.

Finally, to preserve type-safety, Ccured relies upon a conservative garbage collector to reclaim all heap-allocated memory objects. The issue is that if a programmer manually deallocates an object, then the type system cannot ensure that other references to the object will not be used in the future. Thus, the data cannot truly be recycled to hold an ob-

[1] To be fair, Microsoft also employs some infrequent integrity checks on the heap to try to avoid this problem, but the checks are only run infrequently.

ject of an alternative type. Of course, the need for a garbage collector also makes Ccured difficult to use in contexts such as a real-time kernel due to the potential for pauses. The garbage collector also imposes a relatively healthy space overhead. And finally, certain run-time services, such as garbage collection itself, cannot be easily written in Ccured.

Cyclone takes a different approach to achieving soundness than Ccured. In particular, programmers must make extensive changes to the code by adding additional typing annotations that refine the types of objects. For example, Cyclone makes a distinction between fat pointers and normal pointers. Only fat pointers can be used with arbitrary pointer arithmetic, as only fat pointers provide the necessary meta-data to perform the needed run-time checks. Thus, whereas Ccured infers where fat and thin pointers are needed, Cyclone requires that programmers make the choice explicit.

The disadvantage of the Cyclone approach is clear: it is not longer easy to apply the compiler to existing C code, but rather, programmers must make extensive changes. The advantage of explicit annotations is that type-checking becomes modular (i.e., does not require the whole program) and programmers are made aware of data representations. For instance, they can be assured that thin pointers really will match the native pointer representation, which makes interfacing with legacy code and devices a bit simpler.

Cyclone also includes support for some manual memory management: it incorporates a region-based, type-and-effect system, derived from the work of Tofte and Talpin [15], that allows programmers to allocate and deallocate collections of objects. The type system tracks which collections are live at each program point, and uses the effect system in conjunction with region and effect polymorphism to keep type-checking modular.

The original region type system of Tofte and Talpin only supported lexically-scoped regions. Thus computations that were "tail-recursive" in a region (such as CPS code or event-processing code) could only deallocate regions at the end of the program, leading to massive space leaks. Cyclone worked around this problem by incorporating a more general form of regions protected by linear capabilities. The resulting system was strong enough that one could code a copying garbage collector safely in the language without needing to rely upon a meta-level collector.

In practice, the extra annotations needed on Cyclone code and the region-based type-and-effect system resulted in a complicated language, where porting existing C code of any substantial size was not very practical. In contrast, Ccured could be used to port relatively large applicaitons. On the other hand, for new system code or code intended for real-time settings or where control over resources was critical, Cyclone is a relatively good fit.

References

[1] Aleph One. Smashing the Stack for Fun and Profit. Phrack 7(49). Available at http://www.phrack.org/archives/49/P49-14.

[2] Andrew W. Appel and Amy P. Felty. A Semantic Model of Types and Machine Instructions for Proof-Carrying Code. In 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 243-253, January 2000.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63–78, San Antonio, TX, January 1998.

[4] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05), Alexandria, VA, November 2005.

[5] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In Proc. of the Conference on Programming Language Design and Implementation, pages 95–107, May 2000.

[6] Karl Crary. Toward a Foundational Typed Assembly Language. In ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 2003.

[7] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In Proceedings of the 7th Usenix Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, Washington, November 2006.

[8] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In Proceedings of the 15th Usenix Security Symposium, Vancouver, British Columbia, August 2006.

[9] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. USENIX Annual Technical Conference, Monterey, 2002.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. Conference Record of POPL âĂŹ98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, pp. 85âĂŞ97, 1998.

[11] G. Necula and P. Lee. Safe Untrusted Agents Using Proof-Carrying Code. LNCS 1419, p. 61, 1988

[12] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrotting of Legacy Code. Twenty-Ninth ACM Symposium on Principles of Programming Languages, 2002

[13] Fred B. Schendier. Enforceable Security Policies. In ACM Transactions on Information Systems Security, 3(1), p. 30–50, 2000.

[14] Zhaozhong Ni and Zhong Shao. Certified Assembly Programming with Embedded Code Pointers. In ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 2006.

[15] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. Information and Computation, 132(2):109–176, February 1997.

[16] Hongwei Xi and Frank Pfenning, Dependent Types in Practical Programming. In Proceedings of Symposium on Principles of Programming Languages(POPL '99), San Antonio, Texas, January 1999

[17] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203–216, December 1993.

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Reflecting Quantifier Elimination for Linear Arithmetic

Tobias NIPKOW
Institut für Informatik, TU München

Abstract. This paper formalizes and verifies quantifier elimination procedures for dense linear orders and for real and integer linear arithmetic in the theorem prover Isabelle/HOL. It is a reflective formalization because it can be applied to HOL formulae themselves. In particular we obtain verified executable decision procedures for linear arithmetic. The formalization for the various theories is modularized with the help of *locales*, a structuring facility in Isabelle.

1. Introduction

This research is about adding decision procedures to theorem provers in a reliable manner, i.e. without having to trust the decision procedure. The traditional LCF approach [16] involves programming the decision procedure in the implementation language of the theorem prover using the basic inference rules of the logic. This is safe but tricky to write and maintain. There are two alternatives: checking externally generated certificates (for an example see [22]), and *reflection*, i.e. the formalization and verification of the decision procedure in the logic itself. The focus of this paper is reflection, partly because the theories we consider do not lend themselves to certificate checking: there are no short certificates, i.e. checking the certificates is as expensive as generating them in the first place.

The mathematical subject matter of the paper is quantifier elimination, i.e. the process of computing a quantifier-free equivalent of a quantified formula, yielding in particular a decision procedure for closed formulae. Many numeric theories enjoy quantifier elimination. The most celebrated instance is quantifier elimination for real closed fields, i.e. $(\mathbb{R}, +, *)$, due to Tarski [31]. We reflect quantifier elimination procedures for dense linear orders, and linear real and integer arithmetic.

Everything has been formalized and verified in the logic HOL (higher-order logic) of the theorem prover Isabelle [27] and is available online in the Archive of Formal Proofs at afp.sf.net. In particular we have made use of *locales*, a structuring facility akin to parameterized theories. Locales are a fairly recent addition to Isabelle [1,2] and this article demonstrates locales in a serious application.

In summary, the article makes the following contributions:

1. A detailed exposition of a formalization of quantifier elimination for linear real and integer arithmetics in HOL.
2. Reflective implementations of quantifier elimination.

3. A modular development based on locales.

Note that our presentation aims for simplicity and minimality of concepts, not for practical efficiency. For example, we restrict to as few atomic propositions as possible, typically by having only one of $\leq$ and $<$. In practice one would avoid this as it tends to lead to inefficiencies due to coding, e.g. if $s = t$ is replaced by $s \leq t \wedge t \leq s$. Nevertheless our presentation provides a convenient starting point for more efficient implementations, as demonstrated elsewhere [8], where the reflective implementation is two orders of magnitude faster than the LCF approach.

The core of the article is structured as follows: We start with an abstract generic account of logical formulae (§4); quantifier elimination is given as a locale parametric in the specific logical theory of interest. This locale is instantiated four times: for dense linear orders (§5), for linear real arithmetic (Fourier-Motzkin elimination in §6.2 and Ferrante and Rackoff's procedure in §6.4), and for linear integer arithmetic (§7).

2. Reflection, Informally

Reflection means to perform a proof step by computation inside the logic rather than inside some external programming language (ML). Inside the logic it is not possible to write functions by pattern matching over the syntax of formulae because two syntactically distinct formulae may be logically equivalent. Hence the relevant fragment of formulae must be represented (*reflected*) inside the logic as a datatype. We call it *rep*, the representation.

The two levels of formulae must be connected by two functions:

I (a HOL function) maps an element of *rep* to the formula it represents, and
$reify$ (an ML function) maps a formula to its representation.

The two functions should be inverses of each other. Informally $I(\mathit{reify}(P)) = P$ should hold. More precisely, taking the ML representation of a formula P and applying *reify* to it yields an ML representation of a term p of type *rep* such that $I(p) = P$ holds.

Typically, the formalized proof step is some equivalence $P \leftrightarrow P'$ where P is given and P' is some simplified version of P (e.g. the elimination of quantifiers). This transformation is now expressed as a recursive function *simp* of type *rep* $\rightarrow$ *rep*. We prove (typically by induction on *rep*) that *simp* preserves the interpretation: $I(\mathit{simp}(p)) \leftrightarrow I(p)$. To apply this theorem to a given formula P we proceed as follows:

1. Create a *rep*-term p from P using *reify*. This *reification* step must be performed in ML.
2. Prove $P \leftrightarrow I(p)$. Usually this is trivial by rewriting with the definition of I.
3. Instantiate *simp*'s correctness theorem $I(\mathit{simp}(p)) \leftrightarrow I(p)$, compute the result p' of evaluating $\mathit{simp}(p)$ and obtain the theorem $I(p') \leftrightarrow I(p)$ (and by symmetry $I(p) \leftrightarrow I(p')$). This is the *evaluation* step.
4. Simplify $I(p')$, again by rewriting with the definition of I, yielding a theorem $I(p') \leftrightarrow P'$

The final theorem $P \leftrightarrow P'$ holds by transitivity.

The evaluation step is crucial for efficiency as all other steps are typically linear-time. We employ Isabelle's recent code generator [18] for compiling and evaluating

simp(p) in ML. Other approaches include evaluation via LISP (Boyer and Moore's "metafunctions" [5], the mother of all reflections) and the use of an internal λ-calculus evaluator [17] as in Coq.

There is also the practical issue of where *reify* comes from. In general, the implementor of the reflected proof procedure must program it in ML and link it into the above chain of deductions. But because *reify* must be the inverse of I, it is often possible to automate this step. Isabelle implements a sufficiently general inversion scheme for I such that for all of the examples in this paper, *reify* is performed automatically.

In principle the reader may now forget about the details of reflection and merely keep in mind that all the algorithms in this paper, although expressed on some representation of formulae, carry over to HOL formulae automatically.

3. Basic Notation

HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The basic types of truth values, natural numbers, integers and reals are called *bool*, *nat*, *int* and *real*. The space of total functions is denoted by $\Rightarrow$. Type variables are denoted by α, β, etc. The notation $t{::}\tau$ means that term t has type τ.

Sets over type α, type α *set*, follow the usual mathematical convention.

Lists over type α, type α *list*, come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in *s* usually stand for lists. In addition to the standard functions *map* and *filter*, Isabelle/HOL also supports Haskell-style list comprehension notation, with minor differences: instead of `[e | x <- xs, ...]` we write $[e.\ x \leftarrow xs, \ldots]$, and $[x{\leftarrow}xs. \ldots]$ is short for $[x.\ x{\leftarrow}xs, \ldots]$.

Finally note that $=$ on type *bool* means "iff".

Although all our algorithms and formal theorems conform to HOL syntax, we frequently switch to everyday mathematical notation during informal explanations.

4. Logic

The data type of formulae is defined in the usual manner:

$$\begin{array}{rcl} \alpha\ fm & = & \mathit{TrueF} \mid \mathit{FalseF} \mid \mathit{Atom}\ \alpha \\ & \mid & \mathit{And}\ (\alpha\ fm)\ (\alpha\ fm) \\ & \mid & \mathit{Or}\ (\alpha\ fm)\ (\alpha\ fm) \\ & \mid & \mathit{Neg}\ (\alpha\ fm) \\ & \mid & \mathit{ExQ}\ (\alpha\ fm) \end{array}$$

This representation provides the customary logical operators but leaves the type of atoms open by making it a parameter α. Variables are represented by de Bruijn indices: quantifiers do not explicitly mention the name of the variable being bound because that is implicit. For example, *ExQ* (*ExQ* ... *0* ... *1* ...) represents a formula $\exists x_1.\exists x_0.\ \ldots x_0 \ldots x_1 \ldots$. Note that the only place where variables can appear is inside

atoms. The only distinction between free and bound variables is that the index of a free variable is larger than the number of enclosing binders.

Further logical operators can be introduced as abbreviations, in particular $AllQ\ \varphi \equiv Neg\ (ExQ\ (Neg\ \varphi))$.

4.1. Auxiliary Functions

The set of atoms is computed by the (easy to define) function

$$atoms :: \alpha\ fm \Rightarrow \alpha\ set.$$

Conjunctions and disjunctions of lists of formulae are created by the functions

$$\begin{array}{ll} list\text{-}conj & :: \alpha\ fm\ list \Rightarrow \alpha\ fm \\ list\text{-}disj & :: \alpha\ fm\ list \Rightarrow \alpha\ fm \end{array}$$

Their definition is straightforward:

$list\text{-}conj\ [\varphi_1,\ldots,\varphi_n] = and\ \varphi_1\ (and\ \ldots\ \varphi_n)$

where *and* is an intelligent version of *And*:

$$\begin{array}{ll} and\ FalseF\ \varphi & = FalseF \\ and\ \varphi\ FalseF & = FalseF \\ and\ TrueF\ \varphi & = \varphi \\ and\ \varphi\ TrueF & = \varphi \\ and\ \varphi_1\ \varphi_2 & = And\ \varphi_1\ \varphi_2 \end{array}$$

Similar for *list-disj* and *or*, an optimized version of *Or*. For convenience the following abbreviation is introduced:

$Disj\ us\ f \equiv list\text{-}disj\ (map\ f\ us)$

More interesting is the conversion to DNF:

$$dnf :: \alpha\ fm \Rightarrow \alpha\ list\ list$$

$$\begin{array}{l} dnf\ TrueF = [[]] \\ dnf\ FalseF = [] \\ dnf\ (Atom\ \varphi) = [[\varphi]] \\ dnf\ (Or\ \varphi_1\ \varphi_2) = dnf\ \varphi_1\ @\ dnf\ \varphi_2 \\ dnf\ (And\ \varphi_1\ \varphi_2) = [d_1\ @\ d_2.\ d_1 \leftarrow dnf\ \varphi_1, d_2 \leftarrow dnf\ \varphi_2] \end{array}$$

The resulting list of lists represents the disjunction of conjunctions of atoms. Working with lists rather than type *fm* has the advantage of a well-developed library and notation.

Note that *dnf* assumes that its argument contains neither quantifiers nor negations. Most of our work will be concerned with quantifier-free formulae where all negations have not just been pushed right in front of atoms but actually into them. This is easy for linear orders because $\neg(x < y)$ is equivalent with $y \leq x$. This conversion will be described later on because it depends on the type of atoms. The (easy to define) predicates

$$\begin{array}{ll} qfree & :: \alpha\ fm \Rightarrow bool \\ nqfree & :: \alpha\ fm \Rightarrow bool \end{array}$$

check whether their argument is free of quantifiers (*qfree*), and free of negations and quantifiers (*nqfree*).

There is also a mapping functional

$$map_{fm} :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ fm \Rightarrow \beta\ fm$$

which recurses down a formula, e.g.

$map_{fm}\ h\ (And\ \varphi_1\ \varphi_2) = And\ (map_{fm}\ h\ \varphi_1)\ (map_{fm}\ h\ \varphi_2)$

until it finds an atom: $map_{fm}\ h\ (Atom\ a) = Atom\ (h\ a)$.

4.2. Interpretation

The interpretation or semantics of a *fm* is defined via the obvious homomorphic mapping to an HOL formula: *And* becomes $\wedge$, *Or* becomes $\vee$, etc. The interpretation of atoms is a parameter of this mapping. Atoms may refer to variables and are thus interpreted w.r.t. a valuation. Since variables are represented as natural numbers, the valuation is naturally represented as a list: variable i refers to the ith entry in the list (starting with 0). This leads to the following interpretation function:

$interpret :: (\alpha \Rightarrow \beta\ list \Rightarrow bool) \Rightarrow \alpha\ fm \Rightarrow \beta\ list \Rightarrow bool$

$interpret\ h\ TrueF\ xs = True$
$interpret\ h\ FalseF\ xs = False$
$interpret\ h\ (Atom\ a)\ xs = h\ a\ xs$
$interpret\ h\ (And\ \varphi_1\ \varphi_2)\ xs = (interpret\ h\ \varphi_1\ xs \wedge interpret\ h\ \varphi_2\ xs)$
$interpret\ h\ (Or\ \varphi_1\ \varphi_2)\ xs = (interpret\ h\ \varphi_1\ xs \vee interpret\ h\ \varphi_2\ xs)$
$interpret\ h\ (Neg\ \varphi)\ xs = (\neg\ interpret\ h\ \varphi\ xs)$
$interpret\ h\ (ExQ\ \varphi)\ xs = (\exists x.\ interpret\ h\ \varphi\ (x{\cdot}xs))$

In the equation for *ExQ* the value of the bound variable x is added at the front of the valuation. De Bruijn indexing ensures that in the body 0 refers to x and $i + 1$ refers to bound variable i further up.

4.3. Atoms

Atoms are more than a type parameter α. They come with an *interpretation* (their semantics), and a few other specific functions. These functions are also parameters of the generic part of quantifier elimination. Thus the further development will be like a module parameterized with the type of atoms and some functions on atoms. These parameters will be instantiated later on when applying the framework to various linear arithmetics.

In Isabelle this parameterization is achieved by means of a **locale** [1], a named context of types, functions and assumptions about them. We call this context *ATOM*. It provides the following functions

I_a	$:: \alpha \Rightarrow \beta\ list \Rightarrow bool$
$aneg$	$:: \alpha \Rightarrow \alpha\ fm$
$depends_0$	$:: \alpha \Rightarrow bool$
$decr$	$:: \alpha \Rightarrow \alpha$

with the following intended meaning:

I_a *a xs* is the interpretation of atom *a* w.r.t. valuation *xs*, where variable *i* (*i* :: *nat*!) is assigned the *i*th element of *xs*.

aneg negates an atom. It returns a formula which should be free of negations. This is strictly for convenience: it means we can eliminate all negations from a formula. In the worst case we would have to introduce negated versions of all atoms, but in the case of linear orders this is not necessary because we can turn, for example, $\neg(x < y)$ into $(y < x) \vee (y = x)$.

$depends_0$ *a* checks if atom *a* contains (depends on) variable 0, and *decr a* decrements every variable in *a* by 1.

Within context *ATOM* we introduce the abbreviation $I \equiv interpret\ I_a$. The assumptions on the parameters of *ATOM* can now be stated quite succinctly:

$$I\ (aneg\ a)\ xs = (\neg\ I_a\ a\ xs) \qquad nqfree\ (aneg\ a)$$
$$\neg\ depends_0\ a \Longrightarrow I_a\ a\ (x{\cdot}xs) = I_a\ (decr\ a)\ xs$$

Function *aneg* must return a quantifier and negation-free formula whose interpretation is the negation of the input. And when interpreting an atom not containing variable 0 we can drop the head of the valuation and decrement the variables without changing the interpretation.

These assumptions must be discharged when the locale is instantiated. We do not show this in the text because the proofs are straightforward in all cases.

The *negation normal form* (NNF) of a formula is defined in the customary manner by pushing negations inwards. We show only a few representative equations:

$$nnf :: \alpha\ fm \Rightarrow \alpha\ fm$$

$$nnf\ (Neg\ (Atom\ a)) = aneg\ a$$
$$nnf\ (Or\ \varphi_1\ \varphi_2) = Or\ (nnf\ \varphi_1)\ (nnf\ \varphi_2)$$
$$nnf\ (Neg\ (Or\ \varphi_1\ \varphi_2)) = And\ (nnf\ (Neg\ \varphi_1))\ (nnf\ (Neg\ \varphi_2))$$
$$nnf\ (Neg\ (And\ \varphi_1\ \varphi_2)) = Or\ (nnf\ (Neg\ \varphi_1))\ (nnf\ (Neg\ \varphi_2))$$

The first equation differs from the usual definition and gets rid of negations altogether — see the explanation of *aneg* above.

The fact that *nnf* preserves interpretations is a trivial inductive consequence of the assumptions about the locale parameters: $I\ (nnf\ \varphi)\ xs = I\ \varphi\ xs$.

4.4. Quantifier Elimination

The elimination of all quantifiers from a formula is achieved by eliminating them one by one in a bottom-up fashion. Thus each step needs to deal merely with the elimination of a single quantifier in front of a quantifier-free formula. This step is theory-dependent and hard. The lifting to arbitrary formulae is simple and can be defined once and for all. We assume we are given a function $qe :: \alpha\ fm \Rightarrow \alpha\ fm$ for the elimination of a single *ExQ*, i.e. $I\ (qe\ \varphi) = I\ (ExQ\ \varphi)$ if *qfree* φ. Note that *qe* is not applied to $ExQ\ \varphi$ but just to φ, *ExQ* remains implicit. Lifting *qe* is straightforward:

$$\textit{lift-nnf-qe} :: (\alpha\ \textit{fm} \Rightarrow \alpha\ \textit{fm}) \Rightarrow \alpha\ \textit{fm} \Rightarrow \alpha\ \textit{fm}$$

$$\begin{aligned}
&\textit{lift-nnf-qe qe}\ (\textit{And}\ \varphi_1\ \varphi_2) = \textit{and}\ (\textit{lift-nnf-qe qe}\ \varphi_1)\ (\textit{lift-nnf-qe qe}\ \varphi_2)\\
&\textit{lift-nnf-qe qe}\ (\textit{Or}\ \varphi_1\ \varphi_2) = \textit{or}\ (\textit{lift-nnf-qe qe}\ \varphi_1)\ (\textit{lift-nnf-qe qe}\ \varphi_2)\\
&\textit{lift-nnf-qe qe}\ (\textit{Neg}\ \varphi) = \textit{neg}\ (\textit{lift-nnf-qe qe}\ \varphi)\\
&\textit{lift-nnf-qe qe}\ (\textit{ExQ}\ \varphi) = \textit{qe}\ (\textit{nnf}\ (\textit{lift-nnf-qe qe}\ \varphi))\\
&\textit{lift-nnf-qe qe}\ \varphi = \varphi
\end{aligned}$$

To simplify life for *qe* we put its argument into NNF.

We can go even further and put the argument of *qe* into DNF because then we can pull the disjunction out of the existential quantifier as follows (using customary logical notation):

$$(\exists x. \bigvee_i \bigwedge_j a_{ij}) \;=\; (\bigvee_i \exists x. \bigwedge_j a_{ij})$$

where a_{ij} are the atoms of the DNF. Thus *qe* can be applied directly to a conjunction of atoms. Using

$$(\exists x. A \wedge B(x)) \;=\; (A \wedge (\exists x.\ B(x)))$$

where A does not depend on x, we can push the quantifier right in front of a conjunction of atoms all of which depend on x. This simplifies matters for *qe* as much as possible.

Now we look at the formalization of this second lifting procedure:

$$\textit{lift-dnf-qe} :: (\alpha\ \textit{list} \Rightarrow \alpha\ \textit{fm}) \Rightarrow \alpha\ \textit{fm} \Rightarrow \alpha\ \textit{fm}$$

Because we represent the DNF via lists of lists of atoms, the first argument of *lift-dnf-qe* takes a list rather than a conjunction of atoms.

The separation of a list (conjunction) of atoms into those that do contain 0 and those that do not, and the application of *qe* to the former is performed by an auxiliary function:

$$\begin{aligned}
&\textit{qelim qe as} = (\textsf{let}\ \textit{qf} = \textit{qe}\ [a \leftarrow \textit{as}.\ \textit{depends}_0\ a];\\
&\qquad\qquad \textit{indep} = [\textit{Atom}(\textit{decr}\ a).\ a \leftarrow \textit{as}, \neg\ \textit{depends}_0\ a]\\
&\qquad\ \textsf{in}\ \textit{and qf}\ (\textit{list-conj indep}))
\end{aligned}$$

Because the innermost quantifier is eliminated, all references to other quantifiers need to be decremented. For the atoms independent of the innermost quantifier this needs to be done explicitly, for the other atoms this must happen inside *qe*.

The main function *lift-dnf-qe* recurses down the formula (we omit the obvious equations) until it finds an *ExQ* φ, removes the quantifiers from φ, puts the result into NNF and DNF, and applies *qelim qe* to each disjunct:

$$\textit{lift-dnf-qe qe}\ (\textit{ExQ}\ \varphi) = \textit{Disj}\ (\textit{dnf}\ (\textit{nnf}\ (\textit{lift-dnf-qe qe}\ \varphi)))\ (\textit{qelim qe})$$

4.4.1. Correctness

Correctness of these lifting functions is roughly expressed as follows: if *qe* eliminates one existential while preserving the interpretation, then *lift qe* eliminates all quantifiers while preserving the interpretation.

For compactness we employ a set theoretic language for expressing properties of functions: $A \to B$ is the set of functions from A to B, *lists* A the set of lists over A, $-A$ the complement of A, and $|P| \equiv \{x \mid P\,x\}$.

First we look at *lift-nnf-qe*. Elimination of all quantifiers is easy:

Lemma 1 *If* $qe \in |nqfree| \to |qfree|$ *then* $qfree\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)$.

Preservation of the interpretation is slightly more involved:

Lemma 2 *If* $qe \in |nqfree| \to |qfree|$ *and* $nqfree\ \varphi \Longrightarrow I\ (qe\ \varphi)\ xs = (\exists x.\ I\ \varphi\ (x{\cdot}xs))$ *for all* φ *and* xs, *then* $I\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$.

For *lift-dnf-qe* the statements are a bit more involved still, but essentially analogous to those for *lift-nnf-qe*. The only difference is that *qe* applies to lists of atoms *as* instead of a formula φ.

Lemma 3 *If* $qe \in lists\ |depends_0| \to |qfree|$ *then* $qfree\ (lift\text{-}dnf\text{-}qe\ qe\ \varphi)$.

Lemma 4 *If* $qe \in lists\ |depends_0| \to |qfree|$ *and* $\forall as \in lists\ |depends_0|$. *is-dnf-qe qe as, then* $I\ (lift\text{-}dnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$.

where $is\text{-}dnf\text{-}qe\ qe\ as \equiv \forall xs.\ I\ (qe\ as)\ xs = (\exists x.\ \forall a{\in}set\ as.\ I_a\ a\ (x{\cdot}xs))$. The right-hand side is equal to $\exists x.\ I\ (list\text{-}conj\ (map\ Atom\ as))\ (x{\cdot}xs)$.

All proofs are straightforward inductions using a number of additional lemmas.

4.4.2. Complexity

Conversion to DNF may (unavoidably) cause exponential blowup. Since this can happen every time a quantifier is eliminated, even if *qe* runs in linear time, the worst case running time of *lift-dnf-qe qe* is non-elementary in the size of the formula, i.e. a tower of exponents $2^{\cdot^{\cdot^{2}}}$ whose height is the size of the formula. In contrast, conversion to NNF is linear. This leads to more reasonable upper bounds. For example, if *qe* takes quadratic time, the worst case running time of *lift-nnf-qe qe* is only doubly exponential. Thus we have the choice between an essentially infeasible lifting function *lift-dnf-qe* which allows each quantifier elimination step to focus on conjunctions of atoms, or a potentially feasible lifting function *lift-nnf-qe* which requires each quantifier elimination step to deal with arbitrary combinations of conjunctions and disjunctions.

4.4.3. Equality

We can generalize quantifier elimination via DNF even further based on the predicate calculus law

$$(\exists x.\ x = t \wedge \phi) = \phi[t/x] \tag{1}$$

provided x does not occur in t. In two of our theories this will enable us to remove equalities completely: in linear real arithmetic, any equation containing variable x is either independent of the value of x (e.g. $x = x$ or $x = x + 1$) or can be brought into the form $x = t$ with x not in t. But even if one cannot remove all equalities, as in most nonlinear theories, it is useful to deal with $x = t$ separately for obvious efficiency reasons. Hence we extend locale *ATOM* to locale *ATOM-EQ* containing the following additional parameters

$$\begin{array}{ll} \mathit{solvable}_0 & :: \alpha \Rightarrow \mathit{bool} \\ \mathit{trivial} & :: \alpha \Rightarrow \mathit{bool} \\ \mathit{subst}_0 & :: \alpha \Rightarrow \alpha \Rightarrow \alpha \end{array}$$

with the following intended meaning expressed by the corresponding assumptions:

- For solvable atoms, any valuation of the variables > 0 can be extended to a satisfying valuation: $\mathit{solvable}_0\ \mathit{eq} \Longrightarrow \exists\, e.\ I_a\ \mathit{eq}\ (e{\cdot}\mathit{xs})$.
- Trivial atoms satisfy every valuation: $\mathit{trivial}\ \mathit{eq} \Longrightarrow I_a\ \mathit{eq}\ \mathit{xs}$.
- Function subst_0 substitutes its first argument, a solvable equality, into its second argument. This is expressed by requiring that the substitution lemma must hold under certain conditions: *If* $\mathit{solvable}_0\ \mathit{eq}$ *and* $\neg\ \mathit{trivial}\ \mathit{eq}$ *and* $I_a\ \mathit{eq}\ (x{\cdot}\mathit{xs})$ *and* $\mathit{depends}_0\ a$ *then* $I_a\ (\mathit{subst}_0\ \mathit{eq}\ a)\ \mathit{xs} = I_a\ a\ (x{\cdot}\mathit{xs})$. And substituting a solvable atom into itself results in a trivial atom: $\mathit{solvable}_0\ \mathit{eq} \Longrightarrow \mathit{trivial}\ (\mathit{subst}_0\ \mathit{eq}\ \mathit{eq})$.

Now we can define a lifting function that takes a quantifier elimination procedure *qe* on lists of atoms and extends it to lists containing trivial atoms (by filtering them out) and solvable atoms (by substituting them in):

$$\begin{array}{l} \mathit{lift\text{-}eq\text{-}qe}\ \mathit{qe}\ \mathit{as} = \\ (\mathsf{let}\ \mathit{as} = [a{\leftarrow}\mathit{as}.\ \neg\ \mathit{trivial}\ a] \\ \ \mathsf{in}\ \mathsf{case}\ [a{\leftarrow}\mathit{as}.\ \mathit{solvable}_0\ a]\ \mathsf{of} \\ \quad [] \Rightarrow \mathit{qe}\ \mathit{as} \\ \quad \mid \mathit{eq} \cdot \mathit{eqs} \Rightarrow (\mathsf{let}\ \mathit{ineqs} = [a{\leftarrow}\mathit{as}.\ \neg\ \mathit{solvable}_0\ a] \\ \qquad\qquad \mathsf{in}\ \mathit{list\text{-}conj}\ (\mathit{map}\ (\mathit{Atom} \circ \mathit{subst}_0\ \mathit{eq})\ (\mathit{eqs}\ @\ \mathit{ineqs})))) \end{array}$$

>From the assumptions of locale *ATOM-EQ* it is not hard to prove that if *qe* performs quantifier elimination on any list of unsolvable atoms depending on variable 0, then *lift-eq-qe qe* is a quantifier elimination procedure on any list of atoms depending on 0:

Lemma 5 *If* $\forall\, \mathit{as} \in \mathit{list}(|\mathit{depends}_0| \cap -|\mathit{solvable}_0|).$ *is-dnf-qe qe as then* $\forall\, \mathit{as} \in \mathit{list}|\mathit{depends}_0|.$ *is-dnf-qe* $(\mathit{lift\text{-}eq\text{-}qe}\ \mathit{qe})$ *as.*

In our instantiations, the unsolvable atoms will be the inequalities ($<$) and *qe* will only need to deal with them; $=$ is taken care of completely by this lifting process.

