Thesis for the Degree of Licentiate of Philosophy

Semantics-based Program Security and Aspects of Program Analysis

Andrei Sabelfeld

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, March 2000

Semantics-based Program Security
and Aspects of Program Analysis
Andrei Sabelfeld

ISBN 91-7197-898-4

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, March 2000

Abstract

The first part of the thesis investigates *confidentiality*, i.e. the problem of determining whether a given program (possibly received from an untrusted source) has *secure information flow.* We propose an extensional semantics-based formal specification of secure information-flow properties in sequential and concurrent programs based on representing degrees of security by partial equivalence relations *(pers)*.

By making use of the machinery of powerdomain semantics, the approach is extended to handle nondeterminism and *probabilistic covert channels* for a small sequential language. The treatment of security for concurrent programs uses operational semantics and pers as probabilistic-bisimulation-like relations. We specify security for a small multi-threaded programming language with dynamic thread creation and generalise the security condition in order to define robust security with respect to a wide class of schedulers and program-controlled thread priorities. For both denotational and operational approaches, the usefulness of the compositional nature of the security specifications is illustrated by straightforward correctness proofs for simple type-based security analyses.

The second part of the thesis examines some analyses and transformations for a simple call-by-name functional and a logic programming language. For the functional language, we present two general frameworks of semantic property analysis – *forwards* and *backwards* analyses – and give a marking technique for solving a subset of these problems. We study equivalence relations for the subclass of functional programs with finite unfolding and present complete systems of equivalent transformations for the tree and functional equivalence of such programs.

Finally, we catalogue some known transformations (along with their soundness proofs) and introduce two transformation rules (copying/merging and contextual replacement by equal term) for a logic programming language.

Keywords: *semantics, security, confidentiality, partial equivalence relations, noninterference, powerdomains, probabilistic bisimulation, probabilistic covert channels, program analysis, program transformation.*

Preface

The first part of the thesis investigates *confidentiality*, i.e. the problem of determining whether a given program (possibly received from an untrusted source) has *secure information flow*. Here is a possible scenario for motivating the problem. You have received a program from an untrusted source. Let us call it company M. M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that you permit a log-file containing a summary of your usage of the program to be automatically emailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has secure information flow. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database.

The problem of secure information flow in the context of systems with multiple levels of confidentiality has been studied much since Denning's pioneering work [Den76, DD77]. Denning's solution is to apply a static analysis suitable for inclusion into a compiler. The basic idea is that security levels are represented as a lattice (for example the two point lattice $\mathit{PublicDomain} \leq \mathit{TopSecret}$). The aim of the static analysis is to ensure that information from inputs, variables or processes of a given security level only flows to outputs, variables or processes which have been assigned a higher or equal security level.

In order to verify a program analysis or a specific proof a program's security one must have a formal specification of what constitutes secure information flow. The value of a semantics-based specification for secure information flow is that it contributes significantly to the reliability of and the confidence in such activities, and can be used in the systematic design of program analyses. Many approaches to Denning-style analyses (including the original articles) contain a fair degree of formalism but arguably are lacking a rigorous soundness proof. Volpano *et al* [VSI96] claim to give the first satisfactory treatment of soundness of Denning's analysis. Such a claim rests on the dissatisfaction with soundness arguments based on an instrumented operational e.g., [Ørb95] or denotational semantics e.g., [MS92], or on "axiomatic" approaches which define security in terms of a program logic [AR80] without any models to relate the logic to the semantics of the programming language. The problem here is that an "instrumented semantics" or a "security logic" is just a definition, not subject to any further mathematical justification. For a convincing assurance of a program's security, one therefore needs a specification of security that is defined in terms of the original noninstrumented programming language semantics.

This motivates the notion of *noninterference* introduced introduced in [GM82]. Intuitively, a program has the noninterference property whenever any behaviour observable by the attacker (e.g., low input, low output, termination, etc.) is independent from the initial high data. Many noninterference approaches have been phrased in terms of abstract, and sometimes rather ad hoc models of computation. We adopt the noninterference view of security and present a formalisation of noninterference based on the machinery of programming language semantics for a small sequential and a small concurrent language. We further show how this approach extends to formalise *covert channels* – implicit information that the attacker might learn about the program execution – such as timing, probability distribution of low output, etc.

Paper **A** proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations *(pers)*. The approach is inspired by (and in the deterministic case equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher-order functions and "partially confidential data". The approach is extended to handle nondeterminism by using powerdomain semantics and *probabilistic covert channels* by using probabilistic powerdomain semantics.

Further investigation concerns a probability-sensitive confidentiality specification – a form of *probabilistic noninterference* – for a small multi-threaded programming language with dynamic thread creation. In principle, the denotational semantics machinery should be sufficient to handle concurrent languages, but the mathematical overheads are high. Thus, in Paper **B** we turn to operationally-based use of pers as probabilistic-bisimulation-like relations. Probabilistic covert channels arise from a scheduler which is probabilistic. Since scheduling policy is typically outside the language specification for multi-threaded languages, we describe how to generalise the security condition in order to define robust security with respect to a wide class of schedulers and program-controlled thread priorities.

In both denotational and operational approaches, the security conditions satisfy compositionality properties which facilitate straightforward proofs of correctness for e.g. security type systems. The usefulness of the compositional nature of the security specifications is illustrated in both Papers **A** and **B** by straightforward correctness proofs for simple type-based security analyses.

The second part of the thesis examines some analyses and transformations for a simple call-by-name functional and a logic programming language. For the functional language, Paper **X** presents two general frameworks of semantic property analysis – *forwards* and *backwards* analyses – and gives a marking technique for solving a subset of these problems. The forwards analysis framework is exemplified by solving the *strictness analysis* problem and the backwards analysis framework is illustrated by solving the *dead code elimination* problem. An investigation of the *formal parameter dependence analysis* which cannot be be described as in either forwards or backwards analysis framework concludes the case study.

Paper **Y** studies equivalence relations for the subclass of functional programs with finite unwinding and presents complete systems of equivalent transformations for the tree and functional equivalences of such programs. The paper gives the upper bound $C^{2^{O(n)}}$ for the size of the full unwinding of a program of size n from the class in question and shows that this bound is also the lower bound.

Finally, Paper **Z** catalogues some known transformations (along with their soundness proofs) and introduces two new transformation rules (Copying/Merge of Copies and Contextual Replacement by Equal Term) for a logic programming language.

References

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[GM82] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

[MS92] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[Ørb95] Peter Ørbæk. Can you Trust your Data? In M. I. Schwartzbach P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAP-SOFT/FASE'95 Conference*, LNCS 915, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Acknowledgements

I count myself lucky to be supervised by Dave Sands. Besides being a sharp and broad-minded researcher, he is also an exciting versatile character who is second-to-none in the wide range from talk-giving to coffee-machine handling. It was in early 1998 when, thanks to discussions with Dave, I realised and started strongly believing that programming language techniques can be very usefully applied to the area of security. It has been truly stimulating and fruitful to work with Dave pursuing this goal.

This work has also benefited from discussions with John Hughes, Johan Agat, Magnus Carlsson, Jörgen Gustavsson, Walid Taha, Peter Dybjer, Makoto Takeyama, Ilya Beylin and Koen Claesson.

Going further afield, I would like to thank Martín Abadi, Jon Riecke, Rustan Leino, Greg Nelson, Yuri Gurevich, Andrei Voronkov, Bengt Jonsson, Peter Ryan, Riccardo Focardi and Paul Syverson for fruitful discussions and valuable feedback.

I would like to express my gratitude to my mother who is always my best support and my father who has also been a great collaborator.

Special thanks to my friends Lars Pareto, Tuomo Takkula, Nadja Carlsson, Dave Rutter, Boris Koldehofe, Gordon Pace, Karol Ostrovsky and my charming office mate Ana Bove for a great job at making my time in Gothenburg full of fun.

Finally, many thanks from the bottom of my heart go to the man of high and lonely fate akin only to Elric's (only he killed considerably less people), Gandalf's (only he claims he dresses with much less style), and The Man That Says: "And now for something completely different".

Andrei Sabelfeld
17 March 2000

List of Included Papers

This thesis is based on work reported in the following papers:

A Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming, ESOP'99*, LNCS 1576, Amsterdam, March 1999, Springer-Verlag.

B Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press. To appear.

X Andrei Sabelfeld and Viktor Sabelfeld. Simple Semantic Analysis Problems for Functional Programs. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, June 1997. ACM Press.

Y Andrei Sabelfeld and Viktor Sabelfeld. Equivalent transformations of recursive programs with finite unfolding. *Programming and Computer Software*, 23 (2):70–77, 1997.

Z Andrei Sabelfeld. Correct transformations of logic programs. *Joint Bulletin of Novosibirsk Computer Center and the Institute of Informatics Systems*, 5:55–67, 1996.

Contents

Part I

Semantics-based Program Security

A Per Model of Secure Information Flow in Sequential Programs *

Andrei Sabelfeld and David Sands

Chalmers†

Abstract

This paper proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations (pers). The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by (and in the deterministic case equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher-order functions and "partially confidential data". We extend the approach to handle nondeterminism by using powerdomain semantics and show how *probabilistic security properties* can be formalised by using probabilistic powerdomain semantics. We illustrate the usefulness of the compositional nature of the security specifications by presenting a straightforward correctness proof for a simple type-based security analysis.

*This is an extended version of the paper from the *Proceedings of European Symposium on Programming, ESOP'99*, LNCS 1576, Amsterdam, March 1999. Springer-Verlag.

†Department of Computer Science, Chalmers University of Technology and the University of Göteborg, 412 96 Göteborg, Sweden. E-mail: {andrei,dave}@cs.chalmers.se

1 Introduction

1.1 Motivation

You have received a program from an untrusted source. Let us call it company M. M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that you permit a log-file containing a summary of your usage of the program to be automatically emailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has *secure information flow*. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database.

This paper proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations (pers[1]). The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by and, in the deterministic case, equivalent to the use of partial equivalence relations in specifying binding-time analysis [HS91], and is thus able to specify security properties of higher-order functions and "partially confidential data" (e.g. one's financial database could be deemed to be partially confidential if the number of entries is not deemed to be confidential even though the entries themselves are). We show how the approach can be extended to handle nondeterminism, and illustrate how the various choices of powerdomain semantics affects the kinds of security properties that can be expressed, ranging from termination-insensitive properties (corresponding to the use of the Hoare (partial correctness) powerdomain) to *probabilistic security properties*, obtained when one uses a probabilistic powerdomain.

[1]A partial equivalence relation is symmetric and transitive but not necessarily reflexive

1.2 Background

The study of information flow in the context of systems with multiple levels of confidentiality was pioneered by Denning [Den76, DD77] in an extension of Bell and LaPadula's early work [BL76]. Denning's approach is to apply a static analysis suitable for inclusion into a compiler. The basic idea is that security levels are represented as a lattice (for example the two point lattice $\mathit{PublicDomain} \leq \mathit{TopSecret}$). The aim of the static analysis is to ensure that information from inputs, variables or processes of a given security level only flows to outputs, variables or processes which have been assigned a higher or equal security level.

1.3 Semantic Foundations of Information Flow Analysis

In order to verify a program analysis or a specific proof a program's security one must have a formal specification of what constitutes secure information flow. The value of a semantics-based specification for secure information flow is that it contributes significantly to the reliability of and the confidence in such activities, and can be used in the systematic design of program analyses. Many approaches to Denning-style analyses (including the original articles) contain a fair degree of formalism but arguably are lacking a rigorous soundness proof. Volpano *et al* [VSI96] claim to give the first satisfactory treatment of soundness of Denning's analysis. Such a claim rests on the dissatisfaction with soundness arguments based on an instrumented operational e.g., [Ørb95] or denotational semantics e.g., [MS92], or on "axiomatic" approaches which define security in terms of a program logic [AR80] without any models to relate the logic to the semantics of the programming language. The problem here is that an "instrumented semantics" or a "security logic" is just a definition, not subject to any further mathematical justification. McLean points out [McL90b] in a related discussion about the (non language-specific) Bell and LaPadula model:

> One problem is that ... they *[the Bell LaPadula security properties]* constitute a possible implementation of security, ... , rather than an abstract specification of what all secure systems must satisfy. By concerning themselves with particular controls over files inside the computer, rather than limiting themselves to the relation between input and output, they make it harder to reason about the requirements, ...

This criticism points to more abstract, *extensional* notions of soundness, based on, for example, the idea of *noninterference* introduced in [GM82].

1.4 Semantics-based Models of Information Flow

The problem of secure information flow, or "noninterference" is now quite mature, and very many specifications exist in the literature – see [McL94] for a tutorial overview. Many approaches have been phrased in terms of abstract, and sometimes rather ad hoc models of computation. Only more recently have attempts been made to rephrase and compare various security conditions in terms of well-known semantic models, e.g. the use of labelled transition systems and bisimulation semantics in [FG94]. In this paper we consider the problem of information-flow properties of sequential systems, and use the framework of *denotational semantics* as our formal model of computation. Along the way we consider some relations to specific static analyses, such as the ***S**ecurity **Lam**bda Calculus* [HR98] and an alternative semantic condition for secure information flow proposed by Leino and Joshi [LJ98].

1.5 Overview

The rest of the paper is organised as follows.

Section 2 shows how the per-based condition for soundness of binding times analysis is also a model of secure information flow. We show how this provides insight into the treatment of higher-order functions and structured data.

Section 3 shows how the approach can be adapted to the setting of a nondeterministic imperative language by appropriate use of a powerdomain-based semantics. We show how the choice of powerdomain (upper, lower or convex) affects the nature of the security condition.

Section 4 focuses on an alternative semantic specification due to Leino and Joshi. Modulo some technicalities we show that Leino's condition – and a family of similar conditions – are in agreement with, and can be represented using our form of specification.

Section 5 considers the problem of preventing unwanted *probabilistic* information flows in programs. We show how this can be solved in the same framework by utilising a probabilistic semantics based on the probabilistic powerdomain [JP89].

Section 6 shows how the probabilistic security specification satisfies compositionality properties which facilitates straightforward proofs of correctness for compositional analyses. We illustrate the usefulness of the compositional nature of the security specifications by presenting a straightforward correctness proof for a type-based security analysis.

Section 7 concludes.

2 A Per Model of Information Flow

In this section we introduce the way that *partial equivalence relations* (pers) can be used to model dependencies in programs. The basic idea comes from Hunts use of pers to model and construct abstract interpretations for strictness properties in higher-order functional programs [Hun90, Hun91], and in particular its use to model dependencies in *binding-time* analysis [HS91]. Related ideas already occur in the denotational formulation of live-variable analysis [Nie90].

2.1 Binding Time Analysis as Dependency Analysis

Given a description of the parameters in a program that will be known at partial evaluation time (called the *static* arguments), a binding-time analysis (BTA) must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). The safety condition for binding time analysis must ensure that there is no dependency between the *dynamic* (i.e., non-static) arguments and the parts of the program that are deemed to be static. Viewed in this way, binding time analysis is purely an analysis of dependencies.[2]

Dependencies in Security In the security field, the property of absence of unwanted dependencies is often called *noninterference*, after [GM82]. Many problems in security come down to forms of dependency analysis. For example, in the case of *confidentiality*, the aim is to show that the outputs of a program which are deemed to be of low confidentiality do not have any dependence on inputs of a higher degree of confidentiality. In the case of *integrity* (*trust*), one must ensure that the value of some trusted data does not depend on some untrusted source.

[2]Unfortunately, from the perspective of a partial evaluator, BTA is not *purely* a matter of dependencies; in [HS95] it was shown that the pure dependency models of [Lau89] and [HS91] are not quite adequate to ensure the safety of partial evaluation.

Some intuitions about information flow Let us consider a program modelled as a function from some input domain to an output domain. Now consider the following simple functions mapping inputs to outputs: $\mathsf{snd} : D \times E \to E$ for some sets (or domains) D and E, and shift and test, functions in $\mathbf{N} \times \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ and $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$, defined by

$$\begin{aligned} \mathsf{snd}(x, y) &= y \\ \mathsf{shift}(x, y) &= (x + y, y) \\ \mathsf{test}(x, y) &= \mathsf{if}\ x > 0\ \mathsf{then}\ y\ \mathsf{else}\ y + 1 \end{aligned}$$

Now suppose that (h, l) is a pair where h is some high security information, and l is low, "public domain", information. Without knowing about what the actual values h and l might be, we know about the result of applying function snd will be a low value, and, in the case that we have a pair of numbers, the result of applying shift will be a pair with a high first component and a low second component.

Note that the function test does not enjoy the same security property that snd does, since although it produces a value which is constructed from purely low-security components, the actual value is dependent on the first component of the input. This is what is known as an indirect information flow [Den76].

It is rather natural to think of these properties as "security types":

$$\mathsf{snd} : \mathit{high} \times \mathit{low} \to \mathit{low}$$
$$\mathsf{shift} : \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low}$$
$$\mathsf{test} : \mathit{high} \times \mathit{low} \to \mathit{high}$$

But what notion of "type", and what interpretation of "high" and "low" can formalise these more intuitive type statements? Interpreting types as sets of values is not adequate to model "high" and "low". To track degrees of dependence between inputs and outputs we need a more dynamic view of a type as a degree of variation. We must vary (parts of) the input and observe which (parts of) the output vary. For the application to confidentiality we want to determine if there is possible information leakage from a high level input to the parts of an output which are intended to be visible to a low security observer. We can detect this by observing whether the "low" parts of the output vary in any way as we vary the high input.

The simple properties of the functions snd and shift described above can be be captured formally by the following formulae:

$$\forall x, x', y.\, \mathsf{snd}(x, y) = \mathsf{snd}(x', y) \tag{1}$$
$$\forall x, x', y.\, \mathsf{snd}(\mathsf{shift}(x, y)) = \mathsf{snd}(\mathsf{shift}(x', y)) \tag{2}$$

Indeed, this kind of formula forms the core of the correctness arguments for the security analyses proposed by e.g., Volpano and Smith et al [VSI96, SV98], and also for the extensional correctness proofs in core of the *Slam-calculus* [HR98].

High and Low as Equivalence Relations We show how we can interpret "security types" in general as partial equivalence relations. We will interpret *high* (for values in D) as the equivalence relation All_D, and *low* as the relation Id_D where for all $x, x' \in D$:

$$x \; All_D \; x' \tag{3}$$

$$x \; Id_D \; x' \iff x = x'. \tag{4}$$

For a function $f : D \to E$ and binary relations $P \in Rel(D)$ and $Q \in Rel(E)$, we write $f : P \twoheadrightarrow Q$ iff

$$\forall x, x' \in D. \, x \; P \; x' \Longrightarrow (f \; x) \; Q \; (f \; x').$$

For binary relations P, Q we define the relation $P \times Q$ by:

$$(x, y) \; P \times Q \; (x', y') \iff x \; P \; x' \;\; \& \;\; y \; Q \; y'.$$

Now the security property of snd described by (1) can be captured by

$$\mathsf{snd} : All_D \times Id_E \twoheadrightarrow Id_E,$$

and (2) is given by

$$\mathsf{shift} : All_{\mathbf{N}} \times Id_{\mathbf{N}} \twoheadrightarrow All_{\mathbf{N}} \times Id_{\mathbf{N}}$$

2.2 From Equivalence Relations to Pers

We have seen how the equivalence relations *All* and *Id* may be used to describe security "properties" *high* and *low*. It turns out that these are exactly the same as the interpretations given to the notions "dynamic" and "static" given in [HS91]. This means that the binding-time analysis for a higher-order functional language can also be read as a security information-flow analysis. This connection between security and binding time analysis is already folk-law (See e.g. [TK97] for a comparison of a particular security type system and a particular binding-time analysis, and [DRH95] which shows how the incorporation of indirect information flows from Dennings security analysis can improve binding time analyses).

It is worth highlighting a few of the pertinent ideas from [HS91]. Beginning with the equivalence relations All and Id to describe *high* and *low* respectively, there are two important extensions to the basic idea in order to handle *structured data types* and *higher-order functions*. Both of these ideas are handled by the analysis of [HS91] which rather straightforwardly extends Launchbury's projection-based binding-time analysis [Lau89] to higher types. To some extent [HS91] anticipates the treatment of partially-secure data types in the SLam calculus [HR98], and the use of logical relations in their proof of noninterference.

For structured data it is useful to have more refined notions of security than just *high* and *low*; we would like to be able to model various *degrees* of security. For example, we may have a list of records containing name-password pairs. Assuming passwords are considered *high*, we might like to express the fact that although the whole list cannot be considered *low*, it can be considered as a $(low \times high)list$. Constructing equivalence relations which represent such properties is straightforward – see [HS91] for examples (which are adapted directly from Launchbury's work), and [Hun91] for a more general treatment of finite lattices of "binding times" for recursive types.

To represent security properties of higher-order functions we use a less restricted class of relations than the equivalence relations. A *partial equivalence relation* (per) on a set D is a binary relation on D which is *symmetric* and *transitive*. If P is such a per let $|P|$ denote the domain of P, given by

$$|P| = \{x \in D \mid x\ P\ x\}\,.$$

Note that the domain and range of a per P are both equal to $|P|$ (so for any $x, y \in D$, if $x\ P\ y$ then $x\ P\ x$ and $y\ P\ y$), and that the restriction of P to $|P|$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $|P| = D$). Partial equivalence relations over various applicative structures have been used to construct models of the polymorphic lambda calculus (see, for example, [AP90]). As far as we are aware, the first use of pers in static program analysis is that presented in [Hun90].

For a given set D let $Per(D)$ denote the partial equivalence relations over D. $Per(D)$ is a meet semi-lattice, with meets given by set-intersection, and top element All.

Given pers $P \in Per(D)$ and $Q \in Per(E)$, we may construct a new per

$(D \twoheadrightarrow E) \in Per(D \rightarrow E)$ defined by:

$$f\ (P \twoheadrightarrow Q)\ g$$
$$\iff$$
$$\forall x, x' \in D.\, x\ P\ x' \implies (f\ x)\ Q\ (g\ x').$$

If P is a per, we will write $x : P$ to mean $x \in |P|$. This notation and the above definition of $P \twoheadrightarrow Q$ are consistent with the notation used previously, since now

$$\begin{array}{ll} f : P \twoheadrightarrow Q & \\ \iff & f\ (P \twoheadrightarrow Q)\ f \\ \iff & \forall x, x' \in D.\, x\ P\ x' \implies (f\ x)\ Q\ (f\ x'). \end{array}$$

Note that even if P and Q are both total (i.e., equivalence relations), $P \twoheadrightarrow Q$ may be partial. A simple example is $All \twoheadrightarrow Id$. If $f : All \twoheadrightarrow Id$ then we know that given a *high* input, f returns a *low* output. A constant function $\lambda x.42$ has this property, but clearly not all functions satisfy this.

2.3 Observations on Strictness and Termination Properties

We are interested in the security properties of functions which are the denotations of programs (in a Scott-style denotational semantics), and so there are some termination issues which should address. The formulation of security properties given above is sensitive to termination. Consider, for example, the following function $\mathsf{f} : \mathbf{N}_\perp \rightarrow \mathbf{N}_\perp$

$$\mathsf{f} = \lambda x.\mathsf{if}\ x > 0\ \mathsf{then}\ x\ \mathsf{else}\ f x$$

Clearly, if the argument is high then the result must be high. Now consider the security properties of the function $\mathsf{g} \circ \mathsf{f}$ where g the constant function $\mathsf{g} = \lambda x.2$. We might like to consider that g has type *high* $\rightarrow$ *low*. However, if function application is considered to be strict (as in ML) then g is not in $|All_{\mathbf{N}_\perp} \twoheadrightarrow Id_{\mathbf{N}_\perp}|$ since $\perp\ All_{\mathbf{N}_\perp}\ 1$ but $\mathsf{g}(\perp) \neq \mathsf{g}(1)$. Hence the function $\mathsf{g} \circ \mathsf{f}$ does *not* have security type *high* $\rightarrow$ *low* (in our semantic interpretation). This is correct, since on termination of an application of this function, the low observer will have learned that the value of the high argument was positive.

The specific security analysis of e.g. the first calculus of Smith and Volpano [SV98] is termination sensitive – and this is enforced by a rather sweeping measure: all “while”-loop conditions must be low and all “while”-loop bodies must be low commands.

On the other hand, the type system of the SLam calculus [HR98] is not termination sensitive in general. This is due to the fact that it is based on a call-by-value semantics, and indeed the composition $\mathsf{g} \circ \mathsf{f}$ could be considered to have a security type corresponding to "*high* $\rightarrow$ *low*". The correctness proof for noninterference carefully avoids saying anything about nonterminating executions. What is perhaps worth noting here is that had they chosen a non-strict semantics for application then the *same* type-system would yield termination sensitive security properties! So we might say that lazy programs are intrinsically more secure than strict ones. This phenomenon is closely related to properties of parametrically polymorphic functions [Rey83][3]. From the type of a polymorphic function one can predict certain properties about its behaviour – the so-called "free theorems" of the type [Wad89]. However, in a strict language one must add an additional condition in order that the theorems hold: the functions must be *bottom-reflecting* ($f(a) = \bot \implies a = \bot$). The same side condition can be added to make the e.g. the type system of the Slam-calculus termination-sensitive.

To make this observation precise we introduce one further constructor for pers. If $R \in Per(D)$ then we will also let R denote the corresponding per on $D_\bot$ without explicit injection of elements from D into elements in $D_\bot$. We will write $R_\bot$ to denote the relation in $Per(D_\bot)$ which naturally extends R by $\bot\, R\, \bot$.

Now we can be more precise about the properties of g under a strict (call-by-value) interpretation: $\mathsf{g} : (All_{\mathbf{N}})_\bot \twoheadrightarrow Id_{\mathbf{N}_\bot}$, which expresses that g is a constant function, modulo strictness. More informatively we can say that that $\mathsf{g} : (All_{\mathbf{N}}) \twoheadrightarrow Id_{\mathbf{N}}$ which expresses that g is a non-bottom constant function.

It is straightforward to express per properties in a subtype system of compositional rules (although we don't claim that such a a system would be in any sense complete). Pleasantly, all the expected subtyping rules are sound when types are interpreted as pers and the subtyping relation is interpreted as subset inclusion of relations. For the abstract interpretation presented in [HS91] this has already been undertaken by e.g. Jensen [Jen92] and Hankin and Le Métayer [HL94].

[3]Not forgetting that the use of Pers in static analysis was inspired, in part, by Abadi and Plotkin's Per model of polymorphic types [AP90]

3 Nondeterministic Information Flow

In this section we show how the per model of security can be extended to describe nondeterministic computations. We see nondeterminism as an important feature as it arises naturally when considering the semantics of a concurrent language (although the treatment of a concurrent language remains outside the scope of the present paper.)

In order to focus on the essence of the problem we consider a very simplified setting – the analysis of commands in some simple imperative language containing a nondeterministic choice operator. We assume that there is some discrete (i.e., unordered) domain **St** of states (which might be viewed as finite maps from variables to discrete values, or simply just a tuple of values).

3.1 Secure Commands in a Deterministic Setting

In the deterministic setting we can take the denotation of a command C, written $[\![C]\!]$, to be a function in $[\mathbf{St}_\perp \to \mathbf{St}_\perp]$, where by $[D_\perp \to E_\perp]$ we mean the set of strict and continuous maps between domains $D_\perp$ and $E_\perp$. Note that we could equally well take the set of all (trivially continuous) functions in $\mathbf{St} \to \mathbf{St}_\perp$, which is isomorphic.

Now suppose that the state is just a simple partition into a high-security half and a low-security half, so the set of states is the product $\mathbf{St}_{high} \times \mathbf{St}_{low}$. Then we might define a command C to be secure if no information from the high part of the state can leak into the low part:

$$C \text{ is secure } \iff$$
$$[\![C]\!] : (All \times Id)_\perp \twoheadrightarrow (All \times Id)_\perp$$

Which is equivalent to saying that $[\![C]\!] : (All \times Id) \twoheadrightarrow (All \times Id)_\perp$ since we only consider strict functions. Note that this does not imply that $[\![C]\!]$ terminates, but what it does imply is that the termination behaviour is not influenced by the values of the high part of the state. It is easy to see that the sequential composition of secure commands is a secure command, since firstly, the denotation of the sequential composition of commands is just the function-composition of denotations, and secondly, in general for functions $g : D \to E$ and $f : E \to F$, and pers $P \in Per(D)$, $Q \in Per(E)$ and $R \in Per(F)$ it is easy to verify the soundness of the inference rule:

$$\frac{g : P \twoheadrightarrow Q \qquad f : Q \twoheadrightarrow R}{f \circ g : P \twoheadrightarrow R}$$

3.2 Powerdomain Semantics for Nondeterminism

A standard approach to giving meaning to a nondeterministic language – for example Dijkstra's guarded command language – is to interpret a command as a mapping which yields a *set* of results. However, when defining an ordering on the results in order to obtain a domain, there is a tension between the internal order of $\mathbf{St}_\perp$ and the subset order of the powerset. This is resolved by considering a suitable *powerdomain* structure [Plo76, Smy78]. The idea is to define a preoreder on the finitely generated subsets[4] of $S_\perp$ in terms of the order on their elements. By quotienting "equivalent sets" one obtains a partial ordering, each depending on a different view of what sets of values should be considered equivalent. Consider the following three programs (an example from [Plo81])

$$(1)\ x := 1 \qquad\qquad (2)\ x := 1 \mathbin{[]} \mathsf{loop} \qquad\qquad (3)\ \mathsf{loop}$$

In the "Hoare" or *partial correctness* interpretation the first two programs are considered to be equal since, ignoring nontermination, they yield the same sets of outcomes. This view motivates the definition of the *Hoare* or *lower* powerdomain, $\mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]$.

In the "Smyth" or *total correctness* interpretation, programs (2) and (3) are considered equal (equally bad!) because neither of them can guarantee an outcome. In the general case this view motivates the *Smyth* or *upper* powerdomain, $\mathcal{P}_{\mathsf{U}}[\mathbf{St}_\perp]$ [Smy78].

In the "Egli-Milner" interpretation (leading to the *convex* or *Plotkin* powerdomain in the general case) all three programs are considered to have distinct denotations.

The three powerdomains are built from a domain D by starting with the *finitely generated* (f.g.) subsets of $D_\perp$ (those non-empty subsets which are either finite, or contain $\perp$), and a preorder on these sets. Quotienting the f.g. sets using the associated equivalence relation yields the corresponding domain. We give each construction in turn, and give an idea about the corresponding *discrete powerdomain* $\mathcal{P}[\mathbf{St}_\perp]$.

- *Lower powerdomain* Let $u \preceq_{\mathsf{L}} v$ iff $\forall x \in u.\, \exists y \in v.\, x \sqsubseteq y$. In this case the induced discrete powerdomain $\mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]$ is isomorphic to the powerset of $\mathbf{St}$ ordered by subset inclusion. This means that the domain $[\mathbf{St}_\perp \to \mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]]$ is isomorphic to all subsets of $\mathbf{St} \times \mathbf{St}$ – i.e. the relational semantics.

[4] If a set is infinite then it must contain $\perp$.

- *Upper powerdomain* The upper ordering on f.g. sets u, v, is given by

$$u \preceq_{\mathsf{U}} v \iff \forall y \in v.\, \exists x \in u.\, x \sqsubseteq y.$$

  Here the induced discrete powerdomain $\mathcal{P}_{\mathsf{U}}[\mathbf{St}_\perp]$ is isomorphic to the set of finite non-empty subsets of $\mathbf{St}$ together with $\mathbf{St}_\perp$ itself, ordered by superset inclusion.

- *Convex powerdomain* Let $u \preceq_{\mathsf{C}} v$ iff $u \preceq_{\mathsf{U}} v$ and $u \preceq_{\mathsf{L}} v$. This is also known as the Egli-Milner ordering. The resulting powerdomain $\mathcal{P}_{\mathsf{C}}[\mathbf{St}_\perp]$ is isomorphic to the f.g. subsets of $\mathbf{St}_\perp$, ordered by:

$$\begin{array}{lcl} A \sqsubseteq_{\mathsf{C}} B & \iff & \text{either } \perp \notin A \ \ \& \ \ A = B, \\ & & \text{or } \perp \in A \ \ \& \ \ A \setminus \{\perp\} \subseteq B \end{array}$$

A few basic properties and definitions on powerdomains will be needed. For each powerdomain constructor $\mathcal{P}[-]$ define the order-preserving "unit" map $\eta_D : D_\perp \to \mathcal{P}[D_\perp]$ which takes each element $a \in D$ into (the powerdomain equivalence class of) the singleton set $\{a\}$. For each function $f \in [D_\perp \to \mathcal{P}[E_\perp]]$ there exits a unique extension of f, denoted f^* where $f^* \in [\mathcal{P}[D_\perp] \to \mathcal{P}[E_\perp]]$ which is the unique mapping such that

$$f = f^* \circ \eta.$$

In the particular setting of the denotations of commands, it is worth noting that $[\![C_1; C_2]\!]$ would be given by:

$$[\![C_1; C_2]\!] = [\![C_2]\!]^* \circ [\![C_1]\!].$$

3.3 Pers on Powerdomains

Give one of the discrete powerdomains, $\mathcal{P}[\mathbf{St}_\perp]$, we will need a "logical" way to lift a per $P \in Per(\mathbf{St}_\perp)$ to a per in $Per(\mathcal{P}[\mathbf{St}_\perp])$.

Definition 1 *For each $R \in Per(D_\perp)$ and each choice of power domain $\mathcal{P}[-]$, let $\mathcal{P}[R]$ denote the relation on $\mathcal{P}[D_\perp]$ given by*

$$\begin{array}{lcl} A\ \mathcal{P}[R]\ B & \iff & \forall a \in A.\, \exists b \in B.\, a\ R\ b \\ & \& & \forall b \in B.\, \exists a \in A.\, a\ R\ b \end{array}$$

It is easy to check that $\mathcal{P}[R]$ is a per, and in particular that $\mathcal{P}[Id_{D_\perp}] = Id_{\mathcal{P}[D_\perp]}$.

Henceforth we shall restrict our attention to the semantics of simple commands, and hence the three discrete powerdomains $\mathcal{P}[\mathbf{St}_\perp]$.

Proposition 1 *For any* $f \in [\mathbf{St}_\perp \to \mathcal{P}[\mathbf{St}_\perp]]$ *and any* R, $S \in Per(\mathbf{St}_\perp)$,

$$f : R \twoheadrightarrow \mathcal{P}[S] \iff f^* : \mathcal{P}[R] \twoheadrightarrow \mathcal{P}[S]$$

From this it easily follows that the following inference rule is sound:

$$\frac{[\![C_1]\!] : P \twoheadrightarrow \mathcal{P}[Q] \qquad [\![C_2]\!] : Q \twoheadrightarrow \mathcal{P}[R]}{[\![C_1; C_2]\!] : P \twoheadrightarrow \mathcal{P}[R]}$$

3.4 The Security Condition

We will investigate the implications of the security condition under each of the powerdomain interpretations. Let us suppose that, as before the state is partitioned into a high part and a low part: $\mathbf{St} = \mathbf{St}_{high} \times \mathbf{St}_{low}$. With respect to a particular choice of powerdomain let the security "type" $C : high \times low \to high \times low$ denote the property

$$[\![C]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{P}[(All \times Id)_\perp].$$

In this case we say that C is secure. Now we explore the implications of this definition on each of the possible choices of powerdomain:

1. In the lower powerdomain, the security condition describes in a weak sense termination-insensitive information flow. For example, the program

   $$\textsf{if } h = 0 \textsf{ then skip } [\,] \textsf{ loop else skip}$$

   (h is the high part of the state) is considered secure under this interpretation but the termination behaviours is influenced by h (it can fail to terminate only when $h = 0$).

2. In the upper powerdomain nontermination is considered catastrophic. This interpretation seems completely unsuitable for security unless one only considers programs which are "totally correct" – i.e. which must terminate on their intended domain. Otherwise, a possible nonterminating computation path will mask any other insecure behaviours a term might exhibit. This means that for *any* program C, the program C [] $\textsf{loop}$ is secure!

3. The convex powerdomain gives the appropriate generalisation of the deterministic case in the sense that it is termination sensitive, and does not have the shortcomings of the upper powerdomain interpretation.

4 Relation to an Equational Characterisation

In this section we relate the Per-based security condition to a proposal by Leino and Joshi [LJ98]. Following their approach, assume for simplicity we have programs with just two variables: h and l of high and low secrecy respectively. Assume that the state is simple a pair, where h refers to the first projection and l is the second projection.

In [LJ98] the security condition for a program C is defined by

$$HH; C; HH = C; HH,$$

where "=" stands for semantic equality (the style of semantic specification is left unfixed), and HH is the program that "assigns to h arbitrary values" – aka "Havoc on H". We will refer to this equation as the equational security condition. Intuitively, the equation says that we cannot learn anything about the initial values of the high variables by variation of the low security variables. The postfix occurrences of HH on each side mean that we are only interested in the final value of l. The prefix HH on the left-hand side means that the two programs are equal if the final value of l does not depend on the initial value of h.

In relating the equational security condition to pers we must first decide upon the denotation of HH. Here we run into some potential problems since it is necessary in [LJ98] that HH always terminates, but nevertheless exhibits unbounded nondeterminism. Although this appears to pose no problems in [LJ98] (in fact it goes without mention), to handle this we would need to work with non-ω-continuous semantics, and powerdomains for unbounded nondeterminism. Instead, we side-step the issue by assuming that the domain of h, $\mathbf{St}_{high}$, is finite.

4.1 Equational Security and Projection Analysis

A first observation is that the the equational security condition is strikingly similar to the well-known form of static analysis for functional programs known as projection analysis [WH87]. Given a function f, a projection analysis aims to find projections (continuous lower closure operators on the domain) α and β such that

$$\beta \circ f \circ \alpha = \beta \circ f$$

For (generalised) strictness analysis and dead-variable analysis, one is given β, and α is to be determined; for binding time analysis [Lau89] it is a forwards analysis problem: given α one must determine some β.

For strict functions (e.g., the denotations of commands) projection analysis is not so readily applicable. However, in the convex powerdomain HH is rather projection-like, since it effectively hides all information about the high variable; in fact it is an embedding (an upper closure operator) so the connection is rather close.

4.2 The Equational Security Condition Is Subsumed by the Per Security Condition

Hunt [Hun90] showed that projection properties of the form $\beta \circ f \circ \alpha = \beta \circ f$ could be expressed naturally as a per property of the form $f : R_\alpha \twoheadrightarrow R_\beta$ for equivalence relations derived from α and β by relating elements which get mapped to the same point by the corresponding projection.

Using the same idea we can show that the per-based security condition subsumes the equation specification in a similar manner.

We will establish the following:

Theorem 1 *For any command* C

$$[\![HH; C; HH]\!] = [\![C; HH]\!]$$
$$\textit{iff}$$
$$C : \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low}.$$

The idea will be to associate an equivalence relation to the function HH. More generally, for any command C let $ker(C)$, the *kernel* of C, denote the relation on $\mathcal{P}[\mathbf{St}_\perp]$ satisfying

$$s_1 \; ker(C) \; s_2 \iff [\![C]\!]s_1 = [\![C]\!]s_2.$$

Define the extension of $ker(C)$ by

$$A \; ker^*(C) \; B \iff [\![C]\!]^*A = [\![C]\!]^*B.$$

Recall the per interpretation of the type signature of C.

$$C : \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low}$$
$$\iff$$
$$[\![C]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{P}[(All \times Id)_\perp].$$

Observe that $(All \times Id)_\perp = ker(HH)$ since for any h, l, h', l' it holds $[\![HH]\!](h,l) = [\![HH]\!](h',l')$ iff $l = l'$ iff $(h,l)(All \times Id)_\perp(h',l')$.

The proof of the theorem is based on this observation and on the following two facts:

- $\mathcal{P}[All \times Id]_\perp = ker^*(HH)$ and
- $[\![HH; C; HH]\!] = [\![C; HH]\!] \iff$
  $[\![C]\!] : ker(HH) \twoheadrightarrow ker^*(HH)$.

Let us first prove the latter fact by proving a more general statement similar to Proposition 3.1.5 from [Hun91] (the correspondence between projections and per-analysis). Note that we do not use the specifics of the convex powerdomain semantics here, so the proof is valid for any of the three choices of powerdomain.

Theorem 2 *Let us say that a command B is* idempotent *iff* $[\![B; B]\!] = [\![B]\!]$. *For any commands C and D, and any idempotent command B*

$$[\![B; C; D]\!] = [\![C; D]\!] \iff [\![C]\!] : ker(B) \twoheadrightarrow ker^*(D)$$

Proof. $(\Rightarrow)$: Assume $[\![B; C; D]\!] = [\![C; D]\!]$. Let s_1, s_2 denote arbitrary states such that $s_1\ ker(B)\ s_2$. We are required to show that $[\![C]\!]s_1\ ker^*(D)\ [\![C]\!]s_2$. Now

$$\begin{array}{ll}
[\![C; D]\!]s_1 & \\
= [\![B; C; D]\!]s_1 & \text{Assn.} \\
= [\![C; D]\!]^* \circ [\![B]\!]s_1 & \\
= [\![C; D]\!]^* \circ [\![B]\!]s_2 & ker(B)\ \text{defn.} \\
= [\![B; C; D]\!]s_2 & \\
= [\![C; D]\!]s_2. &
\end{array}$$

Now $[\![C; D]\!]s_1 = [\![C; D]\!]s_2$ implies

$$[\![D]\!]^*([\![C]\!]s_1) = [\![D]\!]^*([\![C]\!]s_1)$$

i.e.,

$$[\![C]\!]s_1\ ker^*(D)\ [\![C]\!]s_2$$

as required.

($\Leftarrow$:) Assume $[\![C]\!] : ker(B) \twoheadrightarrow ker^*(D)$. Now $[\![B]\!]^*([\![B]\!]^*A) = [\![B]\!]^*A$ by idempotence, which implies

$$\begin{array}{ll}
[\![B]\!]^*A\ ker^*(B)\ A & \\
\Longrightarrow [\![C]\!]^*([\![B]\!]^*A)\ ker^*(D)\ [\![C]\!]^*A & \text{assn.} \\
\iff [\![D]\!]^*([\![C]\!]^*([\![B]\!]^*A)) = [\![D]\!]^*([\![C]\!]^*A) & \\
\Longrightarrow [\![B; C; D]\!]^*A = [\![C; D]\!]^*A & \square
\end{array}$$

Corollary. Since $[\![HH]\!]$ is idempotent we can conclude that

$$[\![HH;C;HH]\!] = [\![C;HH]\!] \iff$$
$$[\![C]\!] : ker(HH) \twoheadrightarrow ker^*(HH).$$

It remains to establish the first fact.

Theorem 3 $\mathcal{P}[All \times Id]_\perp = ker^*(HH)$

Proof. ($\Rightarrow$): Suppose $A\ \mathcal{P}[All \times Id]_\perp\ B$, need to show $[\![HH]\!]^* A = [\![HH]\!]^* B$. Cases

$$\begin{array}{ll}
\perp \in [\![HH]\!]^* A & \\
\quad \Longrightarrow \ \perp \in A & HH\ \perp\text{-reflecting} \\
\quad \Longrightarrow \ \perp \in B & \\
\quad \Longrightarrow \ \perp \in [\![HH]\!]^* B & \\
(h, l) \in [\![HH]\!]^* A & \\
\quad \Longrightarrow \ \exists h_0 .\, (h_0, l) \in A & \text{defn. } (\cdot)^* \\
\quad \Longrightarrow \ \exists h_1 .\, (h_1, l) \in B & \\
\quad \Longrightarrow \ \forall h' .\, (h', l) \in [\![HH]\!]^* B &
\end{array}$$

($\Leftarrow$): For the other direction assume $[\![HH]\!]^* A = [\![HH]\!]^* B$.

$$\begin{array}{lll}
\perp \in A & \Longrightarrow & \perp \in B \\
(h, l) \in A & \Longrightarrow & \exists h_0 .\, (h_0, l) \in [\![HH]\!]^* A \\
 & \Longrightarrow & \exists h_0 .\, (h_0, l) \in [\![HH]\!]^* B \\
 & \Longrightarrow & \exists h_1 .\, (h_1, l) \in B \qquad \square
\end{array}$$

Thus, the equational and per security conditions in this simple case are equivalent.

In a more recent extension of the paper, [LJ99], Leino and Joshi update their relational semantics to handle termination-sensitive leakages and introduce abstract variables — a way to support partially confidential data. Abstract variables h and l are defined as functions of the concrete variables in a program. For example, for a list of low length and high elements l would be the length of the list and h would be the list itself. In the general case the choice of h and l could be independent, so an *independence condition* must be verified.

Abstract variables are easily represented in our setting. Suppose that some function $g \in \mathbf{St} \to D$ yields the value (in some domain D) of the abstract low variable from any given state, then we can represent the security condition on abstract variables by: $[\![C]\!] : R_g \twoheadrightarrow \mathcal{P}_{\mathsf{C}}[(All \times Id)_\perp]$ where $s_1 R_g s_2 \iff g\, s_1 = g\, s_2$.

5 A Probabilistic Security Condition

There are still some weaknesses in the security condition when interpreted in the convex powerdomain when it comes to the consideration of nondeterministic programs. In the usual terminology of information flow, we have considered *possibilistic information flows.* The probabilistic nature of an implementation may allow *probabilistic information flows* for "secure" programs. Consider the program

$$h := h \ \mathtt{mod}\ 100; (l := h \mathbin{[\!]} l := \mathtt{rand}(99)).$$

This program is secure in the convex powerdomain interpretation since regardless of the initial value of h, the final value of l can be any value in the range $\{0 \dots 99\}$. But with a reasonably fair implementation of the nondeterministic choice and of the randomised assignment, it is clear that a few runs of the program, for a fixed input value of h, could yield a rather clear indication of its value by observing only the possible final values of l, e.g.:

$$17, 2, 45, 2, 2, 33, 2, 97, 2, 8, 57, 2, 2, 66, \dots$$

– from which we might reasonably conclude that the value of h was 2. This observation is not new to this paper. It was first addressed in an abstract setting by Gray [Gra90]. The above example is adapted from McLean [McL90a]. In the setting of a concrete imperative language, Volpano and Smith recently devised a probabilistic security type-system [VS99] with a soundness proof based on a probabilistic operational semantics. They do not provide a *definition* of what it means for a program to be secure, and the security condition implicit in their correctness argument is not directly comparable – due to the fact that they consider parallel deterministic threads and a non-compositional semantics.

To counter this problem indicated by the example we consider *probabilistic powerdomains* [JP89] which allow the probabilistic nature of choice to be reflected in the semantics of programs, and hence enable us to capture the fact that varying the value of h causes *a change in the probability distribution* of values of l.

5.1 Probabilistic Powerdomain of Distributions

In the "possibilistic" setting we had the denotation of a command C to be a continuous function in $[\mathbf{St}_\perp \to \mathcal{P}_{\mathsf{C}}[\mathbf{St}_\perp]]$. In the probabilistic case, given an input to C not only do we keep track of possible outputs, but also of probabilities at which they appear. Thus, we consider a domain $\mathcal{E}[\mathbf{St}_\perp]$ of distributions over $\mathbf{St}_\perp$. The denotation of C is going to be a function in $[\mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]]$.

The general probabilistic powerdomain construction from [JP89] on an inductive partial order $\mathcal{E}[D]$ is taken to be the domain of *evaluations*, which are certain continuous functions on $\Omega(D) \to [0,1]$, where $\Omega(D)$ is the lattice of open subsets of D. We will omit a description of the general probabilistic powerdomain of evaluations since for the present paper it is sufficient and more intuitive to work with discrete powerdomains[5], and hence a simplified notion of probabilistic powerdomain in terms of distributions. Evaluations rather than distributions would be needed to give a probabilistic semantics to a concurrent language based on the domain of *resumptions* [JP89] – but this is beyond the scope of the present article.

If S is a set (e.g., the domain of states for a simple sequential language) then we define the probabilistic powerdomain of $S_\perp$, written $\mathcal{E}[S_\perp]$ to be the domain of *distributions* on $S_\perp$, where a distribution μ, to be a function from $S_\perp$ to $[0,1]$ such that $\sum_{d \in S_\perp} \mu d = 1$. The ordering on $\mathcal{E}[S_\perp]$ is defined pointwise by $\mu \leq \nu$ iff $\forall d \neq \perp . \mu d \leq \nu d$. This structure is isomorphic to Jones and Plotkin's probabilistic powerdomain of evaluations for this special case.

As a simple instance of the probabilistic powerdomain construction from [JP89], one can easily see that $\mathcal{E}[S]$ is an inductively complete partial order with directed lubs defined pointwise, and with a least element $\omega = \eta_S(\perp)$, where η_S is the *point-mass distribution* defined for an $x \in S$ by

$$\eta_S(x)d = \begin{cases} 1, & \text{if } d = x, \\ 0, & \text{otherwise.} \end{cases}$$

To lift a function $f : D_1 \to \mathcal{E}[D_2]$ to type $\mathcal{E}[D_1] \to \mathcal{E}[D_2]$ we define the *extension* of f by

$$f^*(\mu)(y) = \sum_{x \in D_1} f(x)(y) * \mu(x).$$

The structure $(\mathcal{E}[D], \eta_D(x), *)$ is a *Kleisli triple*, and thus we have a canonical way of composing the probabilistic semantics of any two given programs. Suppose $f : D_1 \to \mathcal{E}[D_2]$ and $g : D_2 \to \mathcal{E}[D_3]$ are such. Then the lifted composition $(g^* \circ f)^*$ can be computed by one of the Kleisli triple laws as $g^* \circ f^*$.

[5] By *discrete powerdomain* we mean the powerdomain of a flat domain, such as $\mathbb{N}_\perp$.

5.2 Pers on Probabilistic Powerdomains

The next step towards the security condition is to define how pers work on discrete probabilistic powerdomains. To lift pers to $\mathcal{E}[D]$ we need to consider a definition which takes into consideration the whole of each R-equivalence class in one go. The intuition is that an equivalence class of a per is a set of points that are indistinguishable by a low-level observer. For a given evaluation, the probability of a given observation by a low level user is thus the sum of probabilities over all elements of the equivalence class.

Define the per relation $\mathcal{E}[R]$ on $\mathcal{E}[D]$ for $\mu, \nu \in \mathcal{E}[D]$ by

$$\mu\ \mathcal{E}[R]\ \nu \text{ iff}$$
$$\text{for all } R\text{-equivalence classes } e : \sum_{d \in e} \mu d = \sum_{d \in e} \nu d.$$

Naturally, $\mu\ \mathcal{E}[Id]\ \nu \iff \mu = \nu$ and $\forall \mu, \nu \in \mathcal{E}[D].\, \mu\ \mathcal{E}[All]\ \nu$.

As an example, consider $\mathcal{E}[(All \times Id)_\perp]$. Two distributions μ and ν in $(All \times Id)_\perp \to [0,1]$ are equal if the probability of any given low value l in the left-hand distribution, given by $\sum_h \mu(h,l)$, is equal to the probability in the right-hand distribution, namely $\sum_h \nu(h,l)$.

The probabilistic security condition is indeed a strengthening of the possibilistic one – when we consider programs whose possibilistic and probabilistic semantics are in agreement.

Theorem 4 *Suppose we have a possibilistic (convex) semantics* $[\![\cdot]\!]_{\mathsf{C}}$ *and a probabilistic semantics* $[\![\cdot]\!]_{\mathcal{E}}$*, which satisfy a basic consistency property that for any command* C*, if* $[\![C]\!]_{\mathcal{E}} i\, o > 0$ *then* $o \in [\![C]\!]_{\mathsf{C}} i$*.*

Now suppose that R *and* S *are equivalence relations on* D*. Suppose further that* C *is any command such that possibilistic behaviour agrees with its probabilistic behaviour, i.e.,* $o \in [\![C]\!]_{\mathsf{C}} i \implies [\![C]\!]_{\mathcal{E}} i\, o > 0$*. Then we have that* $[\![C]\!]_{\mathcal{E}} : R \twoheadrightarrow \mathcal{E}[S]$ *implies* $[\![C]\!]_{\mathsf{C}} : R \twoheadrightarrow \mathcal{P}_{\mathsf{C}}[S]$*.*

Proof. For some $a, b \in D$ let $\mu = [\![C]\!]_{\mathcal{E}} a$ and $\nu = [\![C]\!]_{\mathcal{E}} b$. By $[\![C]\!]_{\mathcal{E}} : R \twoheadrightarrow \mathcal{E}[S]$ deduce $a\ R\ b \implies \forall d \in D.\, \sum_{e \in [d]_S} \mu e = \sum_{e \in [d]_S} \nu e$. What we need to prove is $a\ R\ b \implies [\![C]\!]_{\mathsf{C}} a\ \mathcal{P}_{\mathsf{C}}[S]\ [\![C]\!]_{\mathsf{C}} b$. So, assume $a\ R\ b$ and let us show that

$$\forall x \in [\![C]\!]_{\mathsf{C}} a.\, \exists y \in [\![C]\!]_{\mathsf{C}} b.\, x\ S\ y.$$

Take any $x \in [\![C]\!]_{\mathsf{C}} a$. Observe, that $\mu x > 0$, since if x is a possible output of program C run on data a, then the probability of getting this output must be greater then 0. Therefore, $\sum_{e \in [x]_S} \mu e = \sum_{e \in [x]_S} \nu e$. There must exist a $y \in [x]_S$ such that $\nu y > 0$. Thus, y is a possible output of program C run on data b, and $y \in [\![C]\!]_{\mathsf{C}} b$. $y \in [x]_S$ implies $x\ S\ y$. $\square$

5.3 Probabilistic Security Condition for the Case of Two Variables

Let us derive the probabilistic powerdomain security condition for the case of two variables h and l and domain $D = \mathbf{St}_\perp$. C is secure iff

$$\begin{array}{ll} & [\![C]\!]_\mathcal{E} : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp] \\ \iff & (i_h, i_l)\ (All \times Id)_\perp\ (i'_h, i'_l) \Longrightarrow \\ & [\![C]\!]_\mathcal{E}(i_h, i_l)\ \mathcal{E}[(All \times Id)_\perp]\ [\![C]\!]_\mathcal{E}(i'_h, i'_l) \\ \iff & \forall i_h, i'_h, i_l.\, [\![C]\!]_\mathcal{E}(i_h, i_l)\ \mathcal{E}[(All \times Id)_\perp]\ [\![C]\!]_\mathcal{E}(i'_h, i_l) \\ & \text{Let } \mu = [\![C]\!]_\mathcal{E}(i_h, i_l) \text{ and } \nu = [\![C]\!]_\mathcal{E}(i'_h, i_l) \\ \iff & \forall (o_h, o_l) \in \mathbf{St}_\perp.\ \sum_{e \in [(o_h, o_l)]_{(All \times Id)_\perp}} \mu e = \\ & \sum_{e \in [(o_h, o_l)]_{(All \times Id)_\perp}} \nu e \\ \iff & \forall o_l \in \mathbf{St}_{low}.\mu\perp = \nu\perp\ \ \& \\ & \sum_{h_1 \in \mathbf{St}_{high}} \mu(h_1, o_l) = \sum_{h'_1 \in \mathbf{St}_{high}} \nu(h'_1, o_l) \end{array}$$

So, a command C is secure iff

$$[\![C]\!]_\mathcal{E}(i_h, i_l)\ \perp = [\![C]\!]_\mathcal{E}(i'_h, i_l)\ \perp \quad \text{and}$$

$$\sum_{h \in \mathbf{St}_{high}} [\![C]\!]_\mathcal{E}(i_h, i_l)(h, o_l) = \sum_{h \in \mathbf{St}_{high}} [\![C]\!]_\mathcal{E}(i'_h, i_l)(h, o_l)$$

for any i_l, i_h, i'_h and o_l. Intuitively the equation means that if you vary i_h the distribution of low variables (the sums provide "forgetting" the highs) does not change.

Let us introduce probabilistic powerdomain semantics definitions for some language constructs. Here we omit the $\mathcal{E}$-subscripts to mean the probabilistic semantics. Given two programs C_1, C_2 such that $[\![C_1]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ and $[\![C_2]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ the composition of two program semantics is defined by:

$$[\![C_1; C_2]\!]\ i\ o = \sum_{s \in \mathbf{St}_\perp} ([\![C_1]\!]\ i\ s) \cdot ([\![C_2]\!]\ s\ o).$$

The semantics of the uniformly distributed nondeterministic choice $C_1 [\!] C_1$ is defined by:

$$[\![C_1 [\!] C_2]\!]\ i\ o = 0.5[\![C_1]\!]\ i\ o + 0.5[\![C_2]\!]\ i\ o.$$

Consult [JP89] for an account of how to define the semantics of other language constructs.

$$
\begin{aligned}
C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid C_1 \, [\!]_p \, C_2 \\
& \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C
\end{aligned}
$$

Figure 1: Command syntax

Example. Recall the program

$$h := h\ \mathsf{mod}\ 100; (l := h \, [\!] \, l := \mathsf{rand}(99))$$

Now we investigate the security condition by varying the value of h from 0 to 1. Take $i_l = 0, i_h = 0, i'_h = 1$ and $o_l = 0$. The left-hand side is

$$\sum_{h \in [0,\dots,100]} [\![C]\!]_{\mathcal{E}}(0,0)(h,0) = 0.5 \cdot 1 + 0.5 \cdot 0.01 = 0.505,$$

whereas the right-hand side is

$$\sum_{h \in [0,\dots,100]} [\![C]\!]_{\mathcal{E}}(1,0)(h,0) = 0.5 \cdot 0 + 0.5 \cdot 0.01 = 0.005.$$

So, the security condition does not hold and the program must be rejected.

6 Analysis for Security Properties

The compositionality of the security condition can be fruitfully exploited when proving correctness of various compositional analyses. Instances of such are type-system-based analyses. In this section we investigate the compositionality properties of the security condition and show how an example type system of [VS97], compositionally extended to handle probabilistic choice, can be proved correct with respect to the probabilistic security condition of Section 5.

6.1 A Simple Sequential Language with Probabilistic Choice

Consider a very simple sequential language with probabilistic choice ($[\!]_p$), described by the grammar in Figure 1. The semantics of the language is defined in Figure 2. The definition is an adoptation of the probabilistic semantics of Jones and Plotkin [JP89]. The $eval(Exp, s)$ function evaluates

$$
\begin{aligned}
[\![\mathsf{skip}]\!] &= \lambda i.\eta_i \\
[\![Id := Exp]\!] &= \lambda i.\eta_{i'},\ \text{where } i' = [eval(Exp, i)/Id]i \\
[\![C_1; C_2]\!] &= [\![C_2]\!]^* \circ [\![C_1]\!],\ \text{where } (f^*\ \mu)o = \sum_{s \in \mathbf{St}} (\mu s) \cdot (f\ s\ o) \\
[\![C_1 \ [\!]_p\, C_2]\!] &= p \cdot [\![C_1]\!] + (1-p) \cdot [\![C_2]\!] \\
[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] &= \lambda i.\mathbf{if}\ eval(B, i)\ \mathbf{then}\ [\![C_1]\!]\ i\ \mathbf{else}\ [\![C_2]\!]\ i \\
[\![\mathsf{while}\ B\ \mathsf{do}\ C]\!] &= \mu f.\lambda s.\mathbf{if}\ eval(B, s)\ \mathbf{then}\ (f^* \circ [\![C]\!])s\ \mathbf{else}\ \eta_s
\end{aligned}
$$

Figure 2: Probabilistic semantics of commands

a boolean or arithmetic expression Exp in a state s. A state is a tuple of variable values. Given an input state, executing commands skip and $Id := Exp$ results in the point-mass distribution for the input state (updated in the case of assignment $Id := Exp$).

Similarly to Section 3, for each function $f \in [D_\perp \to \mathcal{E}[E_\perp]]$ there exits a unique extension of f, denoted f^* where $f^* \in [\mathcal{E}[D_\perp] \to \mathcal{E}[E_\perp]]$ defined by

$$(f^*\ \mu)o = \sum_{s \in D_\perp} (\mu s) \cdot (f\ s\ o).$$

The extension is the unique mapping such that $f = f^* \circ \eta$. Using the extension we define the semantics of the sequential composition.

When running $C_1 \ [\!]_p\, C_2$, command C_1 is chosen for execution with probability p and C_2 is chosen with probability $1-p$. The existence of the least fixpoint μ in the definition of the semantics for while is guaranteed by the ipo construction of [JP89].

Let us first establish the compositional nature of the probabilistic security condition and then go on with a straightforward proof of correctness of the type system.

6.2 Hook-up Properties

We start with a proposition that will be useful when reasoning about sequential composition of security types.

Proposition 2 *For any $f \in [\mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]]$ and any R, $S \in Per(\mathbf{St}_\perp)$,*

$$f : R \twoheadrightarrow \mathcal{E}[S] \iff f^* : \mathcal{E}[R] \twoheadrightarrow \mathcal{E}[S]$$

Proof. ($\Rightarrow$): Assume $f : R \rightarrow\!\!\!\!\!\!\triangleright\ \mathcal{E}[S]$. It means $i_1\ R\ i_2 \Longrightarrow f(i_1)\ \mathcal{E}[S]\ f(i_2)$ or, using the low $f = f^* \circ \eta$:

$$i_1\ R\ i_2 \Longrightarrow \forall e_S \in \mathbf{St}_\perp/S.\,(f^*\eta_{i_1})e_S = (f^*\eta_{i_2})e_S.$$

Suppose $\mu\ \mathcal{E}[R]\ \nu$. Need to show $(f^*\mu)\ \mathcal{E}[S]\ (f^*\nu)$ or, unwinding the definition of the per $\mathcal{E}[S]$: $\forall e_S \in \mathbf{St}_\perp/S.\,(f^*\mu)e_S = (f^*\nu)e_S$. For any $e_S \in \mathbf{St}_\perp/S$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(f^*\mu)e_S} \\
 & \text{rewrite } \mu(x) = \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r)\cdot\eta_r(x) \\
= & (f^*(\sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r)\cdot\eta_r))e_S \\
 & \text{distributivity of } \cdot \text{ and } + \text{ over } f^* \\
= & (\sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r)\cdot(f^*\eta_r))e_S \\
 & \text{distributivity of } \cdot \text{ and } + \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r)\cdot((f^*\eta_r)e_S) \\
 & \text{lhs of proposition} \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r)\cdot((f^*\eta_{r_0})e_S) \\
 & \text{for some } r_0 \text{ such that } [r_0]_R = e_R \\
= & \sum_{e_R \in \mathbf{St}_\perp/R}((f^*\eta_{r_0})e_S)\cdot\sum_{r \in e_R}\mu(r) \\
= & \sum_{e_R \in \mathbf{St}_\perp/R}((f^*\eta_{r_0})e_S)\cdot\mu(e_R) \\
 & \text{use } \forall e_R.\,\mu(e_R) = \nu(e_R) \\
= & \sum_{e_R \in \mathbf{St}_\perp/R}((f^*\eta_{r_0})e_S)\cdot\nu(e_R) \\
= & (f^*\nu)e_S
\end{array}$$

($\Leftarrow$): Suppose $r_1\ R\ r_2$. Need to show $f(r_1)\ \mathcal{E}[S]\ f(r_2)$.

$$\begin{array}{lll}
\multicolumn{3}{l}{r_1\ R\ r_2} \\
\Longrightarrow & \eta_{r_1}\ \mathcal{E}[R]\ \eta_{r_2} & \\
\Longrightarrow & f^*(\eta_{r_1})\ \mathcal{E}[S]\ f^*(\eta_{r_2}) & \text{rhs of proposition} \\
\Longrightarrow & f(r_1)\ \mathcal{E}[S]\ f(r_2) & f = f^* \circ \eta \quad \square
\end{array}$$

Corollary. From this proposition it easily follows that the following inference rule is sound:

$$\frac{[\![C_1]\!] : P \rightarrow\!\!\!\!\!\!\triangleright\ \mathcal{E}[Q] \qquad [\![C_2]\!] : Q \rightarrow\!\!\!\!\!\!\triangleright\ \mathcal{E}[R]}{[\![C_1; C_2]\!] : P \rightarrow\!\!\!\!\!\!\triangleright\ \mathcal{E}[R]}$$

In the standard security terminology, the *hook-up* property [McC87] says that if two programs are secure then their composition (sequential or some other kind) is secure is well. We will investigate what kinds of composition are allowed. These properties are important since they are the key to the utility of the specification for the purpose of proving the correctness of syntax-directed program analyses.

Recall the probabilistic powerdomain security condition for the case of two variables h and l and domain $D = \mathbf{St}_\perp$. A command C is secure iff

$$[\![C]\!]_{\mathcal{E}}(i_h, i_l) \perp = [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \perp \quad \text{and}$$

$$\sum_{h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i_h, i_l)(h, o_l) = \sum_{h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i'_h, i_l)(h, o_l)$$

for any i_l, i_h, i'_h and o_l.

We will need some definitions to study various kinds of composition. A (boolean or arithmetic) expression Exp is *low* iff $\lambda s.eval(Exp, s) : All \times Id \twoheadrightarrow Id$. Otherwise, the expression is *high*.

Not any context can be used for plugging secure programs to obtain a secure program. (Take, for example, any ground context that is insecure.) The security property is preserved by *secure contexts*, i.e. contexts built with secure components. Let $[\bullet]$ be a hole for a command. A context $\mathbb{C}[\bullet_1, \bullet_2]$ is secure iff it has one of the following forms:

$$\begin{aligned} \mathbb{C}[\bullet_1, \bullet_2] ::= {} & \mathsf{skip} \\ & \mid h := Exp \\ & \mid l := Exp \ (Exp \text{ is low}) \\ & \mid [\bullet_1]; [\bullet_2] \\ & \mid [\bullet_1] \ [\!]_p \ [\bullet_2] \\ & \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low }) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low }) \end{aligned}$$

Theorem 5 (Hook-up) *If C_1, C_2 are secure and $\mathbb{C}[\bullet_1, \bullet_2]$ is a secure context, then $\mathbb{C}[C_1, C_2]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] =$ if B then $[\bullet_1]$ else $[\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided*

$$[\![C_1]\!] \ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]) \ [\![C_2]\!].$$

If $\mathbb{C}[\bullet_1, \bullet_2] =$ while B do $[\bullet_1]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided

$$\begin{array}{rl} (i) & \forall i, j \in \mathbf{N}.\, [\![C_1]\!]^i \ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]) \ [\![C_1]\!]^j \\ (ii) & \forall s.\, [\![\mathbb{C}[C_1, C_2]]\!] \ s \perp = 0. \end{array}$$

Proof. Cases on $\mathbb{C}[C_1, C_2] = C$.

skip

Start with s_1 and s_2 ($s_1 =_L s_2$). With either state, the computation of skip terminates in one step with the same states. So, the final states are low-equal as well.

$l := Exp$ **(**Exp **is low)**

Since Exp is a low expression, the computation of the command with different initial states s_1 and s_2 ($s_1 =_L s_2$) terminates with the same states that are still low-equal.

$h := Exp$

Exp can be arbitrary. Computing the command will not change the low component of the states.