Finally we compose *lift-dnf-qe* and *lift-eq-qe*:

$$\mathit{lift\text{-}dnfeq\text{-}qe} = \mathit{lift\text{-}dnf\text{-}qe} \circ \mathit{lift\text{-}eq\text{-}qe}$$

and obtain a corollary to lemmas 4 and 5:

Corollary 1 *If* $\mathit{qe} \in \mathit{lists}\ |\mathit{depends}_0| \rightarrow |\mathit{qfree}|$ *and* $\forall\, \mathit{as} \in \mathit{lists}(|\mathit{depends}_0| \cap -|\mathit{solvable}_0|).$ *is-dnf-qe qe as then* $I\ (\mathit{lift\text{-}dnfeq\text{-}qe}\ \mathit{qe}\ \varphi)\ \mathit{xs} = I\ \varphi\ \mathit{xs}.$

In the same manner we obtain

Corollary 2 *If* $\mathit{qe} \in \mathit{list}\ |\mathit{depends}_0| \rightarrow |\mathit{qfree}|$ *then qfree* $(\mathit{lift\text{-}dnfeq\text{-}qe}\ \mathit{qe}\ \varphi).$

5. Dense Linear Orders

The theory of dense linear orders (without endpoints) is an extension of the theory of linear orders with the axioms

$$y < z \Longrightarrow \exists x.\, y < x \wedge x < z \qquad \exists u.\, x < u \qquad \exists l.\, l < x$$

It is the canonical example of quantifier elimination [23] and the basis for the arithmetic theories to come. The equivalence $(\exists x.\ y < x \wedge x < z) = (y < z)$ is an easy consequence of the axioms. It generalizes to arbitrary conjunctions of inequalities containing the quantified variable: partition the inequalities into those of the form $l_i < x$ and those of the form $x < u_j$ and combine all pairs:

$$(\exists x.\, (\bigwedge_i l_i < x) \wedge (\bigwedge_j x < u_j)) \;=\; (\bigwedge_{ij} l_i < u_j) \tag{2}$$

The *only-if* direction holds by transitivity. The *if* direction follows because the right-hand formula is just another way of saying that the maximum of the l_i is less than the minimum of the u_j. By denseness there must exists a value in between, which is the witness for the existential formula.

Now we formalize this theory and its quantifier elimination procedure. We concentrate on quantifier elimination via DNF, thus obtaining a non-elementary procedure.

5.1. Atoms

There are just the two relations $<$ and $=$ and no function symbols. Thus atomic formulae can be represented by the following datatype:

$$\mathit{atom} = \mathit{Less\ nat\ nat} \mid \mathit{Eq\ nat\ nat}$$

Because there are no function symbols, the arguments of the relations must be variables. For example, *Less i j* represents the atom $x_i < x_j$ in de Bruijn notation. We define two auxiliary predicates *is-Less* and *is-Eq* which do what their name suggests.

Now we can instantiate locale *ATOM*. Type parameter α becomes type *atom*. The interpretation function I_a becomes I_{dlo} where

$I_{dlo}\ (\mathit{Eq\ i\ j})\ \mathit{xs} = (\mathit{xs}_{[i]} = \mathit{xs}_{[j]})$
$I_{dlo}\ (\mathit{Less\ i\ j})\ \mathit{xs} = (\mathit{xs}_{[i]} < \mathit{xs}_{[j]})$

The notation $\mathit{xs}_{[i]}$ means selection of the *i*th element of *xs*. The type of I_{dlo} is explicitly restricted such that *xs* must be a list of elements over a dense linear order, where the latter is formalized as a type class [19] with the axioms shown at the start of this section. Thus all valuations in this section are over dense linear orders. Parameter *aneg* becomes neg_{dlo}:

$\mathit{neg}_{dlo}\ (\mathit{Less\ i\ j}) = \mathit{Or}\ (\mathit{Atom}\ (\mathit{Less\ j\ i}))\ (\mathit{Atom}\ (\mathit{Eq\ i\ j}))$
$\mathit{neg}_{dlo}\ (\mathit{Eq\ i\ j}) = \mathit{Or}\ (\mathit{Atom}\ (\mathit{Less\ i\ j}))\ (\mathit{Atom}\ (\mathit{Less\ j\ i}))$

The instantiation of the parameters *adepends* and *adecr* is obvious:

$depends_{dlo}\ (Eq\ i\ j) = (i = 0 \vee j = 0)$
$depends_{dlo}\ (Less\ i\ j) = (i = 0 \vee j = 0)$

$decr_{dlo}\ (Less\ i\ j) = Less\ (i - 1)\ (j - 1)$
$decr_{dlo}\ (Eq\ i\ j) = Eq\ (i - 1)\ (j - 1)$

It is straightforward to show that this instantiation satisfies all the axioms of *ATOM*. The extension to *ATOM-EQ* (see §4.4.3) is easy: $solvable_0$ becomes $\lambda Eq\ i\ j \Rightarrow i{=}0 \vee j{=}0 \mid a \Rightarrow False$, *trivial* becomes $\lambda Eq\ i\ j \Rightarrow i{=}j \mid a \Rightarrow False$ and $subst_0$ is defined as follows:

$subst_0\ (Eq\ i\ j)\ (Less\ m\ n) = Less\ (subst\ i\ j\ m)\ (subst\ i\ j\ n)$
$subst_0\ (Eq\ i\ j)\ (Eq\ m\ n) = Eq\ (subst\ i\ j\ m)\ (subst\ i\ j\ n)$

$subst\ i\ j\ k = (\mathsf{if}\ k = 0\ \mathsf{then\ if}\ i = 0\ \mathsf{then}\ j\ \mathsf{else}\ i\ \mathsf{else}\ k) - 1$

Discharging the assumptions of *ATOM-EQ* is straightforward.

5.2. Quantifier Elimination

The quantifier elimination procedure sketched above assumes that it is given a list, i.e. conjunction of atoms. Variable 0, the innermost one, is to be eliminated. Because *lift-dnfeq-qe* already takes care of equalities, we can concentrate on the case where all atoms are *Less*:

qe-less as =
  (*if Less 0 0* ∈ *set as* **then** *FalseF* **else**
   let *lbs* = [*i*. *Less* (*Suc i*) *0* ← *as*];
       ubs = [*j*. *Less 0* (*Suc j*) ← *as*];
       pairs = [*Atom*(*Less i j*). *i* ← *lbs*, *j* ← *ubs*]
   in *list-conj pairs*)

This is exactly the above informal algorithm, except that we also take care of the unsatisfiable atom $x_0 < x_0$ and we decrement the variables to compensate for the eliminated quantifier. Instead of detecting only the contradiction $x_0 < x_0$ one could (and should) return *FalseF* upon finding any $x_i < x_i$.

5.3. Correctness

Theorem 1 $\forall a{\in}$*set as*. *is-Less a* $\wedge$ $depends_{dlo}\ a \Longrightarrow$ *is-dnf-qe qe-less as*

Remember that *is-dnf-qe* abbreviates an equivalence (see §4.4.1). The proof of the $\rightarrow$-direction of the equivalence distinguishes whether *lbs* or *ubs* are empty (in which case the lack of endpoints guarantees the existence of *x*) or not (in which case density comes to the rescue). The other direction follows via transitivity.

Defining *dlo-qe* = *lift-dnfeq-qe qe-less* we obtain the main result

Corollary 3 $I\ (dlo\text{-}qe\ \varphi)\ xs = I\ \varphi\ xs$

as a consequence of Corollary 1, Theorem 1 and the lemma *qfree* (*qe-less as*).

6. Linear Real Arithmetic

Linear real arithmetic is concerned with terms built up from variables, constants, addition, and multiplication with constants. Relations between such terms can be put into a normal form $r \bowtie c_0 * x_0 + \cdots c_n * x_n$ with $\bowtie \in \{=, <\}$ and $r, c_0, \ldots, c_n \in \mathbb{R}$. It is this normal form we work with in this section.

Note that although we phrase everything in terms of the real numbers, the rational number work just as well. In fact, any ordered, divisible, torsion free, Abelian group will do.

This time we will present two quantifier elimination procedures: one resembling the one for DLO, so called Fourier-Motzkin elimination, and a clever algorithm due to Ferrante and Rackoff [14] which brings the complexity down from non-elementary to doubly exponential.

6.1. Atoms

Type *atom* formalizes the normal forms explained above:

$$atom = Less\ real\ (real\ list)\ \mid\ Eq\ real\ (real\ list)$$

The second constructor argument is the list of coefficients [c_0,. . .,c_n] of the variables *0* to *n* — remember de Bruijn! Coefficient lists should be viewed as vectors and we define the usual vector operations on them:

$x *_s xs$ is the componentwise multiplication of a scalar x with a vector xs.
$xs + ys$ and $xs - ys$ are componentwise addition and subtraction on two vectors.
$\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x*y)$ is the inner product of two vectors, i.e. the sum over the componentwise products.

If the two vectors involved in an operation are of different length, the shorter one is padded with 0s (as in Obua's treatment of matrices [28]). We can prove all the algebraic properties we need, like $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$.

Now we instantiate locale *ATOM* just like for DLO in §5.1. The main function is the interpretation I_R of atoms, which is straightforward:

$I_R\ (Less\ r\ cs)\ xs = (r < \langle cs,xs\rangle)$
$I_R\ (Eq\ r\ cs)\ xs = (r = \langle cs,xs\rangle)$

Although this is irrelevant in our context, note that our lists do not form a vector space. Clearly the 0 vector would have to be [], but then there are almost no inverses: [x] + [$-x$] is [*0*] but not []. For a vectors space we would need to remove trailing 0s after each operation. And certain laws like $xs + 0 = xs$ would only hold for vectors without trailing 0s. A proper treatment of vectors spaces requires lists of different lengths to have different types. A solution along these lines is given by Harrison [21].

It is easy to extend the instantiation of *ATOM* to *ATOM-EQ* (see §4.4.3): $solvable_0$ is any *Eq* whose head coefficient is nonzero ($\lambda Eq\ r\ (c{\cdot}cs) \Rightarrow c \neq 0 \mid a \Rightarrow False$), *trivial* is any *Eq* where both sides are zero ($\lambda Eq\ r\ cs \Rightarrow r{=}0 \wedge (\forall c \in set\ cs.\ c{=}0) \mid a \Rightarrow False$), and $subst_0$ is defined as follows:

$subst_0\ (Eq\ r\ (c{\cdot}cs))\ (Less\ s\ (d{\cdot}ds)) = Less\ (s - r * d\ /\ c)\ (ds - (d\ /\ c) *_s cs)$
$subst_0\ (Eq\ r\ (c{\cdot}cs))\ (Eq\ s\ (d{\cdot}ds)) = Eq\ (s - r * d\ /\ c)\ (ds - (d\ /\ c) *_s cs)$

Discharging the assumptions of *ATOM-EQ* is straightforward.

6.2. Fourier-Motzkin Elimination

Fourier-Motzkin Elimination is a procedure discovered by Fourier [15].[1] Essentially, it works like for dense linear orders. You put the formula into DNF and for each conjunct the inequalities are split into those of the form $l < x$ and those of the form $x < u$, and then you "multiply out" exactly as in (2). Except that one has to transform the inequalities into the form $l < x$ and $x < u$ explicitly and the l and u can be proper terms, not just variables.

Quantifier elimination for the special case of a list of atoms *as*, all of which are of the form *Less*, is a one-liner

$qe\text{-}less\ as = list\text{-}conj\ [Atom(combine\ p\ q).\ p{\leftarrow}lbounds\ as, q{\leftarrow}ubounds\ as]$

where *lbounds* and *ubounds* select the inequalities where variable 0 has respectively a positive and a negative coefficient

$lbounds\ as = [(r/c, (-1/c) *_s cs).\ Less\ r\ (c{\cdot}cs) \leftarrow as, c{>}0]$
$ubounds\ as = [(r/c, (-1/c) *_s cs).\ Less\ r\ (c{\cdot}cs) \leftarrow as, c{<}0]$

and they are combined as explained above:

$combine\ (r_1, cs_1)\ (r_2, cs_2) = Less\ (r_1 - r_2)\ (cs_2 - cs_1)$

The correctness theorem

Theorem 2 $\forall a{\in}set\ as.\ is\text{-}Less\ a \wedge depends_R\ a \Longrightarrow is\text{-}dnf\text{-}qe\ qe\text{-}less\ as$

is proved along the same lines as its counterpart Theorem 1, except that linear arithmetic reasoning is necessary now.

The extension with equality is provided by locale *ATOM-EQ*. Defining $lin\text{-}qe = lift\text{-}dnfeq\text{-}qe\ qe\text{-}less$ we obtain the main result

Corollary 4 $I\ (lin\text{-}qe\ \varphi)\ xs = I\ \varphi\ xs$

as a consequence of Corollary 1, Theorem 2 and the lemma $qfree\ (qe\text{-}less\ as)$.

Above we transformed inequalities into $l < x$ and $x < u$ by dividing with the coefficient of x. Alternatively one can combine $r_1 < c_1x + t_1$ and $r_2 < c_2x + t_2$ into $c_1r_2 - c_2r_1 < c_1t_2 - c_2t_1$ provided $c_1 > 0$, $c_2 < 0$, and x does not occur in the t_i.

[1] Motzkin [26] and Farkas [13] cited Fourier but were concerned with the algebraic background, not the algorithm.

6.3. An Optimization

The above code is correct but produces horribly bloated results: even if the initial formula is closed, the result will not just be *TrueF* or *FalseF* but some complicated formula equivalent to that. As a trivial example take $\exists x.1 < x \wedge x < 2$. It is converted to $1 < 2$. To be able to cope with larger inputs, it is essential to simplify intermediate results as much as possible: at the very least, unsatisfiable atoms should be replaced by *FalseF* and tautological ones by *TrueF*. This is very easy to spot: *Less r cs* is unsatisfiable/tautological iff all elements of *cs* are 0 and $r \geq 0 / r < 0$. Here is a corresponding function which simplifies individual atoms to *TrueF* or *FalseF* whenever it can:

asimp (*Less r cs*) =
(*if* $\forall$ *c*∈*set cs*. *c* = *0* *then if r* < *0* *then* *TrueF* *else* *FalseF* *else* *Atom* (*Less r cs*))
asimp (*Eq r cs*) =
(*if* $\forall$ *c*∈*set cs*. *c* = *0* *then if r* = *0* *then* *TrueF* *else* *FalseF* *else* *Atom* (*Eq r cs*))

This simplification is applied when lower and upper bounds are combined:

qe-less′ as = *list-conj* [*asimp*(*combine p q*). *p*←*lbounds as*, *q*←*ubounds as*]

The definition of *list-conj* via *and* ensures that any *TrueF* is dropped and any *FalseF* propagates to the output.

It is not hard to prove that *I* (*qe-less′ as*) *xs* = *I* (*qe-less as*) *xs*, from which the analogous version of Corollary 4 for *lin-qe′* = *lift-dnfeq-qe qe-less′* instead of *lin-qe* follows easily.

6.4. Ferrante and Rackoff

Fourier-Motzkin elimination has non-elementary complexity because of the repeated DNF conversions. Ferrante and Rackoff [14], inspired by Cooper [11], avoid putting the formula explicitly into DNF but still capitalize on the fact that it has a DNF. Below, let ϕ be some quantifier-free formula with a free variable x. Substituting x by some r is written $\phi(r)$.

When eliminating x from ϕ, we can partition the atoms of ϕ that depend on x into 3 categories: $l < x$, $x < u$ and $x = t$. Let $LB(\phi)$ denote the set of all such l in ϕ, $UB(\phi)$ the set of such u, and $EQ(\phi)$ the set of such t. The DNF of a formula over these atoms can be seen as a finite union of finite intersections of half-open intervals (l, ∞) and $(-\infty, u)$ and points t. Each such intersection is equivalent to either a single interval (l, ∞), $(-\infty, u)$ or (l, u), or to a point t — or it is empty, in which case we can silently forget about it. Thus there are 4 possibilities why ϕ can hold: $\phi(x)$ holds for any sufficiently large x (case (l, ∞)), $\phi(x)$ holds for any sufficiently small x (case $(-\infty, u)$), $\phi(x)$ holds for all $x \in (l, u)$ for some $l \in LB(\phi)$ and $u \in UB(\phi)$, or $\phi(t)$ for some $t \in EQ(\phi)$. This leads to the following optimized version of the equivalence due to Ferrante and Rackoff: [2]

[2] The special treatment of equality is missing in Ferrante and Rackoff's work, probably to simplify matters. The asymptotic complexity remains unaffected.

$$(\exists x.\phi(x)) \;=\; (\phi(-\infty) \vee \phi(\infty) \vee \bigvee_{\substack{l\in LB(\phi)\\ u\in UB(\phi)}} \phi(\frac{l+u}{2}) \vee \bigvee_{t\in EQ(\phi)} \phi(t)) \qquad (3)$$

The choice of $(l+u)/2$ is arbitrary: any value in (l,u) will do.

Notation $\phi(-\infty)$ and $\phi(\infty)$ is merely suggestive syntax for the following form of "substitution":

$$\begin{array}{ll} (-\infty < u) = True & (\infty < u) = False \\ (l < -\infty) = False & (l < \infty) = True \\ (-\infty = t) = False & (\infty = t) = False \end{array}$$

Ferrante and Rackoff only sketch the proof of (3). We examine some of the delicate details. The proof of the $\leftarrow$-direction is obvious in case the witness is $(l+u)/2$ or t. For $-\infty$ and ∞, the following lemmas provide the witness:

$$\exists x.\forall y \le x.\ \phi(-\infty) = \phi(y) \qquad \exists x.\forall y \ge x.\ \phi(\infty) = \phi(y)$$

They are proved by induction on ϕ.

The proof of the $\rightarrow$-direction is more subtle. We have $\phi(x)$. Assuming $x \notin EQ(\phi)$, $\neg\phi(-\infty)$ and $\neg\phi(\infty)$, we have to show that $\phi((l+u)/2)$ for some $l \in LB(\phi)$ and $u \in UB(\phi)$. In fact, we show that there are l and u such that $l < u$ and $\phi(y)$ for all $y \in (l,u)$. From the assumptions it follows by induction on ϕ that there must be $l_0 \in LB(\phi)$ and $u_0 \in UB(\phi)$ such that $x \in (l_0,u_0)$. Now we show (by induction on ϕ) the lemma that "innermost" intervals (l,u) completely satisfy ϕ:

Lemma 6 *If* $\phi(x)$, $x \in (l,u)$, $x \notin EQ(\phi)$, $(l,x) \cap LB(\phi) = \emptyset$ *and* $(x,u) \cap UB(\phi) = \emptyset$, *then* $\forall y \in (l,u).\ \phi(y)$.

Given $x \in (l_0,u_0)$ we define $l = \max\{l \in LB(\phi) \mid l < x\}$ and $u = \min\{u \in UB(\phi) \mid x < u\}$. It is easy to see that this satisfies the premises of the lemma and the desired conclusion follows.

Now we describe the implementation of Ferrante and Rackoff's procedure, starting at the top with (3):

$$\begin{array}{l} FR_1\ \varphi = (\mathsf{let}\ as = atoms_0\ \varphi;\ lbs = lbounds\ as;\ ubs = ubounds\ as; \\ \qquad intvs = [subst\ \varphi\ (between\ p\ q)\ .\ p \leftarrow lbs, q \leftarrow ubs]; \\ \qquad eqs = [subst\ \varphi\ rcs\ .\ rcs \leftarrow ebounds\ as] \\ \quad \mathsf{in}\ list\text{-}disj\ (inf_-\ \varphi \cdot inf_+\ \varphi \cdot intvs\ @\ eqs)) \end{array}$$

Function FR_1 expects a formula φ in NNF. Function $atoms_0$ collects the atoms of φ that depend on variable 0. Functions LB, UB and EQ are realized by *lbounds*, *ubounds* (see above) and *ebounds*:

$$ebounds\ as = [(r/c, (-1/c) *_s cs).\ Eq\ r\ (c{\cdot}cs) \leftarrow as, c{\neq}0]$$

Function *between* picks the mid-point between two points:

$$between\ (r, cs)\ (s, ds) = ((r+s)\ /\ 2, (1\ /\ 2) *_s (cs + ds))$$

Substitution, as usual for variable 0, is first defined for atoms

$$asubst\ (r, cs)\ (Less\ s\ (d{\cdot}ds)) = Less\ (s - d * r)\ (d *_s cs + ds)$$
$$asubst\ (r, cs)\ (Eq\ s\ (d{\cdot}ds)) = Eq\ (s - d * r)\ (d *_s cs + ds)$$
$$asubst\ (r, cs)\ (Less\ s\ []) = Less\ s\ []$$
$$asubst\ (r, cs)\ (Eq\ s\ []) = Eq\ s\ []$$

and then lifted to formulae: $subst\ \varphi\ rcs \equiv map_{fm}\ (asubst\ rcs)\ \varphi$. The characteristic lemma is

$$qfree\ \varphi \Longrightarrow I\ (subst\ \varphi\ (r, cs))\ xs = I\ \varphi\ ((r + \langle cs,xs\rangle){\cdot}xs)$$

It remains to define the substitution of $-\infty$ for 0:

$$inf_-\ (And\ \varphi_1\ \varphi_2) = and\ (inf_-\ \varphi_1)\ (inf_-\ \varphi_2)$$
$$inf_-\ (Or\ \varphi_1\ \varphi_2) = or\ (inf_-\ \varphi_1)\ (inf_-\ \varphi_2)$$
$$inf_-\ (Atom\ (Less\ r\ (c{\cdot}cs))) =$$
$$(\mathsf{if}\ c < 0\ \mathsf{then}\ TrueF\ \mathsf{else}\ \mathsf{if}\ 0 < c\ \mathsf{then}\ FalseF\ \mathsf{else}\ Atom\ (Less\ r\ cs))$$
$$inf_-\ (Atom\ (Eq\ r\ (c{\cdot}cs))) = (\mathsf{if}\ c = 0\ \mathsf{then}\ Atom\ (Eq\ r\ cs)\ \mathsf{else}\ FalseF)$$

The remaining cases are the identity. The definition of inf_+ is dual.

The proof of the main correctness theorem

$$nqfree\ \varphi \Longrightarrow I\ (FR_1\ \varphi)\ xs = (\exists x.\ I\ \varphi\ (x{\cdot}xs))$$

is essentially the proof of (3). Defining $FR = lift\text{-}nnf\text{-}qe\ FR_1$ we obtain the overall correctness as a corollary to Lemma 2: $I\ (FR\ \varphi)\ xs = I\ \varphi\ xs$.

Ferrante and Rackoff show that their procedure executes in space $O(2^{cn})$ and hence time $O(2^{2^{dn}})$ where n is the size of the input. This significant improvement over the non-elementary complexity of Fourier's procedure becomes relevant in the context of deeply nested and alternating quantifiers because that is the situation where conversion to DNF can blow up repeatedly.

7. Presburger Arithmetic

Presburger Arithmetic is linear integer arithmetic. Presburger [29] showed that this theory has quantifier elimination. In contrast to linear real arithmetic we need an additional predicate to obtain quantifier elimination: there is no quantifier-free equivalent of $\exists x.\ x + x = y$ if we restrict to linear arithmetic. The way out is to allow the divisibility predicate as well, but only of the form $d \mid t$ where d is a constant. Now $\exists x.\ x + x = y$ is equivalent with $2 \mid x$. Alternatively one can introduce congruence relations $s \equiv t\ (mod\ d)$ instead of divisibility. On the other hand we do not need both $<$ and $=$ (or $\leq$) on the integers because $i < j$ is equivalent with $i + 1 \leq j$. Hence we restrict our attention to $\leq$. All atoms are assumed to be of the form $i \leq k_0 * x_0 + \cdots + k_n * x_n$ or $d \parallel i + k_0 * x_0 + \cdots k_n * x_n$, where $\parallel$ is $\mid$ or $\nmid$, and $d, i, k_0, \ldots, k_n \in \mathbb{Z}$ and $d > 0$. The negated atom $i \not\leq j$ is equivalent with $j + 1 \leq i$.

7.1. Atoms

The above language of atoms is formalized as follows:

$$atom = Le\ int\ (int\ list) \mid Dvd\ int\ int\ (int\ list) \mid NDvd\ int\ int\ (int\ list)$$

Atoms are interpreted w.r.t. a list of variables as usual:

$I_Z\ (Le\ i\ ks)\ xs = (i \leq \langle ks,xs \rangle)$
$I_Z\ (Dvd\ d\ i\ ks)\ xs = (d\ dvd\ i + \langle ks,xs \rangle)$
$I_Z\ (NDvd\ d\ i\ ks)\ xs = (\neg\ d\ dvd\ i + \langle ks,xs \rangle)$

where *dvd* is HOL's divisibility predicate. Note that we can reuse the polymorphic vector, i.e. list operations like $\langle .,. \rangle$ introduced for linear real arithmetic.

There is a slight complication here: We want to exclude the atoms *Dvd 0 i ks* and *NDvd 0 i ks* because they behave anomalously and the algorithm does not generate them either. Catering for them would complicate the algorithm with case distinctions. In order to restrict attention to a subset of atoms, locale *ATOM* in fact has another parameter not mentioned so far: $anormal :: \alpha \Rightarrow bool$ with the axioms

$$anormal\ a \Longrightarrow \forall\, b{\in}atoms\ (aneg\ a).\ anormal\ b$$
$$\neg\ depends_0\ a \Longrightarrow anormal\ a \Longrightarrow anormal\ (decr\ a)$$

In words: negation and decrementation do not lead outside the normal atoms.

A formula is defined as normal iff all its atoms are:

$normal\ \varphi = (\forall\, a{\in}atoms\ \varphi.\ anormal\ a)$

With the help of the above axioms the following modified version of Lemma 4 can be proved:

Lemma 7 *If* $qe \in lists\ |depends_0| \rightarrow |qfree|$ *and* $qe \in lists\ (|depends_0| \cap |anormal|) \rightarrow |normal|$ *and* $\forall\, as \in lists(\ |depends_0| \cap |anormal|\ )$. *is-dnf-qe qe as then normal* φ *implies I* $(lift\text{-}dnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$.

The parameters of locale *ATOM* are instantiated as follows. The interpretation of atoms is given by function I_Z above, their negation by

$neg_Z\ (Le\ i\ ks) = Atom\ (Le\ (1 - i)\ (-\ ks))$
$neg_Z\ (Dvd\ d\ i\ ks) = Atom\ (NDvd\ d\ i\ ks)$
$neg_Z\ (NDvd\ d\ i\ ks) = Atom\ (Dvd\ d\ i\ ks)$

and their decrementation by

$decr_Z\ (Le\ i\ ks) = Le\ i\ (tl\ ks)$
$decr_Z\ (Dvd\ d\ i\ ks) = Dvd\ d\ i\ (tl\ ks)$
$decr_Z\ (NDvd\ d\ i\ ks) = NDvd\ d\ i\ (tl\ ks)$

Parameter $depends_0$ becomes $\lambda a.\ hd\text{-}coeff\ a \neq 0$ where

hd-coeff (*Le i ks*) = (*case ks of* [] ⇒ *0* | *k*·*x* ⇒ *k*)
hd-coeff (*Dvd d i ks*) = (*case ks of* [] ⇒ *0* | *k*·*x* ⇒ *k*)
hd-coeff (*NDvd d i ks*) = (*case ks of* [] ⇒ *0* | *k*·*x* ⇒ *k*)

and parameter *anormal* becomes λ*a*. *divisor a* ≠ *0* where

divisor (*Le i ks*) = *1*
divisor (*Dvd d i ks*) = *d*
divisor (*NDvd d i ks*) = *d*

7.2. Algorithm

In this section we describe and formalize a DNF-based algorithm. It differs from Presburger's original algorithm because that one covers only $=$ (and congruence) — Presburger merely states that it can be extended to $<$. Our algorithm resembles Enderton's version [12], except that the main case split is different: Enderton distinguishes if there are congruences or not, we distinguish if there are lower bounds or not.

Input to the algorithm is $P(x)$, a conjunction of atoms. As an example we pick $l \leq 2x \wedge 3x \leq u$. The algorithm consists of the following steps:

1. Set all coefficients of x to the positive least common multiple (lcm) of all coefficients of x in $P(x)$. Call the result $Q(m * x)$.
   Example: $Q(6 * x) = (3l \leq 6x \wedge 6x \leq 2u)$.
2. Set $R(x) = Q(x) \wedge m \mid x$.
   Example: $R(x) = (3l \leq x \wedge x \leq 2u \wedge 6 \mid x)$.
3. Let δ be the lcm of all divisors d in $R(x)$ and let L be the set of lower bounds for x in $R(x)$. If $L \neq \emptyset$ then return $\bigvee_{t \in T} R(t)$ where $T = \{l+n \mid l \in L \wedge 0 \leq n < \delta\}$. If $L = \emptyset$ return $\bigvee_{t \in T} R'(t)$ where R' is R without $\leq$-atoms and $T = \{n \mid 0 \leq n < \delta\}$.
   Example: $\delta = 6$, $L = \{3l\}$ and the result is $\bigvee_{0 \leq n < 6} R(3l + n)$.

Instead of lower bounds, one may just as well choose upper bounds. In fact, as a local optimization one typically picks the smaller of the two sets.

The first two steps of the algorithm are clearly equivalence preserving. Now we have a conjunction $R(x)$ of atoms where x has coefficient 1 everywhere. Equivalence preservation of the last step is proved in both directions separately.

First we assume the returned formula and show $R(t)$ for some t. If $L \neq \emptyset$ then $R(t)$ for some $t \in T$ and we are done. Now assume $L = \emptyset$. By assumption there must be some $0 \leq n < \delta$ such that $R'(n)$. If there are no upper bounds for x in $R(x)$ either, then $R(x)$ contains no $\leq$-atoms, $R' = R$, and hence $R(n)$. Otherwise let U be the set of all upper bounds of x in $R(x)$, let m be the minimum of U and let $t = n - ((n - m)\, div\, \delta + 1)\delta$. We show $R(t)$. >From $R'(n)$ and the definition of R' and t, $R'(t)$ follows. All $\leq$-atoms must be upper bound constraints $x \leq u$ and hence $m \leq u$. Because $(n - m)\, mod\, \delta < \delta$ we obtain $t \leq m \leq u$. Thus t satisfies all $\leq$-atoms, and hence $R(t)$.

Now assume that $R(z)$ for some z. In this direction it is important to note that (non)divisibility atoms $a(x)$ are cyclic in their divisor d, i.e. $a(x)$ is equivalent with $a(x\, mod\, d)$ because the coefficient of x is 1. This carries over to any multiple of d, in particular δ. If $L = \emptyset$ we obtain $R'(z\, mod\, \delta)$ with $0 \leq z\, mod\, \delta < \delta$ as required because $R'(x)$ consists only of (non)divisibility atoms. If $L \neq \emptyset$ we show $R(t)$ where $t = m + n$

where m is the maximum of L and $n = (z - m)\ mod\ \delta$. Let $a(x)$ be some atom in $R(x)$. If a is a lower bound atom for x, $a(t)$ follows because $t \geq m$ and m is the maximum of L. If a is an upper bound atom for x, $a(t)$ follows because $t \leq z$ and $a(z)$. If a is a (non)divisibility atom, $a(t)$ follows from $a(z)$ because $t\ mod\ \delta = z\ mod\ \delta$.