$C_1; C_2$

We know that $[\![C_i]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]$. Applying the inference rule of the corollary of Proposition 2 yields $[\![C_2]\!]^* \circ [\![C_1]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]$. Thus, C is secure.

$C_1 \,[\!]_p\, C_2$

Trivial, as computation does not depend on h.

if B **then** C_1 **else** C_2 **(**B **is low)**

Start with s_1 and s_2 ($s_1 =_L s_2$). B has the same value in either state. Thus, one step of computation will lead to a configuration with the same command (C_1 or C_2 in both cases) and low-equal states. C_1 and C_2 are secure and, therefore so is C.

if B **then** C_1 **else** C_2 **(**B **is high)**

Start with s_1 and s_2 ($s_1 =_L s_2$). Here the value of B might differ for s_1 and s_2, but, due to the assumption of the theorem, we have $[\![C_i]\!]\,(\mathcal{E}[(All \times Id)_\perp])\,[\![C_j]\!]$ for $i, j \in \{1, 2\}$. This guarantees the security of C.

while B **do** C_1 **(**B **is low)**

Start with s_1 and s_2 ($s_1 =_L s_2$). B has the same value in either state. Thus, the fixpoints in the semantics of the while under both initial states will be the same.

while B **do** C_1 **(**B **is high)**

Start with s_1 and s_2 ($s_1 =_L s_2$). The value of B might differ for s_1 and

s_2. However, from the assumption for this case we know that (ii) the while-loop terminates and (i) any number of C_1-iterations will result in the same distribution of the final values of l. □

6.3 Type-system-based Analysis

Let us now exemplify how the compositional nature of the security condition can be used for straightforward proofs of correctness of compositional analyses. In this subsection we consider a nontermination-covert-flow-sensitive type system by Volpano and Smith's [VS97], adapted to our language and extended to handle probabilistic choice. Note that this is an instance of a particular analysis that will reject some secure programs. The compositionality of the security condition can be used for justification of any compositional analysis of arbitrary precision. The point of this subsection is not to develop a powerful analysis, but rather to illustrate how simple the proof technique is using the compositional security condition.

Let us now describe the type system. The basic idea is to give security types (note the difference between these types and semantic security types used in security specification) to expressions (*high* or *low*) and commands (*high cmd* or *low cmd*) and make sure that these types do not mismatch when composing the commands. The main job of the types is to make sure that (i) an expression of type *low* does not contain an occurrence of the high variable h, (ii) a command of type *high cmd* does not have an occurrence of assignment to the low variable l and (iii) while-loops have the type *low cmd*. These properties can be easily proved by induction.

The type system is presented in Figure 3. The rule Assign_{low} prevents direct insecure information flows – for example, the assignment $l := h$ is not typable. The rule If prevents indirect insecure flows – the command if $h = 0$ then $l := 0$ else $l := 1$ cannot be typed since the guard has type *high* and the branches have type *low cmd*. In this particular system, the guard of the while-loop has to be low to prevent nontermination leaks. The while-loop has to have type *low cmd*. This guarantees that a while cannot be placed in a branch of a high conditional, with could also introduce a potential nontermination leak.

Let us now prove the security of the type system. We start off with a simple lemma. The lemma intuitively says that the behaviour of two commands with no loops or assignments to l can only differ in the high output as one varies the high input.

Lemma 1 $C_1 : \mathit{high\ cmd}, C_2 : \mathit{high\ cmd} \Longrightarrow$
$[\![C_1]\!]\ ((\mathit{All} \times \mathit{Id})_\perp \twoheadrightarrow \mathcal{E}[(\mathit{All} \times \mathit{Id})_\perp])\ [\![C_2]\!]$

[Var]	$h : high \qquad l : low$
[Exp]	$n : \tau \qquad Exp : high$
[Arith$_{low}$]	$\dfrac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$
[Skip]	$\mathsf{skip} : \tau\ cmd$
[Assign$_{low}$]	$\dfrac{Exp : low}{l := Exp : low\ cmd}$
[Assign$_{high}$]	$h := Exp : \tau\ cmd$
[Seq]	$\dfrac{C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{C_1; C_2 : \tau\ cmd}$
[Choice]	$\dfrac{C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{C_1 \,[\,]_p\, C_2 : \tau\ cmd}$
[If]	$\dfrac{B : \tau \quad C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \tau\ cmd}$
[While]	$\dfrac{B : low \quad C : \tau\ cmd}{\mathsf{while}\ B\ \mathsf{do}\ C : low\ cmd}$

Figure 3: Type system for security

Proof. We need to show

$$[\![C_1]\!]_{\mathcal{E}}(i_h, i_l)\ \bot = [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)\ \bot \quad \text{and}$$

$$\sum_{h \in \mathbf{St}_{high}} [\![C_1]\!]_{\mathcal{E}}(i_h, i_l)(h, o_l) = \sum_{h \in \mathbf{St}_{high}} [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)(h, o_l)$$

for any i_l, i_h, i'_h and o_l. Clearly, $[\![C_1]\!]_{\mathcal{E}}(i_h, i_l)\ \bot = 0 = [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)\ \bot$ since nontermination is only possible in the presence of a while-loop. By property (iii) of the type system, no while-loop can occur in a high command. From the fact that there are no assignments to l ((ii)-property), deduce

$$\sum_{h \in \mathbf{St}_{high}} [\![C_i]\!]_{\mathcal{E}}(i_h, i_l)(h, o_l) = \begin{cases} 1, & \text{if } i_l = o_l, \\ 0, & \text{otherwise.} \end{cases}$$

for both $i = 1, 2$. This implies the second equality. $\square$

The following theorem is a straightforward utilisation of the hook-up properties shown in Theorem 5.

Theorem 6 (Security of the Analysis)

$C : \tau\ cmd \implies C$ *is secure.*

Proof. Induction on C and direct application of the hook-up theorem. Let us consider the cases by which $C : \tau\ cmd$. The cases $\mathsf{skip} : \tau\ cmd$, $l := Exp : low\ cmd$ $(Exp : low)$ and $h := Exp : \tau\ cmd$ are immediate since C is a ground secure context. (The case $l := Exp : low\ cmd$ $(Exp : low)$ relies on the (i)-property of the type system which guarantees $Exp : low \implies Exp$ is low.)

The cases $C_1; C_2 : \tau\ cmd$, $C_1\ [\!]_p\ C_2 : \tau\ cmd$, $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : low\ cmd$ $(B : low)$ and $\mathsf{while}\ B\ \mathsf{do}\ C : low\ cmd$ $(B : low)$ provide secure contexts $[\bullet_1]; [\bullet_2]$, $[\bullet_1] [\!]_p [\bullet_2]$, $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ and $\mathsf{while}\ B\ \mathsf{do}\ [\bullet]$ (B is low by in the two latter by the (i)-property) respectively. By the the induction hypothesis the sub-commands are secure, so the conclusion follows by the hook-up theorem.

The last case is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : high\ cmd$. Both C_1 and C_2 have type $high\ cmd$. By Lemma 1 we have $[\![C_1]\!]\ ((All \times Id)_\bot \twoheadrightarrow \mathcal{E}[(All \times Id)_\bot])\ [\![C_2]\!]$. Apply the if-on-high case of Theorem 5. $\square$

7 Conclusions

We have developed an extensional semantics-based specification of secure information flow in sequential programs, by embracing and extending earlier work on the use of partial equivalence relations to model binding times in [HS91]. We have shown how this idea can be extended to handle nondeterminism and also probabilistic information flow.

A final remark on some recent related work: Abadi, Banerjee, Heintze and Riecke [ABHR99] show that a single calculus (DCC), based on Moggi's computational lambda calculus, can capture a number of specific static analyses for security, binding-time analysis, program slicing and call-tracking. Although their calculus does not handle nondeterministic language features, it is notable that the semantic model given to DCC is Per-based, and the logical presentations of the abstract interpretation for Per-based BTA from [HS91, Jen92, HL94] readily fit this framework (although this specific analysis is not one of those considered in [ABHR99]). They also show that what we have called "termination insensitive" analyses (Section 2.3) can be modelled by extending the semantic relations to relate bottom (nontermination) to every other domain point (without insisting on transitivity). It is encouraging to note that – at least in the deterministic setting – the per-based approach can be weakened to a termination-insensitive condition without significant technical difficulties. We do not, however, see any obvious way to make the probabilistic security condition insensitive to termination in a similar manner.

We conclude by considering a few possible extensions and limitations:

Multi-level security There is no problem with handling lattices of security levels rather than the simple *high-low* distinction. But one cannot expect to assign any intrinsic semantic meaning to such lattices of security levels, since they represent a "social phenomenon" which is external to the programming language semantics. In the presence of multiple security levels one must simply formulate conditions for security by considering information flows between levels in a pairwise fashion (although of course a specific static analysis is able to do something much more efficient).

Downgrading and Trusting There are operations which are natural to consider but which cannot be modelled in an obvious way in an extensional framework. One such operation is the downgrading of information from high to low without losing information – for example representing the secure encryption of high level information. This seems impossible since an encryption operation does not lose information about a value and yet should have type *high* $\rightarrow$ *low* – but the only functions of type *high* $\rightarrow$ *low* are the constant functions. An analogous problem arises with Ørbæk and Palsberg's `trust` primitive if we try to use pers to model their *integrity analysis* [ØP97].

Operational Semantics We are not particularly married to the denotational perspective on programming language semantics. In [SS00] we investigate operational formulations of pers on a multi-threaded language, based on partial bisimulations.

Constructing Program Analyses Although the model seems useful to compare other formalisations, further work is needed to show that it can assist in the systematic design of program analyses.

Concurrency Handling nondeterminism can be viewed as the main stepping stone to formulating a language-based security condition for concurrent languages. We have made an investigation of per-based security conditions for a multi-threaded language, using an operational rather than denotational semantic model of security [SS00].

References

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.

[AP90] M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*. IEEE, 1990.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[BL76] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. MTR-2997, Rev. 1, The MITRE Corporation, Bedford, Mass., 1976.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[DRH95] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 100–110, La Jolla, California, June 1995. ACM.

[FG94] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994.

[GM82] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

[Gra90] J.W. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[HL94] C. L. Hankin and D. Le Métayer. A type-based framework for program analysis. In *Proceedings of the First Static Analysis Symposium*, volume 864 of *LNCS*. Springer-Verlag, 1994.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 19–21, 1998.

[HS91] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 154–164, September 1991. ACM SIGPLAN Notices 26(9).

[HS95] F. Henglein and D. Sands. A semantic model of binding times for safe partial evaluation. In Manuel Hermenegildo and S.D̃oaitse Swierstra, editors, *Proc. Programming Languages: Implementations, Logics and Programs (PLILP), Utrecht, The Netherlands*, volume 982 of *Lecture Notes in Computer Science*, pages 299–320. Springer-Verlag, September 1995.

[Hun90] S. Hunt. PERs generalise projections for strictness analysis. In *Draft Proceedings of the Third Glasgow Functional Programming Workshop*, Ullapool, 1990.

[Hun91] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[Jen92] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, University of London, November 1992. Available as DIKU Report 93/11 from DIKU, University of Copenhagen.

[JP89] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, 1989.

[LJ98] K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. In *MPC'98*, Springer Verlag LNCS, 1998.

[LJ99] K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 1999. To appear.

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[McL90a] John McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, California, 1990. IEEE Computer Society Press.

[McL90b] John McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

[McL94] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering.* Wiley & Sons, 1994.

[MS92] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[Nie90] F. Nielson. Two-level semantics and abstract interpretation — fundamental studies. *Theoretical Computer Science*, (69):117–242, 90.

[ØP97] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(4), 1997.

[Ørb95] Peter Ørbæk. Can you Trust your Data? In M. I. Schwartzbach P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[Plo81] G. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. North-Holland, Amsterdam, 1983.

[Smy78] Michael B. Smyth. Powerdomains. *Journal of Computer and Systems Sciences*, 16(1):23–36, February 1978.

[SS99] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming, ESOP'99*, LNCS 1576, pages 40–58, Amsterdam, March 1999. Springer-Verlag.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press. To appear.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.

[TK97] P. Thiemann and H. Klaeren. Binding-time analysis by security analysis. Universität Tübingen, November 1997.

[VS97] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, pages 385–407, Portland, Oregon, September 1987.

Probabilistic Noninterference for Multi-threaded Programs *

Andrei Sabelfeld David Sands

Chalmers†

Abstract

We present a probability-sensitive confidentiality specification – a form of *probabilistic noninterference* – for a small multi-threaded programming language with dynamic thread creation. Probabilistic covert channels arise from a scheduler which is probabilistic. Since scheduling policy is typically outside the language specification for multi-threaded languages, we describe how to generalise the security condition in order to define robust security with respect to a wide class of schedulers, not excluding the possibility of deterministic (e.g. round-robin) schedulers and program-controlled thread priorities. The formulation is based on an adaptation of Larsen and Skou's notion of *probabilistic bisimulation*. We show how the security condition satisfies compositionality properties which facilitate straightforward proofs of correctness for e.g. security type systems. We illustrate this by defining a security type system which improves on previous multi-threaded systems, and by proving it correct with respect to our stronger scheduler independent security condition.

*This is an extended version of the paper to appear in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press.

†Department of Computer Science, Chalmers University of Technology and the University of Göteborg, 412 96 Göteborg, Sweden. {andrei,dave}@cs.chalmers.se

1 Introduction

1.1 Motivation

When is an untrusted program safe to use? One aspect of safety is confidentiality. Given you have some confidential (high) data and some public (low) data in your computer, you want to make sure the attacker – the supplier of the untrusted code – will not learn anything about your personal data, despite the fact that the *application* (e.g. a spreadsheet) may require legitimate access to the confidential data in order to perform its task, and legitimate communication with the supplier of the code (e.g. a registration process for all users).

In this study we assume that the attacker is external to the (trusted) system upon which the program is run. Our aim is to specify when a program is safe to run – from the point of view of its confidentiality properties – with an aim to providing automatic methods for certifying programs prior to execution.

This paper describes some security specifications for multi-threaded programs executing on a single processor. Programs which execute several concurrent processes not only introduce nondeterminism, but also the potential for an attacker to observe the internal timing behaviours of programs, via the effect that the running time of commands might have on the scheduler's choice of when various threads are executed. Since a scheduler's behaviour might be probabilistic, this opens up an opportunity for the attacker to set up a probabilistic covert channel whereby information is obtained by statistical inferences drawn from the relative frequency of outcomes of a repeated computation.

We describe how *confidentiality* (the security property of not having insecure information flows) can be specified operationally for a simple imperative language with dynamic thread creation, and how the compositional nature of the security condition facilitates a straightforward proof of correctness of type-based program analyses.

An important contribution is to abstract over the scheduler's behaviour, so as to obtain a notion of security which holds for a broad class of schedulers, possibly including both probabilistic and deterministic behaviours. We illustrate this by defining a security type system which improves on a previous system of Volpano and Smith [VS99] for a simpler language), and by proving it correct with respect to our stronger explicitly scheduler-independent security condition.

1.2 Classification of Insecure Information Flows

It will be useful to informally classify a number of insecure information flows that might arise in programs. Let us restrict ourselves to a simplified setting: two variables h and l of high and low confidentiality respectively. Consider the examples of insecure information flows that security specifications and specific program analyses might attempt to rule out:

Direct: $l := h$
Direct information flow from the initial value of h to the final value of l.

Indirect: $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$
Secret information *indirectly* leaks to the final value of l.

Termination behaviour: $\mathsf{if}\ h = 1\ \mathsf{then\ loop}$
If the program terminates, the attacker knows that h was not 1.

Probabilistic:
$h := h\ \mathsf{mod}\ 2; (l := h \, [\!]_{\frac{1}{2}}\, (l := 0 \, [\!]_{\frac{1}{2}}\, l := 1))$
where $[\!]_{\frac{1}{2}}$ is a coin-flip choice operator. Here there is no information flow if we only consider the *possible* behaviours of the system, but the final value of l will reveal information about h with a probability of $\frac{3}{4}$.

Externally observable timing:
$\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100$
The attacker with a stop-watch can learn information about h. This paper does not consider externally observable timing.

Internally observable timing:
$((\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100); l := 1) \mid l := 0$
where $\mid$ denotes the concurrent execution of the two commands. With many schedulers, the timing leaks will translate into an effect on the (distribution of) final value(s) of l. The information flow in this case is a combination of the timing and potentially probabilistic flows in the case that the scheduler exhibits stochastic behaviour. The ability to dynamically create threads also leads to similar possibilities.

1.3 Related Work

There is a substantial body of work in the security community in studying definitions and reasoning principles relating to information flow and confidentiality. For an overview see e.g. [McL90]. The basic approach we take is based on an extentional characterisation usually known as *noninterference*. The particular flavour of noninterference studied here is most closely related to Gray's notion of *P-Restrictiveness* [Gra90], since it aims to eliminate dependence between classified data and the probability distribution of unclassified outputs.

The use of program analysis as a means to eliminate insecure flows was pioneered by Denning [Den76, DD77]. A modern incarnation of this work which recasts Denning's analysis as a type system and proves its correctness is [VSI96].

Semantically, our security condition is founded on a probabilistic notion of bisimulation. Focardi and Gorrieri have promoted the use of (non-probabilistic) bisimulation in formalising and analysing security conditions in process calculus setting (see e.g., [FG95]).

Focusing on concurrent imperative programs, there have been some recent considerations of concurrent programming languages. Banâtre, Bryce and Le Métayer [BBL95] consider a programming language with static process structure and CSP-like communication primitives (see also [AR80]), although there is no extensional specification of correctness in these studies, and probabilistic information flows are not considered. Heintze and Riecke [HR98] present an analysis for a security-oriented lambda calculus, extended with threads and imperative features. Although they prove a noninterference property for the functional core of the language, they (quote) *"have not proved a noninterference theorem for the concurrent setting because ... the notion is unclear in the concurrent setting"*. Smith and Volpano [SV98] extend their previous work ([VSI96]) with static threads. They prove a possibilistic correctness result for their analysis, and note that the analysis does not eliminate probabilistic covert channels (other than by making it overly restrictive). A later work [VS99] shows how to lift this restriction and proves some correctness properties based on a probabilistic semantics. We will return to consider this last work in more detail. In [SS99] we showed how partial equivalence relations (pers) and denotational semantics could capture security properties in sequential languages, including probabilistic information flows (noninterference-style) with the help of probabilistic powerdomains. In principle this machinery should be sufficient to handle concurrent languages, but the mathematical overheads are high. In this paper we also use partial equivalence relations, but this time operationally-based using probabilistic-bisimulation-like relations.

1.4 Overview

The rest of the paper is organised as follows. **Section 2** introduces syntax and probabilistic semantics of a multi-threaded language. **Section 3** defines the security condition based on probabilistic bisimulation. In **Section 4**, we introduce a scheduler-independent security condition. We go on to strengthen it by defining a strong security and illustrate its properties in **Section 5**. **Section 6** considers the applicability of the conditions to proving the soundness of compositional security analyses. **Section 7** concludes.

2 A Simple Concurrent Language

In order to be concrete – and in order to explore the compositionality properties of some of our security conditions – we consider a very simple shared-variable language described by the grammar in Figure 1. Let $C, D, E, \ldots$ range over commands *Com*, and let $\vec{C}$ denote a vector of commands of the form $\langle C_0 \ldots C_{n-1} \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\overrightarrow{Com} = \cup_{n \in \mathbb{N}} Com^n$, the set of thread pools (or programs).

We consider a small-step semantics for the language. A *configuration* c $(d, e, \ldots)$ is a pair of a command and a state (memory). In this paper, a *state* $s \in \mathbf{St}$ is a finite mapping from variables to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables respectively. In concrete examples we will often take just variables h and l, and in this case we will write the state as a pair. Define low-equivalence $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same. Let *Config* denote $\overrightarrow{Com} \times \mathbf{St}$. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoonup$ transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C'\vec{D}, s' |\!\rangle$. Here, one step of computation starting with command C in a state s gives a new main thread C', (possibly empty) vector of spawned processes $\vec{D}$ and a new state s'. Command $\mathsf{fork}(C\vec{D})$ dynamically creates a new vector of processes $\vec{D}$ which run in parallel with the main thread C. This has the effect of adding the vector of processes to the configuration.

The concurrent part of the semantics is presented in Figure 3. The $\rightarrow_p$ transitions are probabilistic. The scheduler modelled here is purely probabilistic; it chooses to execute one of the threads of $\vec{C}$ with equal probability. We call this a *uniform* scheduler. In Section 4 we will pa-

$$
\begin{aligned}
C ::= \; & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \\
& \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\vec{D})
\end{aligned}
$$

Figure 1: Command syntax

$$\langle\!|\mathsf{skip}, s|\!\rangle \rightharpoonup \langle\!|\langle\rangle, s|\!\rangle$$

$$\frac{\langle\!|Exp, s|\!\rangle \downarrow n}{\langle\!|Id := Exp, s|\!\rangle \rightharpoonup \langle\!|\langle\rangle, [Id = n]s|\!\rangle}$$

$$\frac{\langle\!|C_1, s|\!\rangle \rightharpoonup \langle\!|\langle\rangle, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightharpoonup \langle\!|C_2, s'|\!\rangle} \qquad \frac{\langle\!|C_1, s|\!\rangle \rightharpoonup \langle\!|C_1'\vec{D}, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightharpoonup \langle\!|(C_1'; C_2)\vec{D}, s'|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{True}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \rightharpoonup \langle\!|C_1, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{False}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \rightharpoonup \langle\!|C_2, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{True}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle \rightharpoonup \langle\!|C; \mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{False}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle \rightharpoonup \langle\!|\langle\rangle, s|\!\rangle}$$

$$\langle\!|\mathsf{fork}(C\vec{D}), s|\!\rangle \rightharpoonup \langle\!|C\vec{D}, s|\!\rangle$$

Figure 2: Small-step deterministic semantics of commands

$$\frac{\langle\!|C_i, s|\!\rangle \rightharpoonup \langle\!|\vec{C}, s'|\!\rangle}{\langle\!|\langle C_0 \ldots C_{n-1}\rangle, s|\!\rangle \rightarrow^{i}_{\frac{1}{n}} \langle\!|\langle C_0 \ldots C_{i-1}\vec{C}C_{i+1} \ldots C_{n-1}\rangle, s'|\!\rangle}$$

$$(\Sigma)\frac{p = \sum \{\!|q \,|\, \langle\!|\vec{C}, s|\!\rangle \rightarrow^{i}_{q} \langle\!|\vec{D}, s'|\!\rangle|\!\}}{\langle\!|\vec{C}, s|\!\rangle \rightarrow_{p} \langle\!|\vec{D}, s'|\!\rangle}$$

Figure 3: Uniform probabilistic semantics of thread pools

rameterise over the choice of scheduler. Write $\langle\!|\vec{C}, s|\!\rangle \to \langle\!|\vec{D}, s'|\!\rangle$ whenever $\exists p > 0.\, \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle$.

As is standard, in this paper we use multiset comprehensions $\{\!|p| \dots |\!\}$ when summing probabilities.

Labelling the transitions in the rules for parallel composition is performed in order to ensure that the sum of probabilities of transitions from a given nonterminal configuration is one. If there were no labels or rule Σ, then there would be $\langle\!|\langle \mathsf{skip} \mid \mathsf{skip}\rangle, s|\!\rangle \to_{1/2} \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$ rather than $\langle\!|\langle \mathsf{skip} \mid \mathsf{skip}\rangle, s|\!\rangle \to_1 \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$, where $\mid$ is used for separating threads in a thread pool. That is due to the fact that transition $\langle\!|\langle \mathsf{skip}, \mathsf{skip}\rangle, s|\!\rangle \to_{1/2} \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$ could be performed by two different rules. That cannot be observed without the labelling or by making similar distinctions between different ways of obtaining the same transition. Similar constructions can be found in other probabilistic semantics.

The probabilistic semantics satisfies the following "soundness" properties for all nonterminal programs $\vec{C}$:

$$(i) \forall s.\, \sum \{\!|p| \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle |\!\} = 1,$$
$$(ii) \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle, \langle\!|\vec{C}, s|\!\rangle \to_q \langle\!|\vec{D}, s'|\!\rangle \Longrightarrow p = q.$$

Observe that rule Σ guarantees property (ii) while property (i) must be independently enforced (and is dependent on expressions being total). Given these soundness conditions, the semantics can be viewed as an unlabelled *probabilistic transition system* [LS91], or a simple probabilistic automaton.

3 Probabilistic Noninterference

3.1 Background

Now that we have defined the syntax and semantics of the multi-threaded language, we are ready to specify what it means for a program to be secure. We are taking the view of *noninterference* to express the confidentiality property of a program. The central idea of noninterference [GM82] is that a program is secure whenever varying the initial values of high variables cannot change the low-observable (observable by the attacker) behaviour of the program.

A natural covert channel generalising the nontermination channel is the timing channel. A realistic version of noninterference should include timing as a possibly low-observable action. The first step in the direction of incorporating timing into noninterference was made by Volpano and Smith

[VS99]. Volpano and Smith's noninterference is based on a lock-step execution of *probabilistic states*. A distribution μ of a set X (write $\mu \in \mathcal{D}(X)$) is a function $\mu \in X \to [0,1]$ such that $\sum_{x \in X} \mu x = 1$. A probabilistic state μ is a distribution of configurations $\mu \in \mathcal{D}(\vec{C} \times \mathbf{St})$. The computation is modelled by a Markov chain of probabilistic states. The initial probabilistic state is $\langle\!|\vec{C}, s|\!\rangle \mapsto 1$ where s is the initial memory and $\vec{C}$ is a static pool of threads (they do not handle new thread generation). Every following probabilistic state μ' is computed by multiplying the row vector of probabilities from μ by the stochastic transition matrix T. The rows and columns of T correspond to possible configurations reachable from the initial one. There might be a countably infinite number of such states, so the matrix might be countably infinite. The elements $T(i,j)$ of T are the probabilities of a transition from configuration i to configuration j.

From the point of view of the low observer two probabilistic states are indistinguishable (write $\sim_l$) iff projecting out high variables (which involves collapsing several probabilistic states into one in case the commands and low parts of the memories are the same; see [VS99] for the formal definition) makes the two states the same.

From the main correctness theorem we can extract the following probabilistic noninterference condition for a program $\vec{C}$ with stochastic transition matrix T:

$$\mu_1 \sim_l \mu_2 \text{ implies } \mu_1 T \sim_l \mu_2 T. \qquad (*)$$

While intuitive and adequate to prove the security of Volpano and Smith's type-system based analysis as an analysis-independent *definition* of security it suffers from the three drawbacks:

Noncompositional The probabilistic states are distributions of configurations (elements of $\mathcal{D}(\vec{C} \times \mathbf{St})$). This makes compositional reasoning impossible — commands are parts of configurations together with memories.

Imprecise Syntactic equality for commands is involved in the definition of low-equivalence. Consider the following example (or indeed any program with conditionals and no low variables):

$$\mathsf{if}\ h < 0\ \mathsf{then}\ h := h + 1\ \mathsf{else}\ h := h - 1.$$

Intuitively, for the low observer this program has the same behaviour regardless of the initial value of h - since the program does not modify

l. However, it is considered insecure if one takes $(*)$ as a definition. Indeed, take $\mu_1 = \{\langle\!| \mathsf{if}\ h < 0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1, (7,0) |\!\rangle \mapsto 1\}$ and $\mu_2 = \{\langle\!| \mathsf{if}\ h < 0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1, (-7,0) |\!\rangle \mapsto 1\}$. Note $\mu_1 \sim_l \mu_2$. Now, $\mu_1 T = \{\langle\!| h := h-1, (7,0) |\!\rangle \mapsto 1\}$ and $\mu_2 T = \{\langle\!| h := h+1, (-7,0) |\!\rangle \mapsto 1\}$. After projecting out the high variable we are still left with syntactically different commands which implies $\mu_1 T \not\sim_l \mu_2 T$. To make the above example secure according to $(*)$, Volpano and Smith require that any such conditional with a high guard be executed atomically.

Scheduler Specific The security condition is specific to a particular scheduler, namely the scheduler with uniform probability of picking every thread. If the scheduler is changed then secure programs may become insecure. [1]

In the remainder of this section we begin by addressing the question of *precision* in the definition of security. The central problem of $(*)$ is that it does not abstract sufficiently from the syntactic program state. To abstract from the state we should factor in this (probabilistic) equivalence, in the manner of Gray's P-Restrictiveness condition [Gra90]. We will develop probabilistic noninterference criteria based on probabilistic bisimulation. The idea is based on the definition of bisimulation on probabilistic transition systems [LS91],[BK99].

3.2 Relational Definitions and Notation

Let us first introduce some standard definitions and notation for relations.

Let $Rel(A)$ denote the set of binary relations on A. If $R \in Rel(A)$ is an equivalence relation then A/R is the set of $R-$equivalence classes, $A/R \subseteq \wp(A)$. Id_A denotes the identity relation on some set A; we write just Id when A is clear from the context. As usual, R^+ denotes the transitive closure of R and R^* denotes the transitive and reflexive closure of R. For binary relations $P \in Rel(A)$, $Q \in Rel(B)$ we define the relation $P \times Q$ on $A \times B$ by:

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x'\ \&\ y\ Q\ y'.$$

A *partial equivalence relation* (per) on a set A is a binary relation on A which is *symmetric* and *transitive*. If P is such a per let $dom(P)$ denote the

[1] Note that this is not a problem for the actual security type system presented by Volpano and Smith; the proofs of correctness of their type system are easily seen to go through with other choices of probabilities in the transition matrix.

domain of P, given by

$$dom(P) = \{x \in A \mid x \; P \; x\}.$$

Note that the domain and range of a per P are both equal to $dom(P)$ (so for any $x, y \in A$, if $x \; P \; y$ then $x \; P \; x$ and $y \; P \; y$), and that the restriction of P to $dom(P)$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $dom(P) = A$).

3.3 Probabilistic Bisimulation

In this subsection we recall the definition of probabilistic bisimulation, specialising it to the case of unlabelled systems.

An *unlabelled probabilistic transition system* (following [LS91]) is a set of states $\mathbb{S}$ and a set of transitions of the form $\mathbb{s} \to \mu$ where $\mathbb{s} \in \mathbb{S}$ and $\mu \in \mathcal{D}(\mathbb{S})$. The probabilistic semantics corresponds to a transition system where probabilistic transition states are identified with configurations. A small-step transition from a configuration to a distribution on configurations $\langle\!|\vec{C}, s|\!\rangle \to \mu$ where $\mu \in \mathcal{D}(\mathit{Config})$ can be viewed as a transition in the unlabelled probabilistic transition system such that $\mu(\langle\!|\vec{D}, s'|\!\rangle) = p$ iff $\langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle$.

Define small-step semantics transitions from a configuration c to a *set* of configurations S by:

$$c \to_p S \iff p = \sum \{\!|q | c \to_q d, d \in S|\!\}.$$

Inspired by the definition of Larsen and Skou, let us define probabilistic bisimulation on program configurations.

Definition 1 ([LS91]) *An equivalence relation $R \in \mathit{Rel}(\mathit{Config})$ is a* probabilistic bisimulation *if whenever $c \; R \; d$ then*

$$\forall S \in \mathit{Config}/R. \;\; c \to_p S \Longrightarrow d \to_p S.$$

Two configurations are probabilistically bisimilar if there is a *probabilistic bisimulation* which relates them. In the remainder of this article we will use a weakening of the definition of probabilistic bisimulation by relaxing the reflexivity condition.

Definition 2 *A per $R \in \mathit{Rel}(\mathit{Config})$ is a* partial probabilistic bisimulation *if whenever $c \; R \; d$ then*

$$(i) \;\; \forall S \in \mathit{Config}/R. \;\; c \to_p S \Longrightarrow d \to_p S,$$
$$(ii) \;\; c \to_0 \mathit{Config} \setminus dom(R).$$

The partiality introduced here can be thought of as simply removing "unreachable" parts of the relation. The modification will be useful when we specify security, but otherwise does not significantly change the definition. Note that any probabilistic bisimulation is also a partial probabilistic bisimulation. In the other direction we have:

Proposition 1 *R is a partial probabilistic bisimulation implies R^* is a probabilistic bisimulation.*

From this it follows that any two configurations which are contained in some partial probabilistic bisimulation R are also probabilistically bisimilar, and vice-versa.

Since we are interested in studying programs rather than configurations, we recover an equivalence relation on programs (vectors of commands) as follows.

Definition 3 *A per $R \in Rel(\vec{Com})$ is a* partial probabilistic bisimulation on thread pools *iff $R \times Id_{\mathbf{St}}$ is a partial probabilistic bisimulation on configurations.*

We write $c \sim d$ (resp. $C \sim D$) iff there exists a partial probabilistic bisimulation on configurations (commands) relating c and d (C and D).

It is possible to show that both $\sim$ relations are themselves (partial) probabilistic bisimulations (c.f. [BK99]). We sketch a proof of this fact in Appendix A.

The partial probabilistic bisimulation on thread pools could form the basis of a compositional semantics – although one must at least restrict it to relations between equal-length vectors to obtain compositionality. We will not pursue this investigation further in this paper – since we are not particularly interested in the uniform scheduler.

3.4 Noninterference Based on Partial Probabilistic Bisimulation

Our aim at this point is to construct a modification of probabilistic bisimulation that reflects the "equivalence" on program behaviour visible for the attacker. The intuition behind the construction is that a program is secure iff for any two states which differ only in the values of high variables, two configurations containing the program and each of the states, execute in such a way that their behaviour is indistinguishable (or "low-equivalent") from the attacker's observation of the low parts of the state and the probability with which they occur. Let us formally define such a "low-equivalence" as a partial probabilistic bisimulation. The development parallels Section 3.3, with the modification that the states of equivalent configurations can be different in the high components.

Definition 4 *A per $R \in Rel(\vec{Com})$ is a* partial probabilistic low bisimulation *iff $R \times (=_L)$ is a partial probabilistic bisimulation on configurations.*

Write $C \sim_L D$ (C and D are *low-bisimilar*) iff there exists a partial probabilistic low bisimulation relating programs C and D.

There is an alternative and equivalent way to define this partial probabilistic low bisimulation directly in terms of the operational semantics:

Proposition 2 *A per R is a partial probabilistic bisimulation on commands iff whenever $\vec{C}\ R\ \vec{D}$ then*

$$
\begin{aligned}
&\forall s_1 =_L s_2 . \langle\!| \vec{C}, s_1 |\!\rangle \to \langle\!| \vec{C'}, s'_1 |\!\rangle \Longrightarrow \\
&\quad \exists \vec{D'}, s'_2 .\ \langle\!| \vec{D}, s_2 |\!\rangle \to \langle\!| \vec{D'}, s'_2 |\!\rangle \\
&\qquad (i) \sum \{\!| p \,|\, \langle\!| \vec{C}, s_1 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{C'}]_R, s =_L s'_1 |\!\} = \\
&\qquad\quad \sum \{\!| p \,|\, \langle\!| \vec{D}, s_2 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{D'}]_R, s =_L s'_2 |\!\} \\
&\qquad (ii)\ \ \vec{C'}\ R\ \vec{D'}, s'_1 =_L s'_2,
\end{aligned}
$$

where $[\vec{E}]_R$ stands for the $R-$equivalence class which contains $\vec{E}$.

With this formulation the (informal) connection to Gray's P-restrictiveness condition becomes clearer.

Analogously to Section 3.3 we obtain the result that $\sim_L$ is itself a low partial probabilistic bisimulation and it is the gfp of the corresponding functor. We are now ready for the culmination of this section, the security specification for a thread pool:

$$\vec{C} \text{ is } \textit{secure} \iff \vec{C} \sim_L \vec{C}.$$

The observations about proof techniques for $\sim$ from Appendix A hold for the case of $\sim_L$ as well.

4 Scheduler-independent Noninterference

The security specification above captures the probabilistic and internal timing covert channels. We have so far presumed the uniform scheduler is used for choosing the next thread. However, the scheduler is typically not specified by the language definition, and may vary from implementation to implementation. In different implementations, different probabilistic policies of a scheduler can be used, and these specific policies can be exploited by the attacker. Under the worst case assumption the security condition should be scheduler-independent – in other words we should assume that the attacker chooses the scheduler. In this section, we generalise the security condition to be robust with respect to any particular scheduler used – for a reasonable class of schedulers. The schedulers in question will be defined as a function from the history of the computation (and the low variables) to a probability distribution on the set of live threads.

Let us start with an example that illustrates that the uniform scheduler assumption does not imply security for other schedulers. Suppose that h is a boolean. Consider the following program: $\mathsf{fork}(l := h\langle l := \neg h\rangle)$. While this is a secure program for the uniform scheduler that picks threads with equal probability, it is insecure for any other scheduler. For example, a round-robin scheduler might execute threads in a deterministic order. Although the attacker may not know this in advance, he may easily be able to determine it with reasonable certainty (using a program which does not touch high data), and thus deduce the value of h. This motivates the introduction of a *scheduler-independent low-bisimulation* ($\approx_L$).

4.1 Semantics with Schedulers

In order to obtain a security condition which is robust with respect to schedulers we extend the semantics of the language to explicitly schedule threads.

But what is a scheduler? Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property – i.e. its behaviour is independent of high data. Why should a scheduler have access to the state at all? By allowing the scheduler to depend on the low variable, it becomes possible to model threads with *dynamic priorities* – i.e. where the program can control the scheduler's behaviour using the values of certain distinguished low program variables.

The scheduler needs more than the low part of the state however. A minimum requirement of any reasonable scheduler is that it also knows the number of threads from which it can choose. This information will be part of the *history* of the computation so far. A scheduler will be defined to be a function with arguments (i) the *history* of the computation so far, and (ii) the low part of a state, and yielding as its result a probability distribution on the number of threads currently in the configuration.

Let us first inductively define the set of possible schedule histories $Hist$. A *history* H is a sequence of pairs $H \in (\mathbb{N} \times \mathbb{N})^*$, representing information about the computation steps so far. In each pair of the sequence, the first component is the index of the thread last chosen for computation, and the second component is the total number of threads that remained in the configuration after that thread's computation step. Not all sequences describe valid histories. In any two adjacent pairs in a history, $\ldots(i,m)(j,n)\ldots$, the tread j that is selected must be in the range $0\ldots m-1$, and the number of threads remaining after j's step (n) cannot be smaller than $m-1$, since at most one thread can die in any computation step (although many may be created). Assuming without loss of generality that a computation always starts with a single thread and the empty history, we have the following inductive rules that define $Hist$:

$$\epsilon \in Hist \qquad (0,n) \in Hist \qquad \frac{H(i,m) \in Hist}{H(i,m)(j,n) \in Hist}(j \leq m-1 \leq n)$$

Define the number of live processes for a given history by function

$$live(\epsilon) = 1, \qquad live(H(i,n)) = n.$$

A scheduler σ is any function that given a history H and the low part l of a state returns a probability distribution on live processes. The dependent type of $\sigma(H,l)$ is

$$\sigma(H,l) \in \mathcal{D}(\{0\ldots live(H)-1\}).$$

Given a scheduler σ not all histories are feasible under σ. Define the set of σ-*feasible* histories $Hist_\sigma \subseteq Hist$ by:

$$\epsilon \in Hist_\sigma \qquad \frac{H \in Hist_\sigma \quad H(i,m) \in Hist \quad \exists l.\, \sigma(H,l)i > 0}{H(i,m) \in Hist_\sigma}$$

At each step of a computation with low-state l, only those threads can be picked for which $\sigma(H,l) > 0$. $Hist_\sigma$ will be used in the definition of σ-specific low-bisimulation – although it should be noted that $Hist_\sigma$ is still, in general, a superset of the histories which are *actually* feasible.

To introduce the scheduler to the semantics, we need to update the computation history with each successive computation step. We extend configurations to contain a history along with a command vector and a state. An initial configuration has the form $\langle\!|\epsilon, C, s|\!\rangle$. Now we replace the semantics of thread pools in Figure 3 by the rule

$$\frac{\langle\!| C_i, s |\!\rangle \twoheadrightarrow \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| H, \langle C_0 \ldots C_i \ldots C_{n-1}\rangle, s |\!\rangle \rightarrow_p \langle\!| H', \langle C_0 \ldots \vec{C} \ldots C_{n-1}\rangle, s' |\!\rangle}$$

where $H' = H(i, n + |\vec{C}| - 1), p = \sigma(H, l)\, i$ and $s = (h, l)$ for some h. The history is updated with the number of the thread being picked and the new thread-pool size. In a context where we are discussing several different schedulers, we will explicitly label the transition arrow thus $\rightarrow_p^\sigma$.

This model of the semantics with schedulers is general enough to describe any scheduler that uses the full history of thread creation/deletion to generate probabilities for picking the threads. Note however that it cannot model a scheduler which uses real-time. Consider some examples of common schedulers. Any scheduler must pick the first (and the only) thread given the empty history, so we can safely omit this case in the following examples.

For any history H, the deterministic *round robin* scheduler is defined by

$$roundr(H(i,n), l)\, x = \begin{cases} 1, & \text{if } x = (i+1) \ mod \ n, \\ 0, & \text{otherwise.} \end{cases}$$

A round-robin scheduler which chooses the same thread for several steps before switching would need to look further back in the history.

The *uniform scheduler* presented in Section 2 is simply

$$uni(H, l)x = 1/live(H),$$

for any $x \in \{0 \ldots live(H) - 1\}$. As in the example above, the previous history and the value of the low variable are disregarded. Note that the history contains sufficient information to calculate the individual lifetimes of threads, or the creation hierarchy of the threads in order that these may be used to influence – either deterministically or stochastically – the policy of the scheduler.

4.2 Probabilistic Noninterference with Schedulers

Given a scheduler σ, let us define the scheduler-specific partial probabilistic low-bisimulation. This bisimulation is a kind of lock-step execution as we have already seen for $\sim_L$. For two bisimilar configurations, if one makes a transition to an equivalence class then the other configuration should also be able to make a transition with the same probability to the same class. However, the history in the second configuration can vary from the one in the first configuration. Let us capture the degree of how the history can vary by defining a relation $=_\sigma$ on σ-feasible histories. It relates σ-feasible histories which are indistinguishable for σ. Define $=_\sigma \in Rel(Hist_\sigma)$ to be the largest relation satisfying:

$$\frac{H =_\sigma H'}{\forall l.\, \sigma(H, l) = \sigma(H', l)} \qquad \frac{H =_\sigma H'}{H(i, m) =_\sigma H'(i, m)}$$

For example, $=_{uni}$ relates any two histories such that the number of live threads is the same.

Using the method of Section 3 define the scheduler-specific partial probabilistic bisimulation on configurations who now include histories. We have now everything we need to define a scheduler-specific partial probabilistic low-bisimulation on thread pools.

Definition 5 *Given a scheduler σ, a per $R \in Rel(\vec{Com})$ is a* scheduler-specific partial probabilistic low-bisimulation *iff $(=_\sigma) \times R \times (=_L)$ is a scheduler-specific partial probabilistic bisimulation on configurations.*

Write $\vec{C} \sim_L^\sigma \vec{D}$ ($\vec{C}$ and $\vec{D}$ are *low-bisimilar under* σ) iff there exists a partial probabilistic low-bisimulation relating programs $\vec{C}$ and $\vec{D}$.

Note that for the uniform scheduler *uni*, we have $\sim_L^{uni} = \sim_L$, where $\sim_L$ is the low-bisimulation from Section 3. Given a scheduler σ and a program $\vec{C}$,

$$\vec{C} \text{ is } \sigma\textit{-secure} \iff \vec{C} \sim_L^\sigma \vec{C}.$$

4.3 Scheduler-independent Noninterference

As we have argued, a realistic security condition should be scheduler-independent. We are now ready to define the *scheduler-independent* low-bisimulation $\approx_L$ and the scheduler-independent security (SI-security). Define

$$\vec{C} \approx_L \vec{D} \iff \forall \sigma. \vec{C} \sim_L^\sigma \vec{D}, \text{ and}$$

$$\vec{C} \text{ is } \textit{SI-secure} \iff \vec{C} \approx_L \vec{C}.$$

Trivially, $\approx_L \subseteq \sim_L$. The example in the beginning of this section shows that the reverse inclusion does not hold. As a corollary, we have $\vec{C}$ is SI-secure implies $\vec{C}$ is secure.

On a side note, the quantification over schedulers is somewhat reminiscent of the "strategies" in the adversary model used by Syverson and Gray [SG95], which are also functions from computation histories to distributions. Note however that we are not using them to model attacker behaviour (e.g. I/O) – but rather an internal component of the system.

We define yet another low-bisimulation along with a security condition as an aid to prove various analyses secure. This condition is potentially the strongest of all considered in the paper; and proofs of the soundness of particular analyses go through easiest just with this one. Call the new low-bisimulation *strong*.

In order to achieve a compositional bisimulation we want to restrict the strong low-bisimulation so that any two strongly low-bisimilar thread pools must be of equal size and must create/kill exactly the same number of processes at each step under any scheduler. This means that we do not need the machinery for explicit scheduling. In fact, the new low-bisimulation is purely nondeterministic.

Definition 6 *Define the* strong *low-bisimulation* $\cong_L$ *to be the union of all symmetric relations* R *on thread pools of equal size, such that whenever* $\langle C_0 \ldots C_{n-1}\rangle\ R\ \langle D_0 \ldots D_{n-1}\rangle$ *then*

$$\begin{aligned}\forall s_1 =_L s_2 \forall i. \langle\!| C_i, s_1 |\!\rangle \rightarrow \langle\!| \vec{C}', s_1' |\!\rangle \Longrightarrow \\ \exists \vec{D}', s_2'. \langle\!| D_i, s_2 |\!\rangle \rightarrow \langle\!| \vec{D}', s_2' |\!\rangle, \\ \vec{C}'\ R\ \vec{D}', s_1' =_L s_2'.\end{aligned}$$

The strong security specification goes:

$$\vec{C} \text{ is } \textit{strongly secure} \Longleftrightarrow \vec{C} \cong_L \vec{C}.$$

We are going to use this specification of security in the proofs of analyses correctness. Let us first show that the strong bisimulation is indeed stronger than the scheduler-independent one.

Proposition 3 $\vec{C}$ *is strongly secure* $\Longrightarrow \vec{C}$ *is SI-secure.*

We suspect that the reverse inclusion may also hold in case the SI-security condition is required to be compositional. Notice for example that the quantification over all schedulers in the SI-security condition forces the restriction on the number of threads of two bisimilar programs, i.e. if $\vec{C} \approx_L \vec{D}$ then $|\vec{C}| = |\vec{D}|$. Indeed, otherwise there is a scheduler that can use the difference in the number of threads to modify the low variable which can be used to leak information. For example, it holds that $\mathsf{skip} \mid \mathsf{skip} \approx_L \mathsf{skip}; \mathsf{skip}$ – for any scheduler the two programs execute deterministically in lock-step. However, the parallel composition of the each of the two with the thread $l := 5$ will not preserve the bisimulation $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\approx_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$ since the probability of l to become 5 depends on the number of threads and many schedulers will reflect it in the semantics. This applies, for example, to the uniform scheduler: $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\sim_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$.

5 Hook-up Properties

In this section, we investigate the hook-up properties of strong security. Strong bisimulation is constructed to be suitable for compositional reasoning and yet to be general enough to be able to prove security of known analyses. The type-based program analyses that we have considered all satisfy the strongest notion of security. For the rest of the paper "secure" will mean strongly secure.

In the standard security terminology, the *hook-up* property [McC87] says that if two programs are secure then their composition (parallel, sequential or some other kind) is secure is well. We will show what kinds of composition are allowed. These properties are important since they are the key to the utility of the specification for the purpose of proving the correctness of syntax-directed program analyses.

We will need some definitions to study various kinds of composition. A (boolean or arithmetic) expression Exp is *low* iff $\forall s_1 =_L s_2. \exists n. \langle\!| Exp, s_1 |\!\rangle \downarrow n \ \&\ \langle\!| Exp, s_2 |\!\rangle \downarrow n$. Otherwise, the expression is *high*.

The low-bisimulation is not a congruence. (Take, for example, any ground context that is insecure.) The low-bisimulation is preserved by *secure contexts* – i.e. contexts built with secure components. Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is secure iff it has one of the following forms:

$$\begin{aligned}\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \\ & \mid [\bullet_1]; [\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low }) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low }) \mid \mathsf{fork}([\bullet_1][\vec{\bullet_2}]) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle\end{aligned}$$

Theorem 1 (Secure congruence) *If $\vec{C_1} \approxeq_L \vec{C_1'}$, $\vec{C_2} \approxeq_L \vec{C_2'}$ and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then it is the case that $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$ provided $\vec{C_1} \approxeq_L \vec{C_2}$.*

Proof. A detailed proof is given in Appendix B. □

An immediate corollary is the hook-up property result.

Corollary 1 (Hook-up) *If $\vec{C_1}, \vec{C_2}$ are secure and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure provided $\vec{C_1} \approxeq_L \vec{C_2}$.*

Proof. Observe that $\vec{C_1}, \vec{C_2}$ are secure implies $\vec{C_i} \approxeq_L \vec{C_i}$ for $i = 1, 2$. Thus by the security congruence theorem $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1}, \vec{C_2}]$ i.e. $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure.□

6 Proving Analyses Secure

The compositionality of the security condition can be fruitfully exploited when proving the correctness of various compositional program analyses. In this section we develop one such analysis.

A popular form for presenting program analysis is as a nonstandard type system – as exemplified in the present context by Volpano, Smith and Irvine's type system [VSI96] expressing Denning's flow analysis [Den76]. The type system was extended to a statically-threaded language with a uniform scheduler in [VS99]. The style of such type systems is to give security types to expressions (*high* or *low*) and commands (*high cmd* or *low cmd*) and make sure that these types do not mismatch when composing the commands.

The hook-up properties of Corollary 1 are sufficient to make the correctness proof of Volpano and Smith's analysis extremely simple – and the proof thereby includes correctness with respect to e.g. round-robin schedulers and schedulers using dynamic priorities (via dependence on low "priority" variables in the program).

As an example we will consider a useful improvement of Volpano and Smith's system (for our slightly richer language) based on ideas from a recent analysis developed by Johan Agat [Aga00] which shows how timing channels can be eliminated from sequential programs by a combination of typing and code transformation.

The improvement in question concerns the treatment of the security of conditional expressions in the case when the boolean depends on high variables. Consider a command of the form $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $B : \mathit{high}$. Volpano and Smith's analysis requires that (i) no low assignments are made within C_1 or C_2, (ii) no while loops occur within the branches, and (iii) the whole conditional is executed atomically (i.e., without interruption by another thread.) The last of these points is enforced in the Volpano-Smith system by an additional language construct called protect. A disadvantage of using protect is that it is not a typical synchronisation primitive of modern concurrent languages.[2]

[2]The protect statement also appears prohibitively expensive to implement using e.g. locks, since in order to block all threads during a protect, *every* atomic command C in the program must be transformed into $\mathsf{getlock}; C; \mathsf{releaselock}$.

We will instead use a technique due to Agat to avoid the need for protect statements, and obtain an analysis that is more permissive than Volpano and Smith's analysis.

6.1 Agat's Approach

As Agat observed in the sequential case, each of the restrictions on a high conditional can be lifted to some degree by observing that it is sufficient for C_1 and C_2 to be low bisimilar. The hook-up result shows that this is true in the present setting also.

Agat's approach is to *transform* the branches of the conditional so that they become bisimilar in a manner that is easily checked by a syntax directed set of rules. The essence of the transformation is to pad the branches of the conditionals. In our case this padding will add skip instructions and possibly dummy forks. Agat's aim was the elimination of *external* timing attacks; we can also argue this here, although we simply focus on the internal timing leaks.

In the rest of the section we present the combined analysis and transformation system, prove its correctness with respect to strong security and show how this approach eliminates the need for protect's.

6.2 The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct and indirect insecure flows then it might be accepted by the system and transformed into a program free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, C' is the result of its transformation and $\vec{Sl}$ is the "type" of $\vec{C'}$. The types of programs are their *low slices*. A low slice is essentially a copy of a secure program in which assignments to high variables have been replaced by skip's. The slice Sl does not have any occurrences of h, but has the same structure as C' and models the timing behaviour of C', as observable by other threads.

The typing and transformation rules are presented in Figure 4. The variables h and l have the types *high* and *low* respectively. Integer literals n may be considered as either *high* or *low*. An arbitrary expression Exp may be considered as *high*. In all but If_{high} rules, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. Command skip is its own low slice and therefore its own

[Var] $h : high \qquad l : low$

[Exp] $n : \tau \qquad Exp : high$

[Arithm$_{low}$] $$\frac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$$

[Skip] $\mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$

[Assign$_{low}$] $$\frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$$

[Assign$_{high}$] $h := Exp \hookrightarrow h := Exp : \mathsf{skip}$

[Seq] $$\frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$$

[While] $$\frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

[Par] $$\frac{C_1 \hookrightarrow C_1 : Sl_1 \ldots C_n \hookrightarrow C_n : Sl_n}{\langle C_1 \ldots C_n \rangle \hookrightarrow \langle C_1' \ldots C_n' \rangle : \langle Sl_1 \ldots Sl_n \rangle}$$

[Fork] $$\frac{C_1 \hookrightarrow C_1' : Sl_1 \quad \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\mathsf{fork}(C_1 \vec{C_2}) \hookrightarrow \mathsf{fork}(C_1' \vec{C_2'}) : \mathsf{fork}(Sl_1 \vec{Sl_2})}$$

[If$_{low}$] $$\frac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$$

[If$_{high}$] $$\frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

Figure 4: Transformation to eliminate timing leaks

type. The rule Assign_{low} prevents direct insecure information flows – the assignment $l := h$ is not typable. The rule Assign_{high} types an assignment to the high variable with the low slice skip. The rules Arithm_{low}, Seq, If_{low}, Par and Fork propagate types compositionally. The guard of the while-loop in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard.

The rule If_{high} prevents indirect insecure flows and timing flows. Let $al(C)$ be a predicate returning True whenever there is a syntactic occurrence of an assignment to low variable l in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) = \mathsf{False}$ prevents the indirect leaks.

The interesting rule is the case when the guard of an if command is high. In this case both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure it is also necessary that the two branches be low bisimilar. This is achieved by cross-copying the low slice of one branch into the other. The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

A very simple example of the application of the transformation system is presented in Appendix C.

6.3 Correctness of the Analysis

Let us prove the security of Agat's analysis extended to the multi-threaded language. The following theorem is a straightforward use of the secure congruence properties of the security condition proved in Theorem 1.

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \approxeq_L \vec{Sl}$

Proof. The proof is given in Appendix D. □

Corollary 2 (Security of the Analysis) $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ *is secure.*

6.4 Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? We argue below that the transformed program is a possibilistic refinement of the original program, and that under certain additional assumptions, that it is equivalent.

Clearly, the padding introduced by the transformation can change the timing and thereby the program's probabilistic behaviour, but otherwise it is just additional "stuttering". A more serious problem is that it might introduce extra nontermination, in the case that a nonterminating loop is cross-copied into a branch that would otherwise have terminated.

To make the first point precise, let us define a *possibilistic (bi)simulation* on programs. To obtain a possibilistic transition relation we can simply erase the probability from the uniform scheduler semantics defined in Section 2.

Definition 7 *Define the possibilistic simulation $\preceq$ (resp., bisimulation $\simeq$) to be the the union of all (resp. symmetric) relations R on thread pools such that whenever $\vec{C}\ R\ \vec{D}$ then for all s, S', C', there exists a D' such that*

$$\langle\!| \vec{C}, s |\!\rangle \to \langle\!| \vec{C'}, s' |\!\rangle \Longrightarrow \langle\!| \vec{D}, s |\!\rangle \to^* \langle\!| \vec{D'}, s' |\!\rangle \text{ and } \vec{C'}\ R\ \vec{D'}.$$

Intuitively, $\vec{C} \preceq \vec{D}$ holds whenever $\vec{C}$ performs a (maybe infinitely) slowed down version of the computation of $\vec{D}$.

Lemma 1 *$\preceq$ is reflexive, transitive and preserved by all contexts (not necessarily secure) i.e., $\preceq$ is a precongruence.*

Lemma 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \preceq \vec{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Assign_{low}, Assign_{high} follow from the reflexivity of $\preceq$. The cases Seq, While, Par, Fork and If_{low} are implied by the induction hypotheses and the precongruence of $\preceq$.

The remaining case is If_{high}. By induction we have $C_1' \preceq C_1$ and $C_2' \preceq C_2$. The side condition guarantees that there are no assignments in the slices Sl_1 and Sl_2 of the branches C_1 and C_2 of the if. Thus Sl_1 and Sl_2 contain only dummy computation without changing the state. The insertion of such a computation might only slow down the computation. Thus, $C_1'; Sl_2 \preceq C_1$ and $Sl_1; C_2' \preceq C_2$. The precongruence of $\preceq$ yields $C' \preceq C$. □

6.5 Elimination of the protect

Let us define the notion of a protected program from [VS99]. A program $\vec{C}$ is *protected* iff any conditional on the high variable is wrapped in a protect term. A protect cannot have other protect's or while's within its scope of protection. We are now ready to sketch the protect elimination result.

Theorem 3 (Protect elimination) *If $\vec{C_p}$ is well-typed and protected in Volpano-Smith's type system then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$ where $\vec{C}$ is the result of erasing the protect's from $\vec{C}$, for some secure $\vec{C'}$ and some $\vec{Sl}$ such that $\vec{C} \simeq \vec{C'}$.*

Proof. (Sketch) Although there are only two types of commands in Volpano-Smith's system (*low cmd* and *high cmd*), the type inference is still performed compositionally and similarly to the transformation in Figure 4. Hence we are able to type all programs that are typable in Volpano-Smith's system. Whenever $\vec{C_p}$ is well-typed then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$. By Lemma 2 we immediately have $\vec{C'} \preceq \vec{C}$.

The program $\vec{C}$ has no occurrences of fork (no dynamic thread creation in Volpano-Smith's language). Therefore, the padding transformation cannot introduce new fork's. While-loops cannot be introduced since the only place where the transformation generates new code is the if-on-high case, and since such terms are protected they contain no occurrences of while. What the transformation might introduce in the branches of a high if is a sequential composition of low if's and skip's (these are the only possibilities for the low slices of the branches). The insertion of these commands does not affect the possibilistic semantics or the termination properties of the initial program (we omit the routine proofs of these laws). Using this intuition, the rest of the proof of $\vec{C'} \simeq \vec{C}$ follows analogously to Lemma 2. □

7 Conclusions

We have developed extensional semantics-based specifications of secure information flow for multi-threaded programs. The specifications capture probabilistic covert channels that arise from the scheduling of concurrent threads. We have argued that under the worst case assumption the security condition should be scheduler-independent.

By defining security specifications as noninterference based on a probabilistic bisimulation, we have achieved compositional reasoning (build secure programs from secure components) and precision over approaches proposed in the literature, without the mathematical overheads of a denotational approach using e.g., probabilistic powerdomains. Proving type-system based analyses is simple since such analyses are usually compositional as well.

In this paper, we have been considering unsynchronised multi-threaded concurrency with dynamic thread creation. Future work must include the study of synchronisation primitives.

Another simplification is in our treatment of I/O. Implicitly, we permit the attacker (the external supplier of the program) to observer intermediate low outputs, and even provide the initial low input, but not to selectively provide inputs to the program as it runs. Blocking input gives the attacker(s) the ability to externally control the timing behaviour of threads. To study the essence of these problems, it might be appropriate to extend Focardi and Gorrieri's process-calculus-based study [FG95]) to the probabilistic case. This should also enable a more rigorous connection to be made to Gray's P-Restrictiveness.

References

[Aga00] J. Agat. Transforming out timing leaks. In *POPL'00: The 27:th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53. ACM, ACM Press, January 2000.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[BBL95] J.-P. Banatre, C. Bryce, and D. Le Metayer. An approach to information security in distributed systems. In *Proceedings of the 5th IEEE International Workshop on Future Trends in Distributed Computing Systems*, pages 384–394, 1995.

[BK99] C. Baier and M. Z. Kwiatkowska. Domain equations for probabilistic processes. *Mathematical Structures in Computer Science*, June 1999. To appear.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994/1995.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, April 1982.

[Gra90] J.W. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[Koc96] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[LS91] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[McL90] J. McLean. The specification and modeling of computer security. *Computer*, 23(1), January 1990.

[SG95] P. Syverson and J.W. Gray III. The epistemic representation of information flow security in probabilistic systems. In *Proc. 8th IEEE Computer Security Foundations Workshop*, June 1995.

[SS99] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming, ESOP'99*, LNCS 1576, pages 40–58, Amsterdam, March 1999. Springer-Verlag.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

A Observations on Partial Probabilistic Bisimulation

Here are some observations we made on the way when investigating the properties of partial probabilistic bisimulations. The union of all partial probabilistic bisimulation can itself be shown to be a partial probabilistic bisimulation. This follows from a standard Tarski fixed-point argument (although to our knowledge this argument has not been used elsewhere for probabilistic bisimulation). The key is to note that pers on a given set form a complete lattice (with meet given by set-intersection, and join given by the transitive closure of the set-union), and that the definition can be phrased in terms of a monotonic functional.

Fix an arbitrary unlabelled probabilistic transition system over some set of states $\mathbb{S}$. Let F denote the function from pers to pers (over $\mathbb{S}$) given by: $c\ F(R)\ d$ then

$$\begin{array}{l}(i)\ \forall S \in \mathbb{S}/R.\, c \to_p S \Longrightarrow d \to_p S,\\ (ii)\ c \to_0 \mathbb{S} \setminus dom(R).\end{array}$$

It is straightforward to check that $F(R)$ is indeed transitive and symmetric.

Proposition 4 *Suppose $P, Q \in Rel(\mathbb{S})$ are pers. Then $P \subseteq Q \Longrightarrow F(P) \subseteq F(Q)$.*

Proof. Given $c\ F(P)\ d$, let us show $c\ F(Q)\ d$. Suppose $c \to_q S_Q$ for some $S_Q \in \mathbb{S}/Q$. Since $P \subseteq Q$, we can partition S_Q into $\cup_i S_P^i \cup S$, where the S_P^i are equivalence classes of P, and S is a subset of $\mathbb{S} \setminus dom(P)$. Suppose $c \to_{p_i} S_P^i$. Note $c \to_0 S$. By $c\ F(P)\ d$ we have $d \to_{p_i} S_P^i$ and $d \to_0 S$. This implies $c \to_q S_Q$ by simple summing. Conclude $c\ F(Q)\ d$.□

Let us define $\sim = \cup\{R \mid R \text{ is a per}, R \subseteq F(R)\}$.

Proposition 5 *$\sim$ is the greatest fixed point of F in the lattice of pers.*

The proof is the standard argument, by using the monotonicity of F and appeal to Tarski's fixed-point theorem. When proving a per R to be a partial probabilistic bisimulation (or $R \subseteq \sim$) one can use the the following analogue of the standard "up-to" technique.

Proposition 6 *Given a per $R \in Rel(\mathbb{S})$,*

$$R \subseteq F((R \cup \sim)^+) \Longrightarrow R \subseteq \sim.$$

B Proof of the Secure Congruence Theorem

Theorem 1 (Secure congruence) *If $\vec{C_1} \approxeq_L \vec{C_1'}$, $\vec{C_2} \approxeq_L \vec{C_2'}$ and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] =$ if B then $[\bullet_1]$ else $[\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$ provided $\vec{C_1} \approxeq_L \vec{C_2}$.*

Proof. Cases on $\mathbb{C}$. Consider first the single-threaded case $\mathbb{C}[\vec{C_1}, \vec{C_2}] = C$ and $\mathbb{C}[\vec{C_1}, \vec{C_2}] = C'$. Let us start off with the cases skip, $x := e$, if B then $[\bullet_1]$ else $[\bullet_2]$ $(B : low)$, if B then $[\bullet_1]$ else $[\bullet_2]$ $(B : high)$ and fork$([\bullet_1][\vec{\bullet_2}])$.

In the proof below, let us make use of the so-called "up-to" technique for proving two programs $\vec{I}$ and $\vec{J}$ bisimilar. In such a technique, one constructs a relation $R \in Rel(\vec{Com})$ such that $\vec{I}\ R\ \vec{J}$ and $R \subseteq F_{\approxeq_L}(R \cup \approxeq_L)$ where $F_{\approxeq_L}(\cdot)$ is the corresponding function on relations such that R is a strong low-bisimulation whenever R is a subset of $F(R)$ (c.f. Appendix A).

The standard nondeterministic bisimulation theory guarantees us then that $R \subseteq \approxeq_L$ and thereby $\vec{I} \approxeq_L \vec{J}$. It is indeed enough to have the singleton relation $R = \{(C, C')\}$ for a proof of the first cases in question. Let us inspect these cases of $\mathbb{C}$:

skip

Start with s_1 and s_2 $(s_1 =_L s_2)$. With either state, the computation of skip terminates in one step with the same states. So, the final states are low-equal as well.

$l := Exp$ **(**Exp **is low)**

Since Exp is a low expression, the computation of the command with different initial states s_1 and s_2 $(s_1 =_L s_2)$ terminates in one step with the same states that are still low-equal.

$h := Exp$

Exp can be arbitrary. Computing the command will not change the low component of the states.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : low)$

Start with $\langle$if B then C_1 else $C_2, s_1\rangle$ and $\langle$if B then C_1' else $C_2', s_2\rangle$ $(s_1 =_L s_2)$. B has the same value in either state. Thus, one step of computation will lead to the configuration with bisimilar commands (C_1 and C_1', or C_2 and C_2') and low-equal states. Further, we know $C_1 \approxeq_L C_1'$ and $C_2 \approxeq_L C_2'$.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : high)$

Start with s_1 and s_2 $(s_1 =_L s_2)$. Here the value of B might differ for s_1 and s_2, but since $C_i \approxeq_L C'_j$ for $i, j = 1, 2$ one step of computation ends up in bisimilar commands and and low-equal states in this case as well.

fork$([\bullet_1][\vec{\bullet_2}])$

Trivial, as computation does not depend on h.

We have three cases left:

$[\bullet_1]; [\bullet_2]$

C_1 (and C'_1) might spawn new processes, so we need to have all possible bisimilar postfix vectors after C_2 (C'_2) in R. $R = \{((E; C_2)\vec{G}, (E'; C'_2)\vec{G'}) \mid E \approxeq_L E', \vec{G} \approxeq_L \vec{G'}\}$. Then $R \subseteq F_{\approxeq_L}(R \cup \approxeq_L)$.

while B **do** $[\bullet_1]$ **(B is low)**

Choose R to be the relation $\{((E; \mathsf{while}\ B\ \mathsf{do}\ C_1)\vec{G}, (E'; \mathsf{while}\ B\ \mathsf{do}\ C'_1)\vec{G'}) \mid E \approxeq_L E', \vec{G} \approxeq_L \vec{G'}\}$. Start with s_1 and s_2 $(s_1 =_L s_2)$. B has the same value in either state. Thus, one step of computation will lead both configurations either to the termination of the loop (with low-equal states) or to configurations with commands $C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1$ and $C'_1; \mathsf{while}\ B\ \mathsf{do}\ C'_1$ and low-equal states. C_1 and C'_1 might also spawn new processes.