7.3. Formalization

The above algorithm consist of two steps which we implement separately. First the head coefficients of a list of atoms are set to 1 or -1 and the divisibility predicate is added:

hd-coeffs1 as =
(*let m* = *zlcms* (*map hd-coeff as*) *in Dvd m 0* [*1*]·*map* (*hd-coeff1 m*) *as*)

where *zlcms* computes the positive lcm of a list of integers, *hd-coeff* extracts the head coefficient from an atom (see §7.1), and *hd-coeff1* sets the head coefficient of one atom to 1 or -1:

hd-coeff1 m (*Le i* (*k*·*ks*)) =
(*let m'* = *m div* |*k*| *in Le* (*m'* ∗ *i*) (*sgn k*·*m'* ∗$_s$ *ks*))
hd-coeff1 m (*Dvd d i* (*k*·*ks*)) =
(*let m'* = *m div k in Dvd* (*m'* ∗ *d*) (*m'* ∗ *i*) (*1*·*m'* ∗$_s$ *ks*))
hd-coeff1 m (*NDvd d i* (*k*·*ks*)) =
(*let m'* = *m div k in NDvd* (*m'* ∗ *d*) (*m'* ∗ *i*) (*1*·*m'* ∗$_s$ *ks*))

sgn i = (*if i* = *0 then 0 else if 0* < *i then 1 else* − *1*)

We prove that *hd-coeffs1* leaves the interpretation unchanged:

Lemma 8 *If* ∀ *a*∈*set as. hd-coeff a* ≠ *0 then* (∃ *x.* ∀ *a*∈*set* (*hd-coeffs1 as*). I_Z *a* (*x*·*e*)) = (∃ *x.* ∀ *a*∈*set as.* I_Z *a* (*x*·*e*)).

In the second step the actual quantifier elimination is performed:

```
qe-pres as =
  (let ds = [a←as. is-dvd a]; d = zlcms(map divisor ds); ls = lbounds as
   in if ls = []
      then Disj [0..d−1] (λn. list-conj(map (Atom ∘ asubst n []) ds))
      else Disj ls (λ(i,ks).
             Disj [0..d−1] (λn. list-conj(map (Atom ∘ asubst (i+n) (−ks)) as))))
```

where *is-dvd a* is true iff *a* is of the form *Dvd* or *NDvd*, and *lbounds* collects the lower bounds for variable 0, *lbounds as* = [(*i*,*ks*). *Le i* (*k*·*ks*) ← *as*, *k*>*0*], and *asubst* is substitution:

asubst i' ks' (*Le i* (*k*·*ks*)) = *Le* (*i* − *k* ∗ *i'*) (*k* ∗$_s$ *ks'* + *ks*)
asubst i' ks' (*Dvd d i* (*k*·*ks*)) = *Dvd d* (*i* + *k* ∗ *i'*) (*k* ∗$_s$ *ks'* + *ks*)
asubst i' ks' (*NDvd d i* (*k*·*ks*)) = *NDvd d* (*i* + *k* ∗ *i'*) (*k* ∗$_s$ *ks'* + *ks*)

The following lemma shows that *asubst* is indeed substitution:

$I_Z\ (asubst\ i\ ks\ a)\ xs = I_Z\ a\ ((i + \langle ks,xs\rangle)\cdot xs)$

The actual quantifier elimination procedure is the lifted composition of the two basic steps:

$pres\text{-}qe = lift\text{-}dnf\text{-}qe\ (qe\text{-}pres \circ hd\text{-}coeffs1)$

7.4. Correctness

The main correctness theorem is

Theorem 3 *If* $\forall a{\in}set\ as.\ divisor\ a \neq 0$ *and* $\forall a{\in}set\ as.\ hd\text{-}coeff\text{-}is1\ a$ *then* $I\ (qe\text{-}pres\ as)\ xs = (\exists x.\ \forall a{\in}set\ as.\ I_Z\ a\ (x\cdot xs))$.

Its proof was given in §7.2. Predicate *hd-coeff-is1 a* is true iff the head coefficient of *a* is 1 or -1. Combining this theorem with Lemma 8 (and the lemma that *hd-coeff1* establishes *hd-coeff-is1*) yields: *If* $\forall a{\in}set\ as.\ divisor\ a \neq 0$ *and* $\forall a{\in}set\ as.\ hd\text{-}coeff\ a \neq 0$ *then* $I\ ((qe\text{-}pres \circ hd\text{-}coeffs1)\ as)\ e = (\exists x.\ \forall a{\in}set\ as.\ I_Z\ a\ (x\cdot e))$. Because $depends_0\ a = (hd\text{-}coeff\ a \neq 0)$ and $anormal\ a = (divisor\ a \neq 0)$, Lemma 7 yields as a corollary:

$$normal\ \varphi \Longrightarrow I\ (pres\text{-}qe\ \varphi)\ xs = I\ \varphi\ xs$$

This requires an easy $(qfree\ ((qe\text{-}pres \circ hd\text{-}coeffs1)\ as)$ and a tedious lemma:
if $\forall a{\in}set\ as.\ hd\text{-}coeff\ a \neq 0 \wedge divisor\ a \neq 0$ then $normal((qe\text{-}pres{\circ}hd\text{-}coeffs1)\ as)$.

8. Related Work

This paper is an outgrowth of [9]. One of the many differences of the two papers is the replacement of Cooper's NNF-based algorithm [11] for Presburger arithmetic by a DNF-based one. These two algorithms are related to each other like Ferrante and Rackoff's is to Fourier's. One can also view this article as translating some of the programs in Harrison's forthcoming textbook [20] from OCaml to HOL and verifying them (and a number of additional ones).

Another popular quantifier elimination method for Presburger arithmetic is due to Pugh [30] and takes Fourier's method as a starting point. Linear arithmetic over both reals and integers also admits quantifier elimination [32]. Chaieb reflected this algorithm in Isabelle [7]. The decision problem for first-order arithmetic theories can also be solved by automata theoretic methods. Büchi [6] initiated this approach for Presburger arithmetic. It was later extended to mixed integer and real arithmetic [4].

An LCF-style quantifier elimination procedure for real closed fields has been implemented by McLaughlin [25], a reflective version of Collin's CAD method [10] has been implemented but only partly verified by Mahboubi [24].

The special case of decision procedures for quantifier free linear real arithmetic has received an enormous amount of attention for its practical relevance and because it is solvable in polynomial time. In particular it is possible to generate short certificates that can be checked quickly (e.g. [3]).

Acknowledgements

Amine Chaieb helped me to understand quantifier elimination. He and John Harrison were constant sources of ideas. Clemens Ballarin, Florian Haftmann and Makarius Wenzel conceived and implemented locales and helped me to use them.

References

[1] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lect. Notes in Comp. Sci.*, pages 34–50. Springer-Verlag, 2004.

[2] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. Borwein and W. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, volume 4108 of *Lect. Notes in Comp. Sci.*, pages 31–43. Springer-Verlag, 2006.

[3] F. Besson. Fast reflexive arithmetic tactics the linear case and beyond. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lect. Notes in Comp. Sci.*, pages 48–62. Springer-Verlag, 2007.

[4] B. Boigelot, S. Jodogne, and P. Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Log.*, 6:614–633, 2005.

[5] R. S. Boyer and J. S. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[6] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960.

[7] A. Chaieb. Verifying mixed real-integer quantifier elimination. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lect. Notes in Comp. Sci.*, pages 528–540. Springer-Verlag, 2006.

[8] A. Chaieb and T. Nipkow. Verifying and reflecting quantifier elimination for Presburger arithmetic. In G. Stutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lect. Notes in Comp. Sci.*, pages 367–380. Springer-Verlag, 2005.

[9] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. Technical report, Institut für Informatik, Technische Universität München, 2006. Submitted for publication.

[10] G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lect. Notes in Comp. Sci.*, pages 134–183. Springer-Verlag, 1976.

[11] D. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

[12] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[13] J. Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902.

[14] J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Computing*, 4:69–76, 1975.

[15] J. B. J. Fourier. Solution d'une question particulière du calcul des inégalités. In G. Darboux, editor, *Joseph Fourier - Œuvres complétes*, volume 2, pages 317–328. Gauthier-Villars, 1888–1890.

[16] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.

[17] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Int. Conf. Functional Programming*, pages 235–246. ACM Press, 2002.

[18] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends*. Department of Computer Science, University of Kaiserslautern, 2007.

[19] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lect. Notes in Comp. Sci.*, pages 160–174. Springer-Verlag, 2007.

[20] J. Harrison. *Introduction to Logic and Automated Theorem Proving*. Cambridge University Press. Forthcoming.

[21] J. Harrison. A HOL theory of Eucledian space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2005*, volume 3603 of *Lect. Notes in Comp. Sci.*, pages 114–129. Springer-Verlag, 2005.

[22] J. Harrison. Verifying nonlinear real formulas via sums of squares. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lect. Notes in Comp. Sci.*, pages 102–118. Springer-Verlag, 2007.

[23] C. Langford. Some theorems on deducibility. *Annals of Mathematics (2nd Series)*, 28:16–40, 1927.

[24] A. Mahboubi. *Contributions à la certification des calculs sur* $\mathbb{R}$ *: théorie, preuves, programmation*. PhD thesis, Université de Nice, 2006.

[25] S. McLaughlin and J. Harrison. A proof-producing decision procedure for real arithmetic. In R. Nieuwenhuis, editor, *Automated Deduction — CADE-20*, volume 3632 of *Lect. Notes in Comp. Sci.*, pages 295–314. Springer-Verlag, 2005.

[26] T. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. PhD thesis, Universität Basel, 1936.

[27] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

[28] S. Obua. Proving bounds for real linear programs in Isabelle/HOL. In J. Hurd, editor, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lect. Notes in Comp. Sci.*, pages 227–244. Springer-Verlag, 2005.

[29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[30] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[31] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[32] V. Weispfenning. Mixed real-integer linear quantifier elimination. In *International Symposium Symbolic and Algebraic Computation (ISSAC)*, pages 129–136. ACM Press, 1999.

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Content in Proofs of List Reversal

Helmut SCHWICHTENBERG
Ludwig-Maximilians-Universität, Munich, Germany

Abstract. Berger [2] observed that the well-known linear list reversal algorithm can be obtained as the computational content of a weak (or "classical") existence proof. The necessary tools are a refinement [3] of the Dragalin/Friedman [4,5] A-translation, and uniform (or "non-computational") quantifiers [1]. Both tools are implemented in the Minlog proof assistant (`www.minlog-system.de`), in addition to the more standard realizability interpretation. The aim of the present paper is to give an introduction into the theory underlying these tools, and to explain their usage in Minlog, using list reversal as a running example.

1. Minimal arithmetic in finite types

1.1. Gödel's T

Types are built from base types ι by function type formation $\rho \to \sigma$. As base types we only need the types $\mathbf{N}$ of natural numbers, $\mathbf{B}$ of booleans and the parametrized constructs of the list type $\mathbf{L}(\rho)$ and the pair type $\rho \times \sigma$. For these base types the constructors have standard names, as follows. We also spell out the type of their recursion operators:

$$\mathtt{tt}^{\mathbf{B}} := \mathrm{C}^{\mathbf{B}}_1, \quad \mathtt{ff}^{\mathbf{B}} := \mathrm{C}^{\mathbf{B}}_2,$$
$$\mathcal{R}^{\tau}_{\mathbf{B}} \colon \mathbf{B} \to \tau \to \tau \to \tau,$$

$$0^{\mathbf{N}} := \mathrm{C}^{\mathbf{N}}_1, \quad \mathrm{S}^{\mathbf{N}\to\mathbf{N}} := \mathrm{C}^{\mathbf{N}}_2,$$
$$\mathcal{R}^{\tau}_{\mathbf{N}} \colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau,$$

$$\mathrm{nil}^{\mathbf{L}(\rho)} := \mathrm{C}^{\mathbf{L}(\rho)}_1, \quad \mathrm{cons}^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)} := \mathrm{C}^{\mathbf{L}(\rho)}_2,$$
$$\mathcal{R}^{\tau}_{\mathbf{L}(\rho)} \colon \mathbf{L}(\rho) \to \tau \to (\rho \to \mathbf{L}(\rho) \to \tau \to \tau) \to \tau,$$

$$\big(\times^{+}_{\rho\sigma}\big)^{\rho\to\sigma\to\rho\times\sigma} := \mathrm{C}^{\rho\times\sigma}_1,$$
$$\mathcal{R}^{\tau}_{\rho\times\sigma} \colon \rho \times \sigma \to (\rho \to \sigma \to \tau) \to \tau.$$

One often writes $x :: l$ as shorthand for $\mathrm{cons}\, x\, l$, and $\langle y, z\rangle$ for $\times^{+} yz$. The *terms* of Gödel's T [7] are inductively defined from typed variables x^{ρ} and the constants, that is,

constructors and recursion operators, by abstraction $\lambda_{x^\rho} M^\sigma$ and application $M^{\rho\to\sigma} N^\rho$. For example, the *projections* of a pair to its components can be defined easily:

$$M0 := \mathcal{R}^\rho_{\rho\times\sigma} M^{\rho\times\sigma}(\lambda_{x^\rho,y^\sigma} x^\rho), \quad M1 := \mathcal{R}^\rho_{\rho\times\sigma} M^{\rho\times\sigma}(\lambda_{x^\rho,y^\sigma} y^\sigma).$$

Another example is the *append*-function $:+:$ for lists, which should satisfy the recursion equations

$$\mathrm{nil} :+: v_2 := v_2, \qquad (x :: v_1) :+: v_2 := x :: (v_1 :+: v_2).$$

It can be defined as the term

$$v_1 :+: v_2 := \mathcal{R}^{\mathbf{L}(\alpha)\to\mathbf{L}(\alpha)}_{\mathbf{L}(\alpha)} v_1(\lambda_{v_2} v_2)(\lambda_{x,\_,p,v_2}(x :: (pv_2)))v_2.$$

The conversion relations are the standard ones, and also the notion of a normal form of a term. Clearly normal closed terms are of the form $\mathrm{C}_i\vec{N}$.

1.2. Heyting Arithmetic

We define Heyting Arithmetic HA^ω for our language based on Gödel's T, which is finitely typed. Among the *prime formulas* are those built from terms r of type $\mathbf{B}$ by means of a special operator $\mathrm{atom}(r^{\mathbf{B}})$; we call them *decidable* prime formulas. They include for instance equations between terms of type $\mathbf{N}$, since the boolean-valued binary equality function $=_{\mathbf{N}}\colon \mathbf{N}\to\mathbf{N}\to\mathbf{B}$ can be defined by

$$(0 =_{\mathbf{N}} 0) := \mathtt{tt}, \quad (\mathrm{S}n =_{\mathbf{N}} 0) := \mathtt{ff},$$
$$(0 =_{\mathbf{N}} \mathrm{S}m) := \mathtt{ff}, \quad (\mathrm{S}n =_{\mathbf{N}} \mathrm{S}m) := (n =_{\mathbf{N}} m).$$

For falsity we can take the atomic formula $F := \mathrm{atom}(\mathtt{ff})$ – called *arithmetical falsity* – built from the boolean constant $\mathtt{ff}$. Below we will also need the (logical) falsity $\bot$, which we can view as just a particular propositional symbol. The *formulas* of HA^ω are built from prime formulas by the connectives $\to$ and $\forall$. We define *negation* $\neg A$ by $A\to F$ or $A\to\bot$ (depending on the context), and the *weak* (or "classical") existential quantifier by

$$\tilde{\exists}_x A := \neg\forall_x\neg A.$$

We use natural deduction rules: $\to^+$, $\to^-$, $\forall^+$ and $\forall^-$ of Figure 1. It will be convenient to write derivations as terms, where the derived formula is viewed as the "type" of the term. This representation is known under the name *Curry-Howard correspondence*. From now on we use M, N etc. to range over derivation terms, and r, s etc. for object terms.

We give an inductive definition of derivation terms in Figure 1, where for clarity we have written the corresponding derivations to the left. For the universal quantifier $\forall$ there is an introduction rule $\forall^+ x$ and an elimination rule $\forall^-$, whose right premise is the term

derivation	term
$u\colon A$	u^A
$\dfrac{\begin{array}{c}[u\colon A]\\ \mid M\\ B\end{array}}{A \to B}\,{\to}^+u$	$(\lambda_{u^A} M^B)^{A\to B}$
$\dfrac{\mid M \qquad\qquad \mid N}{A \to B \qquad\qquad A\quad}\,{\to}^-$	$(M^{A\to B} N^A)^B$
$\dfrac{\mid M}{A}\,\forall^+x$ (with var.cond.)	$(\lambda_x M^A)^{\forall_x A}$ (with var.cond.)
$\dfrac{\mid M}{\forall_x A(x) \qquad r}\,\forall^-$	$(M^{\forall_x A(x)} r)^{A(r)}$

Figure 1. Derivation terms for $\to$ and $\forall$

r to be substituted. The rule $\forall^+x$ is subject to the standard *(Eigen-) variable condition*: The derivation term M of the premise A should not contain any open assumption with x as a free variable. The *induction axioms* are

$$\begin{aligned}
&\mathrm{Ind}_{p,A(p)}\colon \forall_p\big(A(\mathtt{tt}) \to A(\mathtt{ff}) \to A(p^{\mathbf{B}})\big),\\
&\mathrm{Ind}_{n,A(n)}\colon \forall_n\big(A(0) \to \forall_n(A(n) \to A(\mathrm{S}n)) \to A(n^{\mathbf{N}})\big),\\
&\mathrm{Ind}_{v,A(v)}\colon \forall_v\big(A(\mathrm{nil}) \to \forall_{x,v}(A(v) \to A(x :: v)) \to A(v^{\mathbf{L}(\rho)})\big),\\
&\mathrm{Ind}_{x,A(x)}\colon \forall_x\big(\forall_{y^\rho,z^\sigma} A(\langle y, z\rangle) \to A(x^{\rho\times\sigma})\big).
\end{aligned}$$

We show that the fragment of HA^ω with decidable prime formulas is "classical" in the sense that the principle of indirect proof holds. Here we make essential use of the fact that our formulas are built with the connectives $\to$ and $\forall$. Recall that negation $\neg A$ and the weak existential quantifier $\tilde{\exists}_x A$ are definable. In the next section we will (inductively)

define the proper (or "constructive") existential quantifier, which will cause proofs to have computational content. In this richer language the principle of indirect proof does not hold any more.

Lemma (Ex falso quodlibet)**.** $\mathrm{HA}^\omega \vdash F \to A$.

Proof. Induction on A, using boolean induction for atomic formulas. □

The following lemma expresses the *principle of indirect proof*.

Lemma (Stability)**.** $\mathrm{HA}^\omega \vdash \neg\neg A \to A$.

Proof. Induction on A. □

1.3. Inductive constructions

In addition to atomic prime formulas of the form $\mathrm{atom}(r^{\mathbf{B}})$, we want to form initial propositions with inductively defined predicates, each of which is given by its clauses. Rather than introducing them in general, for simplicity we here restrict ourselves to just two examples: list reversal Rev and the existential quantifier.

To define inductively the property of two lists that the second is the reversal of the first, the clauses are

$$\mathrm{InitRev}\colon \mathrm{Rev}(\mathrm{nil}, \mathrm{nil}), \tag{1}$$

$$\mathrm{GenRev}\colon \mathrm{Rev}(v, w) \to \mathrm{Rev}(v :+: x:, x :: w). \tag{2}$$

Recall that we use $x :: v$ for the `cons`-operator, and $v :+: w$ for the append function. $x:$ denotes $x :: \mathrm{nil}$, i.e., the singleton list consisting of x. We will view Rev as a predicate "without computational content". The reader should not be confused here: of course the clauses involving Rev do express how a computation of the reverted list should proceed. However, the predicate Rev itself does not require a witness.

Another particularly important example of an inductively defined predicate is the existential quantifier, which takes a formula $A(x^\rho)$ as parameter. It is given by only one clause:

$$\exists^+\colon \forall_x(A(x^\rho) \to \exists_x A(x^\rho)).$$

This time will view the predicate $\exists_x A(x^\rho)$ as one "with computational content", which intuitively is the pair of the witness x^ρ and the content of the proof of $A(x)$. Proper definitions will be given in the next section.

The intended meaning of an inductively defined predicate is that it should be the *least* one satisfying its clauses. This is expressed by means of an elimination scheme, which in the case of the existential quantifier is

$$\exists^-\colon \exists_x A(x^\rho) \to \forall_x(A(x^\rho) \to C) \to C \qquad \text{with } x \notin \mathrm{FV}(C).$$

2. Realizability

Clearly proper existence proofs have computational content. A well-known and natural way to define this concept is the notion of realizability, which can be seen as an incarnation of the Brouwer-Heyting-Kolmogorov interpretation of proofs.

2.1. Computational content

The concept of "computational content" of a proof only makes sense after we have introduced inductively defined predicates (such as the existential quantifier) to our "negative" language, initially involving $\forall$ and $\to$ only. We first define the *realizability type* $\tau(A)$ of a formula A, and when it is *computationally relevant*. Then we go on and define the formula t *realizes* A, written $t\ \mathbf{r}\ A$, for t of type $\tau(A)$.

Every formula A possibly containing inductively defined predicates can be seen as a "computational problem". We define $\tau(A)$ as the type of a potential realizer of A, i.e., the type of the term (or "program") to be extracted from a proof of A. More precisely, we assign to every formula A an object $\tau(A)$ (a type or the "nulltype" symbol ε). In case $\tau(A) = \varepsilon$ proofs of A have no computational content; such formulas A are called *Harrop formulas*, or computationally *irrelevant* (c.i.). Non-Harrop formulas are also called computationally *relevant* (c.r.). The definition can be conveniently written if we extend the use of $\rho \to \sigma$ to the nulltype symbol: $(\rho \to \varepsilon) := \varepsilon$, $(\varepsilon \to \sigma) := \sigma$, $(\varepsilon \to \varepsilon) := \varepsilon$. With this understanding of $\rho \to \sigma$ we can simply write

$$\tau(\mathrm{atom}(r^{\mathbf{B}})) := \varepsilon, \quad \tau(\mathrm{Rev}(v,w)) := \varepsilon,$$
$$\tau(\exists_x A(x^\rho)) := \rho \times \tau(A),$$
$$\tau(A \to B) := (\tau(A) \to \tau(B)), \quad \tau(\forall_{x^\rho} A) := (\rho \to \tau(A)),$$

Let A be a formula and t either a term of type $\tau(A)$ if the latter is a type, or the nullterm symbol ε if $\tau(A) = \varepsilon$. For a convenient definition we extend the use of term application to the nullterm symbol: $\varepsilon t := \varepsilon$, $t\varepsilon := t$, $\varepsilon\varepsilon := \varepsilon$. We define the formula $t\ \mathbf{r}\ A$, to be read t *realizes* A.

$$\begin{aligned}
t\ \mathbf{r}\ \mathrm{atom}(r) &:= \mathrm{atom}(r),\\
t\ \mathbf{r}\ \mathrm{Rev}(v,w) &:= \mathrm{Rev}(v,w),\\
t\ \mathbf{r}\ \exists_x A(x) &:= t0\ \mathbf{r}\ A(t1),\\
t\ \mathbf{r}\ (A \to B) &:= \forall_x(x\ \mathbf{r}\ A\ \to\ tx\ \mathbf{r}\ B),\\
t\ \mathbf{r}\ (\forall_x A) &:= \forall_x\, tx\ \mathbf{r}\ A.
\end{aligned}$$

Formulas which do not contain $\exists$ play a special role in this context; we call them *negative*. Their crucial property is $(\varepsilon\ \mathbf{r}\ A) = A$. Clearly every formula of the form $t\ \mathbf{r}\ A$ is negative.

We now define the *extracted term* $[\![M]\!]$ of a derivation M. For derivations M^A where $\tau(A) = \varepsilon$ (i.e., A is a Harrop formula) let $[\![M]\!] := \varepsilon$ (the *nullterm* symbol). Now assume

that M derives a formula A with $\tau(A) \neq \varepsilon$. Recall our extended use of term application to the nullterm symbol. We also understand that in case $\tau(A) = \varepsilon$, $\lambda_{x_u^{\tau(A)}}[\![M]\!]$ means just $[\![M]\!]$. Then

$$\begin{aligned}
[\![u^A]\!] &:= x_u^{\tau(A)} \quad (x_u^{\tau(A)} \text{ uniquely associated with } u^A),\\
[\![(\lambda_{u^A} M)^{A\to B}]\!] &:= \lambda_{x_u^{\tau(A)}}[\![M]\!],\\
[\![M^{A\to B} N]\!] &:= [\![M]\!][\![N]\!],\\
[\![(\lambda_{x^\rho} M)^{\forall_x A}]\!] &:= \lambda_{x^\rho}[\![M]\!],\\
[\![M^{\forall_x A} r]\!] &:= [\![M]\!]r.
\end{aligned}$$

We also need to define extracted terms for our axioms: induction axioms and for the existential quantifier the clause $\exists^+$ and its elimination axiom $\exists^-$. For the latter this is rather obvious: For $\exists^-\colon \exists_x A(x^\rho) \to \forall_x(A(x^\rho) \to C) \to C$ we take $[\![\exists^-]\!] := \lambda_p\lambda_f(f(p0)(p1))$, assuming $\tau(A) \neq \varepsilon$; here p has type $\rho \times \tau(A)$. For $\exists^+\colon \forall_x(A(x^\rho) \to \exists_x A(x^\rho))$ let $[\![\exists^+]\!] := \lambda_{x,y}\langle x, y\rangle$. For the induction axioms we take the corresponding recursion operators.

Theorem (Soundness)**.** *Let M be a derivation of a formula A from assumptions $u_i\colon C_i$. Then we can find a derivation of $[\![M]\!]$ $\mathbf{r}$ A from assumptions $\bar{u}_i\colon x_{u_i}$ $\mathbf{r}$ C_i.*

Proof. Induction on M. □

2.2. A constructive proof for list reversal

Let Rev be the graph of the list reversal function. We view Rev as an inductively defined predicate without computational content, given by the clauses (1) and (2) above.

A straightforward proof of $\forall_v\exists_w \mathrm{Rev}(v, w)$ proceeds as follows. We first prove a lemma `ListInitLastNat` stating that every non-empty list can be written in the form $v :+: x:$. Using it, $\forall_v\exists_w \mathrm{Rev}(v, w)$ can be proved by induction on the length of v. In the step case, our list is non-empty, and hence can be written in the form $v :+: x:$. Since v has smaller length, the IH (induction hypothesis) yields its reversal w. Then we can take $x :: w$.

2.3. Extraction from the existence proof for list reversal

Here is the term `neterm` (for "normalized extracted term") extracted from a formalization of this proof, with variable names `f` for unary functions on lists and `p` for pairs of lists and numbers:

```
[x0]
 (Rec nat=>list nat=>list nat)x0([v2](Nil nat))
 ([x2,f3,v4]
   [if v4
```

```
(Nil nat)
([x5,v6][let p7 (cListInitLastNat v6 x5)
           (right p7::f3 left p7)])])
```

where the square brackets in `[x]` is a notation for λ-abstraction λ_x. The term contains the constant `cListInitLastNat` denoting the content of the auxiliary proposition, and in the step the function defined recursively calls itself via `f3`. The underlying algorithm defines an auxiliary function g by

$$\begin{aligned} g(0, v) &:= \text{nil}, \\ g(n+1, \text{nil}) &:= \text{nil}, \\ g(n+1, x :: v) &:= \text{let } w :+: y := x :: v \text{ in } y :: g(n, w) \end{aligned}$$

and gives the result by applying g to $\mathrm{lh}(v)$ and v. It clearly takes quadratic time. To run this algorithm one has to normalize the term obtained by applying `neterm` to the length of a list and the list itself:

```
(animate "ListInitLastNat")
(animate "Id")
(pp (nt (mk-term-in-app-form
          neterm (pt "4") (pt "1::2::3::4:"))))
; 4::3::2::1:
```

We have used here of a mechanism to "animate" or "deanimate" lemmata, or more precisely the constants that denote their computational content. This method can be described generally as follows. Suppose a proof of a theorem uses a lemma. Then the proof term contains just the name of the lemma, say `L`. In the term extracted from this proof we want to preserve the structure of the original proof as much as possible, and hence we use a new constant `cL` at those places where the computational content of the lemma is needed. When we want to execute the program, we have to replace the constant `cL` corresponding to a lemma `L` by the extracted program of its proof. This can be achieved by adding computation rules for `cL`. We can be rather flexible here and enable/block rewriting by using `animate`/`deanimate` as desired. To obtain the `let` expression in the term above, we have used implicitely the "identity lemma" `Id`: $P \to P$; its realizer has the form $\lambda_{f,x}(fx)$. If `Id` is not animated, the extracted term has the form $\texttt{cId}(\lambda_x M)N$, which is printed as $[\texttt{let}\ x\ N\ M]$.

3. Substituting for falsity in weak existence proofs

We now aim at finding computational content in weak existence proofs. First we describe a general method that can be employed here; it is sometimes called "refined A-translation". The difference to the treatment in [3] is that we avoid to work with different formal systems (called Z, Z_0, Z^X and Z_0^X there) and only deal with arithmetic in finite types HA^ω based on minimal logic. Then we apply the method to some examples, in particular to our running example of list reversal.

3.1. The refined A-translation method

It is known that any proof of a specification of the form $\forall_x \tilde{\exists}_y B$ with B quantifier-free and the weak (or "classical") existential quantifier $\tilde{\exists}_y$ can be transformed into a proof of $\forall_x \exists_y B$, now with the constructive existential quantifier $\exists_y$. Here is a simple idea of how to prove this. First recall that our given proof is in *minimal* logic, and therefore does not assume anything about $\bot$. Hence we can replace $\bot$ anywhere in the proof by $\exists_y G$. Then the end formula $\forall_y(G \to \bot) \to \bot$ is turned into $\forall_y(G \to \exists_y G) \to \exists_y G$, and since the premise is trivially provable, we have the claim.

Unfortunately, this simple argument is not quite correct. First, G may contain $\bot$, and hence is changed under the substitution $\bot \mapsto \exists_y G$. Second, we may have used axioms or lemmata involving $\bot$ (e.g., $\bot \to A$), which need not be derivable after the substitution. But in spite of this, the simple idea can be turned into something useful.

Assume that the lemmata $\vec{D}$ and the goal formula G are such that we can derive

$$\vec{D} \to D_i[\bot \mapsto \exists_y G], \tag{3}$$

$$G[\bot \mapsto \exists_y G] \to \exists_y G. \tag{4}$$

Assume also that the substitution $\bot \mapsto \exists_y G$ turns the axioms into instances of the same scheme with different formulas, or else into derivable formulas. Then from our given derivation (in minimal logic) of $\vec{D} \to \forall_y(G \to \bot) \to \bot$ we obtain

$$\vec{D}[\bot \mapsto \exists_y G] \to \forall_y(G[\bot \mapsto \exists_y G] \to \exists_y G) \to \exists_y G.$$

Now (3) allows to drop the substitution in $\vec{D}$, and by (4) the second premise is derivable. Hence we obtain as desired

$$\vec{D} \to \exists_y G.$$

We shall identify classes of formulas – to be called *definite* and *goal* formulas – such that slight generalizations of (3) and (4) hold. This will be done in 3.2. In 3.3 we then prove the general theorem about extraction from classical proofs.