$\langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle$

Finally, the case of multiple threads. Take $R = \{(\vec{E_1}\vec{E_2}, \vec{G_1}\vec{G_2}) \mid \vec{E_i} \approxeq_L \vec{G_i}, i = 1, 2\}$. Suppose $\vec{E_i} = \langle E_i^0 \dots E_i^{n_i-1} \rangle$ and $\vec{G_i}' = \langle G_i^0 \dots G_i^{m_i-1} \rangle$ for $i = 1, 2$. Let us prove $R \subseteq F_{\approxeq_L}(\approxeq_L)$ by showing:

$$
\begin{aligned}
&(i) |\vec{E_1}\vec{E_2}| = |\vec{G_1}\vec{G_2}| \\
&(ii)\, \forall s_1 =_L s_2 \forall i, j. \langle\!| E_i^j, s_1 |\!\rangle \rightharpoonup \langle\!| \vec{E}, s'_1 |\!\rangle \Longrightarrow \\
&\qquad\qquad \exists \vec{G}, s'_2. \langle\!| G_i^j, s_2 |\!\rangle \rightharpoonup \langle\!| \vec{G}, s'_2 |\!\rangle, \\
&\qquad\qquad\qquad \vec{E} \approxeq_L \vec{G}, s'_1 =_L s'_2.
\end{aligned}
$$

The (i)-item is straightforward, as $|\vec{E_i}| = |\vec{G_i}|$ for $i = 1, 2$. The (ii)-item follows from the two separate (ii)-items (when i is fixed to be either 1 or 2) which come from the unwinding of the definition of the strong bisimulation for each of the $\vec{E_i} \approxeq_L \vec{G_i}$.$\square$

C An Example Code Transformation

The following example is based on one in [Aga00], and is used to illustrate the effect of the transformation described in Section 6. RSA encryption is based on computing $a^b \ mod \ n$, where a is an integer representing the plaintext and b is the integer encryption key. To efficiently compute $a^b \ mod \ n$, the modular exponentiation algorithm can be used, but as shown by Kocher [Koc96], a careless implementation will leak b through timing. On the left in Figure 5, we give an example of such an implementation. (Assume we have an if without an else with the obvious semantics.) All variables except for i and k in the program are high. The result of computation is stored in d. We represent b as a $k+1$ element long array of secret booleans, with the

Original program	Transformed program
$c := 0;\ d := 1;\ i := k;$	$c := 0;\ d := 1;\ i := k;$
while $i \geq 0$	while $i \geq 0$
do $i := i - 1;$	do $i := i - 1;$
$c := 2 \cdot c;$	$c := 2 \cdot c;$
$d := (d \cdot d) \ mod \ n;$	$d := (d \cdot d) \ mod \ n;$
if $b_i = 1$	if $b_i = 1$
then $c := c + 1;$	then $c := c + 1;$
$d := (d \cdot a) \ mod \ n$	$d := (d \cdot a) \ mod \ n$
	else skip;
	skip

Figure 5: Producing a secure modular exponentiation thread.

most significant bit of b at $b[k]$. The initial program is not secure since the two branches of the if have different timing behaviours (they are not low bisimilar). For some schedulers, a (secure) thread running in parallel with this code may compute different low outputs depending on the values of the b_i (we omit a concrete example for reasons of space). The transformation algorithm presented in Figure 4 (with the minor extension necessary to handle arrays with a low index) will close these timing leaks by transforming the program into the one given on the right in Figure 5. The security of this padded program relies on atomic execution of the assignment operators.

D Proof of Theorem 2

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \approxeq_L \vec{Sl}$

Proof. Proceed by structural induction on the thread pool $\vec{C'}$. Let us consider the cases by which $\vec{C'} : \vec{Sl}$. In the cases $\mathsf{skip} : \mathsf{skip}$, $l := Exp : l := Exp$ $(Exp : low)$ and $h := Exp : \mathsf{skip}$ the program $\vec{C'}$ is low bisimilar to it's low slice. The cases $C_1; C_2$, $\mathsf{while}\ B\ \mathsf{do}\ C_1 : \mathsf{while}\ B\ \mathsf{do}\ Sl'$ $(B : low)$,$\langle C_1 \dots C_n \rangle : \langle Sl_1 \dots Sl_n \rangle$, $\mathsf{fork}(C_1 \vec{C_2})$ and $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2$ $(B : low)$ provide secure contexts $[\bullet_1]; [\bullet_2]$, $\mathsf{while}\ B\ \mathsf{do}\ [\bullet]$, $\langle [\bullet_1] \dots [\bullet_n] \rangle$, $\mathsf{fork}([\bullet_1][\vec{\bullet_2}])$ and $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ respectively. By the the induction hypothesis the sub-commands have bisimilar low slices, so the conclusion follows by the hook-up theorem.

The last case is $\mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2$. In order to apply the special if-on-high case of the hook-up theorem we need to show $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. By the induction hypothesis, $C_i' \approxeq_L Sl_i$ $(i = 1, 2)$. Now since $[\bullet_1]; [\bullet_2]$ is a secure context, we have that $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. Finally, this is sufficient to argue that $\{(\mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2', \mathsf{skip}; Sl_1; Sl_2)\}$ is a strong bisimulation up to $\approxeq_L$ (which requires just a case analysis on the outcome of B). $\square$

Part II

Aspects of Program Analysis

Forwards and backwards analysis for functional programs *

Andrei Sabelfeld
Chalmers University of Technology and
University of Göteborg
Göteborg, Sweden
andrei@cs.chalmers.se

Viktor Sabelfeld
Karlsruhe University
Karlsruhe, Germany
sabelfel@ira.uka.de

Abstract

This paper presents an approach to static analysis of functional programs that lies between abstract interpretation and flow analysis approaches. Two general frameworks of semantic property analysis for recursion schemes are described. We present a marking technique for solving a rather powerful subset of these problems. Illustrative examples that are captured by this technique are *strictness analysis* and *needed/unneeded analysis*. However, some static property analyses of recursive programs described by either flow analysis or abstract interpretation approaches cannot be formulated as either forwards or backwards analysis problems. As an instance, we consider the *formal parameter dependence analysis*.

*An extended abstract of this paper appeared in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, June 1997. ACM Press. The paper is submitted to TCS.

1 Introduction

To check applicability conditions of equivalent program transformations, to detect semantic errors in programs, and to specialise programs efficiently one should learn how to determine semantic properties of programs. Many authors have explored program analysis problems on program models. Most results have been obtained in flow analysis [KU77] and abstract interpretation [CC77, CC92] approaches.

We introduce a new method of semantic analysis for first order call-by-name functional programs (*recursion schemes* [GL71]). We describe two simple classes of semantic property analysis problems, *forwards* and *backwards property analysis problems* for recursion schemes, and use a marking technique for their solution. Our approach is close to Patrick and Radhia Cousot's abstract interpretation approach [CC77, CC92]. We consider the fix-point semantics of recursive programs. Namely, we associate a term sequence called *approximation sequence* to a recursion scheme. Given the properties of terms from the approximation sequence we define the program property as the limit of the constructed sequence of properties. Forwards analysis is applied if we are interested in properties that flow forwards from subexpressions to the big expression that contains them. On the contrary, backwards analysis is useful when properties flow from a large expression to its subexpressions.

Forwards and backwards property analysis problems prove to be undecidable in the general case. However, we give sufficient conditions for solving these problems and present marking algorithms that compute program properties under the conditions required. In Sections 4.2 and 5.2 we give examples of forwards (*strictness analysis*) and backwards (*needed/unneeded analysis*) property analysis problems.

Some static property analyses of recursive programs that are well described either in the flow analysis or abstract interpretation approaches cannot be formulated as either forwards or backwards analysis problems, unless computation traces are propagated. As an instance, in Section 6 we describe the *formal parameter dependence analysis* of recursion schemes as a flow analysis problem.

Section 7 discusses related work and concludes.

2 Recursion scheme definition

Let $\mathcal{X} = \{x, y, z, \dots\}$ be a set of *variables*, $\mathcal{F}_b = \{\omega, f, g, h, \dots\}$ be a set of *basic symbols*, $\mathcal{F}_d = \{F, F_1, F_2, \dots\}$ be a set of *defined symbols*, $\mathcal{F} = \mathcal{F}_b \bigcup \mathcal{F}_d$, and r denote the *rank* function. Each symbol $\phi \in \mathcal{F}$ of rank $r(\phi) = n$ is said to have arity n. The symbol ω stands for "undefined", the least informative term; $r(\omega) = 0$. For a set X of variables, $X \subset \mathcal{X}$, let $\mathcal{T}(X)$ be the set of all *terms* under variables from X formed using basic and defined symbols from $\mathcal{F}$.

A *recursion scheme* is a pair $S = \langle e; DEF \rangle$, where e is a term called *scheme entry* and DEF is a finite set of symbol definitions in $\mathcal{F}_d$. A *definition* of a symbol $F \in \mathcal{F}_d$ has the form $F(x_1, \dots, x_n) \Leftarrow t$, where $n = r(F), x_1, \dots, x_n \in \mathcal{X}$ are different *formal parameters* of F and $t \in \mathcal{T}(x_1, \dots, x_n)$ is the *body* of the symbol F. We assume that each formal parameter is distinct from all the variables appearing in the entry.

A term $F(t_1, \dots, t_n)$, where $n = r(F)$ and $F \in \mathcal{F}_d$ is said to be a *call* to the symbol F. Its subterms $t_1, \dots, t_n$ are *actual parameters* of this call. We require each symbol called in a scheme to be defined in it. Here is an example of a scheme:

$$\left\langle F_1(h, h)\ ;\ \begin{array}{ll} F_1(x, y) & \Leftarrow \mathbf{if}(px, F_2(fx, fy), x) \\ F_2(x, y) & \Leftarrow \mathbf{if}(py, F_1(fy, fx), y) \\ F_3(x) & \Leftarrow \mathbf{if}(px, F_3(fx), x) \end{array} \right\rangle$$

Recall the standard notions of *occurrence* and *substitution* from term rewriting theory [Klo91]. Any *occurrence* of a subterm in a term can be uniquely identified by the path from the root of the tree representation of the term to the subterm. Subterms of the entry and definition bodies in a scheme are identified as follows. The *address* of the the scheme entry (or a tree root if the tree is unique in the context under consideration) is the empty word Λ. The *address* of the body of a symbol F is the word $[F]$. If an occurrence of a term $f(t_1, \dots, t_n)$ (or $F(t_1, \dots, t_n)$) has the address a, then its subterm occurrences $t_1, \dots, t_n$ have the addresses $a.1, \dots, a.n$ respectively. For example, the first occurrence of the term fx in the scheme above has the address $[F_1].2.1$, and the first occurrence of the constant h has the address $\Lambda.1$, or just 1. Two addresses are *independent* iff neither is a prefix of the other.

Suppose a is an address and S is a scheme or a term. Let $S \downarrow a$ stand for the term at a in S, and $S[a \leftarrow t]$ denote the scheme (or term) obtained from S by replacing the term at a by t. If N is a set of mutually independent addresses of subterm occurrences of a term t, then $t[N \leftarrow \tau]$ denotes the term obtained from t by replacing all subterm occurrences at addresses from N by the term τ.

A *substitution* is a mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{X})$ that satisfies the condition $\sigma x \neq x$ for only a finite number of variables x from $\mathcal{X}$. The substitution σ that maps a variable x_i into a term t_i for $i = 1, \ldots, n$ is denoted by $[t_1/x_1, \ldots, t_n/x_n]$. The notion of substitution can be extended in a natural way to arbitrary terms: $\sigma\phi(t_1, \ldots, t_n) = \phi(\sigma t_1, \ldots, \sigma t_n)$ for $\phi \in \mathcal{F}$ and $n = r(\phi)$.

Every scheme S defines a mapping $\pi : \mathcal{T}(\mathcal{X}) \to \mathcal{T}(\mathcal{X})$ which corresponds to the "parallel outermost computation rule". It performs one step unfolding of all outermost calls in a term.

$$\pi t = \begin{cases} x, & \text{if } t = x, \text{ where } x \in \mathcal{X}, \\ f(\pi t_1, \ldots, \pi t_n), & \text{if } t = f(t_1, \ldots, t_n), \text{ where } n = r(f) \text{ and } f \in \mathcal{F}_b, \\ \sigma\tau, & \text{if } t = F(t_1, \ldots, t_n), \text{where } F \in \mathcal{F}_d, \\ & \sigma = [t_1/x_1, \ldots, t_n/x_n], \\ & \text{and } F(x_1, \ldots, x_n) \Leftarrow \tau \text{ is the definition of } F \text{ in } S. \end{cases}$$

3 Interpretation and equivalence

In this Section we define the algebraic semantics of recursive programs. A more detailed presentation can be found, for instance, in [Gue81]. A *domain* is a triple $\langle D, \leq, \perp \rangle$, where D is a set, $\perp \in D$ (the "bottom") and $\leq$ is a partial order on D that satisfies the following two conditions:

- $\forall\, d \in D \quad \perp \leq d$.
- *Completeness condition:* for each chain C in D the *least upper bound* $lub(C)$ belongs to D. ($d \leq lub(C)$ for all d in C, and $lub(C) \leq u$ for all u in D such that $d \leq u$ for all d in C.)

The *universal term domain* $\mathcal{U} = \langle T_U, \sqsubseteq, \omega \rangle$ is constructed in the following way. Consider the set T of all terms under the signature $\langle \mathcal{X}, \mathcal{F}_b \rangle$ with the partial order $\sqsubseteq$ introduced by setting $t_1 \sqsubseteq t_2$ iff there exists a set N of mutually independent addresses in t_2, such that $t_1 = t_2[N \leftarrow \omega]$. Intuitively it means that $t_1 \sqsubseteq t_2$ iff t_1 can be obtained from t_2 by replacing 0 or more subterms by ω. Let T_U be the completion of the set T by least upper bounds of all infinite increasing chains of elements from T. The elements from T_U can be treated as (infinite) terms (trees). One can prove that $\mathcal{U}$ is a domain with the partial order $\sqsubseteq$: $t_1 \sqsubseteq t_2$ iff there is a (possibly infinite) set N of mutually independent addresses in t_2 such that $t_1 = t_2[N \leftarrow \omega]$, and bottom ω.

Suppose D and D' are domains. A function $\varphi : D \to D'$ is *monotone*, iff $\forall\, d, d' \in D\ d \leq_D d' \Rightarrow \varphi d \leq_{D'} \varphi d'$. A monotone function is *continuous* if it preserves the least upper bounds of non-empty linearly ordered subsets of D, i.e. $\varphi(lub(C)) = lub(\varphi(C))$.

An *interpretation* I fixes a domain D and assigns

- a member $I(x) \in D$ to each variable $x \in \mathcal{X}$,
- a continuous function $I(f) : D^{r(f)} \to D$ to each symbol $f \in \mathcal{F}_b$, the constant ω always gets $I(\omega) = \perp_D$.

Using the structural induction we can extend the notion of an interpretation I to arbitrary finite terms by considering all defined symbol applications in the terms to be "unknown":

$$I(t) = \begin{cases} I(x), & \text{if } t = x, \text{ where } x \in \mathcal{X}; \\ I(f)(I(t_1), \ldots, I(t_n)), & \text{if } t = f(t_1, \ldots, t_n), \text{ where } f \in \mathcal{F}_b; \\ \perp & \text{otherwise.} \end{cases}$$

An important instance of an interpretation is the *universal* (or *free*, or *initial*, or *Herbrand*) *interpretation* J on the domain $\mathcal{U}$. It assigns symbolic values to terms. Let $J(x) = x$ for variables $x \in \mathcal{X}$, and for $f \in \mathcal{F}_b, n = r(f)$, and $t_1, \ldots, t_n \in T_U$ set $J(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$.

Define the *approximation sequence* of t by $App_n(S, t) = J(\pi^n t)$, where $n \geq 0, \pi^0 \tau = \tau$, and $\pi^n = \pi^{n-1}\pi$ for $n > 0$. Let us now formally define the full unwinding of a term t with respect to a program S. The *determinant* $det(t)$ *of a term* t is the least upper bound of the approximation sequence $\{App_0(S,t), App_1(S,t), \ldots\}$, and the *determinant* $det(S)$ is the determinant of the entry of the scheme S. For example, the first elements of the approximation sequence of the entry of $\langle F(u); F(x) \Leftarrow f(x, F(gx))\rangle$ are

$$\omega \sqsubseteq f(u,\omega) \sqsubseteq f(u, f(gu,\omega)) \sqsubseteq f(u, f(gu, f(ggu,\omega))) \sqsubseteq \ldots, \text{ etc.}$$

The least upper bound of such a sequence is an element of T_U. This element is the algebraic semantics of the example program.

Two schemes S_1 and S_2 with the entries t_1 and t_2 are *equivalent* ($S_1 \sim S_2$ for short) iff $I(t_1) = I(t_2)$ for all interpretations I.

4 Forwards semantic analysis framework

In this section we will formulate the forwards semantic analysis framework, exemplify it (by considering strictness analysis as a forwards analysis problem), show its undecidability in the general case, present a solution under certain conditions, and, finally, apply the solution to the example.

4.1 Flowing forwards

Let $\mathcal{L} = \langle L, \sqcup, \perp \rangle$ be an upper semi-lattice, where L is a set with a partial order $\sqsubseteq$ on it, $\perp$ is the *bottom* element, and $\sqcup$ is a binary *union* operation that satisfies $\forall x, y, z \in L \; x \sqcup y = y \sqcup x, (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z), x \sqcup x = x, x \sqcup \perp = x$. Define $x \sqsubset y$ iff $x \sqsubseteq y$ and $x \neq y$. A function $f : L \to L$ is *distributive* iff $f(x \sqcup y) = f(x) \sqcup f(y)$. We will consider only semi-lattices satisfying the ascending chain condition. (No infinitely ascending chains are allowed.)

We have an instance of a *forwards semantic analysis problem* for a recursion scheme S by giving:

- A property semi-lattice $\mathcal{L}$ that satisfies the chain condition.
- A *deductive semantic function* $Sem^{\uparrow}$ that assigns a monotone property transformer $Sem^{\uparrow}(f) : L^n \to L, n = r(f)$, to each symbol $f \in \mathcal{F}_b$. Such a property transformer defines the property of a term $f(t_1, \dots, t_n)$ using the properties $s_1, \dots, s_n$ of its subterms $t_1, \dots, t_n \in L$ as $Sem^{\uparrow}(f)$ $(s_1, \dots, s_n)$.

For an address a in a scheme S define "variables in the scope" $Var(a)$ as follows. If a is an address of a subterm of the body of some F, let $Var(a)$ be F's formal parameter set. If a is an address of a subterm of the scheme entry, then define $Var(a)$ to be the set of all variables occurring in the entry. Let $\overline{s}$ denote the tuple $s_1, \dots, s_n$ for some n clear from the context. By Pr_i we denote the *projection function* $Pr_i \in (L^n \to L), Pr_i\overline{s} = s_i$. The deductive semantic function can be extended to arbitrary terms without calls. For such a term t set

$$Sem^*(t)\overline{s} = \begin{cases} s_i, & \text{if } t = x_i, \\ Sem^{\uparrow}(f)(Sem^*(t_1)\overline{s}, \dots, Sem^*(t_n)\overline{s}), & \text{if } \begin{array}{l} f \in \mathcal{F}_b \; \& \\ t = f(t_1, \dots, t_n), \end{array} \end{cases}$$

where $|\overline{s}|$ is the number of free variables of t. For an address a in a scheme S define $d(a, n) = Sem^*(App_n(S, S \downarrow a))$ as an element of the semi-lattice $L^{|Var(a)|} \to L$.

The forwards semantic analysis problem for the scheme S consists of finding the property

$$D(a) = \bigsqcup_{n=0}^{\infty} d(a, n)$$

for all addresses a of the scheme S.

4.2 Example: strictness analysis

This analysis is similar to Mycroft's strictness analysis [Myc80, Myc81]. Suppose that a subset $Strict(f)$ of the set $2^{\{1,\dots,r(f)\}}$ is associated to each basic symbol f, call $Strict(f)$ the set of *strict parameter collections* of the symbol f. An interpretation I is *strict*, if for any $d_1, \dots, d_n \in D_I$ the condition

$$(\forall \Delta \in Strict(f)\ \exists i \in \Delta\ d_i = \bot) \Rightarrow I(f)(d_1, \dots, d_n) = \bot$$

holds for each basic symbol $f \in \mathcal{F}_b$. For example, the natural way to bound the interpretations of the ternary symbol **if** is to settle $Strict(\mathbf{if}) = \{\{1,2\},\{1,3\}\}$. This means that we restrict all possible interpretations of the symbol **if** to functions *cond* with the condition

$$\forall\ d, d' \in D\ (cond(\bot, d, d') = \bot)\ \&\ (cond(d, \bot, \bot) = \bot)$$

imposed. Another example is the "vote" function symbol Γ_2^3 with the set $Strict(\Gamma_2^3) = \{\{1,2\},\{1,3\},\{2,3\}\}$ of strict parameter collection, so that

$$\forall\ d \in D\ I(\Gamma_2^3)(\bot,\bot,d) = \bot\ \&\ I(\Gamma_2^3)(\bot,d,\bot) = \bot\ \&\ I(\Gamma_2^3)(d,\bot,\bot) = \bot$$

for all strict interpretations I. An address a of a scheme S is called *hopeless*, iff $I(S \downarrow a) = \bot$ for all strict interpretations I. For example, if $Strict(f) = Strict(h) = \{\{1\}\}$ and $Strict(g) = \{\{2\}\}$, then in the scheme

$$\langle\ F(u,\omega); F(x,y) \Leftarrow g(hy, F(fx, hy))\ \rangle$$

the address Λ is hopeless.

Now we formulate a forwards analysis problem in order to find all hopeless addresses of a scheme. According to the four dimensional strictness analysis classification [DW90] this algorithm is first-order, flat, high fidelity, and forwards. Take the property lattice $L_{str} = \{\bot, \top\}$, $\bot \sqsubset \top$. $\top$ stands for "might be defined", $\bot$ means "certainly undefined". For such a simple lattice $x \sqcap y$ is always either x or y, and similarly $x \sqcup y$ is either x or y. Let the deductive semantic function $Sem^{\uparrow}_{hope}$ assign the monotone transformer

$$Sem^{\uparrow}_{hope}(f)(s_1, \dots, s_n) = \bigsqcup_{\Delta \in Strict(f)} \prod_{i \in \Delta} s_i$$

to each symbol $f \in \mathcal{F}_b$. Then an address a of a scheme S is hopeless if $D_{hope}(a)(\top, \dots, \top) = \bot$. The general result of the strictness analysis for a term t at some address a is described by the function $Sem^*(t)$ that has type $L_{str}^{|Var(a)|} \to L_{str}$. Let $P \cup Q$ be a partition of $\{1, \ldots, |Var(a)|\}$. If $Sem^*(t)$ applied to a combination $\overline{s}$, where $s_i = \perp$ for $i \in P$ and $s_i = \top$ for $i \in Q$, results in $\perp$, it means that in any strict interpretation if the arguments to t indexed by P are undefined then t applied to those arguments along with arbitrary arguments indexed by Q will be undefined as well.

4.3 Undecidability of the general case

Theorem 1. *The forwards semantic analysis problem described is undecidable in the general case.*

Proof. We modify the construction used in [KU77] to reduce Post's correspondence problem [Pos46] for a system (X, Y):

$$X = (\alpha_1, \ldots, \alpha_n), Y = (\beta_1, \ldots, \beta_n),$$

where $i = 1, \ldots, n$; α_i and β_i are words in the alphabet $\{0, 1\}$, to the forwards semantic analysis problem for

$$\left\langle \begin{array}{l} f_1(F(g_1 d), \ldots, F(g_n d)); \\ F(x) \Leftarrow f_2(hx, F(g_1 x), \ldots, F(g_n x)) \end{array} \right\rangle.$$

Let us consider the property semi-lattice

$$L = \{\perp, \top\} \cup \{0,1\}^* \times \{0,1\}^*, x \sqsubseteq y \text{ iff } x = y \text{ or } x = \perp \text{ or } y = \top.$$

This union $x \sqcup y$ is defined to be y, if $x \sqsubseteq y$, or $\top$ otherwise. The lengths of all increasing chains are, apparently, bounded by 3. The semantic function is defined as follows: for $i = 1, \ldots, n$ set $Sem^\uparrow(g_i)(\perp) = \perp, Sem^\uparrow(g_i)(\top) = \top, Sem^\uparrow(g_i)((\alpha, \beta)) = (\alpha\alpha_i, \beta\beta_i)$,

$$\begin{gathered} Sem^\uparrow(f_1)(s_1, \ldots, s_n) = s_1 \sqcup \ldots \sqcup s_n, \\ Sem^\uparrow(f_2)(s_0, s_1, \ldots, s_n) = s_0 \sqcup s_1 \sqcup \ldots \sqcup s_n, \\ Sem^\uparrow(h)(\perp) = \perp, Sem^\uparrow(h)(\top) = \top, \\ Sem^\uparrow(h)((\alpha, \beta)) = \begin{cases} \top, & \text{if } \alpha = \beta, \\ \perp, & \text{if } \alpha \neq \beta, \end{cases} \end{gathered}$$

$Sem^\uparrow(d) = (\Lambda, \Lambda)$, where Λ is the empty word in the alphabet $\{0, 1\}$. In order to prove the monotony of the functions $Sem^\uparrow(h)$ and $Sem^\uparrow(g_i)$ ($i = 1, \ldots, n$) we use the fact that $x \neq \perp$, and $x \sqsubset y$ for $x, y \in L$ implies $y = \top$. Thus if $x \neq \perp$,

$$x \sqsubset y \Rightarrow Sem^\uparrow(h)(y) = Sem^\uparrow(h)(\top) = \top \sqsupseteq Sem^\uparrow(h)(x), \text{ and}$$

$$x \sqsubset y \Rightarrow Sem^{\uparrow}(g_i)(y) = Sem^{\uparrow}(g_i)(\top) = \top \sqsupseteq Sem^{\uparrow}(g_i)(x).$$

Otherwise,

$$\perp \sqsubset y \Rightarrow Sem^{\uparrow}(h)(\perp) = Sem^{\uparrow}(g_i)(\perp) = \perp \sqsubseteq z \text{ for any } z \in L.$$

The monotony of the functions $Sem^{\uparrow}(d), Sem^{\uparrow}(f_1)$ and $Sem^{\uparrow}(f_2)$ is obvious, they are even distributive.

To complete the proof observe that for the formulated forwards analysis problem the condition $D(\Lambda) = \top$ holds iff the system (X, Y) has at least one solution. If we had an algorithm that solves the forwards analysis problem we would be able to solve Post's correspondence problem. □

4.4 Solution

Theorem 2. *If the semi-lattice L is finite and all property transformers are distributive in each argument, then the forwards analysis problem can be solved using the following marking algorithm.*

A *d-marking* of a scheme S is a mapping μ that assigns a mapping (*d-mark*) $\mu a \in (L^n \to L)$, $n = |Var(a)|$, to each address a in S. The initial d-marking μ_0 assigns the d-mark Pr_i to all addresses of the occurrences of the variable x_i and the d-mark $\perp\!\!\!\perp$ ($\perp\!\!\!\perp(x_1, \ldots, x_n) = \perp$) to all the other addresses in S. The initial d-marking and all d-markings which can be obtained from it by serial application of *d-marking rules* are called *reachable*. An application of a d-marking rule to an address a consists of the following:

Let $S \downarrow a = \phi(t_1, \ldots, t_n)$, where $\phi \in \mathcal{F}, n = r(\phi)$. We replace the d-mark μa of the address a by the d-mark $\mu a \sqcup Q$, where the mapping Q is defined by

$$Q = \begin{cases} Sem^{\uparrow}(\phi) \circ (\mu a.1, \ldots, \mu a.n), & \text{if } \phi \in \mathcal{F}_b, \\ \mu[\phi] \circ (\mu a.1, \ldots, \mu a.n), & \text{if } \phi \in \mathcal{F}_d. \end{cases}$$

Here and later we omit the parenthesis in marks $\mu a.i$ to mean $\mu(a.i)$. A *stationary d-marking* is the reachable d-marking that cannot be changed by any application of the d-marking rule. This stationary marking gives the precise solution of the forwards analysis problem. The premises of the theorem, the chain condition, and the monotony of all property transformers guarantee the existence and uniqueness of the stationary d-marking. A formal proof is sketched in Appendix A.2.

4.5 Application to the example

Note that *cond* is not distributive in all arguments, but is distributive in each argument. Moreover, all monotone functions on the two element domain L_{str} are distributive in each argument. This makes Theorem 2 applicable to the strictness analysis. According to the described above main algorithm for the solution of the forwards analysis problem obtain the following procedure.

The initial marking: $\mu_0 a = Pr_i$ for addresses a of the occurrences of a variable x_i, and $\mu_0 a = \perp\!\!\!\perp$ for all the other addresses a in S.

The marking rule: if $S \downarrow a = \phi(t_1, \dots, t_n)$, where $\phi \in \mathcal{F}, n = r(\phi)$, then the mark μa of an address a is replaced by the mark $\mu' a$:

$$\mu' a = \mu a \sqcup \begin{cases} \bigsqcup_{\Delta \in Strict(\phi)} \bigsqcap_{i \in \Delta} \mu a.i, & \text{if } \phi \in \mathcal{F}_b, \\ \mu[\phi] \circ (\mu a.1, \dots, \mu a.n), & \text{if } \phi \in \mathcal{F}_d. \end{cases}$$

5 Backwards semantic analysis framework

5.1 Flowing backwards

The *image set* $Image(a)$ of an address a of a scheme S is defined as the minimal subset of addresses of subterms occurring in $det(S)$ that satisfies the following conditions:

- $\Lambda \in Image(\Lambda)$.
- If $b \in Image(a), S \downarrow a = f(t_1, \dots, t_n)$ and $f \in \mathcal{F}_b$, then $b.i \in Image(a.i)$ for all $i = 1, \dots, n$.
- If $b \in Image(a)$ and $S \downarrow a = F(t_1, \dots, t_n)$ for a symbol F defined in S, then $b \in Image([F])$.
- If $Image(a) \neq \emptyset, S \downarrow a = F(t_1, \dots, t_n)$ for a symbol F defined in S, c is an address of an occurrence of the i-th formal parameter in the body of F, and $b \in Image(c)$, then $b \in Image(a.i)$.

Such a subset exists by the Knaster-Tarski theorem, since the corresponding defining operator is monotone.

Informally, the image of a program term's address is the set of the addresses of those terms in the determinant that “stem from” the use of the term. The image set can be empty. For instance, for

$$\langle F(u, v); F(x, y) \Leftarrow f(x, F(gx, gy)), F'(x) \Leftarrow f(x) \rangle$$

we get $Image([F']) = \emptyset$ as none of the terms $\pi^n F(u,v)$ invokes F'. Besides, $Image([F].2.2) = \emptyset$ since F's second argument is "unneeded". We will investigate how to find such addresses later on, in Section 5.2.

An instance of a *backwards semantic analysis problem* for a recursion scheme S is given by:

- A property semi-lattice $\mathcal{L}$ that satisfies the chain condition.
- An initial property $h_0 \in L$.
- A *semantic heritage function* $Sem^{\downarrow}$ that assigns a monotone property transformer $Sem^{\downarrow}(f,i) : L \to L$ to each symbol $f \in \mathcal{F}_b$ and $i \in \{1,\ldots,r(f)\}$. Such a transformer defines the property of the i-th subterm of a term $f(\ldots)$ with a given property $s \in L$ as $Sem^{\downarrow}(f,i)(s)$.

The property $h^{\downarrow}(a)$ of an address a in $det(S)$ is defined by

$$h^{\downarrow}(a) = \begin{cases} h_0, & \text{if } a = \Lambda, \\ Sem^{\downarrow}(f,i)(h^{\downarrow}(b)), & \text{if } \begin{array}{l} f \in \mathcal{F}_b \ \& \\ \exists i \ a = b.i \ \& \ det(S) \downarrow b = f(t_1,\ldots,t_n). \end{array} \end{cases}$$

The backwards semantic analysis problem for a scheme S consists of finding the property

$$H^{\downarrow}(a) = \bigsqcup_{b \in Image(a)} h^{\downarrow}(b)$$

for all addresses a in S.