3.2. Definite and goal formulas

A formula is *relevant* if it "ends" with $\bot$. More precisely, relevant formulas are defined inductively by the clauses

- $\bot$ is relevant,
- if C is relevant and B is arbitrary, then $B \to C$ is relevant, and
- if C is relevant, then $\forall_x C$ is relevant.

Clearly we can derive $\bot \to C$ for C relevant. A formula which is not relevant is called *irrelevant*.

We define *goal formulas* G and *definite formulas* D inductively. P ranges over prime formulas (including $\bot$).

$$
\begin{aligned}
G ::= P \mid D \to G \quad & \text{provided } D \text{ irrelevant} \Rightarrow D \text{ quantifier-free} \\
\mid \forall_x G \quad & \text{provided } G \text{ irrelevant,}
\end{aligned}
$$

$$
\begin{aligned}
D ::= P \mid G \to D \quad & \text{provided } D \text{ irrelevant} \Rightarrow G \text{ irrelevant} \\
\mid \forall_x D.
\end{aligned}
$$

Let A^F denote $A[\bot \mapsto F]$.

Lemma. *For definite formulas D and goal formulas G we have derivations from $F \to \bot$ of*

$$((D^F \to F) \to \bot) \to D \quad \textit{for } D \textit{ relevant,} \tag{5}$$

$$D^F \to D, \tag{6}$$

$$G \to G^F \quad \textit{for } G \textit{ irrelevant,} \tag{7}$$

$$G \to (G^F \to \bot) \to \bot. \tag{8}$$

Proof. (5)–(8) can be proved simultaneously, by induction on formulas. □

Lemma. *For goal formulas $\vec{G} = G_1, \dots, G_n$ we have a derivation from $F \to \bot$ of*

$$(\vec{G}^F \to \bot) \to \vec{G} \to \bot. \tag{9}$$

Proof. Assume $F \to \bot$. By (8) we have

$$G_i \to (G_i^F \to \bot) \to \bot$$

for all $i = 1, \dots, n$. Now the assertion follows by minimal logic: Assume $\vec{G}^F \to \bot$ and $\vec{G}$; we must show $\bot$. By $G_1 \to (G_1^F \to \bot) \to \bot$ it suffices to prove $G_1^F \to \bot$. Assume G_1^F. By $G_2 \to (G_2^F \to \bot) \to \bot$ it suffices to prove $G_2^F \to \bot$. Assume G_2^F. Repeating this pattern, we finally have assumptions $G_1^F, \dots, G_n^F$ available, and obtain $\bot$ from $\vec{G}^F \to \bot$. □

3.3. Extraction from weak existence proofs

Theorem (Elimination of $\bot$ from weak existence proofs)**.** *Assume that for arbitrary formulas $\vec{A}$, definite formulas $\vec{D}$ and goal formulas $\vec{G}$ we have a derivation of*

$$\vec{A} \to \vec{D} \to \forall_{\vec{y}}(\vec{G} \to \bot) \to \bot. \tag{10}$$

Then we can also derive

$$(F \to \bot) \to \vec{A} \to \vec{D}^F \to \forall_{\vec{y}}(\vec{G}^F \to \bot) \to \bot.$$

In particular, substitution of the formula

$$\exists_{\vec{y}}\vec{G}^F := \exists_{\vec{y}}(G_1^F \land \cdots \land G_n^F)$$

for $\bot$ *yields*

$$\vec{A}[\bot \mapsto \exists_{\vec{y}}\vec{G}^F] \to \vec{D}^F \to \exists_{\vec{y}}\vec{G}^F. \tag{11}$$

Proof. The first assertion follows from (6) (to infer $\vec{D}$ from $\vec{D}^F$) and (9) (to infer $\vec{G} \to \bot$ from $\vec{G}^F \to \bot$). The second assertion is a simple consequence since $\forall_{\vec{y}}(\vec{G}^F \to \exists_{\vec{y}}\vec{G}^F)$ and $F \to \exists_{\vec{y}}\vec{G}^F$ are both derivable. □

Let M be this derivation of (11). Assume that we have terms $\vec{s}$ and $\vec{t}$ realizing $\vec{A}$ and $\vec{D}$ (with free variables among the parameters of (10)), and derivations of

$$\vec{A} \to \vec{D} \to s_i \mathbf{r} A_i[\bot \mapsto \exists_{\vec{y}}\vec{G}^F(\vec{y})], \qquad \vec{A} \to \vec{D} \to t_j \mathbf{r} D_j^F.$$

Then by the Soundness Theorem for realizability we can derive

$$\vec{A} \to \vec{D} \to [\![M]\!]\vec{s}\vec{t} \mathbf{r} \exists_{\vec{y}}\vec{G}^F(\vec{y})$$

and hence by definition of realizability

$$\vec{A} \to \vec{D} \to [\![M]\!]\vec{s}\vec{t}0 \mathbf{r} \vec{G}^F([\![M]\!]\vec{s}\vec{t}1).$$

In particular, if $\vec{G}^F(\vec{y})$ is a negative Harrop formula, we can derive

$$\vec{A} \to \vec{D} \to \vec{G}^F([\![M]\!]\vec{s}\vec{t}).$$

3.4. Example: Fibonacci numbers

Let α_n be the n-th Fibonacci number, i.e.,

$$\alpha_0 := 0, \quad \alpha_1 := 1, \quad \alpha_n := \alpha_{n-2} + \alpha_{n-1} \quad \text{for } n \geq 2.$$

We give a weak existence proof for the Fibonacci numbers:

$$\forall_n \tilde{\exists}_k G(n,k), \quad \text{i.e.,} \quad \forall_k(G(n,k) \to \bot) \to \bot$$

from clauses expressing that G is the graph of the Fibonacci function:

$$v_0 \colon G(0,0),\ v_1 \colon G(1,1),\ v_2 \colon \forall_{n,k,l}\big(G(n,k) \to G(n+1,l) \to G(n+2,k+l)\big).$$

Clearly the clause formulas are definite and $G(n,k)$ is a goal formula. To construct a derivation, assume further

$$u\colon \forall_k(G(n,k) \to \bot).$$

Our goal is $\bot$. To this end we first prove a strengthened claim in order to get the induction through:

$$\forall_n B(n) \quad \text{with } B(n) := \forall_{k,l}\big(G(n,k) \to G(n+1,l) \to \bot\big) \to \bot.$$

This is proved by induction on n. The base case follows from the first two clauses. In the step case we can assume that we have k, l satisfying $G(n,k)$ and $G(n+1,l)$. We need k', l' such that $G(n+1,k')$ and $G(n+2,l')$. Using the third clause simply take $k' := l$ and $l' := k + l$. – To obtain our goal $\bot$ from $\forall_n B$, it clearly suffices to prove its premise $\forall_{k,l}(G(n,k) \to G(n+1,l) \to \bot)$. So let k, l be given and assume $u_1\colon G(n,k)$ and $u_2\colon G(n+1,l)$. Then u applied to k and u_1 gives our goal $\bot$.

The derivation term is

$$M = \lambda_n \lambda_u^{\forall_k(G(n,k)\to\bot)}.$$
$$\mathrm{Ind}_{n,B} n M_{\mathrm{base}} M_{\mathrm{step}} (\lambda_{k,l} \lambda_{u_1}^{G(n,k)} \lambda_{u_2}^{G(n+1,l)}. u k u_1)$$

where

$$M_{\mathrm{base}} = \lambda_{w_0}^{\forall_{k,l}(G(0,k)\to G(1,l)\to\bot)}. w_0 0 1 v_0 v_1$$
$$M_{\mathrm{step}} = \lambda_n \lambda_w^B \lambda_{w_1}^{\forall_{k,l}(G(n+1,k)\to G(n+2,l)\to\bot)}.$$
$$w(\lambda_{k,l} \lambda_{u_3}^{G(n,k)} \lambda_{u_4}^{G(n+1,l)}. w_1 l (k+l) u_4 (v_2 n k l u_3 u_4)).$$

Indeed, one can interactively generate this proof in the Minlog proof assistant and print its lambda-term (which does not show the formulas involved) by `proof-to-expr`. The result (after renaming bound variables) is given in Figure 2.

As described in the proof of the theorem above, we now can substitute $\bot$ by $\exists_k G(n,k)$ and obtain a proper existence proof $M^\exists$, named (11) above. Therefore

$$[\![M^\exists]\!] = \lambda_n . \mathcal{R}_{\mathbf{N}}^{(\mathbf{N}\to\mathbf{N}\to\mathbf{N})\to\mathbf{N}} n [\![M^\exists_{\mathrm{base}}]\!] [\![M^\exists_{\mathrm{step}}]\!] (\lambda_{k,l} k)$$

where

$$[\![M^\exists_{\mathrm{base}}]\!] = \lambda_{w_0}^{\mathbf{N}\to\mathbf{N}\to\mathbf{N}}. w_0 0 1$$
$$[\![M^\exists_{\mathrm{step}}]\!] = \lambda_n \lambda_w^{(\mathbf{N}\to\mathbf{N}\to\mathbf{N})\to\mathbf{N}} \lambda_{w_1}^{\mathbf{N}\to\mathbf{N}\to\mathbf{N}}. w(\lambda_k \lambda_l . w_1 l (k+l))$$

The term machine extracted from this proof is almost literally the same:

```
[n0]
 (Rec nat=>(nat=>nat=>nat)=>nat)n0([f1]f1 0 1)
 ([n1,H2,f3]H2([n4,n5]f3 n5(n4+n5)))
 ([n1,n2]n1)
```

```
(lambda (m)
  (lambda (u)
    (((((|Ind| m)
        (lambda (w0) ((((w0 0) 1) |Intro|) |Intro|)))
       (lambda (n)
         (lambda (w)
           (lambda (w1)
             (w
               (lambda (k)
                 (lambda (l)
                   (lambda (u3)
                     (lambda (u4)
                       ((((w1 l) (+ k l)) u4)
                         (((((|Intro| n) k) l)
                            u3) u4)))))))))))
      (lambda (k)
        (lambda (l)
          (lambda (u1) (lambda (u2) ((u k) u1))))))))
```

Figure 2. Expression for the Fibonacci proof

with `H` a name for variables of type `(nat=>nat=>nat)=>nat` and `f` of type `nat=>nat=>nat`. The underlying algorithm defines an auxiliary functional G by

$$G(0,f) := f(0,1), \qquad G(n+1,f) := G(n, \lambda_{k,l} f(l, k+l))$$

and gives the result by applying G to the original number and the first projection $\lambda_{k,l} k$. This is a linear algorithm in tail recursive form. It is somewhat unexpected since it passes functions (rather than pairs, as one would ordinarily do), and hence uses functional programming in a proper way. This clearly is related to the use of classical logic, which by its use of double negations has a functional flavour.

3.5. A weak existence proof for list reversal

Assuming (1) and (2) we now prove

$$\forall_v \tilde{\exists}_w \mathrm{Rev}(v,w) \qquad (:= \forall_v \big(\forall_w (\mathrm{Rev}(v,w) \to \bot) \to \bot\big)). \tag{12}$$

Fix v and assume $u\colon \forall_w \neg \mathrm{Rev}(v,w)$; we need to derive a contradiction. To this end we prove that all initial segments of v are non-revertible, which contradicts (1). More precisely, from u and (2) we prove

$$\forall_{v_2} A(v_2) \quad \text{with } A(v_2) := \forall_{v_1} \big(v_1 :+: v_2 = v \to \forall_w \neg \mathrm{Rev}(v_1, w)\big)$$

by induction on v_2. For $v_2 = \mathrm{nil}$ this follows from our initial assumption u. For the step case, assume $v_1 :+: (x :: v_2) = v$, fix w and assume further $\mathrm{Rev}(v_1, w)$. We need to

derive a contradiction. Properties of the append function imply that $(v_1{:}+{:}x{:}){:}+{:}v_2 = v$. The IH for $v_1{:}+{:}x{:}$ gives $\forall_w \neg \mathrm{Rev}(v_1{:}+{:}x{:}, w)$. Now (2) yields the desired contradiction.

We formalize this proof, to see what the result of its A-translation is. For readability we write vw for the result $v :+: w$ of appending the list w to the list v, vx for the result $v :+: x{:}$ of appending the one element list $x{:}$ to the list v, and xv for the result $x :: v$ of constructing a list by writing an element x in front of a list v. The following lemmata will be used.

$$\begin{aligned}
&\mathrm{Compat}\colon \forall_P \forall_{v_1,v_2}\big(v_1 = v_2 \to P(v_1) \to P(v_2)\big),\\
&\mathrm{Symm}\colon \quad \forall_{v_1,v_2}\big(v_1 = v_2 \to v_2 = v_1\big),\\
&\mathrm{Trans}\colon \quad \forall_{v_1,v_2,v_3}\big(v_1 = v_2 \to v_2 = v_3 \to v_1 = v_3\big),\\
&L_1\colon \qquad \forall_v\, v = v\,\mathrm{nil},\\
&L_2\colon \qquad \forall_{v_1,x,v_2}\,(v_1 x)v_2 = v_1(xv_2),
\end{aligned}$$

The proof term is

$$M := \lambda_v \lambda_u^{\forall_w \neg \mathrm{Rev}(v,w)}.\mathrm{Ind}_{v_2,A(v_2)} v v M_{\mathrm{Base}} M_{\mathrm{Step}}\ \mathrm{nil}\ \mathrm{T}^{\mathrm{nil}\, v=v}\ \mathrm{nil}\ \mathrm{InitRev}$$

with

$$\begin{aligned}
M_{\mathrm{Base}} &:= \lambda_{v_1} \lambda_{u_1}^{v_1 \mathrm{nil}=v}.\mathrm{Compat}\,\{\, v \mid \forall_w \neg \mathrm{Rev}(v,w)\,\}\, v v_1\\
&\qquad (\mathrm{Symm}\, v_1 v(\mathrm{Trans}\, v_1(v_1\,\mathrm{nil})v(L_1 v_1)u_1))u,\\
M_{\mathrm{Step}} &:= \lambda_{x,v_2} \lambda_{u_{\mathrm{IH}}}^{A(v_2)} \lambda_{v_1} \lambda_{u_1}^{v_1(xv_2)=v} \lambda_w \lambda_{u_2}^{\mathrm{Rev}(v_1,w)}.\\
&\qquad u_{\mathrm{IH}}(v_1 x)(\mathrm{Trans}\,((v_1 x)v_2)(v_1(xv_2))v(L_2 v_1 x v_2)u_1)\\
&\qquad (xw)\big(\mathrm{GenRev}\, v_1 w x u_2\big).
\end{aligned}$$

Again one can interactively generate this proof in Minlog and print its lambda-term by `proof-to-expr`. The result (after renaming bound variables and writing `LA` for `ListAppend`) is given in Figure 3.

We now have a proof M of $\forall_v \tilde{\exists}_w \mathrm{Rev}(v,w)$ from the clauses $\mathrm{InitRev}\colon D_1$ and $\mathrm{GenRev}\colon D_2$. Both are definite formulas without $\bot$. Also $\mathrm{Rev}(v,w) =: G$ is a goal formula not containing $\bot$. Hence D_i^F is D_i and G^F is G. Moreover D_1, D_2 and G are negative Harrop formulas. Therefore $\vec{t}$ is empty here and for the extracted term $[\![M^{\exists}]\!]$ of the derivation $M^{\exists}$ of $D_1 \to D_2 \to \exists_w \mathrm{Rev}(v,w)$ (obtained from M by substituting $\exists_w \mathrm{Rev}(v,w)$ for $\bot$) we can derive $D_1 \to D_2 \to \mathrm{Rev}(v, [\![M^{\exists}]\!])$. The term `neterm` machine extracted from a formalization of the proof above is (after "animating" `Compat`)

```
[v0]
 (Rec list nat=>list nat=>list nat=>list nat)v0([v1,v2]v2)
 ([x1,v2,g3,v4,v5]g3(v4:+:x1:)(x1::v5))
 (Nil nat)
 (Nil nat)
```

```
(lambda (v0)
  (lambda (u)
    (((((((((|Ind| v0) v0)
           (lambda (v1)
             (lambda (u1)
               (((((|Compat| v0) v1)
                  ((((|Symm| v1) v0)
                    ((((((|Trans| v1) (((|LA| v1) '())) v0)
                       (|LOne| v1))
                     u1)))
                 u))))
          (lambda (x)
            (lambda (v2)
              (lambda (uIH)
                (lambda (v1)
                  (lambda (u1)
                    (lambda (w)
                      (lambda (u2)
                        ((((uIH (((|LA| v1) (cons x '())))
                            ((((((|Trans|
                                  (((|LA|
                                     (((|LA| v1) (cons x '())))
                                    v2))
                                  (((|LA| v1) (cons x v2)))
                                 v0)
                                ((((|LTwo| v1) x) v2))
                               u1))
                             (cons x w))
                            (((((|Intro| v1) w) x) u2)))))))))))
         '())
        |Truth-Axiom|)
       '())
      |Intro|))))
```

Figure 3. Expression for the list reversal proof

with `g` a variable for binary functions on lists. To run this algorithm one has to normalize the term obtained by applying `neterm` to a list:

```
(pp (nt (mk-term-in-app-form neterm (pt "1::2::3::4:"))))
; 4::3::2::1:
```

In fact, the underlying algorithm defines an auxiliary function h by

$$h(\mathrm{nil}, v_2, v_3) := v_3, \qquad h(x :: v_1, v_2, v_3) := h(v_1, v_2 :+: x:, x :: v_3)$$

and gives the result by applying h to the original list and twice nil.

Notice that the second argument of h is not needed. However, its presence makes the algorithm quadratic rather than linear, because in each recursion step $v_2 :+: x:$ is computed, and the list append function $:+:$ is defined by recursion over its first argument. We will be able to get rid of this superfluous second argument by redoing the proof, this time taking "uniformity" into account.

4. Uniformity

Recall that in the weak existence proof for reverted lists in 3.5 we have made use of an auxiliary proposition

$$\forall_{v_2} A(v_2) \quad \text{with } A(v_2) := \forall_{v_1}\big(v_1 :+: v_2 = v \to \forall_w \neg \mathrm{Rev}(v_1, w)\big).$$

It turns out that its proof (by induction on v_2) "does not use v_1 computationally", and hence that we can replace $\forall_{v_1}$ by a "uniform quantifier" $\forall^{\mathsf{U}}_{v_1}$. This will lead to a better algorithm. We first explain the notions involved.

4.1. Uniform proofs

We extend the definition of the extracted term of a derivation to the case where our formulas may involve the uniform universal quantifier $\forall^{\mathsf{U}}$. Using this concept we define the notion of a uniform proof, which gives a special treatment to $\forall^{\mathsf{U}}$. More precisely, for a derivation M, we now simultaneously define

- its *extracted term* $[\![M]\!]$ of type $\tau(A)$, and
- when M is *uniform*.

For derivations M^A where $\tau(A) = \varepsilon$ (i.e., A is a Harrop formula) let $[\![M]\!] := \varepsilon$ (the *null-term* symbol); every such derivation is uniform. Now assume that M derives a formula A with $\tau(A) \neq \varepsilon$. We extend the definition in 2.1 by

$$[\![(\lambda_{x^\rho} M)^{\forall^{\mathsf{U}}_x A}]\!] := [\![M^{\forall^{\mathsf{U}}_x A} r]\!] := [\![M]\!].$$

In all the rules uniformity is preserved, except possibly in the introduction rule for the uniform universal quantifier: $(\lambda_{x^\rho} M)^{\forall^{\mathsf{U}}_x A}$ is uniform if M is and – in addition to the usual variable condition – $x \notin \mathrm{FV}([\![M]\!])$.

Remark. It may happen that a uniform proof has non-uniform subproofs: there are no restrictions concerning $\forall^{\mathsf{U}}$ in subproofs ending with a Harrop formula: all such subproofs are uniform by definition.

4.2. Introducing uniformities

To apply the concept of uniformity to our present proof for list reversal, we certainly need to know which occurrences of universal quantifiers can be made uniform.

Clearly this cannot be done everywhere. For instance, the induction axiom

$$\mathrm{Ind}_{v,A(v)} \colon \forall_v \big(A(\mathrm{nil}) \to \forall_{x,v}(A(v) \to A(x :: v)) \to A(v^{\mathbf{L}(\mathbf{N})})\big)$$

requires non-uniform universal quantifiers $\forall_v$ and $\forall_{x,v}$, because otherwise the Soundness Theorem would not hold. For the same reason the existence introduction and elimination axioms

$$\exists^+ \colon \forall_x(A(x^\rho) \to \exists_x A(x^\rho)),$$
$$\exists^- \colon \exists_x A(x^\rho) \to \forall_x(A(x^\rho) \to C) \to C \qquad \text{with } x \notin \mathrm{FV}(C)$$

need non-uniform quantifiers $\forall_x$.

Having identified some $\forall$-occurrences as non-uniform, we need to propagate this information through the entire proof. Here we follow an idea dating back to Gentzen [6], about occurrences of formulas in a proof being connected ("verbunden"). However, in our case we will need to define when two occurrences of a universal quantifier in a proof are connected. The definition is an inductive one, with the following clauses:

- In an axiom $\forall_P A(P)$ whose predicate variable P is substituted with a formula B, every occurrence of a universal quantifier in B is connected with all its copies in the (possibly many) occurrences of P in $A(P)$.
- In an application of the $\to^+$ rule

$$\dfrac{\begin{array}{c}[u\colon A]\\ \mid M\\ B\end{array}}{A \to B}\,{\to^+}u$$

  each occurrence of a universal quantifier in the assumption $u\colon A$ is connected with the corresponding occurrence of this quantifier in premise A of the end formula $A \to B$. Moreover, each occurrence of $\forall$ in the conclusion B of the end formula $A \to B$ is connected with the corresponding occurrence in the end formula B of M.
- In an application of the $\to^-$ rule

$$\dfrac{\begin{array}{cc}\mid M & \mid N\\ A \to B & A\end{array}}{B}\,{\to^-}$$

  each occurrence of a universal quantifier in the premise A of the end formula $A \to B$ of M is connected with the corresponding occurrence in the end formula A of N. Moreover, each occurrence of $\forall$ in the conclusion B of the end formula

$A \to B$ of M is connected with the corresponding occurrence in the end formula B.

For the rules $\forall^+$ and $\forall^-$ the clauses are similar (and even more straightforward).

Now the first step in introducing uniformities into a proof is to mark as non-uniform all occurrences of $\forall$ which are connected to a non-uniform occurrence of a universal quantifier in an induction axiom or an $\exists^+$ or $\exists^-$ axiom.

The next step consists in considering the remaining occurrences of $\forall$ in formulas of the proof, and to group them in equivalence classes w.r.t. the connection relation. Then we look at a topmost occurrence of a non-marked $\forall$, and try to make it and all others connected with it uniform. After having done this, we check whether the resulting proof is uniform, in the sense of the definition in 4.1. If it is, we keep these uniformities; if not, we change all $\forall$'s connected with our topmost one back to non-uniform ones. This procedure can be iterated until all $\forall$'s in the proof have been considered.

4.3. List reversal with uniformities

We now apply this general method of introducing uniformities to the present case of list reversal. To this end we describe our proof in more detail, particularly by writing proof trees with formulas. Let $\neg^\exists A := A \to \exists_w \mathrm{Rev}(v,w)$. $M^\exists_{\mathrm{Base}}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\mathrm{Compat} \quad \{\, v \mid \forall_w \neg^\exists \mathrm{Rev}(v,w)\,\} \quad v \quad v_1}{v{=}v_1 \to \forall_w \neg^\exists \mathrm{Rev}(v,w) \to \forall_w \neg^\exists \mathrm{Rev}(v_1,w)} \quad \begin{array}{r} \mid N \\ v{=}v_1 \end{array}}{\forall_w \neg^\exists \mathrm{Rev}(v,w) \to \forall_w \neg^\exists \mathrm{Rev}(v_1,w)} \qquad \exists^+ \colon \forall_w \neg^\exists \mathrm{Rev}(v,w)}{\dfrac{\dfrac{\forall_w \neg^\exists \mathrm{Rev}(v_1,w)}{v_1\,\mathrm{nil} = v \to \forall_w \neg^\exists \mathrm{Rev}(v_1,w)} \to^+ u_1}{\forall_{v_1}\big(v_1\,\mathrm{nil} = v \to \forall_w \neg^\exists \mathrm{Rev}(v_1,w)\big) \quad (= A(\mathrm{nil}))}}$$

where N is a derivation involving L_1 with a free assumption $u_1 \colon v_1\,\mathrm{nil}{=}v$.

$M^\exists_{\mathrm{Step}}$ is the derivation in Figure 4, where N_1 is a derivation involving L_2 with free assumption $u_1 \colon v_1(xv_2){=}v$, and N_2 is one involving GenRev with the free assumption $u_2 \colon \mathrm{Rev}(v_1,w)$.

All quantifiers $\forall_w$ are connected to the $\forall_w$ in $\exists^+$ (in $M^\exists_{\mathrm{Base}}$) and hence need to be non-uniform. Also the quantifiers $\forall_x$ and $\forall_{v_2}$ in the end formula of $M^\exists_{\mathrm{Step}}$ must be non-uniform. However, *all* occurrences of the universal quantifier $\forall_{v_1}$ can be marked as uniform.

4.4. Extraction

The extracted term `neterm` then is

```
[v0]
 (Rec list nat=>list nat=>list nat)v0([v1]v1)
 ([x1,v2,f3,v4]f3(x1::v4))
 (Nil nat)
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{u_0\colon A(v_2)\quad v_1x}{(v_1x)v_2{=}v \to \forall_w\neg^{\exists}\mathrm{Rev}(v_1x,w)}\quad (v_1x)v_2{=}v}{\forall_w\neg^{\exists}\mathrm{Rev}(v_1x,w)}\quad xw}{\neg^{\exists}\mathrm{Rev}(v_1x,xw)}\quad \mid N_1}{\mathrm{Rev}(v_1x,xw)}}{\exists_w\mathrm{Rev}(v,w)}}{\neg^{\exists}\mathrm{Rev}(v_1,w)}\to^+u_2}{\forall_w\neg^{\exists}\mathrm{Rev}(v_1,w)}}{v_1(xv_2)=v\to\forall_w\neg^{\exists}\mathrm{Rev}(v_1,w)}\to^+u_1}{\forall_{v_1}\big(v_1(xv_2)=v\to\forall_w\neg^{\exists}\mathrm{Rev}(v_1,w)\big)\quad (=A(xv_2))}}{A(v_2)\to A(xv_2)}\to^+u_0}{\forall_{x,v_2}(A(v_2)\to A(xv_2))}$$

Figure 4. The step derivation with uniformity

with `f` a variable for unary functions on lists. Again, to run this algorithm one has to normalize the term obtained by applying `neterm` to a list:

```
(pp (nt (mk-term-in-app-form neterm (pt "1::2::3::4:"))))
; 4::3::2::1:
```

This time, the underlying algorithm defines an auxiliary function g by

$$g(\mathrm{nil},w) := w, \qquad g(x :: v,w) := g(v,x :: w)$$

and gives the result by applying g to the original list and nil. So we have obtained (by automated extraction from a weak existence proof involving uniformity) the standard linear algorithm for list reversal, with its use of an accumulator.

References

[1] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, Berlin, Heidelberg, New York, 1993.

[2] Ulrich Berger. Uniform Heyting Arithmetic. *Annals Pure Applied Logic*, 133:125–148, 2005.

[3] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

[4] Albert Dragalin. New kinds of realizability. In *Abstracts of the 6th International Congress of Logic, Methodology and Philosophy of Sciences*, pages 20–24, Hannover, Germany, 1979.

[5] Harvey Friedman. Classically and intuitionistically provably recursive functions. In D.S. Scott and G.H. Müller, editors, *Higher Set Theory*, volume 669 of *Lecture Notes in Mathematics*, pages 21–28. Springer Verlag, Berlin, Heidelberg, New York, 1978.

[6] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

[7] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

This page intentionally left blank

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Proof Theory, Large Functions and Combinatorics

Stanley S. Wainer
School of Mathematics, University of Leeds, UK

Abstract. These lecture notes show how appropriate finitary versions of Ramsey's Theorem provide natural measures of the mathematical and computational strength of certain basic arithmetical theories.

Keywords. Proof theory, Peano arithmetic, cut elimination, ordinal analysis, provably recursive functions, fast growing hierarchies, Ramsey's theorem.

1. Introduction – Ramsey's Theorem

These lecture notes bring together, in detail, some well–established results by various authors (the present one included) in a central area of mathematical logic where proofs and computations occur as opposite sides of the same coin. The emphasis here is on the analysis of proof–theoretic complexity (though some of the combinatorial results were first obtained by different methods) and the hope is that our treatment will serve as a further illustration of the fundamental nature of proof theory as a unifying tool in mathematical logic and computer science.

The proofs of certain well known, basic theorems of mathematics demand the existence, or prior computation, of "large" numerical functions. Proof theory, on the other hand, provides complexity, and rate-of-growth classifications of the functions "provably computable" or "provably recursive" in given formal theories basic to the foundations of mathematics. Thus if a mathematical theorem is expressible in the language of a given theory T but necessitates (implies) the existence of computable functions whose rates of growth exceed those provably computable in it, then that theorem is "independent" of T, i.e. not provable. Such independence results serve to measure the mathematical power of the theories in question, and since the famous result of Paris–Harrington [8] published in 1977 (and treated below) a whole industry of "Reverse Mathematics" has developed, see Simpson [11]. A rich source of examples is finite combinatorics, particularly "Ramsey Theory", see Graham, Rothschild and Spencer [4], and in this paper we carry through the proof theoretical analysis of two fundamental first-order theories $\mathrm{I}\Delta_0(\exp)$ and PA, to show how the Finite Ramsey Theorem and, respectively, the Paris–Harrington modification of it, yield independence results.

Ramsey's Theorem [9] (1930) has a wide variety of finite and infinite versions. For infinite sets it says that for every positive integer n, each finite partitioning (or "colouring") of the n-element subsets of an infinite set X has an infinite homogeneous or

"monochromatic" subset $Y \subset X$, meaning all n-element subsets of Y have the same colour (lie in the same partition).

The Finite Ramsey Theorem is usually stated as:

$$\forall_{n,k,l}\, \exists_m\, (m \to (k)^n_l)$$

where, letting $m^{[n]}$ denote the collection of all n-element subsets of $m = \{0, \ldots, m-1\}$, $m \to (k)^n_l$ means that for every function (colouring) $c : m^{[n]} \to l$ there is a subset $Y \subset m$ of cardinality at least k, which is homogeneous for c, i.e. c is constant on the n-element subsets of Y.

Whereas by Jockusch [5] the Infinite Ramsey Theorem (with n varying) is not arithmetically expressible (even by restricting to recursive partitions), the Finite Ramsey Theorem clearly is. For by standard coding, the relation $m \to (k)^n_l$ is easily seen to be elementary recursive (i.e. its characteristic function is definable from addition and truncated subtraction by bounded sums and products) so it is expressible as a formula of arithmetic with bounded quantifiers only. The Finite Ramsey Theorem therefore asserts the existence of a recursive function which computes the least such m from n, k, l. This function is known to have superexponential growth-rate, see [4], so it is primitive recursive but not elementary, because every elementary function is bounded by an iterated exponential of fixed (not varying) stack–height. We show next that the provably recursive functions of $\mathrm{I}\Delta_0(\exp)$ are elementary. Thus the Finite Ramsey Theorem is independent of $\mathrm{I}\Delta_0(\exp)$ though it is provable in the Σ_1–inductive fragment of Peano Arithmetic. Later, we characterize the provably recursive functions of PA itself, and thereby show that the Modified Finite Ramsey Theorem of Paris and Harrington has growth–rate beyond even PA!

2. Basic Arithmetic in $\mathrm{I}\Delta_0(\exp)$

$\mathrm{I}\Delta_0(\exp)$ is a theory in classical logic, based on the language $\{=, 0, S, P, +, \dot{-}, \cdot, \exp_2\}$ where S, P denote the successor and predecessor functions. We shall generally use infix notations $x+1$, $x \dot{-} 1$, 2^x rather than the more formal $S(x)$, $P(x)$, $\exp_2(x)$ etcetera. The axioms of $\mathrm{I}\Delta_0(\exp)$ are the usual axioms for equality, the following defining axioms for the constants:

$$\begin{array}{ll} x+1 \neq 0 & x+1 = y+1 \to x = y \\ 0 \dot{-} 1 = 0 & (x+1) \dot{-} 1 = x \\ x+0 = x & x+(y+1) = (x+y)+1 \\ x \dot{-} 0 = x & x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 \\ x \cdot 0 = 0 & x \cdot (y+1) = (x \cdot y) + x \\ 2^0 = 0+1 & 2^{x+1} = 2^x + 2^x \end{array}$$

and the axiom-scheme of "bounded induction":

$$B(0) \wedge \forall_x (B(x) \to B(x+1)) \to \forall_x B(x)$$

for all "bounded" formulas B as defined below.

Definition 2.1 We write $t_1 \leq t_2$ for $t_1 \dot{-} t_2 = 0$ and $t_1 < t_2$ for $t_1 + 1 \leq t_2$, where t_1, t_2 denote arbitrary terms of the language.

A Δ_0- or *bounded* formula is a formula in the langage of $\mathrm{I}\Delta_0(\exp)$, in which all quantifiers occur bounded, thus $\forall_{x<t}\, B(x)$ stands for $\forall x(x < t \to B(x))$ and $\exists_{x<t}\, B(x)$ stands for $\exists x(x < t \wedge B(x))$ (similarly with $\leq$ instead of $<$).

A Σ_1-*formula* is any formula of the form $\exists_{x_1}\exists_{x_2}\ldots\exists_{x_k} B$ where B is a bounded formula. The prefix of unbounded existential quantifiers is allowed to be empty, thus bounded formulas are Σ_1.

The first task in any axiomatic theory is to develop, from the axioms, those basic algebraic properties which are going to be used frequently without further reference. Thus, in the case of $\mathrm{I}\Delta_0(\exp)$ we need to establish the usual associativity, commutativity and distributivity laws for addition and multiplication, the laws of exponentiation, and rules governing the relations $\leq$ and $<$ just defined. These are mostly tedious and straightforward to prove, using inductions on quantifier–free formulas.

Lemma 2.2 *In* $\mathrm{I}\Delta_0(\exp)$ *one can prove (the universal closures of)*
the associativity law for addition: $x + (y + z) = (x + y) + z$
the associativity law for multiplication: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
the distributivity law: $x \cdot (y + z) = x \cdot y + x \cdot z$
the commutativity laws: $x + y = y + x$ *and* $x \cdot y = y \cdot x$
the law: $x \dot{-} (y + z) = (x \dot{-} y) \dot{-} z$
case-distinction: $x = 0 \vee x = (x \dot{-} 1) + 1$
and the exponentiation law: $2^{x+y} = 2^x \cdot 2^y$.

Lemma 2.3 *The following (and their universal closures) are provable in* $\mathrm{I}\Delta_0(\exp)$*:*

1. $x \leq 0 \leftrightarrow x = 0$ *and* $\neg\, x < 0$
2. $0 \leq x$ *and* $x \leq x$ *and* $x < x + 1$
3. $x < y + 1 \leftrightarrow x \leq y$
4. $x \leq y \leftrightarrow x < y \vee x = y$
5. $x \leq y \wedge y \leq z \to x \leq z$ *and* $x < y \wedge y < z \to x < z$
6. $x \leq y \vee y < x$
7. $x < y \to x + z < y + z$
8. $x < y \to x \cdot (z + 1) < y \cdot (z + 1)$
9. $x < 2^x$ *and* $x < y \to 2^x < 2^y$.

Of course in any theory many new functions and relations can be defined out of the given constants. What we are interested in are those which can not only be *defined* in the language of the theory, but also can be *proven to exist*. This gives rise to one of the main definitions in this paper.

Definition 2.4 We say that a function $f\colon \mathbb{N}^k \to \mathbb{N}$ is *provably* Σ_1 *in an arithmetical theory* T if there is a Σ_1-formula $F(\vec{x}, y)$, called a "defining formula" for f, such that

- $f(\vec{n}) = m$ if and only if $F(\vec{n}, m)$ is true (in the standard model)
- $T \vdash \exists_y F(\vec{x}, y)$
- $T \vdash F(\vec{x}, y) \wedge F(\vec{x}, y') \to y = y'$

Since Σ_1-definable functions are recursive, we shall often use the terms "provably Σ_1" and "provably recursive" synonymously.

If, in addition, F is a bounded formula and there is a bounding term $t(\vec{x})$ for f such that $T \vdash F(\vec{x}, y) \rightarrow y < t(\vec{x})$ then we say that f is *provably bounded in T*. In this case we clearly have $T \vdash \exists_{y<t(\vec{x})} F(\vec{x}, y)$.

The importance of this definition is brought out by the following:

Theorem 2.5 *If f is provably Σ_1 in T we may conservatively extend T by adding a new function symbol for f together with the defining axiom $F(\vec{x}, f(\vec{x}))$.*

Proof. This is simply because any model $\mathcal{M}$ of T can be made into a model $(\mathcal{M}, f)$ of the extended theory, by interpreting f as the function on $\mathcal{M}$ uniquely determined by the second and third conditions above. So if A is a closed formula not involving f, provable in the extended theory, then it is true in $(\mathcal{M}, f)$ and hence true in $\mathcal{M}$. Then by Completeness, A must already be provable in T.

We next develop the stock of functions provably Σ_1 in $\mathrm{I}\Delta_0(\exp)$, and prove that they are exactly the elementary functions.

Lemma 2.6 *Each term defines a provably bounded function of* $\mathrm{I}\Delta_0(\exp)$.

Proof. Let f be the function defined explicitly by $f(\vec{n}) = t(\vec{n})$ where t is any term of $\mathrm{I}\Delta_0(\exp)$. Then we may take $y = t(\vec{x})$ as the defining formula for f, since $\exists y\,(y = t(\vec{x}))$ derives immediately from the axiom $t(\vec{x}) = t(\vec{x})$, and $y = t(\vec{x}) \wedge y' = t(\vec{x}) \rightarrow y = y'$ is an equality axiom. Furthermore, as $y = t(\vec{x})$ is a bounded formula and $y = t \rightarrow y < t + 1$ is provable, f is provably bounded.

Lemma 2.7 *Define $2_k(x)$ by $2_0(x) = x$ and $2_{k+1}(x) = 2^{2_k(x)}$. Then for every term $t(x_1, \ldots, x_n)$ built up from the constants $0, S, P, +, \dot{-}, \cdot, \exp_2$, there is a k such that*

$$\mathrm{I}\Delta_0(\exp) \vdash t(x_1, \ldots, x_n) < 2_k(x_1 + \ldots + x_n).$$

Proof. We can prove in $\mathrm{I}\Delta_0(\exp)$ both $0 < 2^x$ and $x < 2^x$. Now suppose t is any term constructed from subterms t_0, t_1 by application of one of the function constants. Assume inductively that $t_0 < 2_{k_0}(s_0)$ and $t_1 < 2_{k_1}(s_1)$ are both provable, where s_0, s_1 are the sums of all variables appearing in t_0, t_1 respectively. Let s be the sum of all variables appearing in either t_0 or t_1, and let k be the maximum of k_0 and k_1. Then, by the various arithmetical laws in the preceeding lemmas, we can prove $t_0 < 2_k(s)$ and $t_1 < 2_k(s)$, and it is then a simple matter to prove $t_0 + 1 < 2_{k+1}(s)$, $t_0 \dot{-} 1 < 2_k(s)$, $t_0 \dot{-} t_1 < 2_k(s)$, $t_0 + t_1 < 2_{k+1}(s)$, $t_0 \cdot t_1 < 2_{k+1}(s)$ and $2^{t_0} < 2_{k+1}(s)$. Hence $\mathrm{I}\Delta_0(\exp)$ proves $t < 2_{k+1}(s)$.

Lemma 2.8 *Suppose f is defined by composition*

$$f(\vec{n}) = g_0(\, g_1(\vec{n}), \ldots, g_m(\vec{n})\,)$$

from functions $g_0, g_1, \ldots, g_m$, each of which is provably bounded in $\mathrm{I}\Delta_0(\exp)$. *Then f is provably bounded in* $\mathrm{I}\Delta_0(\exp)$.

Proof. By the definition of "provably bounded" there is, for each g_i ($i \leq m$) a bounded defining formula G_i and (by the last lemma) a number k_i such that, for $1 \leq i \leq m$, $\mathrm{I}\Delta_0(\exp) \vdash \exists_{y_i<2_{k_i}(s)} G_i(\vec{x}, y_i)$, where s is the sum of the variables $\vec{x}$; and for $i = 0$,

$$\mathrm{I}\Delta_0(\exp) \vdash \exists_{y<2_{k_0}(s_0)} G_0(y_1,\ldots,y_m,y)$$

where s_0 is the sum of the variables $y_1,\ldots,y_m$. Let $k := \max(k_0,k_1,\ldots,k_m)$ and let $F(\vec{x},y)$ be the bounded formula

$$\exists_{y_1<2_k(s)} \ldots \exists_{y_m<2_k(s)} C(\vec{x},y_1,\ldots,y_m,y)$$

where $C(\vec{x},y_1,\ldots,y_m,y)$ is the conjunction

$$G_1(\vec{x},y_1) \wedge \ldots \wedge G_m(\vec{x},y_m) \wedge G_0(y_1,\ldots,y_m,y).$$

Then clearly, F is a defining formula for f, and by prenex operations,

$$\mathrm{I}\Delta_0(\exp) \vdash \exists_y F(\vec{x},y).$$

Furthermore, by the uniqueness condition on each G_i, we can also prove in $\mathrm{I}\Delta_0(\exp)$

$$\begin{array}{l} C(\vec{x},y_1,\ldots,y_m,y) \wedge C(\vec{x},z_1,\ldots,z_m,y') \\ \rightarrow y_1 = z_1 \wedge \ldots \wedge y_m = z_m \wedge G_0(y_1,\ldots,y_m,y) \wedge G_0(y_1,\ldots,y_m,y') \\ \rightarrow y = y' \end{array}$$

and hence by the quantifier rules of logic,

$$\mathrm{I}\Delta_0(\exp) \vdash F(\vec{x},y) \wedge F(\vec{x},y') \rightarrow y = y'.$$

Thus f is provably Σ_1 with F as a bounded defining formula, and it only remains to find a bounding term. But $\mathrm{I}\Delta_0(\exp)$ proves

$$C(\vec{x},y_1,\ldots,y_m,y) \rightarrow y_1 < 2_k(s) \wedge \ldots \wedge y_m < 2_k(s) \wedge y < 2_k(y_1+\ldots+y_m)$$

and

$$y_1 < 2_k(s) \wedge \ldots \wedge y_m < 2_k(s) \rightarrow y_1 + \ldots + y_m < 2_k(s) \cdot m.$$

Therefore by taking $t(\vec{x}\,)$ to be the term $2_k(2_k(s)\cdot m)$ we obtain

$$\mathrm{I}\Delta_0(\exp) \vdash C(\vec{x},y_1,\ldots,y_m,y) \rightarrow y < t(\vec{x}\,)$$

and hence

$$\mathrm{I}\Delta_0(\exp) \vdash F(\vec{x},y) \rightarrow y < t(\vec{x}\,).$$

This completes the proof.

Lemma 2.9 *Suppose f is defined by bounded minimization*

$$f(\vec{n},m) = \mu k{<}m\,(\,g(\vec{n},k) = 0\,)$$

from a function g which is provably bounded in $\mathrm{I}\Delta_0(\exp)$. *Then f is provably bounded in* $\mathrm{I}\Delta_0(\exp)$.

Proof. Let G be a bounded defining formula for g and let $F(\vec{x}, z, y)$ be the bounded formula

$$y \leq z \wedge \forall_{i<y} \neg G(\vec{x}, i, 0) \wedge (y = z \vee G(\vec{x}, y, 0)).$$

Obviously $F(\vec{n}, m, k)$ is true in the standard model if and only if either k is the least number less than m such that $g(\vec{n}, k) = 0$, or there is no such and $k = m$. But this is exactly what it means for k to be the value of $f(\vec{n}, m)$, so F is a defining formula for f. Furthermore $\mathrm{I}\Delta_0(\exp) \vdash F(\vec{x}, z, y) \rightarrow y < z + 1$, so $t(\vec{x}, z) = z + 1$ can be taken as a bounding term for f. Also it is clear that we can prove

$$F(\vec{x}, z, y) \wedge F(\vec{x}, z, y') \wedge y < y' \rightarrow G(\vec{x}, y, 0) \wedge \neg G(\vec{x}, y, 0)$$

and similarly with y and y' interchanged. Therefore

$$\mathrm{I}\Delta_0(\exp) \vdash F(\vec{x}, z, y) \wedge F(\vec{x}, z, y') \rightarrow \neg y < y' \wedge \neg y' < y$$

and hence, because $y < y' \vee y' < y \vee y = y'$ is provable, we have

$$\mathrm{I}\Delta_0(\exp) \vdash F(\vec{x}, z, y) \wedge F(\vec{x}, z, y') \rightarrow y = y'.$$

It remains to check that $\mathrm{I}\Delta_0(\exp) \vdash \exists_y F(\vec{x}, z, y)$. This is the point where bounded induction comes into play, since $\exists_y F(\vec{x}, z, y)$ is a bounded formula. We prove it by induction on z.

For the basis, recall that $y \leq 0 \leftrightarrow y = 0$ and $\neg i < 0$ are provable. Therefore $F(\vec{x}, 0, 0)$ is provable, and hence so is $\exists_y F(\vec{x}, 0, y)$.

For the induction step from z to $z + 1$, we can prove $y \leq z \rightarrow y + 1 \leq z + 1$ and, using $i < y + 1 \leftrightarrow i < y \vee i = y$,

$$\forall_{i<y} \neg G(\vec{x}, i, 0) \wedge (y = z \wedge \neg G(\vec{x}, y, 0)) \rightarrow \forall_{i<y+1} \neg G(\vec{x}, i, 0) \wedge y + 1 = z + 1$$

from which follows $F(\vec{x}, z, y) \rightarrow F(\vec{x}, z+1, y+1) \vee F(\vec{x}, z+1, y)$ and hence, by logic, $\exists_y F(\vec{x}, z, y) \rightarrow \exists_y F(\vec{x}, z + 1, y)$ which is the induction step. This completes the proof.

Theorem 2.10 *Every elementary function is provably bounded in the theory* $\mathrm{I}\Delta_0(\exp)$.

Proof. The elementary functions can be characterized as those definable from the constants 0, S, P, $+$, $\dot{-}$, $\cdot$, $\exp_2$ by composition and bounded minimization. The above lemmas show that each such function is provably bounded in $\mathrm{I}\Delta_0(\exp)$.

2.1. The provably recursive functions of $\mathrm{I}\Delta_0(\exp)$

Definition 2.11 A closed Σ_1-formula $\exists_{\vec{z}} B(\vec{z})$, with B a bounded formula, is said to be "*true at* m", and we write $m \models \exists_{\vec{z}} B(\vec{z})$, if there are numbers $\vec{m} = m_1, m_2, \ldots, m_l$ all less than m, such that $B(\vec{m})$ is true (in the standard model). A finite set Γ of closed Σ_1-formulas is "*true at* m", written $m \models \Gamma$, if at least one of them is true at m.

If $\Gamma(x_1, \ldots, x_k)$ is a finite set of Σ_1 formulas all of whose free variables occur among $x_1, \ldots, x_k$, and if $f\colon \mathbb{N}^k \rightarrow \mathbb{N}$, then we write $f \models \Gamma$ to mean that for all numerical assignments $\vec{n} = n_1, \ldots, n_k$ to the variables $\vec{x} = x_1, \ldots, x_k$ we have $f(\vec{n}) \models \Gamma(\vec{n})$.

Note 2.12 (Persistence) For sets Γ of closed Σ_1-formulas, if $m \models \Gamma$ and $m < m'$ then $m' \models \Gamma$. Similarly for sets $\Gamma(\vec{x})$ of Σ_1-formulas with free variables, if $f \models \Gamma$ and $f(\vec{n}) \leq f'(\vec{n})$ for all $\vec{n} \in N^k$ then $f' \models \Gamma$.

Lemma 2.13 *If $\Gamma(\vec{x})$ is a finite set of Σ_1-formulas (whose disjunction is) provable in $\mathrm{I}\Delta_0(\exp)$ then there is an elementary function f, strictly increasing in each of its variables, such that $f \models \Gamma$.*

Proof. It is convenient to use a Tait-style [12] formalism for the logic of $\mathrm{I}\Delta_0(\exp)$. Thus we derive finite sets Γ of formulas written in negation normal form, where Γ, A is shorthand for $\Gamma \cup \{A\}$ etcetera. The axioms will be all sets of formulas Γ which contain either a complementary pair of equations $t_1 = t_2, t_1 \neq t_2$, or an identity $t = t$, or an equality axiom $t_1 \neq t_2, \neg e(t_1), e(t_2)$ where $e(t)$ is any equation or inequation with a distinguished subterm t, or a substitution instance of one of the defining axioms for the constants. The logical rules are standard, and the cut rule and induction rule (replacing the equivalent axiom scheme) are respectively:

$$\frac{\Gamma,\ C \quad \Gamma,\ \neg C}{\Gamma} \qquad \frac{\Gamma,\ B(0) \quad \Gamma,\ \neg B(y),\ B(y+1)}{\Gamma,\ B(t)}$$

where C is the cut formula and, in the induction rule, B is any bounded formula, y is not free in Γ and t is any term.

Note that if Γ is provable in $\mathrm{I}\Delta_0(\exp)$ then it has a proof in the formalism just described, in which all cut formulas are Σ_1. For if Γ is classically derivable from non-logical axioms $A_1, \ldots, A_s$ then there is a cut-free proof in Tait-style logic of $\neg A_1,\ \Delta,\ \Gamma$ where $\Delta = \neg A_2, \ldots, \neg A_s$. We show how to cancel $\neg A_1$ using a Σ_1 cut. If A_1 is an induction axiom on the formula B we have a cut-free proof in logic of

$$B(0) \wedge \forall_y(\neg B(y) \vee B(y+1)) \ \wedge\ \exists_x \neg B(x),\ \Delta,\ \Gamma$$

and hence, by inversion, cut-free proofs of $B(0), \Delta, \Gamma$ and $\neg B(y), B(y+1), \Delta, \Gamma$ and $\exists_x \neg B(x), \Delta, \Gamma$. From the first two of these we obtain $B(x), \Delta, \Gamma$ by the induction rule above, then $\forall_x B(x), \Delta, \Gamma$, and then from the third we obtain Δ, Γ by a cut on the Σ_1-formula $\exists_x \neg B(x)$. If A_1 is the universal closure of any other (quantifier-free) axiom then we immediately obtain Δ, Γ by a cut on the Σ_1-formula $\neg A_1$. Having thus cancelled $\neg A_1$ we can similarly cancel each of $\neg A_2, \ldots, \neg A_s$ in turn, so as to yield the desired proof of Γ which only uses cuts on Σ_1-formulas.

Now, choosing such a proof for $\Gamma(\vec{x})$, we proceed by induction on its height, showing at each new proof-step how to define the required elementary function f such that $f \models \Gamma$.

(i) If $\Gamma(\vec{x})$ is an axiom then for all $\vec{n}$, $\Gamma(\vec{n})$ contains a true atom. Therefore $f \models \Gamma$ for any f. To make f sufficiently increasing choose $f(\vec{n}) = n_1 + \ldots + n_k$.

(ii) If $\Gamma,\ B_0 \vee B_1$ arises by an application of the $\vee$-rule from $\Gamma,\ B_0,\ B_1$ then (because of our definition of Σ_1-formula) B_0 and B_1 must both be bounded formulas. Thus by our definition of "true at", any function f satisfying $f \models \Gamma,\ B_0,\ B_1$ must also satisfy $f \models \Gamma,\ B_0 \vee B_1$.

(iii) Only a slightly more complicated argument applies to the dual case where $\Gamma,\ B_0 \wedge B_1$ arises by an application of the $\wedge$-rule from the premises $\Gamma,\ B_0$ and $\Gamma,\ B_1$.

For if $f_0(\vec{n}) \models \Gamma(\vec{n}),\ B_0(\vec{n})$ and $f_1(\vec{n}) \models \Gamma(\vec{n}),\ B_1(\vec{n})$ for all $\vec{n}$, then it is easy to see (by persistence) that $f \models \Gamma,\ B_0 \wedge B_1$ where $f(\vec{n}) = f_0(\vec{n}) + f_1(\vec{n})$.

(iv) If $\Gamma,\ \forall_y B(y)$ arises from $\Gamma,\ B(y)$ by the $\forall$-rule (y not free in Γ) then since all the formulas are $\Sigma_1, \forall_y B(y)$ must be bounded and so $B(y)$ must be of the form $y \not< t \vee B'(y)$ for some term t. Now assume $f_0 \models \Gamma,\ y \not< t, B'(y)$ for some increasing elementary function f_0. Then for all assignments $\vec{n}$ to the free variables $\vec{x}$, and all assignments k to the variable y, $f_0(\vec{n}, k) \models \Gamma(\vec{n}),\ k \not< t(\vec{n}),\ B'(\vec{n}, k)$. Therefore by defining $f(\vec{n}) = \Sigma_{k<g(\vec{n})}\ f_0(\vec{n}, k)$ where g is an increasing elementary function bounding t, we easily see that either $f(\vec{n}) \models \Gamma(\vec{n})$ or else, by persistence, $B'(\vec{n}, k)$ is true for every $k < t(\vec{n})$. Hence $f \models \Gamma,\ \forall_y B(y)$ as required, and clearly f is elementary since f_0 and g are.

(v) Now suppose $\Gamma,\ \exists_y A(y, \vec{x}\,)$ arises from $\Gamma,\ A(t, \vec{x}\,)$ by the $\exists$-rule, where A is Σ_1. Then by the induction hypothesis there is an elementary f_0 such that for all $\vec{n}$, $f_0(\vec{n}) \models \Gamma(\vec{n}),\ A(t(\vec{n}), \vec{n})$. Then either $f_0(\vec{n}) \models \Gamma(\vec{n})$ or else $f_0(\vec{n})$ bounds true witnesses for all the existential quantifiers already in $A(t(\vec{n}), \vec{n})$. Therefore by choosing any elementary bounding function g for the term t, and defining $f(\vec{n}) = f_0(\vec{n}) + g(\vec{n})$, we see that either $f(\vec{n}) \models \Gamma(\vec{n})$ or $f(\vec{n}) \models \exists_y A(y, \vec{n})$ for all $\vec{n}$.

(vi) If Γ comes about by the cut rule with Σ_1 cut formula $C \equiv \exists_{\vec{z}} B(\vec{z})$ then the two premises are $\Gamma,\ \forall_{\vec{z}}\, \neg B(\vec{z})$ and $\Gamma,\ \exists_{\vec{z}}\, B(\vec{z})$. The universal quantifiers in the first premise can be inverted (without increasing proof-height) to give $\Gamma,\ \neg B(\vec{z})$ and since B is bounded the induction hypothesis can be applied to this to give an elementary f_0 such that for all numerical assignments $\vec{n}$ to the (implicit) variables $\vec{x}$ and all assignments $\vec{m}$ to the new free variables $\vec{z}$, $f_0(\vec{n}, \vec{m}) \models \Gamma(\vec{n}),\ \neg B(\vec{n}, \vec{m})$. Applying the induction hypothesis to the second premise gives an elementary f_1 such that for all $\vec{n}$, either $f_1(\vec{n}) \models \Gamma(\vec{n})$ or else there are fixed witnesses $\vec{m} < f_1(\vec{n})$ such that $B(\vec{n}, \vec{m})$ is true. Therefore if we define f by substitution from f_0 and f_1 thus:

$$f(\vec{n}) = f_0(\vec{n}, f_1(\vec{n}), \ldots, f_1(\vec{n}))$$

then f will be elementary, greater than or equal to f_1, and strictly increasing since both f_0 and f_1 are. Furthermore $f \models \Gamma$. For otherwise there would be a tuple $\vec{n}$ such that $\Gamma(\vec{n})$ is not true at $f(\vec{n})$ and hence, by persistence, not true at $f_1(\vec{n})$. So $B(\vec{n}, \vec{m})$ is true for certain numbers $\vec{m} < f_1(\vec{n})$. But then $f_0(\vec{n}, \vec{m}) < f(\vec{n})$ and so, again by persistence, $\Gamma(\vec{n})$ cannot be true at $f_0(\vec{n}, \vec{m})$. This means $B(\vec{n}, \vec{m})$ is false, by the above, and so we have a contradiction.

(vii) Finally suppose $\Gamma(\vec{x}\,),\ B(\vec{x}, t)$ arises by an application of the induction rule on the bounded formula B from premises $\Gamma(\vec{x}\,),\ B(\vec{x}, 0)$ and $\Gamma(\vec{x}\,),\ \neg B(\vec{x}, y),\ B(\vec{x}, y+1)$. Applying the induction hypothesis to each of the premises one obtains increasing elementary functions f_0 and f_1 such that for all $\vec{n}$ and all k, $f_0(\vec{n}) \models \Gamma(\vec{n}),\ B(\vec{n}, 0)$ and $f_1(\vec{n}, k) \models \Gamma(\vec{n}),\ \neg B(\vec{n}, k),\ B(\vec{n}, k+1)$. Define $f(\vec{n}) = f_0(\vec{n}) + \Sigma_{k<g(\vec{n})}\, f_1(\vec{n}, k)$ where g is some increasing elementary bounding function for the term t. Then f is elementary and increasing, and by persistence from the above properties of f_0 and f_1, either $f(\vec{n}) \models \Gamma(\vec{n})$, or else $B(\vec{n}, 0)$ and $B(\vec{n}, k) \rightarrow B(\vec{n}, k+1)$ are true for all $k < t(\vec{n})$. In this latter case $B(\vec{n}, t(\vec{n}))$ is true by induction on k up to the value of $t(\vec{n})$. Either way, we have $f \models \Gamma(\vec{x}\,),\ B(\vec{x}, t(\vec{x}\,))$ and this completes the proof.