5.2 Example: needed/unneeded analysis

We will formulate the needed/unneeded analysis, which finds expressions that are not needed to be evaluated, as a backwards analysis problem. As Hughes mentions in [Hug87], programmers do not usually write functions with unneeded parameters, whereas partial evaluators might well introduce them in residual programs. Let us call an address a in a scheme S *unneeded* iff $Image(a) = \emptyset$. For example, in the scheme

$$\langle F(u,v); F(x,y) \Leftarrow \mathbf{if}(px, x, F(fx, fy))\rangle$$

the addresses 2, $[F]$.3.2 and $[F]$3.2.1 are unneeded. For an unneeded address a, $S \sim S[a \leftarrow t]$ holds for arbitrary terms $t \in \mathcal{T}(Var(a))$. For example,

$$\left\langle \begin{array}{l} F(u,v); \\ F(x,y) \Leftarrow \mathbf{if}(px, x, F(fx, fy)) \end{array} \right\rangle \sim \left\langle \begin{array}{l} F(u,v); \\ F(x,y) \Leftarrow \mathbf{if}(px, x, F(fx, b)) \end{array} \right\rangle.$$

We can formulate a backwards analysis problem in order to find all unneeded addresses of a scheme S. Take the property semi-lattice $L = \{\mathbf{O}, \mathbf{1}\}; \mathbf{O}$ stands for "unneeded", and $\mathbf{1}$ stands for "needed". Let

$$\mathbf{O} \sqsubseteq \mathbf{O}, \mathbf{1} \sqsubseteq \mathbf{1}, \mathbf{O} \sqsubseteq \mathbf{1}, p \sqcup p' = \mathbf{O} \Leftrightarrow p = \mathbf{O} \;\&\; p' = \mathbf{O}.$$

The initial property is $\mathbf{1}$. The semantic heritage function $Sem^{\downarrow}$ assigns the distributive property transformer $Sem^{\downarrow}(f, i) = id_L$ to each basic symbol $f \in \mathcal{F}_b$ and $i \in \{1, \ldots, r(f)\}$. Then

$$H^{\downarrow}_{need}(a) = \bigsqcup_{b \in Image(a)} h^{\downarrow}(b) = \mathbf{O} \text{ iff } Image(a) = \emptyset.$$

5.3 Undecidability of the general case

Theorem 3. *The backwards semantic analysis problem described is undecidable in the general case.*

Proof. In a similar, to the forwards problem, way we can prove that the backwards analysis problem is undecidable in the general case. In fact, the leaves of the determinant-tree for the program

$$\left\langle \begin{array}{l} f_1(F(g_1 d), \ldots, F(g_n d)); \\ F(x) \Leftarrow f_2(hx, F(g_1 x), \ldots, F(g_n x)) \end{array} \right\rangle$$

used in the proof of Theorem 4.3 have the form $hg_{i_1} \ldots g_{i_k} d$, where $i_1 \ldots i_k$ are all finite sequences of numbers from 1 to n. The job of the semantic transformer for h is to check if a given sequence gives a solution. Other determinant nodes just propagate all properties to the root.

We can go the other way letting the information flow backwards and accumulate in all the images of d's. Now h is needed to be the argument of g_i's:

$$\left\langle \begin{array}{l} f_1(F(g_1 hd), \ldots, F(g_n hd)); \\ F(x) \Leftarrow f_2(x, F(g_1 x), \ldots, F(g_n x)) \end{array} \right\rangle.$$

Take the same as in Theorem 4.3 semi-lattice L and the initial property $\perp$. Having the semantic transformers "reversed" we reduce Post's correspondence problem to a backwards analysis problem, and thus prove the latter to be undecidable. □

5.4 Solution

Theorem 4. *If the semi-lattice L is finite and all property transformers are distributive, the backwards analysis problem can be solved using the following marking algorithm.*

An *h-marking* of a scheme S is a mapping μ that assigns a property (called *h-mark*) $\mu a \in L$ to each address a in S. The initial h-marking μ_0 assigns the h-mark h_0 to the entry address and the h-mark $\perp$ to all other addresses. The initial h-marking and all h-markings which can be obtained from it by serial application of *h-marking rules* are called *reachable*. An application of an h-marking rule to an address a consists of the following:

1. If $S \downarrow a = f(t_1, \dots, t_n)$, where $f \in \mathcal{F}_b$, then for all $i \in \{1, \dots, n\}$ the h-mark $\mu a.i$ of the address $a.i$ is replaced by the h-mark $\mu a.i \sqcup Sem^{\downarrow}(f, i)(\mu a)$.
2. If $S \downarrow a = F(t_1, \dots, t_n)$, where $F \in \mathcal{F}_d$, then the h-mark $\mu[F]$ of the address $[F]$ is replaced by the h-mark $\mu[F] \sqcup \mu a$.
3. If a is an address of some occurrence of the i-th formal parameter in some F's body, then for each address b of a call to F, that satisfies $\mu b \neq \perp$, the h-mark $\mu b.i$ of the address $b.i$ is replaced by the h-mark $\mu b.i \sqcup \mu a$.

A *stationary h-marking* is a reachable h-marking that cannot be changed by any application of the h-marking rule. This stationary marking gives the precise solution of the backwards analysis problem. By the premises of the theorem, the chain condition, and the monotony of all property transformers the stationary h-marking exists and it is unique. See Appendix A.1 for a formal proof.

For any backwards problem with distributive transformers we have a solution algorithm for free. A natural marking technique comes out of it when we apply it to the example of the needed/unneeded analysis.

6 Formal parameter dependence

If the value of an actual parameter in each call to a symbol F equals the value of a term t, then any occurrence of the corresponding formal parameter in the body of F can be replaced by the term t. Let us describe and solve the formal parameter dependence analysis problem as a flow analysis problem. This problem can be formulated neither as forwards nor as backwards analysis problem in the presented framework since any term in the approximation sequence does not contain formal parameters and, therefore, no formal parameter properties can be taken into account.

In order to solve the problem, we formulate a data flow analysis problem for the directed graph $Graph(S)$ obtained from a scheme S in the following way. The nodes of this graph will be the entry address and the addresses of bodies of the defined symbols of the scheme S. The edges will be the call addresses. We draw an edge a from an address b to an address $[F]$, if a is an address of a call to F that occurs in the term at address b.

Let X be a finite set of variables, $X \subset \mathcal{X}$. We use below the semi-lattice $\mathcal{L}(X)$ of context free grammars G that describes the finite languages $L(G)$ of term equalities of the form $x = t$, where $x \in X$ and $t \in \mathcal{T}(X)$. These grammars have the terminal set

$$\Sigma = \mathcal{F}_b \cup X \cup \{\equiv, (,), ,\}$$

and rules of the following three forms

1. $S \to x \equiv A$ with nonterminals S, A; S is the initial nonterminal of the grammar; $x \in X$;
2. $A \to x$ with a nonterminal A; $x \in X$;
3. $A \to f(A_1, \ldots, A_n)$ with nonterminals $A, A_1, \ldots, A_n$ and a terminal $f \in \mathcal{F}_b$.

In order to ensure the finiteness of a grammar's language let us require the graph of the dependencies of the grammar's nonterminals to be acyclic.

Example:

$$\begin{array}{l} S \to x \equiv A_1 | y \equiv A_1 | z \equiv A_2 \\ A_1 \to x | y | f(A_2, A_3) \\ A_2 \to z | g(A_3) \\ A_3 \to a \end{array} \quad \text{represents} \quad \left\{ \begin{array}{l} x \equiv x, y \equiv y, \\ x \equiv y, x \equiv f(z,a), y \equiv f(z,a), \\ x \equiv f(g(a),a), y \equiv f(g(a),a) \\ z \equiv z, z \equiv g(a) \end{array} \right\}.$$

Such a grammar G can be reduced in linear time to a *reduced* grammar $Red(G) = \langle N, \Sigma, S, P \rangle$ that satisfies the following conditions

- $L(G) = L(Red(G))$ is a finite language,
- $\forall A, B \in N \; L(A) \cap L(B) = \emptyset$,
- $\forall A \; \in N \; L(A) \neq \emptyset$,
- $\forall A \in N \; \exists \alpha, \beta \in \{N \cup \Sigma\}^* \; S \xrightarrow{*} \alpha A \beta$

The *meet* operation $\sqcap$ on reduced grammars corresponds to the language intersection and can be defined in the following way. Let $G_i = \langle N_i, \Sigma, S_i, P_i \rangle$ for $i = 1, 2$ be two reduced grammars. We define $G = \langle N, \Sigma, S, P \rangle$, where

$N = N_1 \times N_2, S = \langle S_1, S_2\rangle, P = \{S \to x \equiv \langle A_1, A_2\rangle | (S_i \to x \equiv A_i) \in P_i, i = 1,2\} \cup \{\langle A_1, A_2\rangle \to f(\langle B_1^1, B_1^2\rangle, \ldots, \langle B_n^1, B_n^2\rangle) | (A_i \to f(B_1^i, \ldots, B_n^i)) \in P_i, i = 1,2\} \cup \{\langle A_1, A_2\rangle \to x | (A_i \to x) \in P_i, i = 1,2, x \in X\}$. Finally, we define $G_1 \sqcap G_2 \stackrel{def}{=} Red(G)$.

Lemma 1. $L(G_1 \sqcap G_2) = L(G_1) \cap L(G_2)$.

We denote by $\mathcal{L}(X)$ the set of all reduced grammars augmented by a new element $\mathbf{1}$ satisfying $\mathbf{1} \sqcap G = G \sqcap \mathbf{1} = G$. The partial order $\sqsubseteq$ on $\mathcal{L}(X)$ is introduced by the definition $G_1 \sqsubseteq G_2 \stackrel{def}{=} G_1 \sqcap G_2 = G_1$. The grammar $\{S \to x \equiv A_x, A_x \to x \mid x \in X\}$ will be denoted by $\mathbf{O}$, $L(\mathbf{O}) = \{x \equiv x \mid x \in X\}, \forall\ G \in \mathcal{L}(X)\ \mathbf{O} \sqsubseteq G$. We say that a nonterminal A of a grammar *knows* a term t, if $A \stackrel{*}{\to} t$ holds. For a grammar G and a term t, the grammar $Add(G,t)$ that has a nonterminal A which knows t can be built in the following way.

If the grammar G already has a nonterminal that knows t, or $G = \mathbf{1}$, then $Add(G,t) \stackrel{def}{=} G$. Otherwise, if t is a variable x, then add a new nonterminal A and a rule $A \to x$ to the grammar G; if $t = f(t_1, \ldots, t_n)$, build the grammar $G' = Add(Add(\ldots Add(G,t_1), \ldots), t_n)$ and add a new nonterminal A and the rule $A \to f(A_1, \ldots, A_n)$ to the grammar G', where for $i = 1, \ldots, n$ A_i is the nonterminal in G' which knows the term t_i.

For an address a of a call $F(t_1, \ldots, t_n)$ in a scheme we define the grammar transformer

$$[\![call(a)]\!] : \mathcal{L}(Var(a) \cup Var(\Lambda)) \to \mathcal{L}(Var([F]) \cup Var(\Lambda)).$$

If $G = \mathbf{1}$, then $[\![call(a)]\!]G = G$. Otherwise, let $G' = Add(\ldots Add(G,t_1) \ldots t_n)$, let A_i be the nonterminal of G' that knows the term t_i, and let B_i be the nonterminal of G' that knows the i-th formal parameter x_i of the symbol $F (i = 1, \ldots, n)$. For all $i = 1, \ldots, n$, if $A_i \neq B_i$, then delete the rules $S \to x_i \equiv B_i$ and $B_i \to x_i$ from the grammar G' and add the new rules $S \to x_i \equiv A_i$ and $A_i \to x_i$. Finally, define $[\![call(a)]\!]G \stackrel{def}{=} Red(G')$.

Lemma 2. *For all addresses a of a scheme, $[\![call(a)]\!]$ is a distributive grammar transformer, i.e.*

$$[\![call(a)]\!](G_1 \sqcap G_2) = [\![call(a)]\!]G_1 \sqcap [\![call(a)]\!]G_2.$$

Now we formulate a data flow analysis problem for $Graph(S)$ by stating

- the initial marking μ_0 that associates the grammar $\mathbf{O}$ to the entry node Λ and the grammar $\mathbf{1}$ to the other nodes of $Graph(S)$,

- the semantic function which associates a distributive grammar transformer $[\![call(a)]\!]$ to any edge a of $Graph(S)$.

Let w be a path in $Graph(S)$, i.e. a sequence of adjacent edges, and let

$$\Phi_w(G) = \begin{cases} G, & \text{if } w \text{ does not contain any edge,} \\ [\![call(a)]\!]\Phi_{w'}(G), & \text{if } w = w'a, \text{ where } a \text{ is an address of a call.} \end{cases}$$

Our data flow analysis problem consists of finding the 'meet over all path solution'

$$mop([F]) = \bigsqcap_{w \in W} \Phi_w(\mathbf{O})$$

for all nodes $[F]$ in $Graph(S)$, where W is the set of all paths in $Graph(S)$ from the entry node to the node $[F]$.

It is well known [KU77], that a stationary marking μ of $Graph(S)$ can be effectively constructed such that $\mu[F] = mop([F])$ for all nodes $[F]$, and hence, if the condition $(x \equiv t) \in L(\mu[F])$ is valid for a node $[F]$ and for a formal parameter x of F, then $S \sim S[a \leftarrow t]$ holds for all addresses a of occurrences of x in the body of the symbol F. For example, for the stationary marking μ of the scheme

$$\left\langle \begin{array}{l} F(h,h); \\ F(x,y) \Leftarrow \mathbf{if}(px, F(fx, fy), gx) \end{array} \right\rangle$$

$(x \equiv y) \in L(\mu[F])$ holds, which means that actual parameter values coincide in all calls to F.

7 Discussion and Conclusions

Our forwards analysis can be classified to be an abstract interpretation according to Patrick and Radhia Cousot [CC92]. In abstract interpretation all primitive functions get abstracted such that they safely approximate the concrete primitives. Abstract functions constitute the abstract domain. The abstract functions correspond to our deductive semantic functions for forwards analysis. Once approximations are defined, it is possible to find the abstract value for a defined function by computing the fix-point of its definition in the abstract domain. Hughes [Hug87] argued that when doing the full abstract interpretation forwards analysis it takes exponential time in the worst case. However, it is possible to make forwards analysis demand-driven, that is not to compute functions by fix-point iteration entirely, but rather check them only on certain values. Jones and Mycroft have introduced *minimal function graphs* [JM86] to implement the idea. In our framework we can simply choose the analysis that suits a particular problem better. The advantage is that in spite of a more operational nature of backwards analysis, both frameworks are formulated to be dual.

As for complexity, observe that both algorithms formulated are non-deterministic. The worst case is when for a scheme of the size n the algorithm (in either analysis) changes each mark as many times as the length of the longest chain in the semi-lattice $\langle L, \sqcup, \perp \rangle$. Since such an algorithm randomly applies the marking rules to the addresses until stabilisation, its complexity gets another factor of n. Let $l = |L|$. In case of forwards analysis the marks are elements of $L^k \to L$ $(k \leq r)$, where r is the maximal rank of all basic functions, and in case of backwards analysis the marks belong to L. This gives us $\mathcal{O}(n^2 * l^{l^r})$ as the worst case estimate for the forwards and $\mathcal{O}(n^2 * l)$ for the backwards analysis problem. The subexpressions l^{l^r} and l instantiate to 2^{2^r} and 2 in the examples of strictness analysis and needed/unneeded analysis respectively. These are rough estimates. In practice one factor of n disappears once a rule application strategy is settled.

The forwards and backwards analyses can be generalised to higher-order functions. It is possible by allowing the property transformers to be higher-order as well. However, the complexity of the marking algorithms becomes extremely high.

Acknowledgements

Thanks are due to David Sands and John Hughes for the constructive suggestions on both contents and presentation of the paper. Thank you to Sebastian Hunt for providing us with his unpublished note [Hun90] which helped us track other work on algebraic semantics.

References

[CC77] P. Cousot, R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Conference Records of the 4^{th} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252. Los Angeles, California, 1977.

[CC92] P. Cousot, R. Cousot. *Abstract interpretation frameworks.* Journal of Logic and Computation, 2(4):511–547, August 1992.

[CC93] P. Cousot, R. Cousot. *Galois connection based abstract interpretations for strictness analysis.* In D. Bjorner, M. Broy, and I.V. Pottosin, editors, Proceedings of the International Conference on Formal Methods in Programming and their Applications, Academgorodok, Novosibirsk, Russia, 28 June – 2 July 1993. Lecture Notes in Computer Science 735, pages 98–127. Springer-Verlag, 1993.

[CC95] P. Cousot, R. Cousot. *Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation.* In Conference Record of FPCA '95 SIGPLAN/SIGARCH/WG2.8 Conference on Functional Programming and Computer Architecture, pages 170–181, La Jolla, California, U.S.A., 25–28 June 1995. ACM Press, New York, U.S.A, 1995.

[DW90] K. Davis, P. Wadler. *Strictness analysis in 4d.* In Glasgow Functional Programming Workshop, 1990.

[GL71] S.J. Garland, D.C. Luckham. *Program schemes, recursion schemes and formal languages.* Report UCLA-ENG-7154, June 1971, School of Engineering and Applied Science, University of California, Los Angeles, 1971.

[Gue81] I. Guessarian. *Algebraic Semantics.* Volume 99 of LNCS, 1981. Springer-Verlag.

[Hug87] J. Hughes. *Backwards Analysis of Functional Programs.* In Bjørner and Ershov, editors, IFIP Workshop on Partial Evaluation and Mivxed Computation, 1987.

[HL92] J. Hughes, J. Launchbury. *Reversing Abstract Interpretations.* In European Symposium on Programming, volume 582 of LNCS, Rennes, 1992. Springer-Verlag. Also to appear in Science of Computer Programming.

[Hun90] S. Hunt. *Projection analysis and stable functions.* An unpublished note.

[JM79] N.D. Jones, S.S. Muthnik. *Flow analysis and optimization of LISP-like programs.* In the 6^{th} ACM POPL, pages 244–256, 1979.

[JM86] N. D. Jones, A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 296–306, St. Petersburg, Florida, 1986.

[Klo91] J.W. Klop. Term Rewriting Systems. In S. Abramsky, Dov M. Gabbay, T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6, Oxford University, 1991. Oxford University Press.

[KU77] J.B. Kam, J.D. Ullman. *Monotone data flow analysis frameworks.* Acta Informatica *7 : 3*, pages 305–318, 1977.

[Myc80] A. Mycroft. *The theory and practice of transforming call-by-need into call-by-name.* In: Robinet B. ed., Proceedings of the Fourth International Symposium on Programming, Paris, 22–24 April 1980, LNCS *83*, pages 270–281, 1980.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, December 1981.

[Pos46] E.L. Post. *A variant of a recursive unsolvable problem.* Bull. of Am. Math. Soc., Vol. 52, 4, pages 264–268, 1946.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction.* The MIT Press, 1993.

A Appendix

A.1 Proof of Theorem 4

If the semi-lattice L is finite and all property transformers are distributive, the backwards analysis problem can be solved by the h-marking algorithm.

Proof. Let us first prove the stationary h-marking μ_s for any backwards analysis problem to exist and be unique (Theorem 5), and then show that it solves the problem by establishing $H^{\downarrow}(a) \sqsubseteq \mu_s a$ (Theorem 5) and $\mu_s a \sqsubseteq H^{\downarrow}(a)$ (Theorem 6) for all addresses a in a scheme S. □

Theorem 5. *The stationary h-marking μ_s defined in Section 5.4 exists and it is unique for any backwards analysis problem for a program S. $H^{\downarrow}(a) \sqsubseteq \mu_s a$ holds for all addresses a in S.*

Proof. Note that a marking rule application to an address might only increase the mark of an address. Since the set of addresses of S is finite and so are all strictly increasing chains, any marking terminates provided it changes a mark at every application of a marking rule.

Let $\sqsubseteq$ be a partial order introduced on marks of S by

$$\mu_1 \sqsubseteq \mu_2 \stackrel{dfn}{\equiv} \forall a \ \mu_1 a \sqsubseteq \mu_2 a.$$

Observe that the stationary marking is a solution of the equation system

1. $\mu a.i = \mu a.i \sqcup Sem^{\downarrow}(f,i)(\mu a)$
   for all addresses a of S such that
   $S \downarrow a = f(t_1, \dots, t_n)$, where $f \in \mathcal{F}_b, i \in \{1, \dots, n\}$,
2. $\mu[F] = \mu[F] \sqcup (\bigsqcup_{a \in C} \mu a)$
   for all defined symbols F of S,
   where C is the set of the addresses where F is called in S,
3. $\mu c.i = \mu c.i \sqcup (\bigsqcup_{a \in A} \mu a)$
   for all such addresses c of calls to F in S that
   $\mu c \neq \perp$, where A is the set of addresses of occurrences of the ith formal parameter of F in F's body, $1 \leq i \leq r(F)$,

that satisfies $\mu_0 \sqsubseteq \mu_s$. Let us show that any reachable h-marking of S does not exceed any solution μ of the system such that $\mu_0 \sqsubseteq \mu$. For the initial h-marking it holds by the condition for μ. Suppose $\mu' \sqsubseteq \mu$ for a reachable h-marking μ', and the h-marking μ'' is obtained from μ' by applying a marking rule to an address a. If $S \downarrow a = f(t_1, \dots, t_n)$, where $f \in \mathcal{F}_b$, then $\forall i = 1, \dots, n \ \mu'' a.i = \mu' a.i \sqcup Sem^{\downarrow}(f,i)(\mu' a) \sqsubseteq \mu a.i \sqcup Sem^{\downarrow}(f,i)(\mu a) = \mu a.i$. If $S \downarrow a = F(t_1, \dots, t_n)$, where $F \in \mathcal{F}_d$, then $\mu''[F] = \mu'[F] \sqcup \mu' a \sqsubseteq \mu[F] \sqcup \mu a = \mu[F]$. If a is an addresses of the ith formal parameter of F in F's body, $1 \leq i \leq r(F)$, and c is the address of a call to F for which $\mu' c \neq \perp$, then $\mu'' c.i = \mu' c.i \sqcup \mu' a \sqsubseteq \mu c.i \sqcup \mu a = \mu c.i$. Other addresses a do not change their marks when building μ'' out of μ', therefore $\mu'' a = \mu' a \sqsubseteq \mu a$. So, the stationary h-marking is the least solution of the system out of those solutions μ that satisfy $\mu_0 \sqsubseteq \mu$. If μ_1, μ_2 are two stationary h-markings of S, it implies $\mu_1 \sqsubseteq \mu_2$ and $\mu_2 \sqsubseteq \mu_1$, i.e. $\mu_1 = \mu_2$.

In order to prove $\forall a \ H^{\downarrow}(a) \sqsubseteq \mu_s a$ it suffices to show

$$\forall a \ \forall b \in Image(a) \ h^{\downarrow}(b) \sqsubseteq \mu_s a$$

for stationary h-marking μ_s. Let us show it by induction on the address length b.

Let b have length 0. Then $b = \Lambda$, and $h^{\downarrow}(b) = h_0$. On the other hand, since $b \in Image(a)$, we have either $a = \Lambda$ and $\mu_s(\Lambda) \sqsupseteq \mu_0(\Lambda) = h_0$, or a is an address of either the body of a defined symbol or a formal/actual parameter occurrence. In the latter case $\mu_s a \sqsupseteq \mu_s(\Lambda) \sqsupseteq \mu_0(\Lambda) = h_0$.

Let $\forall a\ \forall b \in Image(a)\ h^{\downarrow}(b) \sqsubseteq \mu_s a$ hold for all addresses b shorter than n. Prove the statement for an arbitrary $b \in Image(a)$ of length n. It follows that $b = c.i$ for some c of length $n-1$. If there is an address d such that $a = d.i, S \downarrow d = f(t_1, \dots, t_k)$ for $f \in \mathcal{F}_b$, then the induction hypothesis $h^{\downarrow}(c) \sqsubseteq \mu_s d$ implies $\mu_s a = \mu_s a \sqcup Sem^{\downarrow}(f,i)(\mu_s d) \sqsupseteq Sem^{\downarrow}(f,i)(h^{\downarrow}(c)) = h^{\downarrow}(b)$. Otherwise, there is a sequence of addresses $a = a_1, a_2, \dots, a_k$ such that all of them but last one are addresses of the bodies of defined symbols or parameter occurrences. Besides, $b \in Image(a_i)$ É $\forall\ i = 1, \dots, k-1\ \mu_s a_i = \mu_s a_i \sqcup \mu_s a_{i+1}$ holds for $i = 1, \dots, k$, i.e. $\forall\ i = 1, \dots, k-1\ \mu_s a_i \sqsupseteq \mu_s a_{i+1}$, and for a_k there exists an address d such that $a_k = d.i, S \downarrow d = f(t_1, \dots, t_k)$ with some $f \in \mathcal{F}_b$, and therefore $\mu_s a \sqsupseteq \mu_s a_k = \mu_s a_k \sqcup Sem^{\downarrow}(f,i)(\mu_s d) \sqsupseteq Sem^{\downarrow}(f,i)(h^{\downarrow}(c)) = h^{\downarrow}(b)$. $\Box$

We will establish now the approximation in the other direction. Recall that a function $\varphi : L \to L$ is *distributive*, iff

$$\forall\ x, y \in L\ \varphi(x \sqcup y) = \varphi(x) \sqcup \varphi(y).$$

Theorem 6. *If the semi-lattice L is finite and for all $f \in \mathcal{F}_b$ and for all $i = 1, \dots, r(f)$ the property transformer $Sem^{\downarrow}(f,i) : L \to L$ is distributive, then for the stationary h-marking μ_s and for all addresses a of a program S the inequality $\mu_s a \sqsubseteq H^{\downarrow}(a)$ holds.*

Proof. For a proof it suffices to show that $\forall a\ \mu a \sqsubseteq H^{\downarrow}(a)$ for all reachable h-markings μ. For the initial h-marking μ_0 we have $\mu_0(\Lambda) = h_0 \sqsubseteq H^{\downarrow}(\Lambda)$, since $\Lambda \in Image(\Lambda)$ and $h^{\downarrow}(\Lambda) = h_0$. For other addresses a, $\mu_0 a = \bot \sqsubseteq H^{\downarrow}(a)$.

Let $\forall a\ \mu a \sqsubseteq H^{\downarrow}(a)$ hold for a reachable h-marking μ, and the h-marking μ' is obtained out of μ by applying the marking rule to a. Then in the case

$S \downarrow a = f(t_1, \dots, t_n)$, where $f \in \mathcal{F}_b$, we have

$$\forall\, i \in \{1, \dots, n\}\ \mu' a.i = \mu a.i \sqcup Sem^{\downarrow}(f,i)(\mu a) \sqsubseteq$$
$$H^{\downarrow}(a.i) \sqcup Sem^{\downarrow}(f,i)(H^{\downarrow}(a)) =$$
$$H^{\downarrow}(a.i) \sqcup Sem^{\downarrow}(f,i)(\bigsqcup_{b \in Image(a)} h^{\downarrow}(b)) =$$
$$H^{\downarrow}(a.i) \sqcup (\bigsqcup_{b \in Image(a)} Sem^{\downarrow}(f,i)(h^{\downarrow}(b)))$$
$$= H^{\downarrow}(a.i) \sqcup (\bigsqcup_{b \in Image(a.i)} h^{\downarrow}(b)) =$$
$$H^{\downarrow}(a.i) \sqcup H^{\downarrow}(a.i) = H^{\downarrow}(a.i).$$

Let now $S \downarrow a = F(t_1, \dots, t_n)$, where $F \in \mathcal{F}_d$. Note that

$$H^{\downarrow}(a) = \bigsqcup_{b \in Image(a)} h^{\downarrow}(b) \sqsubseteq \bigsqcup_{b \in Image([F])} h^{\downarrow}(b) = H^{\downarrow}([F]),$$

because any call image is a body image as well. Thus,

$$\mu'[F] = \mu[F] \sqcup \mu a \sqsubseteq H^{\downarrow}([F]) \sqcup H^{\downarrow}(a) \sqsubseteq$$
$$H^{\downarrow}([F]) \sqcup H^{\downarrow}([F]) = H^{\downarrow}([F]).$$

Let a is the address of the ith formal parameter of F, and c is any of F-call addresses for which $\mu c \neq \perp$. Observe that $H^{\downarrow}(a) = \bigsqcup_{b \in Image(a)} h^{\downarrow}(b) \sqsubseteq \bigsqcup_{b \in Image(c.i)} h^{\downarrow}(b) = H^{\downarrow}(c.i)$, since any formal parameter image is a corresponding actual parameter image in a call with a non-empty image set. Hence, $\mu' c.i = \mu c.i \sqcup \mu a \sqsubseteq H^{\downarrow}(c.i) \sqcup H^{\downarrow}(a) \sqsubseteq H^{\downarrow}(c.i) \sqcup H^{\downarrow}(c.i) = H^{\downarrow}(c.i)$. For other addresses d of S we have $\mu' d = \mu d \sqsubseteq H^{\downarrow}(d)$. $\Box$

A.2 Proof of Theorem 2

If the semi-lattice L is finite and all property transformers are distributive in each argument, the forwards analysis problem can be solved by the d-marking algorithm.

Proof. The proof technique for this theorem is similar to the proofs in Appendix A.1 for the backwards analysis. To ensure the termination we need no infinitely ascending chains in the semi-lattice of functions on L, which is guaranteed by the finiteness of L. The part $D(a) \sqsubseteq \mu a$ is proved by showing $\forall a\ \forall\ n \geq 0\ \ d(a,n) \sqsubseteq \mu_s a$ (induction on n). The proof of $\mu a \sqsubseteq D(a)$ is done by induction on the d-marking step. Start off with the least marking, and at every step add nothing that "exceeds" $D(a)$. It is slightly more complicated since the semantic properties are functions, and the deductive semantic function $Sem^{\uparrow}$ is second-order. On the other hand, this is a possibility for a speed-up: the property transformer used in the marking can itself be changed during the marking.

Let us prove that $\mu a \sqsubseteq D(a)$ for any address a and any reachable marking μ in a program S. For the initial marking μ_0 it holds since $\mu_0 a = \perp\!\!\!\perp \sqsubseteq D(a)$ for any term but a variable at address a, and $\mu_0 a = Pr_i = D(a)$ for an occurrence of a variable x_i at a. Let $\forall a.\ \mu a \sqsubseteq D(a)$ hold for a reachable marking μ, and the marking μ' is obtained from μ by applying a marking rule to an address a. If $S \downarrow a = f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}_b$, then $\mu' a = \mu a \sqcup Sem^{\uparrow}(f)(\mu a.1, \ldots, \mu a.n) \sqsubseteq D(a) \sqcup Sem^{\uparrow}(f)(D(a.1), \ldots, D(a.n)) = D(a) \sqcup D(a) = D(a)$. In case $S \downarrow a = F(t_1, \ldots, t_n)$, where $F \in \mathcal{F}_d$, we have $\mu' a = \mu a \sqcup (\mu[F])(\mu a.1, \ldots, \mu a.n) \sqsubseteq D(a) \sqcup D([F])(D(a.1), \ldots, D(a.n)) = D(a) \sqcup D(a) = D(a)$. $\square$

Equivalent Transformations of Recursive Programs with Finite Unfolding *

Andrei Sabelfeld
Chalmers University of Technology
Göteborg, Sweden
andrei@cs.chalmers.se

Viktor Sabelfeld
Karlsruhe University
Karlsruhe, Germany
sabelfel@ira.uka.de

Abstract

We study equivalence relations for so-called finite recursive programs (programs with Finite Unfolding). The paper describes the universal interpretation domain and constructs complete systems for the tree and functional equivalence of finite recursive programs. We obtain the upper bound $C^{2^{O(n)}}$ for the size of the determinant of a finite program of size n and show that this bound is also the lower bound.

*This paper appeared in *Programming and Computer Software*, 23 (2):70–77, 1997.

1 Introduction

The problems of analysis, synthesis, optimisation, verification and debugging of functional programs need a theory of transformations that would allow us to formally describe correct transformations of such programs. Back in the 1960s J. McCarthy started the study of equivalent transformations of functional programs, his work was concerned with the language **LISP**. We aim at constructing a theory that would not be too dependent on the semantic constructions of a particular programming language.

In this paper, the universal model of functional programs, namely, recursive programs, is considered. Two sorts of equivalence are investigated: tree (the canonical forms of unfoldings, determinants, coincide) and functional (the programs result equally for all admissible interpretations).

We consider a wider class of recursive programs than that investigated in the literature, beginning with the classical work of Garland and Luckham [5]. The conditions of conditional terms allow for not only atomic formulas, as was usually assumed in earlier investigations, but also terms of the general form. Besides this, a program limits the set of possible interpretations of any basic symbol by assigning to it a collection of strict parameters: an interpretation should give "undefined" if any strict set contains at least one undefined argument.

We consider the subclass of finite recursive programs, i.e., programs with a finite unfolding. In the paper, we prove the determinant size of such programs to be bounded by $C^{2^{O(n)}}$, where C is the maximum arity of basic symbols involved in the program and n is the program size. We construct complete transformation systems for the tree and functional equivalence relations of finite programs. For tree equivalence, the system consists of three rule schemes: deletion (insertion) of unused definitions, folding (unfolding) and replacement of hopeless terms. For functional equivalence, three more rule schemes are added: context-sensitive replacement, reduction of conditional terms and deletion (insertion) of an unessential test. The algorithms for recognising the functional and tree equivalence of finite recursive programs are based on the application of the formulated transformation rules.

Note, that the problem of tree equivalence is in the general case equivalent to the well-known open problem of equivalence of the deterministic push-down automata, and the problem of functional equivalence is undecidable. One can prove that for finite programs the latter problem is NP-hard.

2 Recursive Programs

Let $\mathcal{X} = \{x, y, z, \dots\}$ be a set of variables, $\mathcal{F}_b = \{\omega, f, g, h, \dots\}$ a set of *basic symbols*, $\mathcal{F}_d = \{F, F_1, F_2, \dots\}$ a set of *defined symbols*, and $\mathcal{F} = \mathcal{F}_b \bigcup \mathcal{F}_d$. Let $r(s)$ denote the *arity* (the number of argument positions) of a symbol $s \in \mathcal{F}$ and $r(\omega) = 0$. By $\mathcal{T}(\mathcal{P}, X)$ we denote the set terms over the functional symbols from the set $\mathcal{P}$ and variables from $X, \mathcal{T} = \mathcal{T}(\mathcal{F}_b, \mathcal{X})$ is the set of basic terms. Basic symbols of arity zero are also called constants. Each basic symbol of a nonzero arity is associated with a nonempty subset $Strict(f)$ of the set $2^{\{1,\dots,r(f)\}}$, which is called a collection of *strict sets of parameters* [3] of the symbol f and limits the set of its possible interpretations. That is, an interpretation of a term $f(d_1, \dots, d_{r(f)})$ is undefined if for each $\Delta \in Strict(f)$ there exists i from Δ such that the value d_i is undefined. The notation $\alpha \prec \beta$ stands for the assertion "term α is a subterm of β".