Theorem 2.14 *A number-theoretic function is elementary if and only if it is provably recursive in* $\mathrm{I}\Delta_0(\exp)$.

Proof. We have already shown that every elementary function is provably bounded, and hence provably Σ_1, in $\mathrm{I}\Delta_0(\exp)$. Conversely suppose f is provably Σ_1. Then there is a Σ_1-formula $F(\vec{x}, y) \equiv \exists_{z_1} \ldots \exists_{z_k} B(\vec{x}, y, z_1 \ldots z_k)$ which defines f and such that $\mathrm{I}\Delta_0(\exp) \vdash \exists_y F(\vec{x}, y)$. By the lemma immediately above, there is an elementary function g such that for every tuple of arguments $\vec{n}$ there are numbers $m_0, m_1, \ldots, m_k$ less than $g(\vec{n})$ satisfying the bounded formula $B(\vec{n}, m_0, m_1, \ldots, m_k)$. Using elementary sequence-coding functions, let $h(\vec{n}) = \langle g(\vec{n}), g(\vec{n}), \ldots, g(\vec{n})\rangle$ so that if $m = \langle m_0, m_1, \ldots, m_k\rangle$ where $m_0, m_1, \ldots, m_k < g(\vec{n})$, then $m < h(\vec{n})$. Then, because $f(\vec{n})$ is the unique m_0 for which there are $m_1, \ldots, m_k$ satisfying $B(\vec{n}, m_0, m_1, \ldots, m_k)$, we can define f as follows:

$$f(\vec{n}) = (\ \mu m{<}h(\vec{n})\, B(\vec{n}, (m)_0, (m)_1, \ldots, (m)_k)\ )_0.$$

Since B is a bounded formula of $\mathrm{I}\Delta_0(\exp)$ it is elementarily decidable, and since the least number operator μ is bounded by the elementary function h, the entire definition of f therefore involves only elementary operations. Hence f is an elementary function.

3. The Provably Recursive Functions of Arithmetic

This section develops the classification theory of the provably recursive functions of arithmetic. The topic has a long history tracing back to Kreisel [7] who, in setting out his "no-counter-example" interpretation, gave the first explicit characterization of the functions "computable in" arithmetic, as those definable by recursions over standard well-orderings of the natural numbers with order-types less than ε_0. Such a characterization was perhaps not so surprising in light of the earlier, groundbreaking work of Gentzen [3], showing that these well-orderings are just the ones over which one can prove transfinite induction in arithmetic, and thereby prove the totality of functions defined by recursions over them. Subsequent independent work by Schwichtenberg and the present author around 1970, extending previous results of Grzegorczyk and Robbin, then provided other complexity characterizations in terms of natural, simply-defined hierarchies of so-called "fast growing" bounding functions. (This is a good place to advertise our forthcoming book [10].) What was surprising was the deep connection later discovered by Ketonen and Solovay [6], between these bounding functions and a variety of combinatorial results related to the "modified" Finite Ramsey Theorem of Paris and Harrington [8]. It is through this connection that one gains immediate access to a range of mathematically meaningful independence results for arithmetic and stronger theories. Thus, classifying the provably recursive functions of a theory not only gives a measure of its computational power, it also serves to delimit its mathematical power in providing natural examples of true mathematical statements it cannot prove.

The theories we shall be concerned with in this chapter are PA (Peano Arithmetic) and its inductive fragments $\mathrm{I}\Sigma_n$. We take, as our formalization of PA, $\mathrm{I}\Delta_0(\exp)$ together with all induction axioms

$$A(0) \wedge \forall_a(A(a) \to A(a+1)) \to A(t)$$

for arbitrary formulas A and (substitutible) terms t. $\mathrm{I}\Sigma_n$ has the same base-theory $\mathrm{I}\Delta_0(\exp)$, but the induction axioms are restricted to formulas A of the form Σ_i or Π_i with $i \leq n$, defined for the purposes of this chapter as follows:

Definition 3.1 Σ_1-formulas have already been defined. A Π_1 formula is the dual or (classically) negation of a Σ_1-formula. For $n > 1$, a Σ_n formula is one formed by prefixing just one existential quantifier to a Π_{n-1} formula, and a Π_n formula is one formed by prefixing just one universal quantifier to a Σ_{n-1} formula. Thus only in the cases Σ_1 and Π_1 do strings of like quantifiers occur. In all other cases, strings of like quantifiers are assumed to have been contracted into one such, using the pairing functions which are available in $\mathrm{I}\Delta_0(\exp)$. This is no real restriction, but merely a matter of convenience for later results.

It doesn't matter whether one restricts to Σ_n or Π_n induction formulas since, in the presence of the subtraction function, induction on a Π_n formula A is reducible to induction on its Σ_n dual $\neg A$, and vice-versa. For if one replaces $A(a)$ by $\neg A(t \dot{-} a)$ in the induction axiom, and then contraposes, one obtains

$$A(t \dot{-} t) \wedge \forall_a(A(t \dot{-} (a+1)) \to A(t \dot{-} a)) \to A(t \dot{-} 0)$$

from which follows the induction axiom for $A(a)$ itself, since $t \dot{-} t = 0$, $t \dot{-} 0 = t$, and $t \dot{-} a = (t \dot{-} (a+1)) + 1$ if $t \dot{-} a \neq 0$.

Historically of course, Peano's Axioms only include definitions of zero, successor, addition and multiplication, whereas the base-theory we have chosen includes predecessor, modified subtraction and exponentiation as well. We do this because $\mathrm{I}\Delta_0(\exp)$ is both a natural and convenient theory to have available from the start. However these extra functions can all be provably Σ_1-defined in $\mathrm{I}\Sigma_1$ from the "pure" Peano Axioms using the Chinese Remainder Theorem, so we are not actually increasing the strength of any of the theories here by including them. Furthermore the results in this chapter would not at all be affected by adding to the base-theory any other elementary (or even primitive recursive) functions one wishes.

3.1. Ordinals below ε_0

Throughout the rest of this chapter, $\alpha, \beta, \gamma, \delta, \ldots$ will denote ordinals less than ε_0. Every such ordinal is either 0 or can be represented uniquely in so-called Cantor Normal Form thus:

$$\alpha = \omega^{\gamma_1} \cdot c_1 + \omega^{\gamma_2} \cdot c_2 + \ldots + \omega^{\gamma_k} \cdot c_k$$

where $\gamma_k < \ldots < \gamma_2 < \gamma_1 < \alpha$ and the coefficients $c_1, c_2, \ldots, c_k$ are arbitrary positive integers. If $\gamma_k = 0$ then α is a successor ordinal, written $\mathrm{Succ}(\alpha)$, and its immediate predecessor $\alpha - 1$ has the same representation but with c_k reduced to $c_k - 1$. Otherwise α is a limit ordinal, written $\mathrm{Lim}(\alpha)$, and it has infinitely-many possible "fundamental sequences", i.e., increasing sequences of smaller ordinals whose supremum is α. However we shall pick out *one particular* fundamental sequence $\{\alpha(n)\}$ for each such limit ordinal α, as follows: first write α as $\delta + \omega^\gamma$ where $\delta = \omega^{\gamma_1} \cdot c_1 + \ldots + \omega^{\gamma_k} \cdot (c_k - 1)$ and $\gamma = \gamma_k$. Assume inductively that when γ is a limit, its fundamental sequence $\{\gamma(n)\}$ has already been specified. Then define, for each $n \in \mathbb{N}$,

$$\alpha(n) = \begin{cases} \delta + \omega^{\gamma-1} \cdot (n+1) & \text{if } \mathrm{Succ}(\gamma) \\ \delta + \omega^{\gamma(n)} & \text{if } \mathrm{Lim}(\gamma). \end{cases}$$

Clearly $\{\alpha(n)\}$ is an increasing sequence of ordinals with supremum α. For ε_0 itself we choose the fundamental sequence $\varepsilon_0(n) = \omega$ if $n = 0$ and $\omega^{\varepsilon_0(n-1)}$ otherwise.

Definition 3.2 With each $\alpha < \varepsilon_0$ and each natural number n, associate a finite set of ordinals $\alpha[n]$ as follows:

$$\alpha[n] = \begin{cases} \emptyset & \text{if } \alpha = 0 \\ (\alpha - 1)[n] \cup \{\alpha - 1\} & \text{if } \mathrm{Succ}(\alpha) \\ \alpha(n)[n] & \text{if } \mathrm{Lim}(\alpha). \end{cases}$$

Lemma 3.3 *For each $\alpha = \delta + \omega^\gamma$ and all n,*

$$\alpha[n] = \delta[n] \cup \{\, \delta + \omega^{\gamma_1} \cdot c_1 + \ldots + \omega^{\gamma_k} \cdot c_k \mid \forall_i(\gamma_i \in \gamma[n] \wedge c_i \leq n) \,\}.$$

Proof. By induction on γ. If $\gamma = 0$ then $\gamma[n]$ is empty and so the right hand side is just $\delta[n] \cup \{\delta\}$, which is the same as $\alpha[n] = (\delta + 1)[n]$ according to the definition above.

If γ is a limit then $\gamma[n] = \gamma(n)[n]$ so the set on the right hand side is the same as the one with $\gamma(n)[n]$ instead of $\gamma[n]$. By the induction hypothesis applied to $\alpha(n) = \delta + \omega^{\gamma(n)}$, this set equals $\alpha(n)[n]$, which is just $\alpha[n]$ again by definition.

Now suppose γ is a successor. Then α is a limit and $\alpha[n] = \alpha(n)[n]$ where $\alpha(n) = \delta + \omega^{\gamma-1} \cdot (n+1)$. This we can write as $\alpha(n) = \alpha(n-1) + \omega^{\gamma-1}$ where, in case $n = 0$, $\alpha(-1) = \delta$. By the induction hypothesis for $\gamma - 1$, the set $\alpha[n]$ therefore consists of $\alpha(n-1)[n]$ together with all ordinals of the form

$$\alpha(n-1) + \omega^{\gamma_1} \cdot c_1 + \ldots + \omega^{\gamma_k} \cdot c_k$$

where for all $i = 1, \ldots, k$, $\gamma_i \in (\gamma - 1)[n]$ and $c_i \leq n$. Similarly for each of $\alpha(n-1)[n]$, $\alpha(n-2)[n]$, ... , $\alpha(1)[n]$. Since for each $m \leq n$, $\alpha(m-1) = \delta + \omega^{\gamma-1} \cdot m$, this last set is just the union of $\delta[n]$ together with all ordinals of the form

$$\delta + \omega^{\gamma-1} \cdot m + \omega^{\gamma_1} \cdot c_1 + \ldots + \omega^{\gamma_k} \cdot c_k$$

where $m \leq n$ and for all $i = 1 \ldots k$, $\gamma_i \in (\gamma - 1)[n]$ and $c_i \leq n$. But this is the set required because $\gamma[n] = (\gamma - 1)[n] \cup \{\gamma - 1\}$. This completes the proof.

Corollary 3.4 *For every limit ordinal $\alpha < \varepsilon_0$ and every n, $\alpha(n) \in \alpha[n+1]$. Furthermore if $\beta \in \gamma[n]$ then $\omega^\beta \in \omega^\gamma[n]$ provided $n \neq 0$.*

Definition 3.5 The *maximum coefficient* of $\beta = \omega^{\beta_1} \cdot b_1 + \ldots + \omega^{\beta_l} \cdot b_l$ is defined inductively to be the maximum of all the b_i and all the maximum coefficients of the exponents β_i.

Lemma 3.6 *If $\beta < \alpha$ and the maximum coefficient of β is $\leq n$ then $\beta \in \alpha[n]$.*

Proof. By induction on α. Let $\alpha = \delta + \omega^\gamma$. If $\beta < \delta$, then $\beta \in \delta[n]$ by IH and $\delta[n] \subseteq \alpha[n]$ by the lemma. Otherwise $\beta = \delta + \omega^{\beta_1} \cdot b_1 + \ldots + \omega^{\beta_k} \cdot b_k$ with $\alpha > \gamma > \beta_1 > \ldots > \beta_k$ and $b_i \leq n$. By IH $\beta_i \in \gamma[n]$. Hence $\beta \in \alpha[n]$ by the lemma.

Definition 3.7 Let $G_\alpha(n)$ denote the cardinality of the finite set $\alpha[n]$. Then immediately from the definition of $\alpha[n]$ we have

$$G_\alpha(n) = \begin{cases} 0 & \text{if } \alpha = 0 \\ G_{\alpha-1}(n)+1 & \text{if Succ}(\alpha) \\ G_{\alpha(n)}(n) & \text{if Lim}(\alpha). \end{cases}$$

The hierarchy of functions G_α is called the "slow growing hierarchy".

Lemma 3.8 *If $\alpha = \delta + \omega^\gamma$ then for all n*

$$G_\alpha(n) = G_\delta(n) + (n+1)^{G_\gamma(n)}.$$

Therefore for each $\alpha < \varepsilon_0$, $G_\alpha(n)$ is the elementary function which results by substituting $n+1$ for every occurrence of ω in the Cantor Normal Form of α.

Proof. By induction on γ. If $\gamma = 0$ then $\alpha = \delta + 1$, so $G_\alpha(n) = G_\delta(n) + 1 = G_\delta(n) + (n+1)^0$ as required. If γ is a successor then α is a limit and $\alpha(n) = \delta + \omega^{\gamma-1} \cdot (n+1)$, so by $n+1$ applications of the induction hypothesis for $\gamma - 1$ we have $G_\alpha(n) = G_{\alpha(n)}(n) = G_\delta(n) + (n+1)^{G_{\gamma-1}(n)} \cdot (n+1) = G_\delta(n) + (n+1)^{G_\gamma(n)}$ since $G_{\gamma-1}(n) + 1 = G_\gamma(n)$. Finally, if γ is a limit then $\alpha(n) = \delta + \omega^{\gamma(n)}$, so applying the induction hypothesis to $\gamma(n)$, we have $G_\alpha(n) = G_{\alpha(n)}(n) = G_\delta(n) + (n+1)^{G_{\gamma(n)}(n)}$ which immediately gives the desired result since $G_{\gamma(n)}(n) = G_\gamma(n)$ by definition.

Definition 3.9 (Coding ordinals) Encode each ordinal $\beta = \omega^{\beta_1} \cdot b_1 + \omega^{\beta_2} \cdot b_2 + \ldots + \omega^{\beta_l} \cdot b_l$ by the sequence number $\bar{\beta}$ constructed recursively as follows:

$$\bar{\beta} = \langle \langle \bar{\beta}_1, b_1 \rangle, \langle \bar{\beta}_2, b_2 \rangle, \ldots, \langle \bar{\beta}_l, b_l \rangle \rangle.$$

The ordinal 0 is coded by the empty sequence number 0. Note that $\bar{\beta}$ is numerically greater than the maximum coefficient of β, and greater than the codes $\bar{\beta}_i$ of all its exponents, and their exponents etcetera.

Lemma 3.10 *There is an elementary function $h(m,n)$ such that, with $m = \bar{\beta}$,*

$$h(\bar{\beta}, n) = \begin{cases} 0 & \textit{if } \beta = 0 \\ \overline{\beta - 1} & \textit{if } \text{Succ}(\beta) \\ \overline{\beta(n)} & \textit{if } \text{Lim}(\beta). \end{cases}$$

Furthermore, for each fixed $\alpha < \varepsilon_0$ there is an elementary well-ordering $\prec_\alpha \subset \mathbb{N}^2$ such that for all $b, c \in \mathbb{N}$, $b \prec_\alpha c$ if and only if $b = \bar{\beta}$ and $c = \bar{\gamma}$ for some $\beta < \gamma < \alpha$.

Thus the principles of transfinite induction and transfinite recursion over initial segments of the ordinals below ε_0, can all be expressed in the language of elementary recursive arithmetic.

3.2. The fast growing hierarchy

Definition 3.11 The "Hardy Hierarchy" $\{H_\alpha\}_{\alpha < \varepsilon_0}$ is defined by recursion on α thus:

$$H_\alpha(n) = \begin{cases} n & \text{if } \alpha = 0 \\ H_{\alpha-1}(n+1) & \text{if Succ}(\alpha) \\ H_{\alpha(n)}(n) & \text{if Lim}(\alpha). \end{cases}$$

The "Fast Growing Hierarchy" $\{F_\alpha\}_{\alpha<\varepsilon_0}$ is defined by recursion on α thus:

$$F_\alpha(n) = \begin{cases} n+1 & \text{if } \alpha = 0 \\ F^{n+1}_{\alpha-1}(n) & \text{if } \mathrm{Succ}(\alpha) \\ F_{\alpha(n)}(n) & \text{if } \mathrm{Lim}(\alpha) \end{cases}$$

where $F^{n+1}_{\alpha-1}(n)$ is the $n+1$-times iterate of $F_{\alpha-1}$ on n.

Note 3.12 The H_α and F_α functions could equally well be defined purely number-theoretically, by working over the well-orderings $\prec_\alpha$ instead of directly over the ordinals themselves. Thus they are examples of ε_0-recursive functions.

Lemma 3.13 *For all α, β and all n,*

1. $H_{\alpha+\beta}(n) = H_\alpha(H_\beta(n))$,
2. $H_{\omega^\alpha}(n) = F_\alpha(n)$.

Proof. The first part is proven by induction on β, the unstated assumption being that the Cantor Normal Form of $\alpha+\beta$ is just the result of concatenating their two separate Cantor Normal Forms, so that $(\alpha+\beta)(n) = \alpha + \beta(n)$. This of course requires that the leading exponent in the normal form of β is not greater than the final exponent in the normal form of α. We shall always make this assumption when writing $\alpha+\beta$.

If $\beta = 0$ the equation holds trivially because H_0 is the identity function. If $\mathrm{Succ}(\beta)$ then by the definition of the Hardy functions and the induction hypothesis for $\beta - 1$,

$$H_{\alpha+\beta}(n) = H_{\alpha+(\beta-1)}(n+1) = H_\alpha(H_{\beta-1}(n+1)) = H_\alpha(H_\beta(n)).$$

If $\mathrm{Lim}(\beta)$ then by the induction hypothesis for $\beta(n)$,

$$H_{\alpha+\beta}(n) = H_{\alpha+\beta(n)}(n) = H_\alpha(H_{\beta(n)}(n)) = H_\alpha(H_\beta(n)).$$

The second part is proved by induction on α. If $\alpha = 0$ then $H_{\omega^0}(n) = H_1(n) = n+1 = F_0(n)$. If $\mathrm{Succ}(\alpha)$ then by the limit case of the definition of H, the induction hypothesis, and the first part above,

$$H_{\omega^\alpha}(n) = H_{\omega^{\alpha-1}\cdot(n+1)}(n) = H^{n+1}_{\omega^{\alpha-1}}(n) = F^{n+1}_{\alpha-1}(n) = F_\alpha(n).$$

If $\mathrm{Lim}(\alpha)$ then the equation follows immediately by the induction hypothesis for $\alpha(n)$. This completes the proof.

Lemma 3.14 *For each $\alpha < \varepsilon_0$, H_α is strictly increasing and $H_\beta(n) < H_\alpha(n)$ whenever $\beta \in \alpha[n]$. The same holds for F_α, with the slight restriction that $n \neq 0$, for when $n = 0$ we have $F_\alpha(0) = 1$ for all α.*

Proof. By induction on α. The case $\alpha = 0$ is trivial since H_0 is the identity function and $0[n]$ is empty. If $\mathrm{Succ}(\alpha)$ then H_α is $H_{\alpha-1}$ composed with the successor function, so it is strictly increasing by the induction hypothesis. Furthermore if $\beta \in \alpha[n]$ then either $\beta \in (\alpha-1)[n]$ or $\beta = \alpha - 1$ so, again by the induction hypothesis, $H_\beta(n) \leq H_{\alpha-1}(n) < H_{\alpha-1}(n+1) = H_\alpha(n)$. If $\mathrm{Lim}(\alpha)$ then $H_\alpha(n) = H_{\alpha(n)}(n) < H_{\alpha(n)}(n+1)$ by the induction hypothesis. But as noted previously, $\alpha(n) \in \alpha[n+1] = \alpha(n+1)[n+1]$, so by applying the induction hypothesis to $\alpha(n+1)$ we have $H_{\alpha(n)}(n+1) < H_{\alpha(n+1)}(n+1) =$

$H_\alpha(n+1)$. Thus $H_\alpha(n) < H_\alpha(n+1)$. Furthermore if $\beta \in \alpha[n]$ then $\beta \in \alpha(n)[n]$ so $H_\beta(n) < H_{\alpha(n)}(n) = H_\alpha(n)$ straightaway by the induction hypothesis for $\alpha(n)$.

The same holds for $F_\alpha = H_{\omega^\alpha}$ provided we restrict to $n \neq 0$ since if $\beta \in \alpha[n]$ we then have $\omega^\beta \in \omega^\alpha[n]$. This completes the proof.

Lemma 3.15 *If $\beta \in \alpha[n]$ then $F_{\beta+1}(m) \leq F_\alpha(m)$ for all $m \geq n$.*

Proof. By induction on α, the zero case being trivial. If α is a successor then either $\beta \in (\alpha-1)[n]$ in which case the result follows straight from the induction hypothesis, or $\beta = \alpha - 1$ in which case it's immediate. If α is a limit then we have $\beta \in \alpha(n)[n]$ and hence by the induction hypothesis, $F_{\beta+1}(m) \leq F_{\alpha(n)}(m)$. But $F_{\alpha(n)}(m) \leq F_\alpha(m)$ either by definition of F in case $m = n$, or by the last lemma when $m > n$ since then $\alpha(n) \in \alpha[m]$.

3.3. Provable recursiveness of H_α and F_α

We now prove that for every $\alpha < \varepsilon_0(i)$, with $i > 0$, the function F_α is provably recursive in the theory $\mathrm{I}\Sigma_{i+1}$.

Since all of the machinery we have developed for coding ordinals below ε_0 is elementary, we can safely assume that it is available to us in an appropriate conservative extension of $\mathrm{I}\Delta_0(\exp)$, and can in fact be defined (with all relevant properties proven) in $\mathrm{I}\Delta_0(\exp)$ itself. In particular we shall again make use of the function h such that, if a codes a successor ordinal α then $h(a,n)$ codes $\alpha - 1$, and if a codes a limit ordinal α then $h(a,n)$ codes $\alpha(n)$. Note that we can decide whether a codes a succesor ordinal ($\mathrm{Succ}(a)$) or a limit ordinal ($\mathrm{Lim}(a)$), by asking whether $h(a,0) = h(a,1)$ or not. It is easiest to develop first the provable recursiveness of the Hardy functions H_α, since they have a simpler, unnested recursive definition. The fast growing functions are then easily obtained by the equation $F_\alpha = H_{\omega^\alpha}$.

Definition 3.16 Let $H(a,x,y,z)$ denote the following $\Delta_0(\exp)$ formula:
$(\mathrm{z})_0 = \langle 0, y\rangle \wedge \pi_2(z) = \langle a, x\rangle \wedge$
$\forall_{i<\mathrm{lh}(z)}(\mathrm{lh}((z)_i) = 2 \wedge (i > 0 \to (z)_{i,0} > 0)) \wedge$
$\forall_{0<i<\mathrm{lh}(z)}(\mathrm{Succ}((z)_{i,0}) \to (z)_{i-1,0} = h((z)_{i,0},(z)_{i,1}) \wedge (z)_{i-1,1} = (z)_{i,1}{+}1) \wedge$
$\forall_{0<i<\mathrm{lh}(z)}(\mathrm{Lim}((z)_{i,0}) \to (z)_{i-1,0} = h((z)_{i,0},(z)_{i,1}) \wedge (z)_{i-1,1} = (z)_{i,1}).$

Lemma 3.17 (Definability of H_α) *$H_\alpha(n) = m$ if and only if $\exists_z H(\bar\alpha, n, m, z)$ is true. Furthermore, for each $\alpha < \varepsilon_0$ we can prove in $\mathrm{I}\Delta_0(\exp)$,*

$$\exists_z H(\bar\alpha, x, y, z) \wedge \exists_z H(\bar\alpha, x, y', z) \to y = y'.$$

Proof. The meaning of the formula $\exists_z H(\bar\alpha, n, m, z)$ is that there is a finite sequence of pairs $\langle \alpha_i, n_i\rangle$, beginning with $\langle 0, m\rangle$ and ending with $\langle \alpha, n\rangle$, such that at each $i > 0$, if $\mathrm{Succ}(\alpha_i)$ then $\alpha_{i-1} = \alpha_i - 1$ and $n_{i-1} = n_i + 1$, and if $\mathrm{Lim}(\alpha_i)$ then $\alpha_{i-1} = \alpha_i(n_i)$ and $n_{i-1} = n_i$. Thus by induction up along the sequence, and using the original definition of H_α, we easily see that for each $i > 0$, $H_{\alpha_i}(n_i) = m$, and thus at the end, $H_\alpha(n) = m$. Conversely, if $H_\alpha(n) = m$ then there must exist such a computation-sequence, and this proves the first part of the lemma.

For the second part notice that, by induction on the length of the computation-sequence s, we can prove, for each n, m, m', s, s' that

$$H(\bar{\alpha}, n, m, s) \to H(\bar{\alpha}, n, m', s') \to s = s' \wedge m = m'.$$

This proof can be formalized directly in $\mathrm{I}\Delta_0(\exp)$ to give

$$H(\bar{\alpha}, x, y, z) \to H(\bar{\alpha}, x, y', z') \to z = z' \wedge y = y'$$

and hence

$$\exists_z H(\bar{\alpha}, x, y, z) \to \exists_z H(\bar{\alpha}, x, y', z) \to y = y'.$$

as required.

Thus in order for H_α to be provably recursive it remains only to prove (in the required theory) $\exists_y \exists_z H(\bar{\alpha}, x, y, z)$.

Lemma 3.18 *In $\mathrm{I}\Delta_0(\exp)$ we can prove*

$$\exists_z H(\omega^a, x, y, z) \to \exists_z H(\omega^a \cdot c, y, w, z) \to \exists_z H(\omega^a \cdot (c+1), x, w, z)$$

where ω^a is the elementary term $\langle\langle a, 1\rangle\rangle$ which constructs, from the code a of an ordinal α, the code for the ordinal ω^α, and $b \cdot 0 = 0$, $b \cdot (z+1) = b \cdot z \oplus b$, with $\oplus$ the elementary function which computes $\overline{\alpha + \beta}$ from $\bar{\alpha}$ and $\bar{\beta}$.

Proof. By assumption we have sequences s, s' satisfying $H(\omega^a, n, m, s)$ and $H(\omega^a \cdot c, m, k, s')$. Add $\omega^a \cdot c$ (in the sense of $\oplus$) to the first component of each pair in s. Then the last pair in s' and the first pair in s become identical. By concatenating the two – taking this double pair only once – construct an elementary term $t(s, s')$ satisfying $H(\omega^a \cdot (c+1), n, k, t)$. We can then prove

$$H(\omega^a, x, y, z) \to H(\omega^a \cdot c, y, w, z') \to H(\omega^a \cdot (c+1), x, w, t)$$

in a conservative extension of $\mathrm{I}\Delta_0(\exp)$, and hence in $\mathrm{I}\Delta_0(\exp)$ derive

$$\exists_z H(\omega^a, x, y, z) \to \exists_z H(\omega^a \cdot c, y, w, z) \to \exists_z H(\omega^a \cdot (c+1), x, w, z).$$

Lemma 3.19 *Let $H(a)$ be the Π_2 formula $\forall_x \exists_y \exists_z H(a, x, y, z)$. Then with Π_2-induction we can prove the following:*

1. $H(\omega^0)$.
2. $\mathrm{Succ}(a) \to H(\omega^{h(a,0)}) \to H(\omega^a)$.
3. $\mathrm{Lim}(a) \to \forall_x H(\omega^{h(a,x)}) \to H(\omega^a)$.

Proof. The term $t_0 = \langle\langle 0, x+1\rangle, \langle 1, x\rangle\rangle$ witnesses $H(\omega^0, x, x+1, t_0)$ in $\mathrm{I}\Delta_0(\exp)$, so $H(\omega^0)$ is immediate.

With the aid of the lemma just proven we can derive

$$H(\omega^{h(a,0)}) \to H(\omega^{h(a,0)} \cdot c) \to H(\omega^{h(a,0)}) \cdot (c+1)$$

Therefore by Π_2 induction we obtain

$$H(\omega^{h(a,0)}) \to H(\omega^{h(a,0)} \cdot (x+1))$$

and then

$$H(\omega^{h(a,0)}) \to \exists_y \exists_z H(\omega^{h(a,0)} \cdot (x+1), x, y, z).$$

But there is an elementary term t_1 with the property

$$\mathrm{Succ}(a) \to H(\omega^{h(a,0)} \cdot (x+1), x, y, z) \to H(\omega^a, x, y, t_1)$$

since t_1 only needs to tagg onto the end of the sequence z the new pair $\langle \omega^a, x \rangle$, thus $t_1 = \pi(z, \langle \omega^a, x \rangle)$. Hence by the quantifier rules,

$$\mathrm{Succ}(a) \to H(\omega^{h(a,0)}) \to H(\omega^a).$$

The final case is now straightforward, since the term t_1 just constructed also gives

$$\mathrm{Lim}(a) \to H(\omega^{h(a,x)}, x, y, z) \to H(\omega^a, x, y, t_1)$$

and so by quantifier rules again,

$$\mathrm{Lim}(a) \to \forall_x H(\omega^{h(a,x)}) \to H(\omega^a).$$

Definition 3.20 (Structural Transfinite Induction) The *structural progressiveness* of a formula $A(a)$ is expressed by $\mathrm{SProg}_a A$, which is the conjunction of the formulas $A(0)$, $\forall_a(\mathrm{Succ}(a) \to A(h(a,0)) \to A(a))$, and $\forall_a(\mathrm{Lim}(a) \to \forall_x A(h(a,x)) \to A(a))$. The principle of *structural transfinite induction* up to an ordinal α is then the following axiom-scheme, for all formulas A:

$$\mathrm{SProg}_a A \to \forall_{a \prec \bar{\alpha}} A(a)$$

where $a \prec \bar{\alpha}$ means a lies in the field of the well-ordering $\prec_\alpha$ (i.e. $a = 0 \vee 0 \prec_\alpha a$).

Note 3.21 The last lemma shows that the Π_2 formula $H(\omega^a)$ is structural progressive, and that this is provable with Π_2-induction.

We now make use of a famous result of Gentzen [3], which says that transfinite induction is provable in arithmetic up to any $\alpha < \varepsilon_0$. We prove this fact in a slightly more general form, where one can recurse to *all* points strictly below the present one, and need not refer to distinguished fundamental sequences.

Definition 3.22 (Transfinite Induction) The (general) *progressiveness* of a formula $A(a)$ is

$$\mathrm{Prog}_a A := \forall_a \big(\forall_{b \prec a} A(b) \to A(a)\big).$$

The principle of *transfinite induction* up to an ordinal α is the scheme

$$\mathrm{Prog}_a A \to \forall_{a \prec \bar{\alpha}} A(a)$$

where again $a \prec \bar{\alpha}$ means a lies in the field of the well-ordering $\prec_\alpha$.

Lemma 3.23 *Structural transfinite induction up to α is derivable from transfinite induction up to α.*

Proof. Let A be an arbitrary formula and assume $\mathrm{SProg}_a A$; we must show $\forall_{a\prec\bar\alpha} A(a)$. Using transfinite induction for the formula $a \prec \bar\alpha \to A(a)$ it suffices to prove