A *recursive program* is a pair $S = \langle e; DEF \rangle$, where e is a term called an *entry* of the program and DEF is a finite set of definitions of symbols from $\mathcal{F}_d$. A *definition* of a symbol $F \in \mathcal{F}_d$ has the following form.

$$F(x_1, \dots, x_n) \Leftarrow t\,,$$

where $n = r(F)$, $x_1, \dots, x_n \in \mathcal{X}$ are mutually distinct *formal parameters* of the symbol F, and the term $t \in \mathcal{T}(\mathcal{F}, \{x_1, \dots, x_n\})$ is called the *body* of the symbol F.

Example:

$$S_0 = \left\langle F_1(h, h)\ ;\ \begin{array}{l} F_1(x, y) \Leftarrow \mathbf{if}(px, F_2(fx, fy), x) \\ F_2(x, y) \Leftarrow \mathbf{if}(py, F_1(fy, fx), y) \\ F_3(x) \quad\ \Leftarrow \mathbf{if}(px, F_3(fx), x) \\ Strict(f) = Strict(p) = \{\{1\}\}, \\ Strict(\mathbf{if}) = \{\{1, 2\}, \{1, 3\}\} \end{array} \right\rangle$$

The special constant ω will be interpreted only by value "undefined" and, thus will also be called "undefined". Below we use the terminology and denotations introduced in [3].

3 Universal domain and equivalence

A *domain* is a triple $\langle D, \leq, \perp \rangle$, where D is a set, $\perp \in D$ (the "bottom") and $\leq$ is a partial order on D that satisfies the following two conditions:

- $\forall\, d \in D \ \ \perp \leq d$.

- *Completeness condition:* for each chain C in D the *least upper bound* $lub(C)$ belongs to D. ($d \le lub(C)$ for all d in C, and $lub(C) \le u$ for all u in D such that $d \le u$ for all d in C.)

Let us construct the *universal domain* $\mathcal{U}$ of all finite and infinite regularly constructed terms. Let T_f be the set of all terms under the signature $\langle \mathcal{X}, \mathcal{F}_b \rangle$ that contains all constants from $\mathcal{F}_b$ and all variables from $\mathcal{X}$ and is closed under application of the rule for any $f \in \mathcal{F}_b$:

$$t_1, \dots, t_n \in T_f \ \& \ \exists \Delta \in Strict(f) \ \forall i \in \Delta \ t_i \neq \omega \Rightarrow f(t_1, \dots, t_n) \in T_f.$$

By $t[N \leftarrow \tau]$ we denote the term obtained from a term t by substituting the term τ for all occurrences of the subterms with addresses from the set N. We introduce a partial order $\sqsubseteq$ on T_f by setting $t_1 \sqsubseteq t_2$ iff there exists a set N of mutually independent addresses in t_2 such that $t_1 = t_2[N \leftarrow \omega]$.

An arbitrary term t that does not contain calls to defined symbols can be *reduced* by repeatedly applying to it the rules from the set

$$R = \{f(t_1, \dots, t_n) \to \omega \mid f \in \mathcal{F}_b \& \ \forall \ \Delta \in Strict(f) \ \exists \ i \in \Delta \ t_i = \omega\}$$

while applicable. The result of reduction of a term t is denoted by $Red(t)$. Obviously, the reduced term belongs to the set T_f. Consider the set $\tilde{T}_f$ of infinite chains of the form $t_0 \sqsubseteq t_1 \sqsubseteq \dots \sqsubseteq t_i \sqsubseteq \dots$ with elements $t_i \in T_f$. Each chain $s \in \tilde{T}_f$ corresponds to a unique, possibly infinite, tree-term $\bar{s}$ which can be defined by using special normal representations of the chains in $\tilde{T}_f$.

Define the *cutting* operator $Cut(t, k)$ of a term t to a depth $k \ge 0$ by

$$Cut(t, k) = t[N_t^k \leftarrow \omega],$$

where N_t^k is the set of addresses of all subterms of the term t at the depth k that are neither variables nor constants. Given a chain $s \in \tilde{T}_f$, we have $s = t_0 \sqsubseteq t_1 \sqsubseteq \dots \sqsubseteq t_i \sqsubseteq \dots$. Since s is monotone and the number of tree nodes at the depth i is finite, conclude

$$\forall i \ \exists k_i \ \forall j \ge k_i \ Cut(t_j, i) = Cut(t_{k_i}, i).$$

Using this property, define the *cutting a chains* to a depth i, $\overline{Cut}(s, i)$, as term $\overline{Cut}(s, i) = Cut(t_{k_i}, i)$. A sequence of terms $\bar{s}_i = Red(\overline{Cut}(s, i))$ for $i = 0, \dots$ is called *normal representation* of the chain s. Since $\forall \ i \ \overline{Cut}(s, i) \sqsubseteq \overline{Cut}(s, i+1)$ for any chain $s \in \tilde{T}_f$ and the reduction operation preserves the order $\sqsubseteq$, we obtain $\forall \ i \ Red(\overline{Cut}(s, i)) \sqsubseteq Red(\overline{Cut}(s, i+1))$, i.e. the normal representation of any chain is also a chain from $\tilde{T}_f$. Let T_U be the set of normal representations of chains from $\tilde{T}_f$. Suppose that $\ll$ is an order relation defined elementwise by

$$\bar{s}^1 \ll \bar{s}^2 \stackrel{dfn}{\equiv} \forall\ i\ \bar{s}^1_i \sqsubseteq \bar{s}^2_i.$$

Lemma 1 *$\langle T_U, \ll\rangle$ is a partially ordered set.*

Proof. The reflexivity, transitivity and antisymmetry of $\ll$ follow from the corresponding properties of the relation $\sqsubseteq$. $\Box$

Theorem 1 *The triple $\mathcal{U} = \langle T_U, \ll, \bar{\omega}\rangle$, where $\bar{\omega}$ is the infinite chain consisting of ω elements, is a domain.*

Proof. By the lemma, $\langle T_U, \ll\rangle$ is a partially ordered set. The $\bar{\omega}$ element is the bottom of T_U, since $\forall\ i\ \omega \sqsubseteq \bar{s}_i$ for any chain $\bar{s} = \{\bar{s}_i\}_{i=1}^{\infty}$. Let us now check the completeness condition. For a chain $\overline{s^0} \ll \overline{s^1} \ll \ldots \ll \overline{s^k} \ll \ldots$ of elements $\overline{s^j} = s^j_0 \sqsubseteq \ldots \sqsubseteq s^j_i \sqsubseteq \ldots$ from T_U let us construct the least upper bound $\overline{q} = q_0 \sqsubseteq \ldots \sqsubseteq q_i \sqsubseteq \ldots \times T_U$. For this purpose, consider the sequences $r_i = \{s^j_i\}_{j=0}^{\infty}$ for $i = 0, \ldots$. Since $\overline{s^j} \ll \overline{s^{j+1}}$, each sequence r_i is a chain. Since all elements of the chain r_i are terms of depth no more than i, there exists an element in this chain $q_i = s^k_i$ such that $s^l_i \sqsubseteq s^k_i$ for all $l \geq 0$. Obviously, q_i is the least upper bound of the chain r_i with respect to the order $\sqsubseteq$. It is also clear that the constructed chain $\overline{q} = q_0 \sqsubseteq \ldots \sqsubseteq q_i \sqsubseteq \ldots$ is a normal representation of itself. The fact that $\overline{q}$ is the least upper bound of the chain $\overline{s^0} \ll \ldots \ll \overline{s^k} \ll \ldots$ wrt $\ll$ follows from the fact that q_i is the least upper bound of the chain r_i wrt $\sqsubseteq$. $\Box$

Let D and D' be domains. A function $\varphi : D \to D'$ is called *monotone* if $\forall\, d, d' \in D\ d \leq d' \Rightarrow \varphi(d) \leq \varphi(d')$. A monotone function is called *continuous* if it preserves the least upper bounds of chains, i.e. $\varphi(lub(L)) = lub(\varphi(L))$.

An *interpretation* I fixes a domain D associates

- each variable $x \in \mathcal{X}$ with an element $I(x) \in D$ and

- each symbol f from $\mathcal{F}_b$ with a continuous function $I(f) : D^{r(f)} \to D$ satisfying the following *strictness* condition.

$$(\forall\ \Delta \in Strict(f)\ \exists\ i \in \Delta\ d_i = \bot) \Rightarrow I(f)(d_1, \ldots, d_n) = \bot.$$

By structural induction, extend the definition of interpretation I to arbitrary terms:

$$I(t) = \begin{cases} I(x), & \text{if } t \text{ is a variable } x \in \mathcal{X}, \\ I(f)(I(t_1), \dots, I(t_n)), & \text{if } t = f(t_1, \dots, t_n), \text{where } f \in \mathcal{F}_b \\ & \text{and } n = r(f), \\ \perp & \text{otherwise.} \end{cases}$$

For "infinite terms" $t = lub(\{t_n : n \geq 0\})$ we set

$$I(t) = lub(\{I(t_n) : n \geq 0\}).$$

An important example of interpretation is the *universal interpretation* J with the domain $\mathcal{U}$ which is defined as follows. Let $J(x) = x$ for the variables $x \in \mathcal{X}$. For $f \in \mathcal{F}_b, n = r(f)$, and $t_1, \dots, t_n \in T_U$, we set

$$J(f)(t_1, \dots, t_n) = \begin{cases} f(t_1, \dots, t_n), & \text{if } \exists\ \Delta \in Strict(f) \\ & \forall\ i \in \Delta\ t_i \neq \omega, \\ \omega & \text{otherwise.} \end{cases}$$

Finally, the *determinant* of a program S is defined as the limit of the sequence of *determinant approximations* $J(\pi^n e)$ by

$$det(S) = lub(\{J(\pi^n e) : n \geq 0\}),$$

where e is the entry and J is the universal interpretation. The mapping π corresponds to the rule of "parallel outermost computation rule". It performs one step unfolding of all outermost calls in a term. It is defined by $\pi^0 t = t$, $\pi^n = \pi^{n-1}\pi$ for $n > 0$.

For the example program

$$\left\langle \begin{array}{l} F(u); F(x) \Leftarrow f(x, F(gx)), \\ Strict(f) = \{\{1\}, \{2\}\}, Strict(g) = \{\{1\}\} \end{array} \right\rangle$$

the first elements of the sequence of determinant approximations are

$$\begin{aligned} J(\pi^0 F(u)) &= \omega, \\ J(\pi^1 F(u)) &= f(u, \omega), \\ J(\pi^2 F(u)) &= f(u, f(gu, \omega)), \\ J(\pi^3 F(u)) &= f(u, f(gu, f(ggu, \omega))), \text{ etc.} \end{aligned}$$

Two recursive programs S_1 and S_2 are *tree equivalent* (written $S_1 \approx S_2$, cf. [6]), if $det(S_1) = det(S_2)$.

An interpretation I is *admissible* if its domain D_I contains at least two distinct maximum elements **F** and **T**, different from $\perp$, and the set of basic symbols $\mathcal{F}_b$ contains two constants **false**, **true** and the special ternary symbol **if**, such that $I(\mathbf{false}) = \mathbf{F}, I(\mathbf{true}) = \mathbf{T}$ and $I(\mathbf{if}) = cond$, where

$$cond(d_1, d_2, d_3) = \begin{cases} d_2, & \text{if } d_1 = \mathbf{T} \ , \\ d_3, & \text{if } d_1 = \mathbf{F} \ , \\ \perp & \text{otherwise.} \end{cases}$$

Denote the set of all admissible interpretations by $\mathcal{I}$. Two programs S_1 and S_2 are *functionally equivalent* (written $S_1 \sim S_2$) if

$$\forall\ I \in \mathcal{I}\ val(S_1, I) = val(S_2, I),$$

where $val(S, I) = lub(\{I(\pi^n e) : n \geq 0\})$, and e is the entry of the program S.

4 Transformations for the Tree Equivalence

The notions of *fragment, fragment occurrence, transformation rule and the scheme of a transformation* are borrowed from [3].

4.1 Deletion of Unused Definitions

The first rule scheme $L1$ (*Deletion of Unused Definitions*) contains all rules of the form $G_1 \leftrightarrow G_2$, where G_1 is a fragment without entries and G_2 is an empty fragment.

Example:

$$\left\langle \begin{array}{l} F_1(u); \\ F_1(x) \Leftarrow f(x) \\ F_2(x) \Leftarrow g(F_1(x)) \end{array} \right\rangle \leftrightarrow \left\langle \begin{array}{l} F_1(u); \\ F_1(x) \Leftarrow f(x) \end{array} \right\rangle$$

4.2 Folding and Unfolding

Suppose G is a fragment that contains exactly one definition

$$F(x_1, \ldots, x_n) \Leftarrow t,$$

and a is the address of an occurrence of a call $F(t_1, \ldots, t_n)$ in a term outside the body of the symbol F. Let $G' = G[a \leftarrow \sigma t]$, where $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ (the term $[t_1/x_1, \ldots, t_n/x_n]t$ is obtained from the term t by simultaneously substituting all occurrences of the variable x_i by the term $t_i, i = 1, \ldots, n$). Then $G \leftrightarrow G'$ belongs to the scheme $L2$ (*Folding and Unfolding*).

Example:

$$\left\langle \begin{array}{l} F_1(u); \\ F_1(x) \Leftarrow f(x) \end{array} \right\rangle \leftrightarrow \left\langle \begin{array}{l} f(u); \\ F_1(x) \Leftarrow f(x) \end{array} \right\rangle$$

4.3 Replacement of Hopeless Terms

An address a in a fragment G is *hopeless* if it satisfies the *tree-emptiness condition* $G/a \approx \omega$ (G/a stands for the term at the address a in the fragment G). In order to find hopeless addresses, one can formulate a problem of forwards semantic analysis of the fragment G as in [2]. The semilattice of properties has only two elements: $L_{st} = \{\bot, \top\}$ so that $\bot \sqcup \top = \top$, $\bot \sqcup \bot = \bot$, $\top \sqcup \top = \top$. Interpret the $\top$ value as "tree-nonemptiness" and the $\bot$ value as "tree-emptiness".

The rule scheme $L3$ (*Replacement of Hopeless Terms*) contains all rules of the form $G \leftrightarrow G[a \leftarrow \omega]$, where a is a hopeless address of the fragment G. It includes, in particular, the rules

$$\{f(t_1, \ldots, t_n) \leftrightarrow \omega |\ f \in \mathcal{F}_b, \forall\ \Delta \in \mathit{Strict}(f)\ \exists\ i \in \Delta\ t_i = \omega\}.$$

Example: If $\mathit{Strict}(f) = \mathit{Strict}(g) = \mathit{Strict}(h) = \{\{1\}\}$, then

$$\left\langle \begin{array}{l} F(u, \omega); \\ F(x, y) \Leftarrow g(hy, F(fx, hy)) \end{array} \right\rangle \leftrightarrow \left\langle \begin{array}{l} \omega; \\ F(x, y) \Leftarrow g(hy, F(fx, hy)) \end{array} \right\rangle$$

If $\mathit{Strict}(g) = \{\{1, 2\}\}$, then $g(\omega, t) \leftrightarrow \omega$; $\mathbf{if}(\omega, s, t) \leftrightarrow \omega$, $\mathbf{if}(t, \omega, \omega) \leftrightarrow \omega$.

Consider the system of transformations Σ_t consisting of all rules generated by thr schemes $L1$–$L3$. It is easy to prove that the transformations from Σ_t preserve tree-equivalence, i.e. we have

Lemma 2 $S_1 \overset{\Sigma_t}{\leftrightarrow} S_2 \;\Rightarrow\; S_1 \approx S_2$.

5 The Class of Finite Recursive Programs

A recursive program S is *finite* if its determinant $\mathit{det}(S)$ is finite.

A recursive program is *reduced* if

- it does not contain any definitions and
- its entry does not contain tree-empty subterms different from ω.

Lemma 3 *The property of finiteness of a program S is decidable. There exists an algorithm that recognises finiteness of a program and, if it is finite, transforms the program into a reduced one by transformations from Σ_t.*

Proof. The labelling algorithm that finds all hopeless addresses of a program is described in [1]. Thus, by applying the Replacement of Hopeless Terms rule, one can replace all terms of a program at hopeless addresses by the term ω. Further, from the obtained program S' construct its *graph of calls* $G_{S'}$ whose nodes are the entry and all definitions of the program S. An edge connects the entry e or a definition $F(\dots) \Leftarrow e$ with a definition B iff the term e contains the symbol defined in B.

Obviously, $S \approx S'$ and the program S' is finite iff the graph $G_{S'}$ contains no cycles accessible from the node corresponding to the entry.

If S' is finite, eliminate all calls by applying the unfolding transformation and delete all unused definitions. The acyclicity of $G_{S'}$ guarantees the termination of this process. As a result, the initial program S is transformed into a reduced program $S^\star$ such that $S \approx S^\star$ in accordance with Lemma 2. $\square$

Example: Let

$$S = \langle F(a, \omega, b); F(x, y, z) \Leftarrow f(F(y, z, x), z)\rangle.$$

If $Strict(f) = \{\{1, 2\}\}$, then the substitution of ω for all occurrences of terms at hopeless addresses gives the program $S' = \langle\omega; F(x, y, z) \Leftarrow \omega,$ whose graph of calls has no edges. In case $Strict(f) = \{\{1\}, \{2\}\}$, the initial program has no hopeless address. Thus, $S' = S$ and the graph of calls has a cycle going through the definition of the symbol F and accessible from the entry. This means that in the latter case the program S is not finite.

5.1 The expansion theorem

Theorem 2 *The size of the determinant of a finite program of size n does not exceed C^{2^n+1}, where $C = \max(2, c)$ and c is the maximum arity of all basic symbols occurring in the program. This bound is achievable in the sense that for any n there exists a program of size n whose determinant has the size $2^{2^{\frac{n-2}{5}}} - 1$.*

Proof. We can assume all terms of program S at hopeless addresses are ω, since the corresponding transformation does not increase the program's size.

By induction on the number of defined symbols l of S, let us establish that for the depth $h(det(S))$ of the determinant of S, it holds that $h(det(S)) \leq 2^n (n = |S|)$.

For a program without definitions $h(det(S)) \leq |det(S)| = |S| = n \leq 2^n$.

Suppose the statement holds for all programs with no more than l definitions. Assume S has $l + 1 > 0$ definitions. Since S is finite, it has at least one definition $F(x_1, \dots, x_n) \Leftarrow \tau$ of some symbol F in whose body there are no calls. Consider the program S' obtained from S by deleting this definition and declaring the symbol F as a basic one. Let $|\tau| = m < n$, then $|S'| \leq n - m$. By the induction hypothesis, we have $h(det(S')) \leq 2^{n-m}$. The determinant of S can be obtained from $det(S')$ by unfolding all calls to F and reducing the result of unfolding. Thus, $h(det(S)) \leq h(det(S')) \cdot m \leq 2^{n-m}m \leq 2^{n-m}n \leq 2^n$. Since the depth of the tree $det(S)$ does not exceed 2^n and each node has no more than c sons, conclude $|det(S)| \leq C^{2^n+1}$.

Consider the program

$$S_m = \left\langle \begin{array}{l} F_m a; \\ F_i x \Leftarrow F_{i-1}F_{i-1}x, 1 \leq i \leq m \\ F_0 x \Leftarrow f(x, x). \end{array} \right\rangle$$

Let us denote by $\varphi_i(x)$ the determinant of the program obtained from S_m by replacing the entry by the term $F_i x, 0 \leq i \leq m$. Obviously, the terms $\varphi_i(x)$ satisfy the equalities $\varphi_0(x, x) = f(x, x)$ and $\varphi_i(x) = \varphi_{i-1}(\varphi_{i-1}(x))$ for $i = 1, \dots, m$. Let us show by induction on i that the term $\varphi_i(x, x)$ is a complete binary tree under the signature $\langle\{x\}, \{f\}\rangle$ with 2^{2^i} leaves x. Indeed, $\varphi_0(x, x) = f(x, x)$. If the term $\varphi_{i-1}(x, x)$ has exactly $2^{2^{i-1}}$ occurrences of x, then after replacing them by $\varphi_{i-1}(x, x)$ we will have in the term $\varphi_i(x, x) = \varphi_{i-1}(x, x)[\varphi_{i-1}(x, x)/x]$ exactly $2^{2^{i-1}} \cdot 2^{2^{i-1}} = 2^{2^i}$ leaves. Hence, the term $det(S_m) = \varphi_m(a)$ has exactly 2^{2^m} leaves and $2^{2^m+1} - 1$ nodes. Since $|S_m| = n = 5m + 7$, we have $|det(S_m| = 2^{2^{\frac{n-7}{5}+1}} - 1 = 2^{2^{\frac{n-2}{5}}} - 1.\square$

5.2 Completeness of the System Σ_t

The algorithm for recognising the tree equivalence of finite programs S_1 and S_2 consists of converting the initial programs by applying transformations from Σ_t into reduced programs (terms) e_1, e_2. It only remains to check whether the reduced programs coincide.

$$S_1 \approx S_2 \ \Leftrightarrow\ S_1^\star \approx S_2^\star \ \Leftrightarrow\ e_1 \approx e_2 \ \Leftrightarrow\ e_1 = e_2.$$

Theorem 3 (Completeness Theorem) $S_1 \overset{\Sigma_t}{\leftrightarrow} S_2 \;\Leftrightarrow\; S_1 \approx S_2$ *for finite* S_1, S_2.

Proof. Lemma 2 gives the forward implication. It remains to show to show how to transform arbitrary tree equivalent programs one into another. The algorithm for recognising tree equivalence was formulated in such a way that when applied to an arbitrary pair of programs, it yields the answer **Yes** along with the sequence of transformations from Σ_t converting one program into the other. □

6 Transformations Preserving Functional Equivalence

6.1 Context Replacement

The rule scheme $F1$ (*Context Replacement*) allows one to eliminate redundant tests in some conditional terms. For example,

$$\mathbf{if}(px, \mathbf{if}(px, \alpha, \beta), \gamma) \sim \mathbf{if}(px, \alpha, \gamma).$$

Here, checking the condition $p(x)$ in the then-alternative of the conditional in the left-hand side is redundant. The transformation formulation also takes into account the fact that if a term σ is "more defined" than the term α, i.e. $\alpha \sqsubseteq \sigma$, and $I(\alpha) \in \{\mathbf{true}, \mathbf{false}\}$, then $I(\sigma) = I(\alpha)$.

The rule scheme *Context Replacement* contains all rules of the form

$$\{\mathbf{if}(\alpha, \beta, \gamma) \leftrightarrow \mathbf{if}(\alpha, \beta[a \leftarrow \mathbf{true}], \gamma) \mid \alpha \sqsubseteq \beta/a]\}, \text{ and}$$

$$\{\mathbf{if}(\alpha, \beta, \gamma) \leftrightarrow \mathbf{if}(\alpha, \beta, \gamma[b \leftarrow \mathbf{false}]) \mid \alpha \sqsubseteq \gamma/b]\},$$

where a and b are addresses of some subterms in β and γ, respectively.
Examples:

$$\mathbf{if}(\alpha, \mathbf{if}(\alpha, \beta, \gamma), \delta) \leftrightarrow \mathbf{if}(\alpha, \mathbf{if}(\mathbf{true}, \beta, \gamma), \delta),$$

if $Strict(g) = \{\{1\}, \{2\}\}$, then

$$\mathbf{if}(g(\omega, x), g(y, x), z) \leftrightarrow \mathbf{if}(g(\omega, x), \mathbf{true}, z).$$

6.2 Reduction of Conditional Terms

The rule scheme $F2$ (*Reduction of Conditional Terms*) contains all rules of the form

$$\mathbf{if}(\mathbf{true}, \beta, \gamma) \leftrightarrow \beta, \quad \mathbf{if}(\mathbf{false}, \beta, \gamma) \leftrightarrow \gamma,$$

where β and γ are arbitrary terms.

6.3 Deletion of an Unessential Test

Consider the following relation on terms.

$$\beta < \alpha \stackrel{def}{=} \forall\, I \in \mathcal{I}\ (I(\beta) \neq \bot \Rightarrow\ I(\alpha) = \mathbf{T} \vee I(\alpha) = \mathbf{F}).$$

It is a decidable relation, since it can be checked, for example, by verifying the implication for a finite set of finite interpretations.

The rule scheme $F3$ (*Deletion of an Unessential Test*) contains all rules of the form $\mathbf{if}(\alpha, \beta, \beta) \leftrightarrow \beta$, where $\beta < \alpha$.

Example:

$$\mathbf{if}(p, x, y) \ \leftrightarrow\ \mathbf{if}(p, \mathbf{if}(p, x, y), \mathbf{if}(p, x, y)).$$

Let Σ_f be the transformation system containing all rules generated by the rule schemes $L1$ — $L3$ and $F1$ — $F3$.

7 Completeness of the System Σ_f

It is easy to prove that applying rules from Σ_f preserves functional equivalence, i.e. we have

Lemma 4 $S_1 \stackrel{\Sigma_f}{\leftrightarrow} S_2 \ \Rightarrow\ S_1 \sim S_2.$

A reduced program (term) t is *normal*, if

- each of its subterms $f(t_1, \dots, t_n)$ with an outer symbol $f \neq \mathbf{if}$ does not contain subterms σ such that $f(t_1, \dots, t_n) < \sigma$;
- each of its subterms $\mathbf{if}(\alpha, \beta, \gamma)$ does not contain subterms σ of the term α different from α such that $\mathbf{if}(\alpha, \beta, \gamma) < \sigma$; and
- the conditions of all conditional subterms of the term t are different from the constants **true** and **false**.

Lemma 5 *Any finite program can be converted into a functionally equivalent normal term by applying the rules from* Σ_f.

Proof. Let us first reduce the initial program and eliminate constant conditions of conditional terms by applying $F3$. If after doing so there is an occurrence of a term t in the program such that $t = f(t_1, \ldots, t_n), f \neq \mathbf{if}, \sigma \prec t$ or $t = \mathbf{if}(\alpha, \beta, \gamma), \sigma \prec \alpha, \sigma \neq \alpha$ for some subterm σ of t, then by applying $F3$ replace this occurrence of t by the term $\mathbf{if}(\sigma, t, t)$ to which we repeatedly apply the following two transformations (in any order, while possible).

- $\mathbf{if}(\sigma, t, t) \xrightarrow{F1} \mathbf{if}(\sigma, t[N_\sigma \leftarrow \mathbf{true}], t[N_\sigma \leftarrow \mathbf{false}])$, where N_σ is the set of addresses of all occurrences of the term σ in the term t.
- By applying $\mathbf{if}(\mathbf{true}, t_1, t_2) \longrightarrow t_1$ and $\mathbf{if}(\mathbf{false}, t_1, t_2) \longrightarrow t_2$ eliminate constant conditions of conditional terms in each of the alternatives of the obtained term.

It is clear that this process will terminate, since the term size monotonely decreases with each transformation step. □

Example:

For the term $t = \mathbf{if}(\mathbf{if}(\sigma, \beta, \gamma), \mu, \nu)$ it holds that $t < \sigma$. Therefore, its normalisation yields

$$\begin{aligned} t \xrightarrow{F3} \mathbf{if}(\sigma, \mathbf{if}(\mathbf{if}(\sigma, \beta, \gamma), \mu, \nu), \mathbf{if}(\mathbf{if}(\sigma, \beta, \gamma), \mu, \nu)) \quad \xrightarrow{F1} \\ \mathbf{if}(\sigma, \mathbf{if}(\mathbf{if}(\mathbf{true}, \beta, \gamma), \mu, \nu), \mathbf{if}(\mathbf{if}(\mathbf{false}, \beta, \gamma), \mu, \nu)) \\ \xrightarrow{F1} \mathbf{if}(\sigma, \mathbf{if}(\beta, \mu, \nu), \mathbf{if}(\gamma, \mu, \nu)). \end{aligned}$$

Example:

If $Strict(g) = \{\{1\}, \{2\}\}$ then for the term $t = \mathbf{if}(g(\mathbf{if}(g(x, y), x, x), y), u, v)$ it holds that $t < g(x, y)$, and thus

$$\begin{aligned} t &\xrightarrow{F3} \mathbf{if}(g(x,y), \mathbf{if}(g(\mathbf{if}(g(x,y), x, x), y), u, v), \mathbf{if}(g(\mathbf{if}(g(x,y), x, x), y), u, v)) \\ &\xrightarrow{F1} \mathbf{if}(g(x,y), \mathbf{if}(g(\mathbf{if}(\mathbf{true}, x, x), y), u, v), \mathbf{if}(g(\mathbf{if}(\mathbf{false}, x, x), y), u, v)) \\ &\xrightarrow{F3} \mathbf{if}(g(x,y), \mathbf{if}(g(x,y), u, v), \mathbf{if}(g(x,y), u, v)) \\ &\xrightarrow{F1} \mathbf{if}(g(x,y), \mathbf{if}(\mathbf{true}, u, v), \mathbf{if}(\mathbf{false}, u, v)) \\ &\xrightarrow{F3} \mathbf{if}(g(x,y), u, v). \end{aligned}$$

Informally, in normal terms, the condition of any conditionals subterm is minimum in the sense that it does not include subterms that are checked during any execution of this conditional.

Let us describe the algorithm below for recognising functional equivalence of finite programs S_1 and S_2. We will be applying it to a pair of normal terms s and t obtained from the initial programs by reducing and consequent normalising of the entries. The algorithm is recursive: in order to determine whether $s \sim t$, it calls itself with the queries $s' \sim t'$ for some term pairs s' and t' such that either $h(s') < h(s)$ or $h(s') = h(s)$ and $h(t') < h(t)$. The notation $[\alpha \lhd \beta] : \mathcal{T} \to \mathcal{T}$ in the algorithm description stands for the following transformation of terms: the term $[\alpha \lhd \beta]\gamma$ is obtained from the term γ by repeatedly applying the following three transformations while at least one of them is applicable.

- All subterms τ such that there exists a term σ such that $\sigma \sim \alpha$ and $\sigma \sqsubseteq \tau$ are replaced by the term β.

- By applying the rules $\mathbf{if}(\mathbf{true}, t_1, t_2) \longrightarrow t_1$ and $\mathbf{if}(\mathbf{false}, t_1, t_2) \longrightarrow t_2$, conditionals with constant conditions are reduced.

- By applying the ruled $L3$, tree-empty subterms are replaced by ω.

Here, α is an arbitrary term and $\beta \in \{\mathbf{true}, \mathbf{false}\}$.

The recognising algorithm.

$s \sim t \;=\; (s = t)$, if s and t are constants or variables.

$$
\begin{array}{rll|l}
\omega \sim \mathbf{if}(\alpha, \beta, \gamma) = & \omega \sim \alpha \vee & & \mathbf{if}(\alpha, \beta, \gamma) \xleftrightarrow{L3} \omega \\
& \omega \sim \beta \;\&\; \omega \sim \gamma \vee & & \mathbf{if}(\alpha, \omega, \omega) \xleftrightarrow{L3} \omega \\
& \mathbf{true} \sim \alpha \;\&\; \omega \sim \beta \vee & & \mathbf{if}(\mathbf{true}, \omega, \beta) \xleftrightarrow{F2} \omega \\
& \mathbf{false} \sim \alpha \;\&\; \omega \sim \gamma & & \mathbf{if}(\mathbf{false}, \alpha, \omega) \xleftrightarrow{F2} \omega
\end{array}
$$

$\omega \sim f(t_1, \dots, t_n) = \forall\, \Delta \in Strict(f)\; \exists\, i \in \Delta\; \omega \sim t_i \quad | \quad f(t'_1, \dots, t'_n) \xleftrightarrow{L3} \omega,$
where $t'_i = \omega$, if $\omega \sim t_i$ and $t'_i = t_i$ otherwise.

If s is a constant ($s \neq \omega$) or a variable, and t is an arbitrary normal term, then $s \sim t = \;(s = t)$.

If $f \neq \mathbf{if}$ and $g \neq \mathbf{if}$, then

$$
\begin{array}{rl}
f(t_1, \dots, t_n) \sim g(t'_1, \dots, t'_m) = & \omega \sim f(t_1, \dots, t_n) \;\&\; \omega \sim g(t'_1, \dots, t'_m) \vee \\
& f = g \;\&\; \forall\, i\ (1 \leq i \leq n)\ t_i \sim t'_i
\end{array}
$$

If $f \neq \mathbf{if}$ and $t = f(t_1, \dots, t_n)$, then $t \sim \mathbf{if}(\alpha, \beta, \gamma)$ does not hold, because assuming $t \sim \mathbf{if}(\alpha, \beta, \gamma)$ implies $t < \alpha$, which contradicts the fact that t normal.

$$\begin{gathered}\mathbf{if}(\alpha,\beta,\gamma) \sim \mathbf{if}(\sigma,\mu,\nu) = \\ h(\mathbf{if}(\sigma,\mu,\nu)) < h(\mathbf{if}(\alpha,\beta,\gamma)) \ \& \ \mathbf{if}(\sigma,\mu,\nu) \sim \mathbf{if}(\alpha,\beta,\gamma) \ \vee \\ \omega \sim \mathbf{if}(\alpha,\beta,\gamma) \ \& \ \omega \sim \mathbf{if}(\sigma,\mu,\nu) \ \vee \\ \alpha \sim \sigma \ \& \ [\rho \triangleleft \mathbf{true}]\beta \sim [\rho \triangleleft \mathbf{true}]\mu \ \& \ [\rho \triangleleft \mathbf{false}]\gamma \sim [\rho \triangleleft \mathbf{false}]\nu \ \vee \\ \mathbf{if}(\alpha,\beta,\gamma) < \sigma \ \& \ [\sigma \triangleleft \mathbf{true}]\mathbf{if}(\alpha,\beta,\gamma) \sim [\sigma \triangleleft \mathbf{true}]\mu \ \& \\ [\sigma \triangleleft \mathbf{false}]\mathbf{if}(\alpha,\beta,\gamma) \sim [\sigma \triangleleft \mathbf{false}]\nu.\end{gathered}$$

Here, ρ is the term with minimum depth among the terms α and σ. If the last disjunctive component is true (and all previous ones are false), the transformation is performed n the following way.

$$\begin{gathered}t = \mathbf{if}(\alpha,\beta,\gamma) \overset{F3}{\longleftrightarrow} \mathbf{if}(\sigma,t,t) \overset{recursion,F1,F2,L3}{\longleftrightarrow} \mathbf{if}(\sigma,[\sigma \triangleleft \mathbf{true}]t,[\sigma \triangleleft \mathbf{false}]t) \\ \overset{recursion}{\longleftrightarrow} \mathbf{if}(\sigma,[\sigma \triangleleft \mathbf{true}]\mu,[\sigma \triangleleft \mathbf{false}]\nu) \overset{recursion}{\longleftrightarrow} \mathbf{if}(\sigma,\mu,\nu).\end{gathered}$$

Theorem 4 *Σ_f is a complete system of transformations for functional equivalence. i.e. $S_1 \overset{\Sigma_f}{\leftrightarrow} S_2 \ \Leftrightarrow \ S_1 \sim S_2$ for finite programs S_1, S_2.*

Proof. Lemma 4 gives the forward implication. It remains to show to show how to transform arbitrary functional equivalent programs one into another. However, in case the above algorithm for recognising functional equivalence finds a pair of programs to be equivalent, it also points to the sequence of rules from Σ_f converting one program into the other. □

Example:

For $s = \mathbf{if}(p,\mathbf{if}(q,x,y),\mathbf{if}(q,u,v))$ and $t = \mathbf{if}(q,\mathbf{if}(p,x,u),\mathbf{if}(p,y,v))$ it holds that $s < q$, hence

$$\begin{gathered}s \overset{F3}{\longrightarrow} \mathbf{if}(q,s,s) = \mathbf{if}(q,\mathbf{if}(p,\mathbf{if}(q,x,y),\mathbf{if}(q,u,v)),\mathbf{if}(p,\mathbf{if}(q,x,y),\mathbf{if}(q,u,v))) \\ \overset{F1}{\longrightarrow} \mathbf{if}(q,\mathbf{if}(p,\mathbf{if}(\mathbf{true},x,y),\mathbf{if}(\mathbf{true},u,v)),\mathbf{if}(p,\mathbf{if}(\mathbf{false},x,y),\mathbf{if}(\mathbf{false},u,v))) \\ \overset{F2}{\longrightarrow} \mathbf{if}(q,\mathbf{if}(p,x,u),\mathbf{if}(p,y,v)) = t.\end{gathered}$$

8 Conclusion

In view of the estimate of the determinant size for finite programs proved in Section 5.1, it is unlikely that there exists a polynomial algorithm for recognising the tree equivalence of such programs. There is even less hope for constructing a polynomial algorithm for recognising functional equivalence of finite programs, because the problem is NP-hard.

Among the rules of the transformation system Σ_f, the most complex rule is the deletion of an unessential test, because its application requires checking the condition $t < s$ for terms t and s. Note, if we restrict ourselves to the class of programs where the conditions of conditional terms are basic terms without conditionals, then $t < s$ can be checked in linear time.

References

[1] V.E. Kotov, V.K. Sabelfeld. The Theory of Program Schemes (In Russian), Moscow, Nauka, 1991.

[2] V.K. Sabelfeld. Transformations for Recursive Program Schemes, in *Mixed Computations and Program Transformations*, pp. 99–111, Novosibirsk, 1986.

[3] V.K. Sabelfeld. Tree Equivalence of Linear Recursive Schemes. *Programmirovanie*, pp. 5–26, 1994.

[4] Courcelle B., Vuillemin J. Semantics and axiomatics of a simple recursive language. Report No. 60, Institute of Informatics and Automatics, Rocquencourt, France, February 1974.

[5] Garland S.J., Luckham D.C. Program schemes, recursion schemes and formal languages. Report UCLA-ENG-7154, School of Engineering and Applied Science, University of California, Los Angeles, June 1971.

[6] Rosen B.K. Program equivalence and context-free grammars. *JCSS*. Vol. 11, pp. 358–374, 1975.

[7] Valiant L.G. The Equivalence problem for deterministic finite-turn pushdown machines, *Information and Control*, No. 5, pp. 123–133, 1975.

Correct Transformations of Logic Programs *

Andrei Sabelfeld

Department of Computer Science
Chalmers University of Technology
412 96, Göteborg, Sweden

andrei@cs.chalmers.se

Abstract

This paper describes a system of transformations that preserves the semantics of logic programs with respect to a fixed goal. We formalise some standard transformations and introduce two new transformation rules: *Copying/Merge of Copies* and *Contextual Replacement by Equal Term*. Correctness of all schemes of the transformation rules is proved. The applicability conditions of the schemes of rules are effectively decidable, that makes it useful for application in practice. Since some transformations simplify programs, the transformation system can be used for automated program transforming.

*This paper appeared in *Joint Bulletin of Novosibirsk Computer Center and the Institute of Informatics Systems*, 5:55–67, 1996.

1 Introduction

A lot of investigations on analysis, synthesis, specialisation, verification, and optimisation of programs relies on program transformations. When deriving a new program from an initial program the semantics should be kept unchanged. The formal approach consists of constructing an equivalence relation on programs and checking that this relation is preserved while transforming.

In this paper we investigate the logic programs described in [2]. However, we assume that a program contains a request and consider the semantics of the program with respect to the fixed request (goal).

Intuitively, the investigated equivalence relation of logic programs is described as follows. Two programs are equivalent iff any answer computed by one program is an answer for the other program. It is important that the equivalence does not depend on how 'the answer' is defined, declaratively or procedurally.

For a notion of such an equivalence we prove the correctness of the following transformation schemes: *Definition Introduction/Elimination, Clause Introduction/Elimination, Variable Renaming, Parameter Introduction/Elimination, Copying/Merge of Copies, Instance Clause Introduction/Elimination, Conjunct Introduction/Elimination, Folding/Unfolding, Contextual Replacement by Equal Term.* The latter scheme uses the powerful method of decision of the flow analysis problem on graphs and the concept of functional nets described in [3].

The applicability conditions of all schemes can be effectively checked. The whole system is aimed to be used in order to improve programs without affecting their semantics. A simple example is considered in the final part of the paper.

2 Concepts and Notations

We consider the *definite logic programs* (see [2] for a definition) with the assumption that all programs are constructed over a fixed first order language with finite number of predicate and functional symbols. We adopt the following notation.

By $Hd(C)$ and $Bd(C)$ we denote the head and the body of a clause C.

The *definition* $Def(P,p)$ of a predicate p in a program P is the set of all clauses C from P such that $Hd(C) = p$.

A predicate p *directly depends* on a predicate q in P iff the program has a clause $p(\dots) \leftarrow B$, and q occurs in B. p *depends* on q iff either p directly depends on q or there exists a predicate r such that p directly depends on r and r depends on q.

A formula F *requires* a predicate p iff either p occurs in F or there exists a predicate occurring in F which depends on p.

The *relevant part* $Rel(P, F)$ of a program P for a formula F is the union of the definitions of the predicates required by F.

$Vars(t)$ denotes the set of variables of a term (atom, goal, or clause) t.

$\forall(F)$ stands for $\forall x_1 \dots \forall x_n F$, where F is a formula and $x_1, \dots, x_n$ are all its free variables.

Let P be a program with a goal $G = \leftarrow A_1, \dots, A_k$ and θ be a substitution on variables occurring in G. Then θ is called *correct* for $P \cup \{G\}$ iff $\forall((A_1 \wedge \dots \wedge A_n)\theta)$ is a logical consequence of the program P.

We say that a substitution θ is *R-computed* for a program P with a goal G iff the result of the computation using the selection rule R is an SLD-refutation for $P \cup \{G\}$ and θ is the restricted on variables from G composition of the most general unifiers of the inference steps.

3 Equivalence

Definition A substitution θ is said to be a *renaming* for a formula F iff $\theta = \{x_1/y_1, \dots, x_n/y_n\}$, where $y_1, \dots, y_n$ are different variables, $x_i \in Vars(F)$ for all $i = 1, \dots, n$, and $(Vars(F) \setminus \{x_1, \dots, x_n\}) \cap \{y_1, \dots, y_n\} = \emptyset$.

Definition A program P_1 with a goal G_1 is *computable* by a program P_2 with a goal G_2 ($P_1 \cup \{G_1\} \rightsquigarrow P_2 \cup \{G_2\}$ for short) iff for any computation rule R and any R-computed substitution θ for $P_1 \cup \{G_1\}$ there exists a computation rule R' such that θ is an R'-computed substitution for $P_2 \cup \{G_2\}$.

Definition Programs P_1 and P_2 with goals G_1 and G_2, respectively, are called *equivalent* ($P_1 \cup \{G_1\} \sim P_2 \cup \{G_2\}$ for short) iff $P_1 \cup \{G_1\} \rightsquigarrow P_2 \cup \{G_2\}$ and $P_2 \cup \{G_2\} \rightsquigarrow P_1 \cup \{G_1\}$.

Theorem 3.1 *Let P_1 and P_2 be programs and $G_1 = \leftarrow L$ and $G_2 = \leftarrow M$ be goals. Then the following statements are equivalent.*

1. $P_1 \cup \{G_1\} \sim P_2 \cup \{G_2\}$.

2. *θ is a correct substitution for $P_1 \cup \{G_1\}$ iff θ is a correct substitution for $P_2 \cup \{G_2\}$.*

3. For any substitution θ

$$P_1 \models \forall(L\theta) \text{ iff } P_2 \models \forall(M\theta)$$

Proof *2.* is equivalent to *3.* by definition of correct substitution. Equivalence of *1.* and *2.* follows from equivalence of the declarative and procedural semantics (theorems 7.1 and 9.5 in [2]). □

4 Transformation Rules

We denote $P_1 \cup G_1 \longleftrightarrow P_2 \cup G_2$ iff a program P_1 with a goal G_1 is transformable to a program P_2 with a goal G_2 or the other way round. In other words, it is allowed to apply transformation rules in both directions. In the case $G_1 = G_2$ we sometimes use a simpler notation: $P_1 \longleftrightarrow P_2$. If a goal is not specified in a transformation rule, then it is assumed that the rule is applicable to an arbitrary goal.

A transformation rule is *correct* iff $P_1 \cup G_1 \longleftrightarrow P_2 \cup G_2$ implies $P_1 \cup G_1 \sim P_2 \cup G_2$. A scheme of transformation rules is correct iff all instances of the scheme are correct.

4.1 Definition Introduction/Elimination

Let P and P' be programs, G be a goal, and there exists a predicate p such that $P' = P \cup Def(P', p)$ holds. Suppose that the predicate symbol p does not occur in $Rel(P', G)$, then the scheme of rules **DefIntr** contains the rule

$$P \longleftrightarrow P'.$$

Example

$$\left\{ \begin{array}{c} r \leftarrow q \\ \leftarrow r \end{array} \right\} \stackrel{DefIntr}{\longleftrightarrow} \left\{ \begin{array}{c} r \leftarrow q \\ p \leftarrow s \\ \leftarrow r \end{array} \right\}.$$

Lemma 4.1 *The scheme of rules* **DefIntr** *is correct.*

Proof Let $P \stackrel{DefIntr}{\longleftrightarrow} P'$. For any computation rule R SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G\}$ do not involve definitions of the predicate p, since p does not occur in $Rel(P', G)$. $P' = P \cup Def(P', p)$, therefore, the derivations in P and in P' by the rule R coincide and the R-computed substitutions coincide. Hence, $P \cup \{G\} \sim P' \cup \{G\}$. □

4.2 Clause Introduction/Elimination

Let $P = \{C_1, \dots, C_n\}$ be a program and G be a goal. We assume that no two clauses from P have common variables. Suppose that for some k one of the following conditions is valid.

- $C_k = p(t_1, \dots, t_m) \leftarrow A_1, \dots, A_l$. There exists no substitution unifying $p(t_1, \dots, t_m)$ and $p(s_1, \dots, s_m)$ for all occurrences of $p(s_1, \dots, s_m)$ in G and in $Bd(C_i)$ for all $i = 1, \dots, k-1, k+1, \dots, n$.

- $C_k = A \leftarrow A_1, \dots, p(t_1, \dots, t_m), \dots, A_l$. There exists no substitution unifying $p(t_1, \dots, t_m)$ and $p(s_1, \dots, s_m)$ for all occurrences of $p(s_1, \dots, s_m)$ in $Hd(C_i)$ for all $i = 1, \dots, k-1, k+1, \dots, n$.

Then the scheme of rules **ClauseIntr** contains the rule

$$P \longleftrightarrow P \setminus \{C_k\}$$

Example

$$\left\{ \begin{array}{l} p(a) \leftarrow \\ p(b) \leftarrow \\ p(a) \leftarrow q \\ \leftarrow p(a) \end{array} \right\} \overset{ClauseIntr}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow \\ p(a) \leftarrow q \\ \leftarrow p(a) \end{array} \right\} \overset{ClauseIntr}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow \\ \leftarrow p(a) \end{array} \right\}.$$

The condition of variable distinction is required here in order to unify correctly. If this condition were violated, then atoms taken from different clauses, for example, $p(x)$ and $p(f(x))$ would not be unifiable.

Lemma 4.2 *The scheme of rules* **ClauseIntr** *is correct.*

Proof Let us denote $P \setminus \{C_k\}$ by P'. Let $P \overset{ClauseIntr}{\longleftrightarrow} P'$. For any computation rule R SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G\}$ do not involve the clause C_k because of one of the formulated conditions. Therefore, the derivations for P and P' by the rule R coincide and the R-computed substitutions coincide. Hence, $P \cup \{G\} \sim P' \cup \{G\}$. $\square$

4.3 Variable Renaming

Let P be a program. Let $C \in P$ and θ be a renaming for variables of C. Then the scheme of rules **Rename** contains the rule

$$P \longleftrightarrow (P \setminus \{C\}) \cup \{C\theta\}.$$

Example

$$\{p(x,y) \leftarrow q(y)\} \overset{Rename}{\longleftrightarrow} \{p(z,x) \leftarrow q(x)\}\,.$$

Lemma 4.3 *The scheme of rules* **Rename** *is correct.*

Proof Let us denote $(P \setminus \{C\}) \cup \{C\theta\}$ by P'. Let $P \overset{Rename}{\longleftrightarrow} P'$. For any goal $G = \leftarrow L$ and any substitution θ

$$P \models \forall(L\theta) \text{ iff } P' \models \forall(L\theta),$$

since the renaming of variables in clauses does not change the declarative semantics. By theorem 3.1, we have $P \cup \{G\} \sim P' \cup \{G\}$. □

4.4 Parameter Introduction/Elimination

Let P be a program, G be a goal, and $Def(P,p) \neq \emptyset$. Assume that no two clauses from P have common variables.

Suppose that there exists a k such that for any $C \in Def(P,p)$ $Hd(C) = p(t_1,\ldots,t_k,\ldots,t_n)$, and all constants and variables appearing in t_k do not occur in $Bd(C)$. Let P' and G' be obtained from P and G, respectively, replacing all subterms $p(s_1,\ldots,s_k,\ldots,s_n)$ by $p(s_1,\ldots,s_{k-1},s_{k+1},\ldots,s_n)$.

Suppose each occurrence of $p(\ldots)$ that is not unifiable with $Hd(C)$ remains ununifiable after the transformation.

Then the scheme of rules **Param** contains the rule

$$P \cup \{G\} \longleftrightarrow P' \cup \{G'\}.$$

Example

$$\left\{\begin{array}{l} p(x,f(y)) \leftarrow q(f(x)) \\ \leftarrow p(a,z) \end{array}\right\} \overset{Param}{\longleftrightarrow} \left\{\begin{array}{l} p(x) \leftarrow q(f(x)) \\ \leftarrow p(a) \end{array}\right\}.$$

If the above requirement is omitted then the transformation will not be always correct. Consider, for example, $z = b$. In this case an unused clause in SLD-derivations for G would become used in SLD-derivations for G' after the transformation.

Lemma 4.4 *The scheme of rules* **Param** *is correct.*

Proof Let $P \cup \{G\} \overset{Param}{\longleftrightarrow} P' \cup \{G'\}$. For any computation rule R SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G'\}$ coincide, except that in derivation for $P' \cup \{G'\}$ the predicate p does not have the k-th parameter. It is guaranteed by the application criterion that the unification on each step of derivation is identical. Therefore, R-computed substitutions coincide too. Hence, $P \cup \{G\} \sim P' \cup \{G'\}$. □

4.5 Copying/Merge of Copies

Let P be a program and G be a goal. Defined in P predicates $p_1, \dots, p_n$ are called *similar* in P iff for any $i, j \in 1, \dots, n$

$$Def(P, p_i)|_{q/p_1, \dots, q/p_n} = Def(P, p_j)|_{q/p_1, \dots, q/p_n},$$

where q is a predicate that does not have definitions in P. $M|_{q/p}$ stands for the set of clauses obtained by replacement of all occurrences of the predicate symbol p in clauses from the set of clauses M by the symbol q. We use the same notation for replacement of occurrences of predicates in one clause.

Let predicates $p_1, \dots, p_n$ be defined in P. Put $D = Def(P, p_1) \cup \dots \cup Def(P, p_n)$. Suppose that $p_1, \dots, p_n$ are similar in P, then the scheme of rules **Copy** contains the rule

$$P \cup \{G\} \longleftrightarrow ((P \setminus D) \cup Def(P, p_1) \cup \{G\})\,|_{p_1/p_2, \dots, p_1/p_n}.$$

Example

$$\left\{ \begin{array}{l} p(a) \leftarrow q(a), r(c) \\ q(a) \leftarrow r(a), p(c) \\ r(a) \leftarrow p(a), q(c) \\ \leftarrow q(a) \end{array} \right\} \overset{Copy}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow p(a), p(c) \\ \leftarrow p(a) \end{array} \right\}.$$

Lemma 4.5 *The scheme of rules* **Copy** *is correct.*

Proof Denote $((P \setminus D) \cup Def(P, p_1))\,|_{p_1/p_2, \dots, p_1/p_n}$ by P' and $G|_{p_1/p_2, \dots, p_1/p_n}$ by G'. Let $P \cup \{G\} \overset{Copy}{\longleftrightarrow} P' \cup \{G'\}$. For any computation rule R SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G'\}$ coincide modulo substitution of similar predicates. Therefore, the R-computed substitutions coincide too. Hence, $P \cup \{G\} \sim P' \cup \{G'\}$. $\square$

4.6 Instance Clause Introduction/Elimination

Let P be a program. Suppose that clauses C and D from P have no common variables and there exists such θ that $C\theta = D$. Then the scheme of rules **Instance** contains the rule

$$P \longleftrightarrow P \setminus \{D\}.$$

Example

$$\left\{ \begin{array}{l} p(a, b) \leftarrow q(a, a, x) \\ p(y, z) \leftarrow q(v, a, w) \end{array} \right\} \overset{Instance}{\longleftrightarrow} \left\{ \; p(y, z) \leftarrow q(v, a, w) \; \right\}.$$

Lemma 4.6 *The scheme of rules* **Instance** *is correct.*

Proof We denote $P \setminus \{D\}$ by P'. Let $P \overset{Instance}{\longleftrightarrow} P'$. For any goal $G = \leftarrow L$ and any substitution η

$$P \models \forall(L\eta) \text{ iff } P' \models \forall(L\eta),$$

since both introduction and elimination of an instance clause do not influence the declarative semantics. By theorem 3.1, we have $P \cup \{G\} \sim P' \cup \{G\}$. □

4.7 Conjunct Introduction/Elimination

Let P be a program and G be a fixed goal. Let either $C = G$ or $C \in P$. Let $Bd(C) = \leftarrow A_1, \ldots, A_i, \ldots, A_i, \ldots, A_n$. Suppose that C' is obtained from C by removing one of the A_i's in the body. Thus, we have $C' = Hd(C) \leftarrow A_1, \ldots, A_i, \ldots, A_n$. Then the scheme of rules **Conj** contains the rule

$$P \cup \{G\} \longleftrightarrow ((P \cup \{G\}) \setminus \{C\}) \cup \{C'\}.$$

Example

$$\left\{ \begin{array}{l} p \leftarrow q(x), p, q(x) \\ \leftarrow p, p \end{array} \right\} \overset{Conj}{\longleftrightarrow} \left\{ \begin{array}{l} p \leftarrow p, q(x) \\ \leftarrow p, p \end{array} \right\} \overset{Conj}{\longleftrightarrow} \left\{ \begin{array}{l} p \leftarrow p, q(x) \\ \leftarrow p \end{array} \right\}.$$

The introduction/elimination of a conjunct does not change the declarative semantics of a program, therefore, by theorem 3.1, we receive the following lemma.

Lemma 4.7 *The scheme of rules* **Conj** *is correct.*

4.8 Folding/Unfolding

This transformation scheme is based on the described in [6] unfold transformation. It has been slightly modified to suit our approach.

Let P be a program and G be a fixed goal. Let either $C = G$ or $C \in P$. Let $C = H \leftarrow L, A, K$, where A is an atom, and L and K are (maybe empty) conjunctions of atoms. Assume that

1. $E_1, \ldots, E_n = \{C \in P \mid A \text{ is unifiable with } Hd(C)\}$, $n \geq 1$, and $\theta_1, \ldots, \theta_n$ are the corresponding most general unifiers;

2. $n = 1$ in the case $G = C$;

3. $D_i = (H \leftarrow L, Bd(E_i), K)\theta_i$, for $i = 1, \ldots, n$;

4. no two clauses from $C, E_1, \ldots, E_n, D_1, \ldots, D_n$ have common variables.

Then the scheme of rules **Fold** contains the rule

$$P \longleftrightarrow (P \setminus \{C\}) \cup \{D_1, \ldots, D_n\}.$$

A step of unfolding corresponds to an application of the SLD-resolution to the clause C with the atom A selected and the input clauses $E_1, \ldots, E_n$.

Example

$$\left\{\begin{array}{l} s \leftarrow q(x) \\ s \leftarrow r(y) \\ t \leftarrow p(z), s \end{array}\right\} \overset{Fold}{\longleftrightarrow} \left\{\begin{array}{l} s \leftarrow q(x) \\ s \leftarrow r(y) \\ t \leftarrow p(z_1), q(x_1) \\ t \leftarrow p(z_2), r(y_1) \end{array}\right\}.$$

The requirement of variable distinction is relevant here. For example, if $x = y = z$ then unfolding would not be correct.

Lemma 4.8 *The scheme of rules* **Fold** *is correct.*

Proof We denote $(P\setminus\{C\})\cup\{D_1, \ldots, D_n\}$ by P'. Let $P\cup\{G\} \overset{Fold}{\longleftrightarrow} P'\cup\{G\}$ for some goal G. As noted above, a step of unfolding corresponds to an application of the SLD-resolution to the clause C with the atom A selected and input clauses $E_1, \ldots, E_n$. Hence, folding/unfolding allows us to extend/reduce the SLD-derivation without influence on the answer. Therefore, $P \cup \{G\} \sim P' \cup \{G'\}$. $\square$

4.9 Contextual Replacement by Equal Term

If the value of one of actual parameters of all occurrences of a predicate p in the right parts of the clauses of a program coincides with the value of some term t from other actual parameters, then it is possible to replace any occurrence of corresponding formal parameter in a clause defining p by term t from other formal parameters. These cases often arise, for instance, in *partial evaluation*, when such *functional dependencies* are discovered after a program specialisation with regard to a particular input.

Let us describe the algorithm based on application of the marking technique for search of the program invariants. The property of functional dependence of parameters will be a base for the construction of transformation by contextual replacement by equal term.

Equality relations

$x = x \; y = y$
$y = h(a)$
$x = f(y, a)$
$x = f(h(a), a)$

are represented by

Functional net

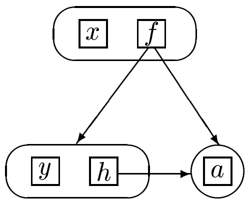

Figure 1: An example of a net

For this purpose let us formulate a flow analysis problem for properties of the directed graph $\Gamma(P)$ constructed according to a program $P \cup G$ as follows. The clauses of the program will be treated as the nodes of the graph.

An edge from a clause C to a clause D (distinct from C) is drawn iff in the right part of C there exists a predicate unifiable with $Hd(D)$. For correctness of unification we require different clauses to have no common variables. An edge from a clause C to itself is drawn iff in the right part of C there exists a predicate unifiable with $Hd(C')$, where C' is the clause obtained from C by replacement of all variables by different variables not contained in $Vars(C)$.

Obviously, edges do not lead to G.

As the semi-lattice of properties we use the semi-lattice $(\mathcal{L}, \sqcap)$ of the reduced acyclic nets described in [3]. More precisely, we allocate the semi-lattice $(\mathcal{D}, \sqcap)$ containing only such nets that represent the sets of equality of the form $x \equiv t$, where x is a variable, and t is a functional term.

An example of a net and term equalities it represents is given on Fig.1.

Let us take advantage of the defined in [3] *assignment effect* converter of nets on the semi-lattice $(\mathcal{D}, \sqcap)$ and introduce two new converters.

- If Z is a finite set of variables, then the application of the converter of nets $[\![Forget(Z)]\!]$ to a net s consists of removal of all elements associated with all variables from Z and in subsequent reduction of the result. It is, naturally, a distributive converter of nets from $\mathcal{D}$.

- $[\![x_n := t_n, \dots, x_1 := t_1]\!] s = [\![x_n := t_n]\!] (\dots ([\![x_1 := t_1]\!] s) \dots)$. This is a distributive converter of nets from $\mathcal{D}$, as far as it represents the superposition of distributive converters.

We get the formulation of the problem of flow analysis of the graph $\Gamma(G)$ by fixing the initial marking μ_0 that associates the mark $\mathbb{1}$ to all nodes and semantic function that associates the distributive converter of properties $Sem(C, D)$ to each pair of incidental clauses.

For defining the semantic function let us consider possible cases.

- Let $C = q(\dots) \leftarrow \dots, p(t_1, \dots, t_n), \dots$, $D = p(s_1, \dots, s_n) \leftarrow \dots$, and $C \neq D$, where C and D have no common variables, then for each edge related to an occurrence of p in the right part of C,

$$Sem(C, D) = Forget(Y)[\![\, z_1 := o_1, \dots, z_n := o_k \,]\!] s_D,$$

  where $Y = Vars(C)$, $\{z_1/o_1, \dots, z_k/o_k\}$ is the most general unifier of $p(t_1, \dots, t_n)$ and $p(s_1, \dots, s_n)$, and s_D is the net mark of the node D.

- Let $C = p(\dots) \leftarrow \dots, p(t_1, \dots, t_n), \dots$; then for each edge associated with an occurrence of p in the right part of C let us consider the clause C' obtained from C by replacement of all variables by different variables not contained in $Vars(C)$.

$$Sem(C, C) = Forget(Y)[\![\, z_1 := o_1, \dots, z_n := o_k \,]\!] s_C,$$

  where $Y = Vars(C')$, $\{z_1/o_1, \dots, z_k/o_k\}$ is the most general unifier of $p(t_1, \dots, t_n)$ and $Hd(C')$, and s_D is the net mark of the node C.

It follows from the results proved in [3] that one can effectively build the stationary marking μ_c of the graph $\Gamma(P)$, that is the exact solution of the flow analysis problem on $\Gamma(P)$.

We formulate the rule of transformation based on the construction of the stationary marking μ_c detecting the functional dependencies of formal parameters as follows. Let C be a clause of a program P and $(x \equiv t) \in assert(\mu_c C)$. We denote $\{x/t\}$ by θ. Then the scheme of rules **Replace** contains the rule

$$P \cup \{G\} \longleftrightarrow (P \cup \{G\}) \setminus \{C\} \cup \{C\theta\}.$$

Note that the formulated scheme of rules allows us to substitute x for t in C as well.

Example

$$\left\{ \begin{array}{l} q(x) \leftarrow p(x, f(x)) \\ p(h(z), v) \leftarrow r(z, v) \end{array} \right\} \overset{Replace}{\longleftrightarrow} \left\{ \begin{array}{l} q(x) \leftarrow p(x, f(x)) \\ p(h(z), f(h(z))) \leftarrow r(z, f(h(z))) \end{array} \right\}.$$

As noticed above, only distributive converters are used in the flow analysis problem. By theorem 2.4 from [3], the stationary marking of the graph can be effectively constructed. The marking is the exact solution of the flow analysis problem. Hence, the following lemma holds.

Lemma 4.9 *The scheme of rules* **Replace** *is correct.*

5 An Example of Program Optimisation

Consider a program

$$\left\{\begin{array}{l} Inter(Y,Z) \leftarrow In(x,Y), In(x,Z) \\ In(x,x.Y) \leftarrow \\ In(x,y.Z) \leftarrow x \neq y, In(x,Z) \\ \leftarrow Inter(Y,Y) \end{array}\right\}.$$

Here sets are represented by lists We have $Inter(Y,Z)$ iff $Y \cap Z \neq \emptyset$ and $In(x,Y)$ iff $x \in Y$.

$Inter(Y,Y)$ is the only call of the predicate $Inter$, therefore we can apply **Replace** and get

$$\left\{\begin{array}{l} Inter(Y,Y) \leftarrow In(x,Y), In(x,Y) \\ In(x,x.Y) \leftarrow \\ In(x,y.Z) \leftarrow x \neq y, In(x,Z) \\ \leftarrow Inter(Y,Y) \end{array}\right\}.$$

Now we can eliminate the conjunct duplication by **Conj**

$$\left\{\begin{array}{l} Inter(Y,Y) \leftarrow In(x,Y) \\ In(x,x.Y) \leftarrow \\ In(x,y.Z) \leftarrow x \neq y, In(x,Z) \\ \leftarrow Inter(Y,Y) \end{array}\right\}.$$

Here we **Unfold** the goal of the program

$$\left\{\begin{array}{l} Inter(Y,Y) \leftarrow In(x,Y) \\ In(x,x.Y) \leftarrow \\ In(x,y.Z) \leftarrow x \neq y, In(x,Z) \\ \leftarrow In(x,Y) \end{array}\right\}.$$

Now we are capable to eliminate the definition of $Inter$, since it is not used anymore. Using **DefIntr** we obtain finally

$$\left\{\begin{array}{l} In(x,x.Y) \leftarrow \\ In(x,y.Z) \leftarrow x \neq y, In(x,Z) \\ \leftarrow In(x,Y) \end{array}\right\}.$$

Observe that our transformation chain proves implicitly that $Y \cap Y \neq \emptyset$ iff $\exists z\ z \in Y$.

6 Related Work

A number of authors have investigated problems of transformation correctness of logic programs following the approach proposed by Tamaki and Sato [1]. This approach consists of imposing restrictions on the use of the transformation rules for ensuring total correctness.

Bossi and Cocco [4] focus their study on definite logic programs which are evaluated using the *PROLOG* leftmost computation rule. They prove that restricted versions of Tamaki and Sato's unfolding and folding preserve both the set of computed answer substitutions and *PROLOG* semantics.

Bossi and Etalle [5] consider general logic programs and show that Tamaki and Sato's rules preserve *acyclicity* of programs.

Recently, advances in the study of logic programs termination have stimulated the investigations of the relationship between program transformation and program termination. The paper by Lau, Ornaghi, Pettorossi, and Proietti [6] has contributed to this area. They introduce the concept of *existential termination.* A program P existentially terminates with respect to a goal G iff there exists an SLD-tree for $P \cup \{G\}$ which either has at least one success branch or it is finitely failed. They describe powerful classes of transformations which appear to be correct if existential termination of an initial program implies existential termination of the program that is final in a transformation chain. Clearly, such a condition is undecidable in general case.

A common approach consists of fixing a (maybe infinite) set of ground goals. In such a definition, conditions necessary for proof of correctness are either very constraining or they require the verification of complex, sometimes undecidable, properties.

Our conception is different. We fix an arbitrary single goal. This approach allows us to check the applicability of transformations and carry out transformations effectively. At the same time, our transformations let us detect non-trivial properties and considerably improve programs. Each scheme can be applied in both directions. Minimal restrictions on the use of transformation rules are imposed.

Acknowledgement

Thank you to Johan Jeuring for his comments and suggestions on an earlier draft of this paper.

References

[1] H. Tamaki, T. Sato. Unfold/fold transformation of logic programs. In *Proc. of the Second International Conference on Logic Programming,* Uppsala, Sweden, pages 127–138. Uppsala University, 1984.

[2] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Second Edition, 1987.

[3] V.E. Kotov, V.K. Sabelfeld. *Theory of program schemes.* Moscow, Nauka, 1991. In Russian.

[4] A. Bossi, N. Cocco. Preserving universal termination through unfold/fold. In *Proc. ALP'94,* LNCS 850, pages 269–286. Springer-Verlag, 1994.

[5] A. Bossi, S. Etalle. Transforming acyclic programs. *ACM Transactions on Programming Languages and Systems,* 16(4):1081-796, July 1994.

[6] K.-K. Lau, M. Ornaghi, A. Pettorossi and M. Proietti. Correctness of Logic Program Transformations Based on Existential Termination. In *Proc. of the Symposium on Logic Programming,* Oregon, USA, pages 480–494. Springer-Verlag, 1995.