$$\forall_a\big(\forall_{b\prec a; b\prec\bar\alpha} A(b) \to a \prec \bar\alpha \to A(a)\big)$$

which is equivalent to

$$\forall_{a\prec\bar\alpha}\big(\forall_{b\prec a} A(b) \to A(a)\big).$$

This is easily proved from $\mathrm{SProg}_a A$, using the properties of the h function, and distinguishing the cases $a = 0$, $\mathrm{Succ}(a)$ and $\mathrm{Lim}(a)$.

We now come to Gentzen's theorem. In the proof we will need some properties of $\prec$ which can all be proved in $\mathrm{I}\Delta_0(\exp)$: irreflexivity and transitivity for $\prec$, and also, following Schütte,
$a \prec 0 \to A$,
$c \prec b \oplus \omega^0 \to (c \prec b \to A) \to (c = b \to A) \to A$,
$a \oplus 0 = a$,
$a \oplus (b \oplus c) = (a \oplus b) \oplus c$,
$0 \oplus a = a$,
$\omega^a 0 = 0$,
$\omega^a(x+1) = \omega^a x \oplus \omega^a$,
$a \neq 0 \to c \prec b \oplus \omega^a \to c \prec b \oplus \omega^{\mathrm{e}(a,b,c)}\mathrm{m}(a,b,c)$,
$a \neq 0 \to c \prec b \oplus \omega^a \to \mathrm{e}(a,b,c) \prec a$.
where e and m are appropriate function constants, easily seen to be elementary by comparing Cantor Normal Forms.

Theorem 3.24 (Gentzen) *For every Π_2 formula F and each $i > 0$ we can prove in $\mathrm{I}\Sigma_{i+1}$ the principle of transfinite induction up to α for all $\alpha < \varepsilon_0(i)$.*

Proof. Starting with any Π_j formula $A(a)$, we construct the formula

$$A^+(a) := \forall_b(\forall_{c\prec b} A(c) \to \forall_{c\prec b\oplus\omega^a} A(c))$$

where, as mentioned above, $\oplus$ is the elementary addition function on ordinal-codes thus: $\bar\alpha \oplus \bar\gamma = \overline{\alpha+\gamma}$. Note that since A is Π_j then A^+ is (provably equivalent to) a Π_{j+1} formula. The crucial point is that

$$\mathrm{I}\Sigma_j \vdash \mathrm{Prog}_a A(a) \to \mathrm{Prog}_a A^+(a).$$

So assume $\mathrm{Prog}_a A(a)$, that is $\forall_a(\forall_{b\prec a} A(b) \to A(a))$, and $\forall_{b\prec a} A^+(b)$. We have to show $A^+(a)$. So assume further $\forall_{c\prec b} A(c)$ and $c \prec b \oplus \omega^a$. We have to show $A(c)$, making use of the various properties of $\prec$ listed above.

If $a = 0$, then $c \prec b \oplus \omega^0$. It suffices to derive $A(c)$ from $c \prec b$ as well as from $c = b$. In the first case it follows by assumption, and in the second case from the progressiveness of A.

If $a \neq 0$, from $c \prec b \oplus \omega^a$ we obtain $c \prec b \oplus \omega^{\mathrm{e}(a,b,c)}\mathrm{m}(a,b,c)$ where $\mathrm{e}(a,b,c) \prec a$. We then obtain $A^+(\mathrm{e}(a,b,c))$ and by the definition of $A^+(x)$ we get

$$\forall_{u\prec b\oplus\omega^{\mathrm{e}(a,b,c)}x} A(u) \to \forall_{u\prec(b\oplus\omega^{\mathrm{e}(a,b,c)}x)\oplus\omega^{\mathrm{e}(a,b,c)}} A(u)$$

and hence

$$\forall_{u \prec b \oplus \omega^{e(a,b,c)} x} A(u) \to \forall_{u \prec b \oplus \omega^{e(a,b,c)}(x+1)} A(u).$$

Also

$$\forall_{u \prec b \oplus \omega^{e(a,b,c)} 0} A(u)$$

and using an appropriate instance of the Π_j induction scheme we can then conclude

$$\forall_{u \prec b \oplus \omega^{e(a,b,c)} m(a,b,c)} A(u)$$

and hence $A(c)$ as required.

Now fix $i > 0$ and (throughout the rest of this proof) let $\prec$ denote the well-ordering $\prec_{\varepsilon_0(i)}$. Given any Π_2 formula $F(v)$ define $A(a)$ to be the formula $\forall_{v \prec a} F(v)$. Then A is also (provably equivalent to) a Π_2 formula and furthermore it is easy to see that $\mathrm{Prog}_v F(v) \to \mathrm{Prog}_a A(a)$ is derivable in $\mathrm{I}\Delta_0(\exp)$. Therefore by iterating the above procedure i times starting with $j = 2$, we obtain successively the formulas A^+, A^{++}, ... $A^{(i)}$ where $A^{(i)}$ is Π_{i+2} and

$$\mathrm{I}\Sigma_{i+1} \vdash \mathrm{Prog}_v F(v) \to \mathrm{Prog}_u A^{(i)}(u).$$

Now fix any $\alpha < \varepsilon_0(i)$ and choose k so that $\alpha \leq \varepsilon_0(i)(k)$. By applying $k+1$ times the progressiveness of $A^{(i)}(u)$, one obtains $A^{(i)}(\overline{k+1})$ without need of any further induction, since k is fixed. Therefore

$$\mathrm{I}\Sigma_{i+1} \vdash \mathrm{Prog}_v F(v) \to A^{(i)}(\overline{k+1}).$$

But by instantiating the outermost universally quantified variable of $A^{(i)}$ to zero we have $A^{(i)}(\overline{k+1}) \to A^{(i-1)}(\omega^{\overline{k+1}})$. Again instantiating to zero the outermost universally quantified variable in $A^{(i-1)}$ we similarly obtain $A^{(i-1)}(\omega^{\overline{k+1}}) \to A^{(i-2)}(\omega^{\omega^{\overline{k+1}}})$. Continuing in this way, and noting that $\varepsilon_0(i)(k)$ consists of an exponential stack of i ω's with $k+1$ on the top, we finally get down (after i steps) to

$$\mathrm{I}\Sigma_{i+1} \vdash \mathrm{Prog}_v F(v) \to A(\overline{\varepsilon_0(i)(k)}).$$

Since $A(\overline{\varepsilon_0(i)(k)})$ is just $\forall_{v \prec \overline{\varepsilon_0(i)(k)}} F(v)$ we have therefore proved, in $\mathrm{I}\Sigma_{i+1}$, transfinite induction for F up to $\varepsilon_0(i)(k)$, and hence up to the given α.

Theorem 3.25 *For each i and every $\alpha < \varepsilon_0(i)$, the fast growing function F_α is provably recursive in $\mathrm{I}\Sigma_{i+1}$.*

Proof. If $i = 0$ then α is finite and F_α is therefore primitive recursive, and so provably recursive in $\mathrm{I}\Sigma_1$.

Now suppose $i > 0$. Since $F_\alpha = H_{\omega^\alpha}$ we need only show, for every $\alpha < \varepsilon_0(i)$, that H_{ω^α} is provably recursive in $\mathrm{I}\Sigma_{i+1}$. But a lemma above shows that its defining Π_2 formula $H(\omega^a)$ is provably progressive in $\mathrm{I}\Sigma_2$, and therefore by the Gentzen result,

$$\mathrm{I}\Sigma_{i+1} \vdash \forall_{a \prec \bar{\alpha}} H(\omega^a).$$

One further application of progressiveness then gives

$$\mathrm{I}\Sigma_{i+1} \vdash H(\omega^{\bar{\alpha}})$$

which, together with the definability of H_α proved above, completes the provable Σ_1-definability of H_{ω^α} in $\mathrm{I}\Sigma_{i+1}$.

3.4. Ordinal Bounds for Provable Recursion in PA

For the converse of the above result we perform an ordinal analysis of PA-proofs in a system which allows higher levels of induction to be reduced, via cut elimination, to Σ_1-inductions. The cost of such reductions is a successive exponential increase in the ordinals involved, but in the end this enables us to read off fast growing bounding functions for provable recursion.

It would be naive to try to carry through cut elimination directly on PA-proofs since the inductions would get in the way. Instead, the trick is to unravel the inductions by means of the ω-rule: from the infinite sequence of premises $\{\, A(n) \mid n \in \mathbb{N} \,\}$ derive $\forall_x A(x)$. The disadvantage is that this embeds PA into a "semi-formal" system with an infinite rule, so proofs will now be well-founded trees with ordinals measuring their heights. The advantage is that this system admits cut elimination, and furthermore it bears a close relationship with the fast growing hierarchy, as we shall see.

3.5. The infinitary system $n\colon N \vdash^\alpha \Gamma$

We shall inductively generate, according to the rules below, an infinitary system of (classical) one-sided sequents $n\colon N \ \vdash^\alpha \ \Gamma$ in Tait–style (i.e., with negation of compound formulas defined by de Morgan's laws) where:

(i) $n\colon N$ is a new kind of atomic formula, declaring a bound on numerical "inputs" from which terms appearing in Γ are computed according to the N-rules and axioms.

(ii) Γ is any finite set of closed formulas, either of the form $m : N$, or else formulas in the language of arithmetic based on $\{=, 0, S, P, +, \dot{-}, \cdot, \exp_2\}$, possibly with the addition of any number of further primitive-recursively-defined function symbols. Recall that Γ, A denotes the set $\Gamma \cup \{A\}$ etc.

(iii) Ordinals $\alpha, \beta, \gamma < \varepsilon_0$ denote bounds on the heights of derivations, assigned in a carefully controlled way due originally to Buchholz [1] (see also [2]) though modified somewhat here. Essentially, the condition is that if a sequent with bound α is derived from a premise with bound β then $\beta \in \alpha[n]$ where n is the declared input bound.

(iv) Any occurrence of a number n in a formula should of course be read as its corresponding numeral, but we need not introduce explicit notation for this since the intention will be clear in context.

The first axiom and rule are "computation rules" for N, and the rest are just formalised versions of the truth definition, with Cut added.

$(N1)$ For arbitrary α,

$$n\colon N \ \vdash^\alpha \ \Gamma, \ m\colon N \ \text{ provided } m \leq n+1$$

$(N2)$ For $\beta, \beta' \in \alpha[n]$,

$$\frac{n\colon N \ \vdash^\beta \ n'\colon N \quad n'\colon N \ \vdash^{\beta'} \ \Gamma}{n\colon N \ \vdash^\alpha \ \Gamma}$$

(Ax) If Γ contains a true atom (i.e., an equation or inequation between closed terms) then for arbitrary α,

$$n{:}\, N \ \vdash^{\alpha} \ \Gamma$$

($\vee$) For $\beta \in \alpha[n]$,

$$\frac{n{:}\, N \ \vdash^{\beta} \ \Gamma,\ A, B}{n{:}\, N \ \vdash^{\alpha} \ \Gamma, A \vee B}$$

($\wedge$) For $\beta, \beta' \in \alpha[n]$

$$\frac{n{:}\, N \ \vdash^{\beta} \ \Gamma,\ A \quad n{:}\, N \ \vdash^{\beta'} \ \Gamma,\ B}{n{:}\, N \ \vdash^{\alpha} \ \Gamma,\ A \wedge B}$$

($\exists$) For $\beta, \beta' \in \alpha[n]$,

$$\frac{n{:}\, N \ \vdash^{\beta} \ m{:}\, N \quad n{:}\, N \ \vdash^{\beta'} \ \Gamma,\ A(m)}{n{:}\, N \ \vdash^{\alpha} \ \Gamma,\ \exists_x A(x)}$$

($\forall$) Provided $\beta_i \in \alpha[\max(n, i)]$ for every i,

$$\frac{\max(n, i){:}\, N \ \vdash^{\beta_i} \ \Gamma,\ A(i) \ \text{ for every } i \in N}{n{:}\, N \ \vdash^{\alpha} \ \Gamma,\ \forall_x A(x)}$$

(Cut) For $\beta, \beta' \in \alpha[n]$,

$$\frac{n{:}\, N \ \vdash^{\beta} \ \Gamma,\ C \quad n{:}\, N \ \vdash^{\beta'} \ \Gamma,\ \neg C}{n{:}\, N \ \vdash^{\alpha} \ \Gamma}$$

(C is called the "cut formula").

Definition 3.26 The functions B_α are defined by the recursion:

$$B_0(n) = n + 1, \quad B_{\alpha+1}(n) = B_\alpha(B_\alpha(n)), \quad B_\lambda(n) = B_{\lambda(n)}(n)$$

where λ denotes any limit ordinal with assigned fundamental sequence $\lambda(n)$.

Note 3.27 Since, at successor stages, B_α is just composed with itself once, an easy comparison with the fast growing F_α shows that $B_\alpha(n) \leq F_\alpha(n)$ for all $n > 0$. It is also easy to see that for each positive integer k, $B_{\omega\cdot k}(n)$ is the 2^{n+1}-times iterate of $B_{\omega\cdot(k-1)}$ on n. Thus another comparison with the definition of F_k shows that $F_k(n) \leq B_{\omega\cdot k}(n)$ for all n. Thus every primitive recursive function is bounded by a $B_{\omega\cdot k}$ for some k. Furthermore, just as for H_α and F_α, B_α is strictly increasing and $B_\beta(n) < B_\alpha(n)$ whenever $\beta \in \alpha[n]$. The next two lemmas show that these functions B_α are intimately related with the infinitary system we have just set up.

Lemma 3.28 *$m \leq B_\alpha(n)$ if and only if $n{:}\, N \ \vdash^{\alpha} \ m{:}\, N$ is derivable by the $N1$ and $N2$ rules only.*

Proof. For the "if" part, note that the proviso on the axiom $N1$ is that $m \leq n+1$ and therefore $m \leq B_\alpha(n)$ is automatic. Secondly if $n\colon N \ \vdash^\alpha \ m\colon N$ arises by the $N2$ rule from premises $n\colon N \ \vdash^\beta \ n'\colon N$ and $n'\colon N \ \vdash^{\beta'} \ m\colon N$ where $\beta, \beta' \in \alpha[n]$ then, assuming inductively that $m \leq B_{\beta'}(n')$ and $n' \leq B_\beta(n)$, we have $m \leq B_{\beta'}(B_\beta(n))$ and hence $m \leq B_\alpha(n)$.

For the "only if" proceed by induction on α, assuming $m \leq B_\alpha(n)$. If $\alpha = 0$ then $m \leq n+1$ and so $n\colon N \ \vdash^\alpha \ m\colon N$ by $N1$. If $\alpha = \beta + 1$ then $m \leq B_\beta(n')$ where $n' = B_\beta(n)$, so by the induction hypothesis, $n\colon N \ \vdash^\beta \ n'\colon N$ and $n'\colon N \ \vdash^\beta \ m\colon N$. Hence $n\colon N \ \vdash^\alpha \ m\colon N$ by $N2$ since $\beta \in \alpha[n]$. Finally, if α is a limit then $m \leq B_{\alpha(n)}(n)$ and so $n\colon N \ \vdash^{\alpha(n)} \ m\colon N$ by the induction hypothesis. But since $\alpha[n] = \alpha(n)[n]$ the ordinal bounds β on the premises of this last derivation also lie in $\alpha[n]$, which means that $n\colon N \ \vdash^\alpha \ m\colon N$ as required.

Definition 3.29 A sequent $n\colon N \ \vdash^\alpha \ \Gamma$ is said to be *term controlled* if every closed term occuring in Γ has numerical value bounded by $B_\alpha(n)$. An infinitary derivation is then *term controlled* if every one of its sequents is term controlled.

Note 3.30 For a derivation to be term controlled it is sufficient that each axiom is term controlled, since in any rule, the closed terms occuring in the conclusion must already occur in a premise (in the case of the $\forall$ rule, the premise $i = 0$). Thus if α is the ordinal bound on the conclusion, every such closed term is bounded by a $B_\beta(n)$ for some $\beta \in \alpha[n]$ and hence is bounded by $B_\alpha(n)$ as required.

Lemma 3.31 (Bounding Lemma) *Let Γ be a set of Σ_1-formulas or atoms of the form $m\colon N$. If $n\colon N \ \vdash^\alpha \ \Gamma$ has a term controlled derivation in which all cut formulas are Σ_1, then Γ is true at $B_{\alpha+1}(n)$. Here, the definition of "true at" is extended to include atoms $m\colon N$ by saying that $m\colon N$ is true at k if $m < k$.*

Proof. By induction over α according to the generation of the sequent $n\colon N \ \vdash^\alpha \ \Gamma$, which we shall denote by S.

(Axioms) If S is either a logical axiom or of the form $N1$ then Γ contains either a true atomic equation or inequation, or else an atom $m\colon N$ where $m < n+2$, so Γ is automatically true at $B_{\alpha+1}(n)$.

($N2$) If S arises by the $N2$ rule from premises $n\colon N \ \vdash^\beta \ n'\colon N$ and $n'\colon N \ \vdash^{\beta'} \ \Gamma$ where $\beta, \beta' \in \alpha[n]$ then, by the induction hypothesis, Γ is true at $B_{\beta'+1}(n')$ where $n' < B_{\beta+1}(n)$. Therefore by persistence, Γ is true at $B_{\beta'+1}(B_{\beta+1}(n))$ which is less than or equal to $B_\alpha(B_\alpha(n)) = B_{\alpha+1}(n)$. So by persistence again, Γ is true at $B_{\alpha+1}(n)$.

($\vee$, $\wedge$) Because of our definition of Σ_1-formulas, the $\vee$ and $\wedge$ rules only apply to bounded ($\Delta_0(\exp)$) formulas, so the result is immediate in these cases (by persistence and the fact that the rules preserve truth).

($\forall$) Similarly, the only way in which the $\forall$ rule can be applied is in a bounded context, where $\Gamma = \Gamma', \forall x(x \not< t \vee A(x))$, t is a closed term, and $A(x)$ a bounded formula. Suppose then, that S arises by the $\forall$ rule from premises $\max(n,i)\colon N \ \vdash^{\beta_i} \ \Gamma',\ i \not< t \vee A(i)$ where $\beta_i \in \alpha[\max(n,i)]$ for every i. Since the derivation is term controlled we know that (the numerical value of) t is less than or equal to $B_\alpha(n)$. Therefore by the induction hypothesis and persistence again: for every $i < t$, the set $\Gamma', A(i)$ is true at $B_{\beta_i+1}(B_\alpha(n))$. But $\beta_i \in \alpha[B_\alpha(n)]$ and so $B_{\beta_i+1}(B_\alpha(n)) \leq B_\alpha(B_\alpha(n)) = B_{\alpha+1}(n)$. Hence Γ is true at $B_{\alpha+1}(n)$ using persistence once more.

($\exists$) If Γ contains a Σ_1-formula $\exists_x A(x)$ and S arises by the $\exists$ rule from premises $n{:}\,N \vdash^{\beta} m{:}\,N$ and $n{:}\,N \vdash^{\beta'} \Gamma,\ A(m)$ then by the induction hypothesisis, $\Gamma, A(m)$ is true at $B_{\beta'+1}(n)$ where $m < B_{\beta+1}(n)$. Therefore, by the definition of "true at", Γ is true at whichever is the greater of $B_{\beta+1}(n)$ and $B_{\beta'+1}(n)$. But since $\beta, \beta' \in \alpha[n]$ both of these are less than $B_{\alpha+1}(n)$, so Γ is again true at $B_{\alpha+1}(n)$.

(Cut) Finally suppose S comes about by a cut on the Σ_1 formula $C \equiv \exists_{\vec{x}} D(\vec{x})$ with D bounded. Then the premises are $n{:}\,N \vdash \Gamma,\ C$ and $n{:}\,N \vdash \Gamma,\ \neg C$ with ordinal bounds $\beta, \beta' \in \alpha[n]$ respectively. By the induction hypothesis applied to the first premise, we have numbers $\vec{m} < B_{\beta+1}(n)$ such that $\Gamma, D(\vec{m})$ is true at $B_{\beta+1}(n)$. By inverting the universal quantifiers in $\neg C \equiv \forall_{\vec{x}} \neg D(\vec{x})$, the second premise gives $\max(n, \vec{m}){:}\,N \vdash^{\beta'} \Gamma,\ \neg D(\vec{m})$. Then by the induction hypothesis (since $\Gamma, \neg D(\vec{m})$ is now a set of Σ_1-formulas) we have $\Gamma, \neg D(\vec{m})$ true at $B_{\beta'+1}(\max(n, \vec{m}))$, which is less than $B_{\beta'+1}(B_{\beta+1}(n))$, which is less than or equal to $B_{\alpha+1}(n)$. Therefore (by persistence) Γ must be true at $B_{\alpha+1}(n)$, for otherwise both $D(\vec{m})$ and $\neg D(\vec{m})$ would be true, and this cannot be.

3.6. Embedding of PA

The Bounding Lemma above becomes applicable to PA if we can embed it into the infinitary system and then (as done in the next sub-section) reduce all the cuts to Σ_1 form. This is standard proof-theoretic procedure. First, comes a simple technical lemma which will be needed frequently.

Lemma 3.32 (Weakening) *If $n{:}\,N \vdash^{\alpha} \Gamma$ and $n \leq n'$ and $\Gamma \subseteq \Gamma'$ and $\alpha[m] \subseteq \alpha'[m]$ for every $m \geq n'$ then $n'{:}\,N \vdash^{\alpha'} \Gamma'$. Furthermore, if the given derivation of $n{:}\,N \vdash^{\alpha} \Gamma$ is term controlled then so will be the derivation of $n'{:}\,N \vdash^{\alpha'} \Gamma'$ provided of course, that all the closed terms occurring in Γ' are bounded by $B_{\alpha'}(n')$.*

Proof. Proceed by induction on α. Note first that if $n{:}\,N \vdash^{\alpha} \Gamma$ is an axiom then Γ, and hence also Γ', contains either a true atom or a declaration $m{:}\,N$ where $m \leq n+1$. Thus $n'{:}\,N \vdash^{\alpha'} \Gamma'$ is an axiom also.

($N2$) If $n{:}\,N \vdash^{\alpha} \Gamma$ arises by the $N2$ rule from premises $n{:}\,N \vdash^{\beta} m{:}\,N$ and $m{:}\,N \vdash^{\beta'} \Gamma$ where $\beta, \beta' \in \alpha[n]$ then, by applying the induction hypothesis to each of these, n can be increased to n' in the first, and Γ can be increased to Γ' in the second. But then since $\alpha[n] \subseteq \alpha[n'] \subseteq \alpha'[n']$ the rule $N2$ can be re-applied to yield the desired $n'{:}\,N \vdash^{\alpha'} \Gamma'$.

($\exists$) If $n{:}\,N \vdash^{\alpha} \Gamma$ arises by the $\exists$ rule from premises $n{:}\,N \vdash^{\beta} m{:}\,N$ and $n{:}\,N \vdash^{\beta'} \Gamma, A(m)$ where $\exists_x A(x) \in \Gamma$ and $\beta, \beta' \in \alpha[n]$ then, by applying the induction hypothesis to each premise, n can be increased to n' and Γ increased to Γ'. The $\exists$ rule can then be re-applied to yield the desired $n'{:}\,N \vdash^{\alpha'} \Gamma'$, since as above, $\beta, \beta' \in \alpha'[n']$.

($\forall$) Suppose $n{:}\,N \vdash^{\alpha} \Gamma$ arises by the $\forall$ rule from premises

$$\max(n, i){:}\,N \vdash^{\beta_i} \Gamma, A(i)$$

where $\forall_x A(x) \in \Gamma$ and $\beta_i \in \alpha[\max(n, i)]$ for every i. Then, by applying the induction hypothesis to each of these premises, n can be increased to n' and Γ increased to Γ'. The $\forall$ rule can then be re-applied to yield the desired $n'{:}\,N \vdash^{\alpha'} \Gamma'$, since for each i, $\beta_i \in \alpha[\max(n', i)] \subseteq \alpha'[\max(n', i)]$.

The remaining rules $\vee$, $\wedge$ and Cut, are handled easily by increasing n to n' and Γ to Γ' in the premises, and then re-applying the rule.

Theorem 3.33 (Embedding) *Suppose* PA $\vdash \Gamma(x_1, \ldots, x_k)$ *where* $x_1, \ldots, x_k$ *are all the free variables occurring in* Γ. *Then there is a fixed number* d *such that, for all numerical instantiations* $n_1, n_2, \ldots, n_k$ *of the free variables, we have a term controlled derivation of*

$$\max(n_1, n_2, \ldots, n_k)\colon N \ \vdash^{\omega \cdot d} \ \Gamma(n_1, n_2, \ldots, n_k).$$

Furthermore, the (non-atomic) cut formulas occurring in this derivation are just the induction formulas which occur in the original PA *proof.*

Proof. We work with a Tait-style formalisation of PA in which the induction axioms are replaced by corresponding rules:

$$\frac{\Gamma, A(0) \quad \Gamma, \neg A(z), A(z+1)}{\Gamma, A(t)}$$

with z not free in Γ and t any term. As in the case of $\mathrm{I}\Delta_0(\exp)$ we may suppose that the given PA-proof of $\Gamma(\vec{x}\,)$ has been reduced to free-cut-free form, wherein the only non-atomic cut formulas are the induction formulas. We simply have to transform each step of this PA-proof into an appropriate, term controlled infinitary derivation.

(Axioms) If $\Gamma(\vec{x}\,)$ is an axiom of PA then with $\vec{n} = n_1, n_2, \ldots, n_k$ substituted for the variables $\vec{x} = x_1, x_2, \ldots, x_k$, there must occur a true atom in $\Gamma(\vec{n})$. Thus we automatically have a derivation of $\max \vec{n}\colon N \ \vdash^{\alpha} \ \Gamma(\vec{n})$ for arbitrary α. However we must choose α appropriately so that, for all $\vec{n}$, this sequent is term controlled. To do this, simply note that, since PA only has primitive-recursively-defined function constants, every one of the (finitely many) terms $t(\vec{x}\,)$ appearing in $\Gamma(\vec{x}\,)$ is primitive recursive, and therefore there is a number d such that for all $\vec{n}$, $B_{\omega \cdot d}(\max \vec{n})$ bounds the value of every such $t(\vec{n})$. So choose $\alpha = \omega \cdot d$.

($\vee$, $\wedge$, Cut) If $\Gamma(\vec{x}\,)$ arises by a $\vee$, $\wedge$ or cut rule from premises $\Gamma_0(\vec{x}\,)$ and $\Gamma_1(\vec{x}\,)$ then, inductively, we can assume that we already have infinitary derivations of $\max \vec{n}\colon N \ \vdash^{\omega \cdot d_0} \ \Gamma_0(\vec{n})$ and $\max \vec{n}\colon N \ \vdash^{\omega \cdot d_1} \ \Gamma_1(\vec{n})$ where d_0 and d_1 are independent of $\vec{n}$. So choose $d = \max(d_0, d_1) + 1$ and note that $\omega \cdot d_0$ and $\omega \cdot d_1$ both belong to $\omega \cdot d[\max \vec{n}]$. Then by re-applying the corresponding infinitary rule, we obtain $\max \vec{n}\colon N \ \vdash^{\omega \cdot d} \ \Gamma(\vec{n})$ as required, and this derivation will again be term controlled provided the premises were.

($\forall$) Suppose $\Gamma(\vec{x}\,)$ arises by an application of the $\forall$ rule from the premise $\Gamma_0(\vec{x}\,), A(\vec{x}, z)$ where $\Gamma = \Gamma_0, \forall_z A(\vec{x}, z)$. Assume that we already have a d_0 such that for all $\vec{n}$ and all m, there is a term controlled derivation of $\max(\vec{n}, m)\colon N \ \vdash^{\omega \cdot d_0} \Gamma_0(\vec{n}), A(\vec{n}, m)$. Then with $d = d_0 + 1$ we have $\omega \cdot d_0 \in \omega \cdot d[\max(\vec{n}, m)]$, and so an application of the infinitary $\forall$ rule immediately gives $\max \vec{n}\colon N \ \vdash^{\omega \cdot d} \ \Gamma(\vec{n})$. This is also term controlled because any closed term appearing in $\Gamma(\vec{n})$ must appear in $\Gamma_0(\vec{n}), A(\vec{n}, 0)$ and so is already bounded by $B_{\omega \cdot d_0}(\max \vec{n})$.

($\exists$) Suppose $\Gamma(\vec{x}\,)$ arises by an application of the $\exists$ rule from the premise $\Gamma_0(\vec{x}\,), A(\vec{x}, t(\vec{x}\,))$ where $\Gamma = \Gamma_0, \exists_z A(\vec{x}, z)$. If the witnessing term t contains any other variables besides $x_1, \ldots, x_k$ we can assume they have been substituted by zero. Thus by the induction we have, for every $\vec{n}$, a term controlled derivation of $\max \vec{n}\colon N \ \vdash^{\omega \cdot d_0}$

$\Gamma_0(\vec{n}), A(\vec{n}, t(\vec{n}))$ for some fixed d_0 independent of $\vec{n}$. Now it is easy to see, by checking through the rules, that any occurrences of the term $t(\vec{n})$ may be replaced by (the numeral for) its value, say m. Furthermore, because the derivation is term controlled, $m \leq B_{\omega \cdot d_0}(\max n)$ and hence $\max \vec{n}: N \vdash^{\omega \cdot d_0} m: N$. Therefore by the infinitary $\exists$ rule we immediately obtain $\max \vec{n}: N \vdash^{\omega \cdot d} \Gamma_0(\vec{n}), \exists_z A(\vec{n}, z)$ where $d = d_0 + 1$, and this derivation is again term controlled.

(Induction) Finally, suppose $\Gamma(\vec{x}) = \Gamma_0(\vec{x}), A(\vec{x}, t(\vec{x}))$ arises by the induction rule from premises $\Gamma_0(\vec{x}), A(\vec{x}, 0)$ and $\Gamma_0(\vec{x}), \neg A(\vec{x}, z), A(\vec{x}, z+1)$. Assume inductively, that we have d_0 and d_1 and, for all $\vec{n}$ and all i, term controlled derivations of

$$\max \vec{n}: N \vdash^{\omega \cdot d_0} \Gamma_0(\vec{n}), A(\vec{n}, 0)$$

$$\max(\vec{n}, i): N \vdash^{\omega \cdot d_1} \Gamma_0(\vec{n}), \neg A(\vec{n}, i), A(\vec{n}, i+1).$$

Now let d_2 be any number $\geq \max(d_0, d_1)$ and such that $B_{\omega \cdot d_2}$ bounds every subterm of $t(\vec{x})$ (again there is such a d_2 because every subterm of t defines a primitive recursive function of its variables). Then for all $\vec{n}$, if m is the numerical value of the term $t(\vec{n})$ we have a term controlled derivation of

$$\max(\vec{n}, m): N \vdash^{\omega \cdot (d_2+1)} \Gamma_0(\vec{n}), A(\vec{n}, m).$$

For, in the case $m = 0$ this follows immediately from the first premise above by weakening the ordinal bound; and if $m > 0$ then by successive cuts on $A(\vec{n}, i)$ for $i = 0, 1, \ldots, m-1$, with weakenings where necessary, we obtain first a term controlled derivation of

$$\max(\vec{n}, m): N \vdash^{\omega \cdot d_2 + m} \Gamma_0(\vec{n}), A(\vec{n}, m)$$

and then, since $m \in \omega[\max(\vec{n}, m)]$, another weakening provides the desired ordinal bound $\omega \cdot (d_2 + 1)$.

Since, by our choice of d_2, $\max(\vec{n}, m) \leq B_{\omega \cdot d_2}(\max \vec{n})$ we also have

$$\max \vec{n}: N \vdash^{\omega \cdot d_2} \max(\vec{n}, m): N$$

and so, combining this with the sequent just derived, the $N2$ rule gives

$$\max \vec{n}: N \vdash^{\omega \cdot (d_2+2)} \Gamma_0(\vec{n}), A(\vec{n}, m).$$

It therefore only remains to replace the numeral m by the term $t(\vec{n})$, whose value it is. But it is easy to check, by induction over the logical structure of formula A, that provided d_2 is in addition chosen to be at least twice the height of the formation tree of A, then for all $\vec{n}$ there is a cut-free derivation of

$$\max \vec{n}: N \vdash^{\omega \cdot d_2} \Gamma_0(\vec{n}), \neg A(\vec{n}, m), A(\vec{n}, t(\vec{n})).$$

Therefore, fixing d_2 accordingly and setting $d = d_2 + 3$, a final cut on the formula $A(\vec{n}, m)$ yields the desired term controlled derivation, for all $\vec{n}$, of

$$\max \vec{n}: N \vdash^{\omega \cdot d} \Gamma_0(\vec{n}), A(\vec{n}, t(\vec{n})).$$

This completes the induction case, and hence the proof, noting that the only non-atomic cuts introduced are on induction formulas.

3.7. Cut elimination

Once a PA proof is embedded in the infinitary system, we need to reduce the cut complexity before the Bounding Lemma becomes applicable. As we shall see, this entails an iterated exponential increase in the original ordinal bound. Thus ε_0, the first exponentially-closed ordinal after ω, is a measure of the proof-theoretic complexity of PA.

Lemma 3.34 ($\forall$-Inversion) *If* $n\colon N \vdash^{\alpha} \Gamma, \forall_a A(a)$ *then* $\max(n,m)\colon N \vdash^{\alpha} \Gamma, A(m)$ *for every* m. *Furthermore if the given derivation is term controlled, so is the resulting one.*

Proof. We proceed by induction on α. Note first that if the sequent $n\colon N \vdash^{\alpha} \Gamma, \forall_a A(a)$ is an axiom then so is $n\colon N \vdash^{\alpha} \Gamma$ and then the desired result follows immediately by weakening.

Suppose $n\colon N \vdash^{\alpha} \Gamma, \forall_a A(a)$ is the consequence of a $\forall$ rule with $\forall_a A(a)$ the "main formula" proven. Then the premises are, for each i,

$$\max(n,i)\colon N \vdash^{\beta_i} \Gamma, A(i), \forall_a A(a)$$

where $\beta_i \in \alpha[\max(n,i)]$. So by applying the induction hypothesis to the case $i = m$ one immediately obtains $\max(n,m)\colon N \vdash^{\beta_m} \Gamma, A(m)$. Weakening then allows the ordinal bound β_m to be increased to α.

In all other cases the formula $\forall_a A(a)$ is a "side formula" occurring in the premise(s) of the final rule applied. So by the induction hypothesis, $\forall_a A(a)$ can be replaced by $A(m)$ and n by $\max(n,m)$. The result then follows by re-applying that final rule.

Note that each transformation preserves term control.

Definition 3.35 We insert a subscript "Σ_r" on the proof-gate thus:

$$n\colon N \vdash^{\alpha}_{\Sigma_r} \Gamma$$

to indicate that, in the infinitary derivation, all cut formulas are of the form Σ_i or Π_i where $i \leq r$.

Lemma 3.36 (Cut Reduction) *Suppose* $n\colon N \vdash^{\alpha}_{\Sigma_r} \Gamma, C$ *and* $n\colon N \vdash^{\gamma}_{\Sigma_r} \Gamma', \neg C$ *where* $r \geq 1$ *and* C *is a* Σ_{r+1} *formula. Suppose also that* $\alpha[n'] \subseteq \gamma[n']$ *for all* $n' \geq n$. *Then*

$$n\colon N \vdash^{\gamma+\alpha}_{\Sigma_r} \Gamma, \Gamma'.$$

Furthermore, if the given derivations are term controlled, so is the resulting one.

Proof. We proceed by induction on α according to the derivation of $n\colon N \vdash^{\alpha}_{\Sigma_r} \Gamma, C$. If this is an axiom then C, being non-atomic, can be deleted and it's still an axiom, and so is $n\colon N \vdash^{\gamma+\alpha}_{\Sigma_r} \Gamma, \Gamma'$. Furthermore this sequent is term controlled if the given ones are, since $B_{\gamma+\alpha}(n)$ is greater than or equal to $B_{\gamma}(n)$ and $B_{\alpha}(n)$.

Now suppose C is the "main formula" proven in the final rule of the derivation. Since $C \equiv \exists_x D(x)$ with D a Π_r formula, this final rule is an $\exists$ rule with premises $n\colon N \vdash^{\beta_0}_{\Sigma_r} m\colon N$ and $n\colon N \vdash^{\beta_1}_{\Sigma_r} \Gamma, D(m), C$ where $\beta_0, \beta_1 \in \alpha[n] \subseteq \gamma[n]$. By the induction hypothesis we then have

$$n{:}\, N \vdash^{\gamma+\beta_1}_{\Sigma_r} \Gamma, D(m), \Gamma' \quad (*)$$

Since $\neg C \equiv \forall_x \neg D(x)$ we can apply $\forall$-inversion to the given derivation of $n{:}\, N \vdash^{\gamma}_{\Sigma_r} \Gamma', \neg C$ to obtain $\max(n,m){:}\, N \vdash^{\gamma}_{\Sigma_r} \Gamma', \neg D(m)$ as inversion does not affect the cut formulas. Hence by the $N2$ rule, using $n{:}\, N \vdash^{\beta_0}_{\Sigma_r} m{:}\, N$ and a weakening,

$$n{:}\, N \vdash^{\gamma+\beta_1}_{\Sigma_r} \Gamma, \neg D(m), \Gamma' \quad (**)$$

Then from $(*)$ and $(**)$ a cut on $D(m)$ gives the desired result:

$$n{:}\, N \vdash^{\gamma+\alpha}_{\Sigma_r} \Gamma, \Gamma'.$$

Notice, however, that $(**)$ requires β_1 to be nonzero so that $\gamma \in \gamma + \beta_1[n]$. If, on the other hand, $\beta_1 = 0$ then either $n{:}\, N \vdash^{\beta_1}_{\Sigma_r} \Gamma$ is an axiom or else $D(m)$ is a true atom, in which case $\neg D(m)$ may be deleted from $\max(n,m){:}\, N \vdash^{\gamma}_{\Sigma_r} \Gamma', \neg D(m)$ and then, by $N2$, $n{:}\, N \vdash^{\gamma+\alpha}_{\Sigma_r} \Gamma'$. Whichever is the case, the desired result follows immediately by weakening.

Finally suppose otherwise, i.e., C is a "side formula" in the final rule of the derivation of $n{:}\, N \vdash^{\alpha}_{\Sigma_r} \Gamma, C$. Then by applying the induction hypothesis to the premise(s), C gets replaced by Γ' and the ordinal bounds β are replaced by $\gamma + \beta$. Re-application of that final rule then yields $n{:}\, N \vdash^{\gamma+\alpha}_{\Sigma_r} \Gamma, \Gamma'$ as required.

It is clear, at each step, that the new derivations introduced are term controlled provided that the assumed ones are.

Theorem 3.37 (Cut Elimination) *If* $n{:}\, N \vdash^{\alpha}_{\Sigma_{r+1}} \Gamma$ *where* $n \geq 1$ *then*

$$n{:}\, N \vdash^{\omega^\alpha}_{\Sigma_r} \Gamma.$$

Furthermore, if the given derivation is term controlled so is the resulting one.

Proof. Proceeding by induction on α, first suppose $n{:}\, N \vdash^{\alpha}_{\Sigma_{r+1}} \Gamma$ comes about by a cut on a Σ_{r+1} or Π_{r+1} formula C. Then the premises are $n{:}\, N \vdash^{\beta_0}_{\Sigma_{r+1}} \Gamma, C$ and $n{:}\, N \vdash^{\beta_1}_{\Sigma_{r+1}} \Gamma, \neg C$ where $\beta_0, \beta_1 \in \alpha[n]$. By an appropriate weakening we may increase whichever is the smaller of β_0, β_1 so that both ordinal bounds become $\beta = \max(\beta_0, \beta_1)$. Applying the induction hypothesis we obtain

$$n{:}\, N \vdash^{\omega^\beta}_{\Sigma_r} \Gamma, C \quad \text{and} \quad n{:}\, N \vdash^{\omega^\beta}_{\Sigma_r} \Gamma, \neg C.$$

Then since one of $C, \neg C$ is Σ_{r+1}, the above Cut Reduction Lemma with $\alpha = \gamma = \omega^\beta$ yields

$$n{:}\, N \vdash^{\omega^\beta \cdot 2}_{\Sigma_r} \Gamma.$$

But $\beta \in \alpha[n]$ and so $\omega^\beta \cdot 2[m] \subseteq \omega^\alpha[m]$ for every $m \geq n$. Therefore by weakening, $n{:}\, N \vdash^{\omega^\alpha}_{\Sigma_r} \Gamma$.

Now suppose $n{:}\, N \vdash^{\alpha}_{\Sigma_{r+1}} \Gamma$ arises by any rule (or axiom) other than a cut on a Σ_{r+1} or Π_{r+1} formula. First, apply the induction hypothesis to the premises (if any), thus reducing $r+1$ to r and increasing ordinal bounds β to ω^β, and then re-apply that final rule to obtain $n{:}\, N \vdash^{\omega^\alpha}_{\Sigma_r} \Gamma$, noting that if $\beta \in \alpha[n]$ then $\omega^\beta \in \omega^\alpha[n]$ provided $n \geq 1$. Note again, that the resulting derivation is term controlled if the original one is.

Theorem 3.38 (Preliminary Cut Elimination) *If* $n\colon N \vdash^{\omega\cdot d+c}_{\Sigma_{r+1}} \Gamma$ *with* $r \geq 1$ *and* $n \geq 1$, *then*

$$n\colon N \vdash^{\omega^d\cdot 2^{c+1}}_{\Sigma_r} \Gamma$$

and this derivation is term controlled if the first one is.

Proof. This is just a special case of the main Cut Elimination Theorem above, where $\alpha < \omega^2$. Essentially the same steps are applied, but with a few extra technicalities.

3.8. The classification theorem

Theorem 3.39 *For each* i *the following are equivalent:*

1. f *is provably recursive in* $\mathrm{I}\Sigma_{i+1}$*;*
2. f *is elementarily definable from* $F_\alpha = H_{\omega^\alpha}$ *for some* $\alpha < \varepsilon_0(i)$.

Proof. We have already shown that if $\alpha < \varepsilon_0(i)$ then F_α is provably recursive in $ISigma_{i+1}$, and since $IDelta_0(\exp)$ provides closure under elementary definability it follows that every function elementarily definable from F_α is provably recursive in $ISigma_{i+1}$.

Conversely suppose that $f\colon \mathbb{N}^k \to \mathbb{N}$ is provably recursive in $\mathrm{I}\Sigma_{i+1}$. Then there is a Σ_1-formula $F(\vec{x}, y)$ such that for all $\vec{n}$ and m, $f(\vec{n}) = m$ if and only if $F(\vec{n}, m)$ is true, and such that

$$\mathrm{I}\Sigma_{i+1} \vdash \exists_y F(\vec{x}, y).$$

By the Embedding Theorem there is a fixed number d and, for all instantiations $\vec{n}$ of the variables $\vec{x}$, a term controlled derivation of

$$\max \vec{n}\colon N \vdash^{\omega\cdot d}_{\Sigma_{i+1}} \exists_y F(\vec{n}, y).$$

Suppose $i > 0$. Let $n = \max(1, \max \vec{n})$. Then by the Preliminary Cut Elimination Theorem with $c = 0$,

$$n\colon N \vdash^{\omega^d\cdot 2}_{\Sigma_i} \exists_y F(\vec{n}, y)$$

and by weakening, since $\omega^d \cdot 2[m] \subseteq \omega^{d+1}[m]$ for all $m \geq n$,

$$n\colon N \vdash^{\omega^{d+1}}_{\Sigma_i} \exists_y F(\vec{n}, y).$$

Now, if $i > 1$, apply the ordinary Cut Elimination Theorem $i - 1$ times, bringing the cuts down to the Σ_1 level and simultaneously increasing the ordinal bound ω^{d+1} by $i-1$ iterated exponentiations to the base ω. This produces

$$n\colon N \vdash^{\alpha}_{\Sigma_1} \exists_y F(\vec{n}, y)$$

with ordinal bound $\alpha < \varepsilon_0(i)$ (recalling that, as defined earlier, $\varepsilon_0(i)$ consists of an exponential stack of $i + 1$ ω's). Since this last derivation is still term controlled, we can next apply the Bounding Lemma to conclude that $\exists_y F(\vec{n}, y)$ is true at $B_{\alpha+1}(n)$, which is less than or equal to $F_{\alpha+1}(n)$. This means that for all $\vec{n}$, $F_{\alpha+1}(n)$ bounds the value

m of $f(\vec{n})$ and bounds witnesses for all the existential quantifiers in the prefix of the Σ_1 defining-formula $F(\vec{n}, m)$. Thus, relative to $F_{\alpha+1}$, the defining formula is bounded and therefore elementarily decidable, and f can be defined from it by a bounded least-number operator. That is, f is elementarily definable from $F_{\alpha+1}$.

In case $i = 0$ the Bounding Lemma applies immediately, without any cut elimination, to give $\exists_y F(\vec{n}, y)$ true at $B_{\omega\cdot d+1}(n)$. But this is less than or equal to $F_{d'}(n)$ for some $d' < \omega = \varepsilon_0(0)$, and f is then elementarily definable from $F_{d'}$. This completes the proof.

Corollary 3.40 *Every function provably recursive in* $\mathrm{I}\Sigma_{i+1}$ *is bounded by an* $F_\alpha = H_{\omega^\alpha}$ *for some* $\alpha < \varepsilon_0(i)$. *Hence* $H_{\varepsilon_0(i+1)}$ *is not provably recursive in* $\mathrm{I}\Sigma_{i+1}$, *for otherwise it would dominate itself.*

4. The Paris–Harrington Independence Result for PA

If the Hardy hierarchy is extended to ε_0 itself by the definition

$$H_{\varepsilon_0}(n) = H_{\varepsilon_0(n)}(n)$$

then clearly (by what we have already done) the provable recursiveness of H_{ε_0} is a consequence of transfinite induction up to ε_0. However this function is obviously not provably recursive in PA, for if it were we would have an $\alpha < \varepsilon_0$ such that $H_{\varepsilon_0}(n) \leq H_\alpha(n)$ for all n, contradicting the fact that $\alpha \in \varepsilon_0[m]$ for some m and hence $H_\alpha(m) < H_{\varepsilon_0}(m)$. Thus, although transfinite induction up to any fixed ordinal below ε_0 is provable in PA, transfinite induction all the way up to ε_0 itself is not. This is Gentzen's result, that ε_0 is the least upper bound of the "provable ordinals" of PA. Together with the Gödel incompleteness phenomena, it forms the foundational basis of all logical independence results for PA and related theories.

The question that remained until the later 1970's was whether there might be other independence results of a more natural and clear mathematical character, i.e. genuine mathematical statements formalizable in the language of arithmetic which, though true, are not provable in PA. The first and most famous one, the Modified Finite Ramsey Theorem of Paris and Harrington [8], is treated below, but whereas their original independence proof used non-standard model theoretic ideas, the proof given here (following Graham, Rothschild and Spencer [4]) is essentially a basic, slimmed–down version of the purely combinatorial analysis by Ketonen and Solovay [6] which first established direct, refined comparisons with the Hardy hierarchy.

4.1. The Modified Finite Ramsey Theorem

The Modified Finite Ramsey Theorem of Paris and Harrington [8] is, like the Finite Ramsey Theorem, also expressible as a Π^0_2-formula, but its growth rate is much higher and it is now independent of full Peano Arithmetic. Their modification is to replace the requirement that the finite homogeneous set Y has cardinality at least k, by the requirement that Y is "*large*" in the sense that its cardinality is at least as big as its smallest element, i.e., $|Y| \geq \min Y$. (Thus $\{5, 7, 8, 9, 10\}$ is large but $\{6, 7, 80, 900, 10^{10}\}$ is not.) We can now (if we wish, and it's simpler to do so) dispense with the parameter k and state the modified version as:

$$\forall_{n,l}\, \exists_m\, (m \to (\text{large})^n_l)$$

where $m \to (\text{large})^n_l$ means that every colouring $c : m^{[n]} \to l$ has a large homogeneous set $Y \subset m$, it being assumed always that Y must have at least $n+1$ elements in order to avoid the trivial case $Y = m = n$.

That the Modified Finite Ramsey Theorem is indeed true follows easily from the Infinite Ramsey Theorem. For assume, toward a contradiction, that it is false. Then there are fixed n and l such that for every m there is a colouring $c_m : m^{[n]} \to l$ with no large homogeneous set. Define a "diagonal" colouring on all $n+1$-element subsets of $\mathbb{N}$ by:

$$d(\{x_0, x_1, \ldots, x_{n-1}, x_n\}) = c_{x_n}(\{x_0, x_1, \ldots, x_{n-1}\})$$

where $x_0, x_1, \ldots, x_{n-1}, x_n$ are written in increasing order. Then by the Infinite Ramsey Theorem, d has an infinite homogeneous set $Y \subset \mathbb{N}$. We can therefore select from Y an increasing sequence $\{y_0, y_1, \ldots, y_{y_0}\}$ with $y_0 \geq n+1$. Now let $m = y_{y_0}$ and choose $Y_0 = \{y_0, y_1, \ldots, y_{y_0-1}\}$. Then Y_0 is a large subset of m and is homogeneous for c_m since $c_m(x_0, \ldots, x_{n-1}) = d(x_0, \ldots, x_{n-1}, m)$ is constant on all $\{x_0, \ldots, x_{n-1}\} \in Y_0^{[n]}$. This is the desired contradiction.

The Paris-Harrington function is

$$PH(n,l) = \mu m\, (m \to (\text{large})^n_l)$$

and we show here that, for a suitable elementary function $l(n)$,

$$H_{\varepsilon_0}(n) \leq PH(n+1, l(n)).$$

Though it does not give the refined bounds of Ketonen-Solovay, this is enough for the independence result since, by the work of the previous section, it shows that PH grows faster than every provably recursive function of PA.

The proof has two parts. First, define certain colourings on finite sets of ordinals below ε_0, for which we can prove that all of their homogeneous sets must be "relatively small". Then use the Hardy functions to associate the interval of numbers x between n and $H_{\varepsilon_0}(n)$ with the strictly decreasing sequence of ordinals $P_x P_{x-1} \ldots P_n(\varepsilon_0)$ where, for any β, $P_i(\beta)$ denotes the maximum element of $\beta[i]$. This crucial correspondence is due to the simple fact that for any $\alpha \neq 0$,

$$H_\alpha(k) = \mu y > k.P_{y-1}P_{y-2}\cdots P_k(\alpha) = 0.$$

By the correspondence one obtains colourings on $n+1$-element subsets of $H_{\varepsilon_0}(n)$ which have no large homogeneous sets. Hence PH must grow at least as fast as H_{ε_0}.

Definition 4.1 Given Cantor Normal Forms $\alpha = \omega^{\alpha_1} \cdot a_1 + \ldots + \omega^{\alpha_r} \cdot a_r$ and $\beta = \omega^{\beta_1} \cdot b_1 + \ldots + \omega^{\beta_s} \cdot b_s$ with $\alpha > \beta$, let $D(\alpha, \beta)$ denote the first (i.e. greatest) exponent α_i at which they differ. Thus $\omega^{\alpha_1} \cdot a_1 + \ldots + \omega^{\alpha_{i-1}} \cdot a_{i-1} = \omega^{\beta_1} \cdot b_1 + \ldots + \omega^{\beta_{i-1}} \cdot b_{i-1}$ and $\omega^{\alpha_i} \cdot a_i > \omega^{\beta_i} \cdot b_i + \ldots + \omega^{\beta_s} \cdot b_s$.

Definition 4.2 For each $n \geq 2$ the function C_n from the $n+1$-element subsets of $\varepsilon_0(n-1)$ into $2^n - 1$ is given by the following induction. The definition of $C_n(\{\alpha_0, \alpha_1, \ldots, \alpha_n\})$ requires that the ordinals are listed in *descending* order; when-

ever we need to emphasise this we write $C_n(\alpha_0, \alpha_1, \ldots, \alpha_n)_>$ instead. Note that if $\alpha, \beta < \varepsilon_0(n-1)$ then $D(\alpha, \beta) < \varepsilon_0(n-2)$.

$$C_2(\alpha_0, \alpha_1, \alpha_2)_> = \begin{cases} 0 \text{ if } D(\alpha_0, \alpha_1) > D(\alpha_1, \alpha_2) \\ 1 \text{ if } D(\alpha_0, \alpha_1) < D(\alpha_1, \alpha_2) \\ 2 \text{ if } D(\alpha_0, \alpha_1) = D(\alpha_1, \alpha_2) \end{cases}$$

and for each $n > 2$,

$$C_n(\alpha_0, \ldots, \alpha_n)_> = \begin{cases} 2 \cdot C_{n-1}(\{\delta_0, \ldots, \delta_{n-1}\}) & \text{if } D(\alpha_0, \alpha_1) > D(\alpha_1, \alpha_2) \\ 2 \cdot C_{n-1}(\{\delta_{n-1}, \ldots, \delta_0\}) + 1 & \text{if } D(\alpha_0, \alpha_1) < D(\alpha_1, \alpha_2) \\ 2^n - 2 & \text{if } D(\alpha_0, \alpha_1) = D(\alpha_1, \alpha_2) \end{cases}$$

where $\delta_i = D(\alpha_i, \alpha_{i+1})$ for each $i < n$.

Lemma 4.3 *If $S = \{\gamma_0, \gamma_1 \ldots, \gamma_r\}_>$ is homogeneous for C_n then, letting $\max(\gamma_0)$ denote the maximum coefficient of γ_0 and $k(n) = 1 + 2 + \cdots + (n-1) + 2$, we have $|S| < \max(\gamma_0) + k(n)$.*

Proof. Proceed by induction on $n \geq 2$.

For the base-case we have $\varepsilon_0(1) = \omega^\omega$ and $C_2 : (\omega^\omega)^{[3]} \to 3$. Since S is a subset of ω^ω the values of $D(\gamma_i, \gamma_{i+1})$, for $i < r$, are integers. Let γ_0, the greatest member of S, have Cantor Normal Form:

$$\gamma_0 = \omega^m \cdot c_m + \omega^{m-1} \cdot c_{m-1} + \ldots + \omega^2 \cdot c_2 + \omega \cdot c_1 + c_0$$

where some of $c_{m-1}, \ldots, c_1, c_0$ may be zero, but $c_m > 0$. Then for each $i < r$, $D(\gamma_i, \gamma_{i+1}) \leq c_m \leq \max(\gamma_0)$. Now if C_2 has constant value 0 or 1 on $S^{[3]}$ then all $D(\gamma_i, \gamma_{i+1})$, for $i < r$, are distinct, and since we have r distinct numbers $\leq \max(\gamma_0)$ it follows that $|S| = r + 1 < \max(\gamma_0) + 3$ as required. If, on the other hand, C_2 has constant value 2 on $S^{[3]}$ then all the $D(\gamma_i, \gamma_{i+1})$ are equal, say to j. But then the Cantor Normal Form of each γ_i contains a term $\omega^j \cdot c_{i,j}$ where $0 \leq c_{r,j} < c_{r-1,j} < \ldots < c_{0,j} = c_j \leq \max(\gamma_0)$. In this case we have $r+1$ distinct numbers $\leq \max(\gamma_0)$ and hence, again, $|S| = r + 1 < \max(\gamma_0) + 3$.

For the induction step assume $n > 2$. Assume also that $r \geq k(n)$, for otherwise the desired result $|S| < \max(\gamma_0) + k(n)$ is automatic.

First, suppose C_n is constant on $S^{[n+1]}$ with even value $< 2^n - 2$. Note that the final $n+1$-tuple of S is $(\gamma_{r-n}, \gamma_{r-n+1}, \gamma_{r-n+2}, \ldots, \gamma_r)_>$. Therefore, by the first case in the definition of C_n,

$$D(\gamma_0, \gamma_1) > D(\gamma_1, \gamma_2) > \ldots > D(\gamma_{r-n+1}, \gamma_{r-n+2})$$

and this set is homogeneous for C_{n-1} (the condition $r \geq k(n)$ ensures that it has more than n elements). Consequently, by the induction hypothesis, $r - n + 2 < \max(D(\gamma_0, \gamma_1)) + k(n-1)$ and therefore, since $D(\gamma_0, \gamma_1))$ occurs as an exponent in the Cantor Normal Form of γ_0,

$$|S| = r + 1 < \max(D(\gamma_0, \gamma_1)) + k(n-1) + (n-1) \leq \max(\gamma_0) + k(n)$$

as required.

Second, suppose C_n is constant on $S^{[n+1]}$ with odd value. Then by the definition of C_n we have

$$D(\gamma_{r-n+1}, \gamma_{r-n+2}) > D(\gamma_{r-n}, \gamma_{r-n+1}) > \ldots > D(\gamma_0, \gamma_1)$$

and this set is homogeneous for C_{n-1}. So by applying the induction hypothesis one obtains $r - n + 2 < \max(D(\gamma_{r-n+1}, \gamma_{r-n+2})) + k(n-1)$ and hence

$$|S| = r + 1 < \max(D(\gamma_{r-n+1}, \gamma_{r-n+2})) + k(n).$$

Now in this case, since $D(\gamma_1, \gamma_2) > D(\gamma_0, \gamma_1)$ it follows that the initial segments of the Cantor Normal Forms of γ_0 and γ_1 are identical down to and including the term with exponent $D(\gamma_1, \gamma_2)$. Therefore $D(\gamma_1, \gamma_2) = D(\gamma_0, \gamma_2)$. Similarly $D(\gamma_2, \gamma_3) = D(\gamma_1, \gamma_3) = D(\gamma_0, \gamma_3)$ and by repeating this argument one obtains eventually, $D(\gamma_{r-n+1}, \gamma_{r-n+2}) = D(\gamma_0, \gamma_{r-n+2})$. Thus $D(\gamma_{r-n+1}, \gamma_{r-n+2})$ is one of the exponents in the Cantor Normal Form of γ_0, so its maximum coefficient is bounded by $\max(\gamma_0)$ and, again, $|S| < \max(\gamma_0) + k(n)$.

Finally suppose C_n is constant on $S^{[n+1]}$ with value $2^n - 2$. In this case all the $D(\gamma_i, \gamma_{i+1})$ are equal, say to δ, for $i < r - n + 2$. Let d_i be the coefficient of ω^δ in the Cantor Normal Form of γ_i. Then $d_0 > d_1 > \ldots > d_{r-n+1} > 0$ and so $r - n + 1 < d_0 \leq \max(\gamma_0)$. Therefore $|S| = r + 1 < \max(\gamma_0) + k(n)$ and this completes the proof.

Lemma 4.4 *For each $n \geq 2$ let $l(n) = 2k(n) + 2^n - 1$. Then there is a colouring $c_n \colon H_{\varepsilon_0(n-1)}(k(n))^{[n+1]} \to l(n)$ which has no large homogeneous sets.*

Proof. Fix $n \geq 2$ and let $k = k(n)$. Recall that

$$H_{\varepsilon_0(n-1)}(k) = \mu y > k.P_{y-1}P_{y-2}\cdots P_k(\varepsilon_0(n-1)) = 0.$$

As i increases from k up to $H_{\varepsilon_0(n-1)}(k) - 1$, the associated sequence of ordinals $\alpha_i = P_iP_{i-1}\cdots P_k(\varepsilon_0(n-1))$ strictly decreases to 0. Therefore, from the above colouring C_n on sets of ordinals below $\varepsilon_0(n-1)$, we can define a colouring d_n on the $(n+1)$-subsets of $\{2k, 2k+1, \ldots, H_{\varepsilon_0(n-1)}(k) - 1\}$ thus:

$$d_n(x_0, x_1, \ldots, x_n)_< = C_n(\alpha_{x_0-k}, \alpha_{x_1-k}, \ldots \alpha_{x_n-k})_>.$$

Clearly, every homogeneous set $\{y_0, y_1, \ldots, y_r\}_<$ for d_n corresponds to a homogeneous set $\{\alpha_{y_0-k}, \alpha_{y_1-k}, \ldots \alpha_{y_r-k}\}_>$ for C_n, and by the previous lemma it has fewer than $\max(\alpha_{y_0-k}) + k$ elements. Now the maximum coefficient of any $P_i(\beta)$ is no greater than the maximum of i and $\max(\beta)$, so $\max(\alpha_{y_0-k}) \leq y_0 - k$. Therefore every homogeneous set $\{y_0, y_1, \ldots, y_r\}_<$ for d_n has fewer than y_0 elements.

From d_n construct $c_n \colon H_{\varepsilon_0(n-1)}(k)^{[n+1]} \to l(n)$ as follows:

$$c_n(x_0, x_1, \ldots, x_n)_< = \begin{cases} d_n(x_0, x_1, \ldots, x_n) & \text{if } x_0 \geq 2k \\ x_0 + 2^n - 1 & \text{if } x_0 < 2k \end{cases}$$

Suppose $\{y_0, y_1, \ldots, y_r\}_<$ is homogeneous for c_n with colour $\geq 2^n - 1$. Then by the second clause, $y_0 + 2^n - 1 = c_n(y_0, y_1, \ldots, y_n) = c_n(y_1, y_2, \ldots, y_{n+1}) = y_1 + 2^n - 1$ and hence $y_0 = y_1$ which is impossible. Therefore any homogeneous set for c_n has least element $y_0 \geq 2k$ and, by the first clause, it must be homogeneous for d_n also. Thus it has fewer than y_0 elements, and hence this colouring c_n has no large homogeneous sets.

Theorem 4.5 (Paris-Harrington 1977) *The Modified Finite Ramsey Theorem is true but not provable in PA.*

Proof. Suppose, toward a contradiction, that $\forall_n \forall_l \exists_m (m \to (\text{large})^n_l)$ were provable in PA. Then the function

$$PH(n,l) = \mu m(m \to (\text{large})^n_l)$$

would be provably recursive in PA, and so also would be the function $f(n) = PH(n+2, l(n+1))$. For each n, $f(n)$ is so big that every colouring on $f(n)^{[n+2]}$ with $l(n+1)$ colours, has a large homogeneous set. The last lemma, with n replaced by $n+1$, gives a colouring $c_{n+1} : H_{\varepsilon_0(n)}(k(n+1))^{[n+2]} \to l(n+1)$ with no large homogeneous sets. Therefore $f(n) > H_{\varepsilon_0(n)}(k(n+1))$ for otherwise c_{n+1}, restricted to $f(n)^{[n+2]}$, would have a large homogeneous set. Since $H_{\varepsilon_0(n)}$ is increasing, $H_{\varepsilon_0(n)}(k(n+1)) > H_{\varepsilon_0(n)}(n) = H_{\varepsilon_0}(n)$. Hence $f(n) > H_{\varepsilon_0}(n)$ for all n, and since H_{ε_0} eventually dominates all provably recursive functions of PA it follows that f cannot be provably recursive. This is the contradiction.

References

[1] W. Buchholz, "An independence result for $(\Pi^1_1 - CA) + (BI)$", Annals of Pure and Applied Logic 23 (1987), 131-155.

[2] W. Buchholz and S.S. Wainer, "Provably computable functions and the fast growing hierarchy", in S.G. Simpson (ed) Logic and Combinatorics, Contemporary Mathematics 65, AMS Providence RI (1987), 179-198.

[3] G. Gentzen, "Die Widerspruchsfreiheit der reinen Zahlentheorie", Math. Annalen 112 (1936), 493-565.

[4] R.L. Graham, B.L. Rothschild and J.H. Spencer, "Ramsey Theory", 2nd. Ed., J. Wiley and Sons, New York (1990).

[5] C.G. Jockusch Jnr., "Ramsey's theorem and recursion theory", Jour. Symbolic Logic 37 (1972), 268-280.

[6] J. Ketonen and R.M. Solovay, "Rapidly growing Ramsey functions", Annals of Mathematics 113 (1981), 267-314.

[7] G. Kreisel, "On the interpretation of non-finitist proofs II", Jour. Symbolic Logic 17 (1952), 43-58.

[8] J.B. Paris and L. Harrington, "A mathematical incompleteness in Peano Arithmetic", in J. Barwise (ed), Handbook of Mathematical Logic, N-Holland, Amsterdam (1977), 1133-1142.

[9] F.P. Ramsey, "On a problem of formal logic", Proc. London Math. Soc. 30 (1930), 264-286.

[10] H. Schwichtenberg and S.S. Wainer, "Proofs and Computations", in preparation.

[11] S.G. Simpson (ed), "Reverse Mathematics 2001", Lecture Notes in Logic, Assoc. for Symbolic Logic (2005).

[12] W.W. Tait, "Normal derivability in classical logic", in J. Barwise (ed) Springer Lecture Notes in Mathematics 72, Springer–Verlag, Berlin (1968), 204-236.

Formal Logical Methods for System Security and Correctness
O. Grumberg et al. (Eds.)
IOS Press, 2008

Author Index

Barthe, G.	1	Hyland, M.	175
Bickford, M.	29	Mitchell, J.C.	199
Constable, R.	29	Morrisett, G.	233
Datta, A.	199	Nipkow, T.	245
Derek, A.	199	Rezk, T.	1
Esparza, J.	53	Roy, A.	199
Grégoire, B.	1	Schwichtenberg, H.	267
Grumberg, O.	89	Seifert, J.-P.	199
Harrison, J.	111	Wainer, S.S.	319
Hofmann, M.	149		

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank