Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Towards a complete transformational toolkit for compilers

J.A. Bergstra, T.B. Dinesh, J. Field and J. Heering

Computer Science/Department of Software Technology

CS-R9646 1996

Report CS-R9646
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Towards a Complete Transformational Toolkit for Compilers

J. A. Bergstra
University of Amsterdam and Utrecht University

T. B. Dinesh
CWI

J. Field
IBM T. J. Watson Research Center

J. Heering
CWI

Abstract

PIM is an equational logic designed to function as a "transformational toolkit" for compilers and other programming tools that analyze and manipulate imperative languages. It has been applied to such problems as program slicing, symbolic evaluation, conditional constant propagation, and dependence analysis. PIM consists of the untyped lambda calculus extended with an algebraic data type that characterizes the behavior of lazy stores and generalized conditionals. A graph form of PIM terms is by design closely related to several intermediate representations commonly used in optimizing compilers.

In this paper, we show that PIM's core algebraic component, PIM_t, possesses a *complete* equational axiomatization (under the assumption of certain reasonable restrictions on term formation). This has the practical consequence of guaranteeing that every semantics-preserving transformation on a program representable in PIM_t can be derived by application of PIM_t rules. We systematically derive the complete PIM_t logic as the culmination of a sequence of increasingly powerful equational systems starting from a straightforward "interpreter" for closed PIM_t terms.

This work is an intermediate step in a larger program to develop a set of well-

---

Authors' addresses: J. A. Bergstra, Vakgroep Programmatuur, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands; email: janb@wins.uva.nl; T. B. Dinesh and J. Heering, Department of Software Technology, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; email: {T.B.Dinesh, Jan.Heering}@cwi.nl; J. Field, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA; email: jfield@watson.ibm.com.

founded tools for manipulation of imperative programs by compilers and other systems that perform program analysis.

CR Subject Classification (1991): D.3.4 [**Programming Languages**]: Processors —*code generation, compilers, optimization*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*

Keywords & Phrases: imperative language, program transformation, partial evaluation, ω-completeness, term rewriting, Knuth-Bendix completion

Notes: This research was supported in part by the European Communities under ESPRIT Basic Research Action 7166 (CONCUR II) and ESPRIT LTR Project 21871 (SAGA), and by the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project.

This is an extensively revised version of Technical Report RC 20342, IBM T. J. Watson Research Center, Yorktown Heights, and Technical Report CS-R9601, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (January 1996). To appear in *ACM Transactions on Programming Languages and Systems*.

An abridged version of this paper has appeared in H. R. Nielson (ed.), *Programming Languages and Systems (ESOP '96)*, vol. 1058 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, pp. 92–107.

1. Introduction

1.1 Overview

Many intermediate representations (IRs) have been proposed as the basis for algorithms used in optimizing compilers for imperative languages. Typically, such IRs are intended to distill a source program down to certain essential semantic components, either to provide a framework for program transformations, to serve as a starting point for analysis algorithms, or to render some program property self-evident from the IR's structure. In studying the body of work that describes or exploits such IRs, it is often difficult to clearly differentiate the role of the IR from the role of the algorithm that uses it, to contrast the expressive or analytic capabilities of one IR with another, or to understand the relationship between a program's semantics and the IR manipulations that (implicitly) reason about it.

This paper is an effort to contribute to a better understanding of these issues by presenting a formal account of properties of an equational logic called PIM [Field 1992]. PIM consists of a *term language* and an associated set of *equations* which together are designed to function as a "transformational toolkit" for compilers and other programming tools that analyze and manipulate Algol-class imperative languages. PIM is a combination of the untyped lambda calculus and an algebraic data type, PIM_t, that characterizes the behavior of lazy stores [Cartwright and Felleisen 1989] and generalized conditionals. PIM_t is parameterized in such a way that a variety of base data types (e.g., integers, booleans, pairs, lists, streams) and "memory models" (e.g., byte addressed or "symbolically" addressed by variable name) may be accommodated in a manner that is orthogonal to the remainder of PIM. Lambda terms are used to model procedural constructs and loops, but are not needed to model other types of control flow. Extended with appropriate

memory models and operations on base types, we believe that the constructs of PIM are sufficient to model the principal dynamic semantic elements of Algol-class languages.

PIM is intended to represent a family of related IRs, each a class of graph representations of PIM terms that is useful for a different application. Several members of this family are by design closely related to existing IRs, particular those based on the Program Dependence Graph (PDG) or Static Single Assignment (SSA) form [Ferrante et al. 1987; Cytron et al. 1991; Ballance et al. 1990; Yang et al. 1990; Weise et al. 1994; Click 1995]. Our long-term goal is to show that a large class of program analysis algorithms can be recast into a combination of semantics-preserving transformations and abstract interpretations on PIM graphs.

As a first step toward that goal, we restrict our attention in this paper to studying the formal properties of various equational systems for PIM_t. Our principal result is exhibiting a *complete* equational axiomatization of PIM_t's semantics, under the assumption of certain reasonable restrictions on term formation. Formally, we show that there exists an ω-*complete* equational axiomatization of PIM_t's final algebra semantics. We systematically derive the complete logic for PIM_t as the culmination of a sequence of increasingly powerful equational systems starting from a straightforward "interpreter" for PIM_t. We also derive several confluent and terminating *rewriting* systems by orienting instances of rules from the equational systems. Such systems can be used not only as partial decision procedures for PIM_t, but also as algorithms for building various classes of IRs based on different PIM_t normal forms.

Obtaining a positive answer to the completeness question, albeit restricted to the first-order core of PIM, gives us some confidence that our transformational toolkit has an adequate supply of tools. In [Field 1992], it was shown that many aspects of the construction and manipulation of compiler intermediate representations could be expressed by *partially evaluating* PIM graphs using rewriting rules formed from oriented instances of PIM equations. Until now, however, we could not be certain that *all* the equations required to manipulate PIM_t terms were present (with or without restrictions on term formation). Obtaining a complete equational axiomatization for PIM_t is also an important prerequisite to designing a decision procedure.

We are aware of three other completeness results for logics for imperative languages, all of which, like our result, are restricted to first-order subsystems: Hoare et al. [1987] present (implicitly) a completeness result for an equational logic that PIM resembles in some respects. However, their language cannot represent abstract (i.e., unknown) stores or addresses, a property we feel is important for representing transformations and analyses commonly required in optimizing compilers. For example, computations on addresses are needed for representing pointer manipulations; abstract stores are needed, e.g., to reason about calls to procedures whose effect on the store is unknown. Mason and Talcott [1989] show that their logic for reasoning about equivalence in a Lisp-like language is complete. Unlike PIM, their logic is a sequent calculus. In addition, their logic axiomatizes a strict store semantics whose properties differ from PIM's lazy store. Finally, Selke [1989b] sketches a completeness result similar to that of Mason and Talcott for a sequent calculus for reasoning about PDGs. As with the language of Hoare et al., the only unknowns about which Selke's system can reason are base values.

1.2 Organization

The remainder of this paper proceeds as follows: In Section 2, we discuss PIM's relation to other work. In Section 3, we introduce the system PIM_t^0, which axiomatizes the operational behavior of PIM's first-order core. PIM_t^0 has the property that orienting its equations from left to right yields an interpreter for closed programs producing observable "base" values as results. We also discuss the behavior of PIM_t's functions using examples written in a simple programming language, and focus particularly on describing PIM's *lazy store* semantics. Section 4 presents several detailed examples of equivalence reasoning in PIM, and gives an overview of the relative power of various equation systems we develop in the sequel. In Section 5, we briefly discuss the higher order system derived from embedding PIM_t in a lambda calculus. While the higher order system is not the focus of this paper, this section gives a feel for the use of PIM in representing real programming languages. Section 6 discusses the relationship between ω-complete equational systems and partial evaluation. In Section 7, we present some basic definitions and illustrate the process by which we develop a complete logic using an example involving a stack data type. Next, in Section 8, we develop an enrichment of PIM_t^0 called PIM_t^+ that characterizes PIM_t's *final algebra* semantics, i.e., one that equates terms that behave the same in all contexts that generate observable values. Such a semantics is natural for developing semantics-preserving transformations on program fragments. In Section 9, we produce our main result: an *ω-complete* axiomatization of PIM_t^+, $\text{PIM}_t^=$. As before, $\text{PIM}_t^=$ is obtained simply by enriching PIM_t^+ with additional equations. The resulting system equates all *open* PIM_t programs that behave the same under every substitution for their free variables, and thus yields the complete transformational toolkit we seek. In Section 10, we show how *rewriting* systems formed by orienting subsystems of $\text{PIM}_t^=$ can be used to prove program equivalences and produce normal forms with various interesting structural properties. We then develop several rewriting systems based on $\text{PIM}_t^=$ and prove that they are confluent and terminating. Finally, Section 11 covers possible extensions and open problems.

2. PIM in Perspective

There has been considerable previous work on calculi and logics of program equivalence. In addition, the program analysis literature contains numerous accounts of intermediate representations and algorithms intended to accommodate efficient and accurate analysis of imperative programs. Drawing to a considerable extent on ideas from both areas, PIM was designed to bridge the gap between the latter area's practical aims and the former's logical rigor. In this section, we review previous work and its relation to PIM.

A graph form of PIM is by design closely related to intermediate representations used in optimizing compilers, such as the PDG [Ferrante, Ottenstein, and Warren 1987], SSA form [Cytron et al. 1991], GSA form [Ballance, MacCabe, and Ottenstein 1990], the PRG [Yang, Horwitz, and Reps 1990], the VDG [Weise et al. 1994], and the representation of Click [1995]. PIM was inspired in particular by the work of Cartwright and Felleisen [1989], who give a foundational account of PDG construction in a denotational (i.e., model-theoretic) setting. Our goal is to provide an operational realization of an extended version of Cartwright and Felleisen's se-

mantics, one that can be used not only for reasoning about equivalences, but also for carrying out partial evaluation and implementing certain classes of program analysis[1].

Selke [1989a], and Ramalingam and Reps [1989] (the latter building on earlier work of Horwitz et al. [1988]) provide semantics for variants of PDGs. Selke's operational semantics combines graph rewriting[2] and propagation of assigned values through graph edges. Ramalingam and Reps give a mathematical semantics for PRGs in terms of mutually recursive equations on streams of values. Both of these view a PDG as a program in itself, which is then evaluated via the semantics. By contrast, PIM can be used both to *derive* various PDG-like IRs (by transforming a fairly conventional store-based translation of the program), and to evaluate the resulting IR.

For structured programs, most of the non-trivial steps required to translate a program to the PIM analogue of one of the IRs mentioned above can be carried out as *source-to-source* transformations in PIM itself, once an initial PIM graph has been constructed from the program using a simple syntax-directed translation. For example, when constructing the PIM equivalent of GSA form [Ballance, MacCabe, and Ottenstein 1990], various equational transformations on PIM graphs correspond to computation of reaching definitions, building of data and control dependence edges, placement of so-called ϕ nodes, and determination of "gating" predicates.

For unstructured programs, the PIM analogue of a traditional IR may be constructed either by first restructuring the program (e.g., using a method such as that of Ammarguellat [1992]), or by using a continuation-passing transformation (e.g., one similar to that used by Weise et al. [1994]) in which unstructured transfers of control such as `goto`s or `break`s are treated as function calls. The translation process is then completed by application of the same transformations used for structured programs. We prefer the restructuring approach to the continuation-passing transformation, since it obviates the need to use higher-order "interprocedural" methods when analyzing first-order programs.

There has been considerable work on calculi for modeling imperative features in applicative languages [Felleisen and Friedman 1989; Felleisen and Hieb 1992; Swarup, Reddy, and Ireland 1991; Odersky, Rabin, and Hudak 1993]. While the details of each of these approaches differ considerably, their imperative features are inextricably bound with their applicative constructs. Felleisen and Friedman [1989], Felleisen and Hieb [1992], and Swarup et al. [1991] use lambda terms and β-reduction to sequence assignments; Odersky et al. [1993] use lambda terms in a "monadic" style to perform computations on dereferenced mutable variables. By contrast, PIM does not rely on the use of lambda expressions or monads to sequence assignments. This permits the use of stronger axioms for reasoning about store-specific sequencing, and makes it possible to study the properties of the imperative features of the language completely independently from the functional ones.

PIM is an *equational logic*, rather than, e.g., a sequent calculus such as that

[1]In this paper, we do not actually use Cartwright and Felleisen's denotational model in our completeness result, preferring instead a final algebra model that makes a more direct link between a language's operational semantics and its logic of program equivalence.

[2]Selke's semantics uses an ad-hoc graph rewriting formalism, rather than term graph rewriting [Barendregt et al. 1987].

of Mason and Talcott [1989] or Selke [1989a, 1989b], the latter system of which, like PIM, was based on the work of Cartwright and Felleisen [1989]. While sequent calculi are natural for carrying out proofs of program equivalence (especially those performed by hand), equational logics have the useful property that every intermediate state of a proof is itself a program. Equational logics are also particularly amenable to mechanical implementation, using such techniques as term graph rewriting [Barendregt et al. 1987], Knuth-Bendix completion, and (equational) unification or narrowing. The latter two properties make it possible to use PIM not only to prove equivalences, but also to model the "standard" operational semantics of a language (using a terminating and confluent rewriting system on ground terms) or to serve as a "semantics of partial evaluation" (by augmenting the operational semantics by oriented instances of the full logic).

Yang, Horwitz, and Reps [1989] present an algorithm that uses structural properties of the PRG IR to determine equivalence among parts of two variants of the same program which are to be combined by a program integration tool. However, even for the restricted class of programs that can be modeled by PIM_t, not all valid equivalences among programs represented by PRGs can be proved by structural equivalence.

The logics of Hoare et al. [1987] (which PIM resembles most closely in many respects) and Boehm [1985] manipulate languages that are syntactically similar to "real" programming languages, in the sense that operations that affect the store (i.e., side-effects) are intermixed in the term structure with pure operations on values. By contrast, PIM contains an explicitly *threaded* store whose operations are distinguished syntactically from operations on values. While this separation of concerns renders PIM wholely unsuitable as a programming language, we believe that this characteristic makes it easier to represent languages in which expressions with and without side effects are intermixed in complicated ways (e.g., C). This also means that it is usually straightforward to extend PIM with new operations and axioms on base values, or change the "memory model" used to represent addresses, since we need have no concern about how these operations are interrelated. Finally, PIM differs from Hoare et al. [1987] and Boehm [1985] in allowing for computed addresses (which arise from pointers and arrays), and providing operations on stores (e.g., store concatenation), the latter of which, while not present in most real programming languages, are extremely useful for modeling various natural transformations on IRs.

For further details on the correspondence between PIM and traditional IR's, see [Field 1992]. For an example of a practical application of PIM, see [Field, Ramalingam, and Tip 1995], which describes a novel algorithm for program *slicing*.

3. How PIM Works

3.1 μC

Consider the program fragments P_1–P_5 depicted in Fig. 1. They are written in a C language subset that we will call μC, whose complete syntax is given in Fig. 24 of Appendix A. μC has standard C syntax and semantics, with the following exceptions:

```
/* int x,y,p;
   const int ?X, ?P; */

{  p = ?P;
   x = ?X;
   y = p;
   x = p;
   if (p)
     x = y;
}
```
P_1

```
/* int x,y,p; */


{  p = 0;
   x = 1;
   y = p;
   x = p;
   if (p)
     x = y;
}
```
P_2

```
/* int x,y,p;
   const int ?P; */

{  p = ?P;
   y = p;
   x = p;
   if (p)
     x = y;
}
```
P_3

```
/* int x,y,p; */


{  p = 0;
   y = p;
   x = p;
   if (p)
     x = y;
}
```
P_4

```
/* int x,y,p;
   const int ?P; */

{  p = ?P;
   y = ?P;
   x = ?P;
}
```
P_5

Figure 1. Some simple μC programs.

- —*Meta-variables*, such as `?P` or `?X`, are used to represent unknown values. Such a variable may be thought of as a simple form of program input (where each occurrence of a meta-variable represents the *same* input value) or as an (immutable) formal parameter of a function.
- —All data in μC are assumed to be integers or pointers (to integer variables or to other pointers).
- —It is assumed that no address arithmetic is used[3].

To make μC examples such as those in Fig. 1 easier to follow, we will add type "declarations" within comment delimiters for the variables and meta-variables used therein; however, these declarations are not part of the syntax of μC. We will use "`const`" in the declarations of meta-variables to emphasize that their values are immutable.

3.2 PIM Terms and Graphs

A directed *term graph* [Barendregt et al. 1987] form of the PIM representation of P_1, S_{P_1}, is depicted in Fig. 2. S_{P_1} is generated by a simple syntax-directed

[3]Although PIM admits arbitrary computation on addresses in general, the completeness results in this paper assume that addresses are not themselves "results" of programs.

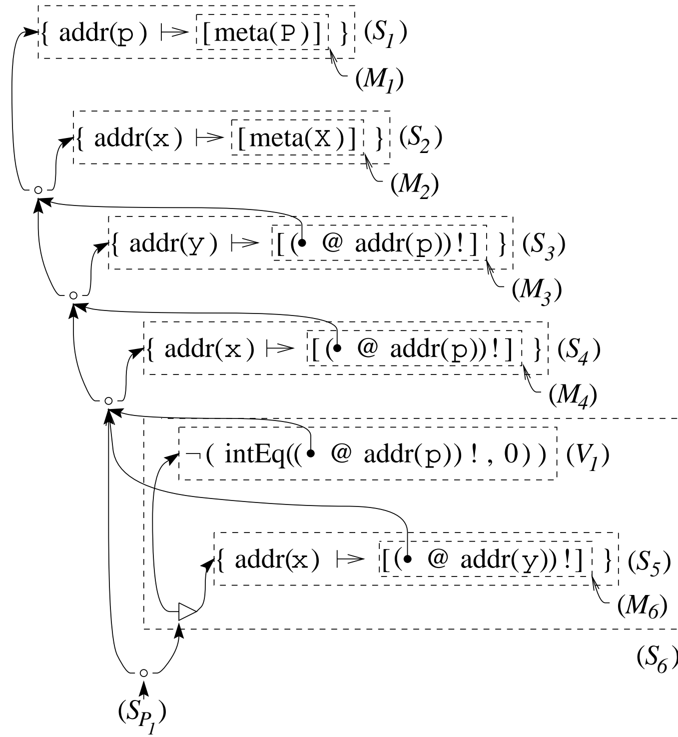

Figure 2: S_{P_1}: The PIM representation of program P_1. Null stores and subscripts on store composition operators are omitted for clarity.

translation, details of which are given in Appendix A. We will discuss S_{P_1} in detail in Section 3.4.

A term graph may be viewed as a conventional tree-structured term by traversing it from its root and replacing all shared subgraphs by separate copies of their term representations. Cycles are also admissible in term graphs, and correspond to infinite terms [Kennaway, Klop, Sleep, and de Vries 1994]. Such cycles arise naturally in conjunction with loops and recursive functions. Shared PIM subgraphs are constructed systematically as a consequence of the translation process. In addition, when term rewriting is extended in a natural way to term graphs [Barendregt et al. 1987], subgraph sharing can be created as a side-effect of the rewriting process.

The properties of the equational systems we consider in this paper are completely independent of whether a PIM term is represented by a tree or a graph. However, when used in implementations, graphs are preferred since the graph representation of a program is invariably much more compact than the tree representation (consider, e.g., the size of the tree form of graph S_{P_1} in Fig. 2). It is also frequently easier to discern the correspondence between related μC constructs and PIM structures when graphs are used in examples; this correspondence will be emphasized by depicting parent nodes in PIM terms *below* their children. The "upside down"

graph orientation also makes the similarity between PIM graphs and traditional IRs more apparent.

3.3 PIM_t: Core PIM

In this paper, we focus on the first-order core subsystem of PIM, denoted by PIM_t. The full version of PIM discussed in [Field 1992] and slightly revised in [Field, Ramalingam, and Tip 1995] augments PIM_t with lambda expressions, an induction rule, and certain additional higher-order *merge distribution* rules that in the more general form given in [Field, Ramalingam, and Tip 1995] propagate conditional "contexts" inside expressions computing base values or addresses. PIM's higher-order constructs allow loops and procedures to be modeled in a straightforward way; we will briefly review these applications in Section 5. In addition, [Field, Ramalingam, and Tip 1995] show how specialized instances of PIM's induction rule can be used to facilitate loop-related transformations required to compute various kinds of *program slices*.

Without the higher-order extensions, PIM_t is not Turing-complete. However, we believe that the constructs in PIM_t alone are sufficient to model the principal control- and data-flow aspects of loop- and function-free programs in Algol-class languages. As we shall see, these constructs have a non-trivial equational axiomatization. Understanding their properties is a prerequisite to studying the properties of higher-order variants.

The signature[4] of PIM_t terms is given in Fig. 3; we will describe the intended interpretation of each of the function symbols in the signature in Section 3.4. The sort structure of terms restricts the form of addresses and predicates in such a way that neither may be the result of an arbitrary PIM computation. Although our completeness result depends on this restriction, the equations in the complete equational system $\text{PIM}_t^=$ remain valid even when the term formation restrictions are dropped[5]. This means, e.g., that we can still use our system to reason about situations in which addresses or predicates are generated by arbitrary computations (or situations in which the values of such expressions are entirely unknown), even though there may be *additional* valid equations (arising as a consequence of the structure of those computations) that we will not necessarily be able to prove.

Fig. 4 depicts the equations[6] of the system PIM_t^0. PIM_t^0 is intended to function as an operational semantics for PIM_t, in the sense that when its equations are oriented from left to right, they form a rewriting system that is confluent on ground terms of sort $\mathcal{V}$, the sort of observable "base" values. PIM_t^0 also serves to define the initial algebra semantics for PIM_t.

PIM can be viewed as a parameterized data type with formal sorts $\mathcal{V}$ and $\mathcal{A}$.

[4]This signature differs slightly from the corresponding signature in [Field 1992]; the differences principally relate to a simplification in the structure of merge expressions.

[5]If address or predicate expressions may contain *nonterminating* computations, there are a number of semantic issues beyond the scope of this paper that must be addressed. In brief, we take the position (usually adopted implicitly by optimizing compilers) that equations remain valid as long as they equate terms that behave the same in the absence of nontermination.

[6]The gaps in the equation numbers used for PIM_t^0, as well as those occurring in other systems discussed in the sequel, are present to ensure compatibility with the equation numbers used in [Field 1992].

sorts

$\mathcal{S}$			(store structures)
$\mathcal{M}$			(merge structures)
$\mathcal{A}$			(addresses)
$\mathcal{B}$			(booleans)
$\mathcal{V}$			(base values)

functions

$\{\mathcal{A} \mapsto \mathcal{M}\}$	$\rightarrow$	$\mathcal{S}$	(store cell)
$\mathcal{B} \rhd_s \mathcal{S}$	$\rightarrow$	$\mathcal{S}$	(guarded store)
$\mathcal{S} \circ_s \mathcal{S}$	$\rightarrow$	$\mathcal{S}$	(store composition)
$\emptyset_s$	$\rightarrow$	$\mathcal{S}$	(null store)
$\mathcal{S} @ \mathcal{A}$	$\rightarrow$	$\mathcal{M}$	(store dereference)
$[\mathcal{V}]$	$\rightarrow$	$\mathcal{M}$	(merge cell)
$\mathcal{B} \rhd_m \mathcal{M}$	$\rightarrow$	$\mathcal{M}$	(guarded merge)
$\mathcal{M} \circ_m \mathcal{M}$	$\rightarrow$	$\mathcal{M}$	(merge composition)
$\emptyset_m$	$\rightarrow$	$\mathcal{M}$	(null merge)
$\alpha_1, \alpha_2, \ldots$	$\rightarrow$	$\mathcal{A}$	(address constants)
$\mathbf{T}, \mathbf{F}$	$\rightarrow$	$\mathcal{B}$	(boolean constants)
$\mathcal{A} \asymp \mathcal{A}$	$\rightarrow$	$\mathcal{B}$	(address comparison)
$\neg\mathcal{B}$	$\rightarrow$	$\mathcal{B}$	(boolean negation)
$\mathcal{B} \wedge \mathcal{B}$	$\rightarrow$	$\mathcal{B}$	(boolean conjunction)
$\mathcal{B} \vee \mathcal{B}$	$\rightarrow$	$\mathcal{B}$	(boolean disjunction)
$\mathcal{M}!$	$\rightarrow$	$\mathcal{V}$	(merge selection)
$c_1, c_2, \ldots$	$\rightarrow$	$\mathcal{V}$	(base value constants)
$?$	$\rightarrow$	$\mathcal{V}$	(unknown base value)

variables

$s, s_1, s_2, \ldots$	$\rightarrow$	$\mathcal{S}$
$m, m_1, m_2, \ldots$	$\rightarrow$	$\mathcal{M}$
$l, l_1, l_2, \ldots$	$\rightarrow$	$\mathcal{S}$
$l, l_1, l_2, \ldots$	$\rightarrow$	$\mathcal{M}$
$a, a_1, a_2, \ldots$	$\rightarrow$	$\mathcal{A}$
$p, p_1, p_2, \ldots$	$\rightarrow$	$\mathcal{B}$
$v, v_1, v_2, \ldots$	$\rightarrow$	$\mathcal{V}$

Figure 3: Signature of PIM_t terms. The notation was borrowed from the ASF+SDF specification language [van Deursen, Heering, and Klint 1996]. The first function entry $\{\mathcal{A} \mapsto \mathcal{M}\} \rightarrow \mathcal{S}$ can be read (i) as a regular function declaration StoreCell : $\mathcal{A} \times \mathcal{M} \rightarrow \mathcal{S}$, with some "abstract" function name StoreCell and sorts $\mathcal{A}$, $\mathcal{M}$, $\mathcal{S}$, or (ii) as a reverse BNF rule $\{\mathcal{A} \mapsto \mathcal{M}\}$ =:: $\mathcal{S}$ with nonterminals $\mathcal{A}$, $\mathcal{M}$, $\mathcal{S}$ and terminals $\{$, $\mapsto$, $\}$. Similarly for the other entries.

$$
\begin{array}{rcll}
\emptyset_\rho \circ_\rho l & = & l & \text{(L1)} \\
l \circ_\rho \emptyset_\rho & = & l & \text{(L2)} \\
l_1 \circ_\rho (l_2 \circ_\rho l_3) & = & (l_1 \circ_\rho l_2) \circ_\rho l_3 & \text{(L3)} \\
\mathbf{T} \rhd_\rho l & = & l & \text{(L5)} \\
\mathbf{F} \rhd_\rho l & = & \emptyset_\rho & \text{(L6)} \\[1ex]
\{a_1 \mapsto m\} \mathbin{@} a_2 & = & (a_1 \asymp a_2) \rhd_m m & \text{(S1)} \\
\{a \mapsto \emptyset_m\} & = & \emptyset_s & \text{(S2)} \\
\emptyset_s \mathbin{@} a & = & \emptyset_m & \text{(S3)} \\
(s_1 \circ_s s_2) \mathbin{@} a & = & (s_1 \mathbin{@} a) \circ_m (s_2 \mathbin{@} a) & \text{(S4)} \\[1ex]
(\alpha_i \asymp \alpha_i) & = & \mathbf{T} \quad (i \geq 1) & \text{(A1)} \\
(\alpha_i \asymp \alpha_j) & = & \mathbf{F} \quad (i \neq j) & \text{(A2)} \\[1ex]
(m \circ_m [v])! & = & v & \text{(M2)} \\
[v]! & = & v & \text{(M3)} \\
\emptyset_m! & = & ? & \text{(M4)} \\[1ex]
\neg\mathbf{T} & = & \mathbf{F} & \text{(B1)} \\
\neg\mathbf{F} & = & \mathbf{T} & \text{(B2)} \\
\mathbf{T} \wedge p & = & p & \text{(B3)} \\
\mathbf{F} \wedge p & = & \mathbf{F} & \text{(B4)} \\
\mathbf{T} \vee p & = & \mathbf{T} & \text{(B5)} \\
\mathbf{F} \vee p & = & p & \text{(B6)}
\end{array}
$$

Figure 4: Equations of PIM_t^0. The equations labeled (Ln) are generic to merge or store structures, i.e., in each case 'ρ' should be interpreted as one of either s or m. Equations (A1) and (A2) are schemes for an infinite set of equations.

These sorts are intended to be instantiated as appropriate to model the data manipulated by a given programming language. Fig. 5 depicts the signature of a small set of functions and auxiliary sorts used to actualize the parameter sorts of PIM to model the integer data manipulated by μC programs. From the point of view of the results presented in the sequel, these additional functions are simply treated as uninterpreted "inert" constructors. In practice, of course, sufficient additional equations would be added to axiomatize the properties of the additional language-specific datatypes (e.g., pairs, lists, records, or streams). In addition, the representation of addresses would have to be extended to allow for such constructs as heap allocated data (which require a function that generates new addresses[7]) and arrays (which require that the address of each array element be distinguished from the address of other elements and the address of the base of the array).

The PIM_t signature given in Fig. 3 as well as the equation systems we will present

[7]It is straightforward to design an "allocation function" that generates new address names simply by encoding some form of counter. However, the equational semantics that result are somewhat unsatisfactory, in that certain simple properties of heap-allocated addresses (e.g., inequality of addresses generated by two consecutive calls to the allocator) must be derived from the properties of the counter used in the allocation function. A more abstract and natural encoding of heap-allocated data remains an open problem.

			sorts
Id			(identifiers)
IntLiteral	$\subseteq$	$\mathcal{V}$	(integer literals; subsort of base values)
			functions
meta(Id)	$\to$	$\mathcal{V}$	(meta-variables constructed from identifiers)
intSum($\mathcal{V}$, $\mathcal{V}$)	$\to$	$\mathcal{V}$	(integer addition)
intDiff($\mathcal{V}$, $\mathcal{V}$)	$\to$	$\mathcal{V}$	(integer subtraction)
intEq($\mathcal{V}$, $\mathcal{V}$)	$\to$	$\mathcal{B}$	(integer equality)
addr(Id)	$\to$	$\mathcal{A}$	(address constants constructed from identifiers)

Figure 5. Signature of μC-Specific PIM Extensions.

in the sequel are sufficiently simple in structure to be implementable with minimal difficulty by many existing equational rewriting systems and theorem provers (e.g., ASF+SDF [van Deursen, Heering, and Klint 1996] and TIP [Fraus 1994]). Other than the possible need to convert infix operators to prefix form, the only other requirements that PIM_t imposes are the ability to encode an infinite set of addresses and to implement equations (A1) and (A2). This can easily be done using a finite number of auxiliary symbols and equations, e.g., by encoding addresses using strings over a fixed alphabet.

3.4 PIM's Parts

In this section, we outline the behavior of PIM's functions and the equations of PIM_t^0 using program P_1 and its PIM translation, S_{P_1}, depicted in Fig. 2.

3.4.1 Stores

The graph S_{P_1} is a PIM *store structure*[8], an abstract representation of memory. S_{P_1} is constructed from the sequential composition of substores corresponding to the statements comprising P_1. The operator '$\circ_s$' is used to concatenate two stores. The subgraphs reachable from the boxes labeled S_1–S_4 in S_{P_1} correspond to the four assignment statements in P_1.

The simplest form of store is a *cell* such as

$$S_1 \equiv \{\text{addr}(\texttt{p}) \mapsto [\text{meta}(\texttt{P})]\}$$

A store cell of this form associates an *address expression* (here addr(p)) with a *value* (here meta(P), the translation of the μC meta-variable `?P`). The operator '$[\cdot]$' encapsulates the value being stored in a *merge structure*; we will discuss such structures in greater detail below. Constant addresses such as addr(p) represent ordinary variables. More generally, address *expressions* may be used when addresses are computed, e.g., in pointer-valued expressions. $\emptyset_s$ is used to denote the null store. Equations (L1) and (L2) of PIM_t^0 indicate that null stores disappear when composed

[8] For clarity, Fig. 2 does not depict certain *null* stores created by the translation process; this elision will be irrelevant in the sequel.

with other stores. Equation (L3) indicates that the store composition operator is associative.

Stores may be guarded, i.e., executed conditionally. The subgraph labeled S_6 in Fig. 2 is such a store, and corresponds to the '`if`' statement in P_1. The guard expression denoted by V_1 corresponds to the `if`'s predicate expression. Consistent with standard C semantics, the guard V_1 tests whether the value of the variable `p` is nonzero. When guarded by the true predicate, **T**, a store structure evaluates to itself. If a store structure is guarded by the false predicate, **F**, it evaluates to the null store structure. These behaviors are axiomatized by equations (L5) and (L6).

3.4.2 Lazy Store Retrieval

A reference to the value of a program variable is modeled in PIM in the usual manner by retrieving the variable's value from the store using the variable's address as a lookup key. However, the details of the retrieval operation are somewhat unusual since PIM axiomatizes a variant of Cartwright and Felleisen's [1989] *lazy* store semantics.

In a traditional (strict) store semantics, the address to which an assignment is made as well as the value being assigned are evaluated *at the point of assignment*. Cartwright and Felleisen observed that it is possible to define an alternative semantics in which the evaluation of an assignment's address, its value, or both are *deferred* until a value in the store is retrieved by a reference to an address.

Cartwright and Felleisen's aim in defining the notion of a lazy store was to provide a (denotational) semantic foundation for program manipulations performed in building Program Dependence Graphs (PDGs) [Ferrante, Ottenstein, and Warren 1987]. However, they also observed that the termination behavior of lazy stores differs from that of strict stores: an assignment that stores a divergent value which is never subsequently retrieved does not cause nontermination in a lazy store semantics, whereas the program would diverge in a strict store semantics. The possible introduction of "gratuitous termination" under a lazy store semantics seems to be of little concern in practice, however, since it is similar to other commonly used dead code elimination steps that also have the potential to remove sources of divergence.

PIM's operational axiomatization of lazy stores decomposes store retrieval into two distinct steps: *dereferencing* (using the '@' operator) and *selection* (using the '!' operator). We see both operators in expressions of the form

$$(s \mathbin{@} a)!$$

in S_{P_1}, where each such expression corresponds to a reference to the value of some variable in P_1.

An expressions of the form $s \mathbin{@} a$ produces an ordered list of all the values that are stored in s at address a prior to dereferencing. This retrieval behavior is codified by equations (S1)–(S4), (A1), and (A2). The list of values produced by the dereferencing operation is deemed a *merge structure*, since values stored at the same address are merged together.

3.4.3 Merge Structures: Sets of Reaching Definitions

The merge structures created by store dereferencing are best thought of as representations of sets of definitions (i.e., assignments) that *reach* a particular use of

some address. For a given use of an address, the set of reaching definitions contains all those assigned values that could be retrieved at the point of use in some valid execution. Due to the existence of conditionals and computed addresses, the set of reaching definitions for a given use of an address cannot always be narrowed to a singleton in an open program (i.e., a program containing meta-variables). A merge structure orders the reaching definition set it represents by "least recent" assignment to "most recent." When applied to a *closed* merge structure m, the selection operator '!' yields the rightmost element in the list represented by m, i.e., the most recently assigned value.

The simplest nonempty form of merge structure is a *merge cell*. The boxes labeled M_1, M_2, M_3, M_4, and M_6 in Fig. 2 are all merge cells. As with store structures, nontrivial merge structures may be built by prepending guard expressions, or by composing merge substructures using the merge composition operator, '$\circ_m$'. $\emptyset_m$ denotes the null merge structure. Some of the characteristics of merge structures are shared by store structures, as indicated by the "polymorphic" equations (L1)–(L6). In the sequel, we will therefore often drop subscripts distinguishing related store and merge constructs when no confusion will arise. Nontrivial merge structures can be used to model various forms of conditional expressions; however, they arise most frequently as *results* of PIM simplifications of other type of expressions.

When the selection operator '!' is applied to a non-empty merge structure m, m must first be evaluated until it has the form

$$m' \circ_m [v]$$

i.e., one in which an *unguarded cell* is rightmost. At this point, the entire expression $m!$ evaluates to v. This behavior is axiomatized by equations (M2) and (M3). Equation (M4) states that attempting to apply the selection operator to a null merge structure yields the special error value '?'.

4. Reasoning with PIM Terms and Graphs

4.1 Example: Manipulating Lazy Stores

To get a feel for equational manipulations in PIM, consider the programs R_1 and R_2 depicted in Fig. 6. In both examples, we will concentrate on the PIM representation of the final reference to the variable `x`. In the case of R_1, the PIM representation of the value of `x` (produced by the translation in Appendix A) is given by the expression

$$((s_1 \circ_s s_2) \circ_s s_3) \ @ \ \mathrm{addr}(\mathtt{x})\,!$$

where

$$\begin{aligned} s_1 &\equiv \mathbf{F} \rhd_s \{\mathrm{addr}(\mathtt{x}) \mapsto [17]\} \\ s_2 &\equiv \mathbf{T} \rhd_s \{\mathrm{addr}(\mathtt{y}) \mapsto [18]\} \\ s_3 &\equiv \mathbf{T} \rhd_s \{\mathrm{addr}(\mathtt{x}) \mapsto [17]\} \end{aligned}$$

For clarity, we have eliminated several null stores introduced by the translation process, and simplified the representation of the integers 0 and 1 used in the conditional statements to their boolean PIM normal forms, '**F**' and '**T**', respectively.

It should be easy to see that PIM stores s_1, s_2, and s_3 represent the correspondingly labeled statements in R_1. The PIM representation of the final reference to `x`

```
/* int x,y; */                      /* int x,y;
                                       const int ?P, ?Q; */

{  if (0) x = 17;   /* s1 */        {  if (!?P) x = 17; /* t1 */
   if (1) y = 18;   /* s2 */           if (?Q) y = 18;  /* t2 */
   if (1) x = 17;   /* s3 */           if (?P) x = 17;  /* t3 */
   ... = x;                            ... = x;
}                                   }
```

R_1 | R_2

Figure 6. μC programs illustrating store and merge manipulations.

first dereferences the composite store using address addr(x), then uses the selection operator '!' as discussed above to extract the first reaching definition of x.

Using the equations of $\textsc{Pim}_t^0$ in Fig. 4, we can simplify the portion of R_1 corresponding to the reference to x as follows (the specific equations of $\textsc{Pim}_t^0$ used at each step are indicated at the right):

$$
\begin{aligned}
&((s_1 \circ_s s_2) \circ_s s_3) \;\; @ \;\; \mathrm{addr}(\mathtt{x})\,! \\
&\quad = (((s_1 \circ_s s_2) \;@\; \mathrm{addr}(\mathtt{x})) \circ_m (\mathbf{T} \rhd_s \{\mathrm{addr}(\mathtt{x}) \mapsto [17]\} \;@\; \mathrm{addr}(\mathtt{x})))\,! && \text{(S4)} \\
&\quad = (((s_1 \circ_s s_2) \;@\; \mathrm{addr}(\mathtt{x})) \circ_m (\{\mathrm{addr}(\mathtt{x}) \mapsto [17]\} \;@\; \mathrm{addr}(\mathtt{x})))\,! && \text{(L5)} \\
&\quad = (((s_1 \circ_s s_2) \;@\; \mathrm{addr}(\mathtt{x})) \circ_m ((\mathrm{addr}(\mathtt{x}) \asymp \mathrm{addr}(\mathtt{x})) \;\rhd_m [17])\,! && \text{(S1)} \\
&\quad = (((s_1 \circ_s s_2) \;@\; \mathrm{addr}(\mathtt{x})) \circ_m (\mathbf{T} \;\rhd_m [17])\,! && \text{(A1)} \\
&\quad = (((s_1 \circ_s s_2) \;@\; \mathrm{addr}(\mathtt{x})) \circ_m [17])\,! && \text{(L5)} \\
&\quad = 17 && \text{(M2)}
\end{aligned}
$$

Note how the lazy store semantics implemented by '@' and '!' retrieves the value of the last assignment to x without simplifying any of the first two assignment statements at all.

While the notion of a lazy store semantics might seem novel, but otherwise inconsequential in the case of R_1, R_2 (Fig. 6) illustrates its considerable utility in simplifying *open* programs. The $\textsc{Pim}$ representation of the final value of x in R_2 is given by the expression

$$((t_1 \circ_s t_2) \circ_s t_3) \;\; @ \;\; \mathrm{addr}(\mathtt{x})\,!$$

where

$$
\begin{aligned}
t_1 &\equiv \neg p \rhd_s \{\mathrm{addr}(\mathtt{x}) \mapsto [17]\} \\
t_2 &\equiv q \rhd_s \{\mathrm{addr}(\mathtt{y}) \mapsto [18]\} \\
t_3 &\equiv p \rhd_s \{\mathrm{addr}(\mathtt{x}) \mapsto [17]\}
\end{aligned}
$$

and

$$
\begin{aligned}
p &\equiv \neg(\mathrm{intEq}(\mathrm{meta}(\mathtt{P}) \,,\, 0)) \\
q &\equiv \neg(\mathrm{intEq}(\mathrm{meta}(\mathtt{Q}) \,,\, 0))
\end{aligned}
$$

As with R_1, we can use the rules of PIM to simplify the initial expression representing the final value of x. However, since the PIM term representing R_2 contains free variables (representing the values of meta-variables ?P and ?Q), the simplification process is somewhat more involved, requiring additional rules not needed for simplifying closed programs. The additional rules are found in Fig. 12, and are part of the ω-complete system $\text{PIM}_t^=$ we derive in Section 9.

The simplification of the PIM term representing R_2 proceeds as follows:

$$
\begin{aligned}
&((t_1 \circ_s t_2) \circ_s t_3) \ @ \ \text{addr}(\mathtt{x})\,! \\
&= \begin{pmatrix} (& (\{\text{addr}(\mathtt{x}) \mapsto (\neg p \rhd_m [17])\}) \circ_s \\ & (\{\text{addr}(\mathtt{y}) \mapsto (q \rhd_m [18])\})\ ) \circ_s \\ & (\{\text{addr}(\mathtt{x}) \mapsto (p \rhd_m [17])\}) \end{pmatrix} @\, \text{addr}(\mathtt{x})\,! && \text{(S5)} \\
&= \begin{pmatrix} (& ((\text{addr}(\mathtt{x}) \asymp \text{addr}(\mathtt{x})) \rhd_m (\neg p \rhd_m [17])) \circ_m \\ & ((\text{addr}(\mathtt{y}) \asymp \text{addr}(\mathtt{x})) \rhd_m (q \rhd_m [18]))\ ) \circ_m \\ & ((\text{addr}(\mathtt{x}) \asymp \text{addr}(\mathtt{x})) \rhd_m (p \rhd_m [17])) \end{pmatrix} ! && \text{(S1)} \\
&= \begin{pmatrix} (& (\mathbf{T} \rhd_m (\neg p \rhd_m [17])) \circ_m \\ & (\mathbf{F} \rhd_m (q \rhd_m [18]))\ ) \circ_m \\ & (\mathbf{T} \rhd_m (p \rhd_m [17])) \end{pmatrix} ! && \text{(A1), (A2)} \\
&= \begin{pmatrix} (& (\neg p \rhd_m [17]) \circ_m \\ & \emptyset_m\ ) \quad \circ_m \\ & (p \rhd_m [17]) \end{pmatrix} ! && \text{(L5), (L6)} \\
&= (\ (\neg p \rhd_m [17]) \circ_m (p \rhd_m [17])\ )\,! && \text{(L2)} \\
&= ((\neg p \vee p) \rhd_m [17])\,! && \text{(L11)} \\
&= (\mathbf{T} \rhd_m [17])\,! && \text{(B13), (B15)} \\
&= [17]\,! && \text{(L5)} \\
&= 17 && \text{(M3)}
\end{aligned}
$$

We see in the step labeled (S5) that the predicates are "pushed inside" the store cells. This enables the store to be simplified in subsequent steps without requiring that the predicates themselves be simplified. In the next four steps (using (S1), (A1), (A2), (L5), (L6), and (L2)), the definitions reaching the final use of x are effectively "gathered together." We can observe from the term in the equation labeled (L2) that two assignments of 17 reach the use of x, and that the two assignments are evaluated under different conditions. However, since the predicates implementing the conditions are logically complementary, we can conclude that 17 is *always* assigned to x. This fact is inferred in the next three steps using rules (L11), (B13), (B15), and (L5). These rules effectively convert the set of two reaching definitions into a single composite reaching definition. This single reaching definition is finally "selected" in the last step using rule (M3).

The collective effect of the rules implementing PIM's lazy store semantics is to enable addresses and predicates (which guard various substores) to be manipulated "orthogonally." This behavior is vital for enabling the final value of x to be simplified to a constant without requiring that the predicates themselves be simplified.

Returning to program P_1 (Fig. 1), we can reason about its PIM representation

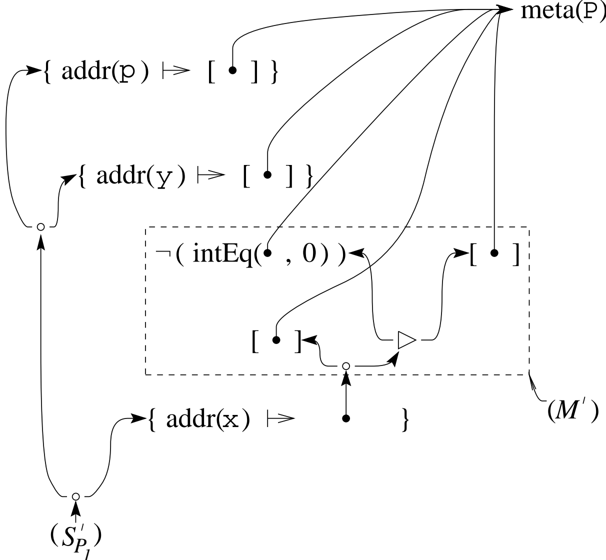

Figure 7. S'_{P_1}: A simplified form of S_{P_1}.

S_{P_1} (Fig. 2) in a manner similar to that of R_2. S_{P_1} can be simplified to the store structure S'_{P_1} depicted in Fig. 7. The merge structure denoted by M' in S'_{P_1} represents all the assignments made to variable `x` in P_1, and can be read roughly as "if the variable to which this expression is bound is ever used, the resulting value will be that of the meta-variable `?P` if `?P` is nonzero (i.e., 'true' in C semantics); otherwise the resulting value will be `?P`." S'_{P_1} is very similar to several compiler IRs; we will revisit this point in more detail in Section 10.1. It should be easy to see that M' can be further simplified to the expression [meta(`P`)], although the resulting graph does not correspond as closely to a compiler IR as does S'_{P_1}.

It is important to note that PIM's axiomatization of lazy stores is independent of whether a lazy (i.e., "outermost") *strategy* is used to rewrite PIM expressions. Indeed, such a strategy was not used in the simplification of R_2 above. It is, however, desirable to use a lazy strategy and graph reduction techniques in rewriting implementations of PIM to ensure that unnecessary rewriting steps are avoided.

4.2 The Relative Power of PIM Subsystems

In order for PIM to make good on its claim to being a "program logic," there must be some semantics about which the logic can reason. The equations of PIM_t^0 in Fig. 4 serve this function for any closed expression of sort $\mathcal{V}$. We can generate an interpreter from these equations simply by orienting the equations in Fig. 4 from left to right, then applying them until a normal form is reached. (It is easily seen that the system is terminating; i.e., *noetherian* and yields unique normal forms on closed terms).

An interpreter generated from PIM_t^0 could have been used to mechanically simplify the expression representing the final value of `x` in program R_1 of Fig. 6 to 17.

For another example, consider program P_2 in Fig. 1. Its PIM representation, S_{P_2}, is the same as S_{P_1}, except that `?P` and `?X` are replaced with 0 and 1, respectively. The value of the variable `x` in the final store produced by evaluation of S_{P_2}, i.e., the final value of `x` after executing P_2 is represented by the expression

$$(S_{P_2} \ @ \ \text{addr}(\mathtt{x}))!$$

and evaluates to the constant 0. The interpreter generated by PIM_t^0 can be used to evaluate the final value of any of P_2's variables in a similar manner.

Consider now the program P_4 of Fig. 1. Although it should be clear that P_4 behaves the same as P_2, the equations of PIM_t^0 are insufficient to equate the PIM translations of the two programs. We will require a more powerful system than PIM_t^0 to axiomatize a *final algebra* semantics, in which all behaviorally equivalent closed terms (such as those representing P_2 and P_4) are equated. PIM_t^+, the equational axiomatization of PIM_t^0's final algebra semantics, will be the subject of Section 8.

Finally, consider program P_5 of Fig. 1. Although it is behaviorally equivalent to both P_1 and P_3, one cannot deduce this fact using PIM_t^+ alone. Intuitively, this is due to the fact that P_1, P_3, and P_5 are all *open* programs. To equate these terms, as well as to prove all other valid equations on open terms, we will need the ω-*complete* system $\text{PIM}_t^=$, which will be developed in Section 9.

In Section 10, we will present several confluent and terminating subsystems of $\text{PIM}_t^=$. In addition to serving as partial decision procedures for equivalence, these systems and variants thereof can alo be used to yield normal forms (i.e., terms or graphs to which no further rewriting rules are applicable) that function in a manner similar to that of a traditional compiler IR, and to implement partial evaluation for optimization purposes.

5. Higher-Order Aspects of PIM

A complete exposition of the higher-order aspects of PIM is outside the scope of this paper. However, to see how PIM_t can be embedded in a higher-order framework, consider the μC program L depicted in Fig. 8.

L contains a simple loop and an initialization. The general scheme for μC `while` loop translation is given by rule (Stmt_6) in Fig. 25 of Appendix A. This scheme embeds a PIM_t expression representing the loop body in a recursive lambda expression. Recursion is expressed formally using a **Y** combinator[9]. In practice, it is advantageous to substitute a self-referential looping graph representation for an explicit **Y** combinator [Peyton Jones 1987]. Note that (Stmt_6) must account for the possibility that the loop predicate has side-effects; this results in a representation that is somewhat more complicated than it would be otherwise.

Fig. 9 depicts the PIM graph S_L that results from translating program L. The store labeled S_1 represents the loop itself, and is modeled by a lambda expression

[9]For procedures, **Y** has type $((\mathcal{S} \to \mathcal{S}) \to (\mathcal{S} \to \mathcal{S})) \to (\mathcal{S} \to \mathcal{S})$, and $\mathbf{Y}f = f(\mathbf{Y}f)$ for any lambda expression $f : (\mathcal{S} \to \mathcal{S}) \to (\mathcal{S} \to \mathcal{S})$.

```
/* int i; */

{  i = 10;
   while (i) i -= 1;
}
```

L

Figure 8. A simple program containing a loop.

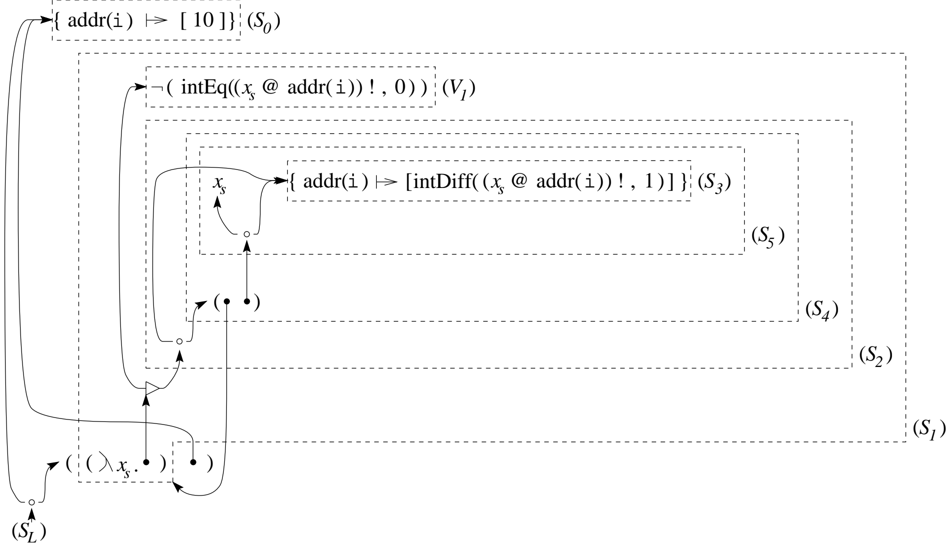

Figure 9. S_L: The PIM representation of program L. Null stores are omitted.

that takes an incoming store x_S and produces a store S_2 representing all the store updates that occur during loop iteration. S_2 is guarded by the loop's termination predicate, V_1. The predicate has no side-effects; the translation rule (Stmt$_6$) accounts for this fact by generating a null substore that is omitted in Fig. 9 for clarity. S_2 is the composition of a store S_3 representing the update of `i` in the current iteration, and a store S_4 representing updates in all subsequent iterations. S_4 contains a recursive reference to the loop's lambda expression, which is applied to a store S_5 representing the incoming store for the next iteration. S_5 thus consists of the current incoming store x_S composed with S_3, the store update produced by the current iteration. Loop iteration is initiated the first time around by applying S_1 to S_0, a store representing the initialization of program variable `i`.

Procedures can be represented in a manner similar to loops. As with loops, local storage within a procedure invocation can be represented simply by introducing a store expression representing the side-effects of the procedure within the lambda expression representing the procedure.

In order to accommodate various common loop simplifications, some form of inductive reasoning is required. Field, Ramalingam, and Tip [1995] provides several induction rules that implement various classes of useful loop transformations. These rules are derived from a more general induction rule given in [Field 1992], which in turn was inspired by work of Wand and Wang [1990].

While we have found that augmenting lambda terms with PIM_t terms allows a natural operational characterization of a variety of higher-order program constructs, more work remains to be done to study the formal properties of higher-order variants of PIM. In Section 11, we outline some possible future directions.

6. Partial Evaluation and ω-Completeness

Since many valid program transformations do not result from the application of evaluation rules alone, an operational semantics does not form an adequate basis for program optimization and transformation. For instance, consider the equation

$$\textbf{if } (p) \textbf{ then } e \textbf{ else } e = e.$$

Some version of this equation is valid in most programming languages (at least if we assume p terminates), yet transforming an instance of the left hand side in a program to the right hand side cannot usually be justified simply by applying an evaluation rule.

The open equations (equations containing variables, such as the one above) valid in a language are not in general equationally derivable from a specification of the language's semantics, but require stronger rules of inference (such as structural induction) for their proofs. It is possible, however, to trade rules of inference for equations. That is, by adding finitely many equations (possibly involving auxiliary sorts and functions) to the specification, the full equational theory of the language may be brought within the scope of equational reasoning [Bergstra and Heering 1994]. Such an enriched specification is called *ω-complete.* Strictly speaking ω-completeness can be achieved only if the equational theory in question is recursively enumerable, but this is to be expected and of no concern for PIM_t, although it will probably become a problem for the full higher-order version of PIM.

Furthermore, equational reasoning (of which term rewriting is a special case) applies equally to closed and open terms. The latter corresponds to transformation of *incomplete* programs with the variables in the input term representing missing data or code. Evaluation of incomplete programs is usually called *partial evaluation* [Ershov 1982; Jones, Gomard, and Sestoft 1993]. In the present context, we are not concerned with binding-time analysis or self-application, but, following [Heering 1986], simply assert that

partial evaluation =	*rewriting of open terms with respect to the intended semantics.*

While partial evaluation has often been advocated as a means for optimizing programs [Ershov 1982; Ershov and Ostrovski 1987; Chambers and Ungar 1989; Berlin and Weise 1990; Nirkhe and Pugh 1992], we know of few results that precisely characterize the transformational capabilities that a particular partial evaluator implements. In our setting, finding an ω-complete specification amounts to showing

that one's partial evaluator has all the rules it needs at its disposal; it will thus be our goal in the sequel to find an ω-complete axiomatization for PIM_t.

7. Algebraic Preliminaries

In this section we give a brief summary of some basic facts of algebraic specification theory which are essential to an understanding of what follows. Good surveys are Meinke and Tucker's survey [1992], Meseguer and Goguen's survey [1985], and Wirsing's survey [1990]. We assume some familiarity on the part of the reader with equational logic and initial algebra semantics.

7.1 ω-Completeness

DEFINITION 1. *An algebraic specification* $\mathbf{S} = (\Sigma, E)$ *with non-void many-sorted signature* Σ*, set of equations* E*, and initial algebra* $I(\mathbf{S})$ *is* ω*-complete if* $I(\mathbf{S}) \models t_1 = t_2$ *iff* $E \vdash t_1 = t_2$ *for open* Σ*-equations* $t_1 = t_2$*.*

According to the definition, all equations valid in the initial algebra of an ω-complete specification may be deduced using only equational reasoning. No structural induction is needed. Trading structural induction for equational reasoning from an enriched equational base has two potential advantages:

—An existing rewrite implementation of equational logic can be used, although an A-, AC-, or some other E-rewriting capability may be needed depending on the specification involved.

—Rewriting may have a sense of direction lacking in structural induction. It may perform useful simplifications of terms without having been given an explicit inductive proof goal beforehand.

One way of proving ω-completeness of a specification is to show that every congruence class modulo E has a representative in canonical form (not necessarily a normal form produced by a rewrite system) such that two distinct canonical forms t_1 and t_2 can always be instantiated to ground terms $\sigma(t_1)$ and $\sigma(t_2)$ that cannot be proved equal from E. Another way is to show by induction on the length (in some sense) of equations that equations valid in $I(\mathbf{S})$ are provable from E. We use both methods in this paper. Additional information about proof techniques for ω-completeness as well as examples of their application can be found in [Heering 1986; Lazrek, Lescanne, and Thiel 1990; Bergstra and Heering 1994].

7.2 Final Algebra Semantics

Final algebra semantics does not make a distinction between elements that have the same observable behavior. We need the following definitions:

DEFINITION 2. *Let* Σ *be a many-sorted signature and* $\mathcal{S}, \mathcal{T} \in \text{sorts}(\Sigma)$*. A* Σ*-context of type* $\mathcal{S} \to \mathcal{T}$ *is an open term of sort* $\mathcal{T}$ *containing a single occurrence of a variable* $\Box$ *of sort* $\mathcal{S}$ *and no other variables.*

The instantiation $C(\Box := t)$ of a Σ-context C of type $\mathcal{S} \to \mathcal{T}$ with a Σ-term t of sort $\mathcal{S}$ will be abbreviated to $C(t)$. If t is a ground term, $C(t)$ is a ground term as well. If t is a Σ-context of type $\mathcal{S}' \to \mathcal{S}$, $C(t)$ is a Σ-context of type $\mathcal{S}' \to \mathcal{T}$.

DEFINITION 3. *Let* $\mathbf{S} = (\Sigma, E)$ *be an algebraic specification with non-void many-sorted signature* Σ*, set of equations* E*, and initial algebra* $I(\mathbf{S})$*. Let* $O \subseteq \mathrm{sorts}(\Sigma)$*. The final algebra* $F_O(\mathbf{S})$ *is the quotient of* $I(\mathbf{S})$ *by the congruence* $\equiv_O$ *defined as follows:*

(i) t_1, t_2 *ground terms of sort* $\mathcal{S} \in O$*:*
$t_1 \equiv_O t_2$ *iff* $I(\mathbf{S}) \models t_1 = t_2$.

(ii) t_1, t_2 *ground terms of sort* $\mathcal{S} \notin O$*:*
$t_1 \equiv_O t_2$ *iff* $I(\mathbf{S}) \models C(t_1) = C(t_2)$ *for all contexts* C *of type* $\mathcal{S} \to \mathcal{T}$ *with* $\mathcal{T} \in O$.

Item (ii) says that terms of nonobservable sorts (sorts not in O) that have the same behavior with respect to the observable sorts (sorts in O) correspond to the same element of $F_O(\mathbf{S})$. It is easy to check that $\equiv_O$ is a congruence.

Definition 3 corresponds to the case $M = I(\mathbf{S})$ of $N(M)$ as defined in [Meseguer and Goguen 1985, p. 488]. Observable sorts are called *visible* in [Meseguer and Goguen 1985] and *primitive* in [Wirsing 1990, Section 5.4].

7.3 Steps Towards a Completeness Result

Our completeness result will require two basic technical steps:

(A) Finding an initial algebra specification of the final model $F_{\mathcal{V}}(\mathrm{PIM}_t^0)$. $F_{\mathcal{V}}(\mathrm{PIM}_t^0)$ is the quotient of the initial algebra $I(\mathrm{PIM}_t^0)$ by behavioral equivalence with respect to the observable sort $\mathcal{V}$ of base values. We add an equational definition of the behavioral equivalence to PIM_t^0, resulting in an initial algebra specification of $F_{\mathcal{V}}(\mathrm{PIM}_t^0)$.

(B) Making the specification obtained in step (A) ω-complete to improve its ability to cope with program transformation and partial evaluation.

We illustrate these steps for a simple data type before attacking PIM_t^0 itself.

7.4 A Simple Example

We perform steps (A) and (B) (Section 7.3) for the specification of a stack data type shown in Figure 10.

Step (A)

Consider the final model $F_{\mathcal{V}}(\mathrm{STACK})$ in the sense of Definition 3 with $O = \{\mathcal{V}\}$. We give an initial algebra specification of it [Heering 1985; Bergstra and Tucker 1995]. The normalized contexts of type $\mathcal{S} \to \mathcal{V}$ are

$$C_n = top(pop^{n-1}(\Box)). \quad (n \geq 1)$$

The constant a_1 is special in view of (St1). By (St1)–(St4)

$$C_n(push(a_1, \emptyset)) = C_n(\emptyset)$$

for all $n \geq 1$. This means $push(a_1, \emptyset)$ and $\emptyset$ have the same $\mathcal{V}$-observable behavior, and the equation

$$push(a_1, \emptyset) = \emptyset \qquad (1)$$

sorts	$\mathcal{V}$			(base values)
	$\mathcal{S}$			(stacks)
functions	$a_1, \ldots, a_N$	$\rightarrow$	$\mathcal{V}$	(base values, $N > 1$)
	$\emptyset$	$\rightarrow$	$\mathcal{S}$	(empty stack)
	$push(\mathcal{V}, \mathcal{S})$	$\rightarrow$	$\mathcal{S}$	(stack constructor)
	$top(\mathcal{S})$	$\rightarrow$	$\mathcal{V}$	(get top element)
	$pop(\mathcal{S})$	$\rightarrow$	$\mathcal{S}$	(delete top element)
variables	v	$\rightarrow$	$\mathcal{V}$	
	s	$\rightarrow$	$\mathcal{S}$	
equations	$top(\emptyset)$	$=$	a_1	(St1)
	$top(push(v, s))$	$=$	v	(St2)
	$pop(\emptyset)$	$=$	$\emptyset$	(St3)
	$pop(push(v, s))$	$=$	s	(St4)

Figure 10. STACK.

which is not valid in $I(\text{STACK})$, holds in $F_{\mathcal{V}}(\text{STACK})$. The rewrite system obtained by interpreting the equations of $\text{STACK}^+ = \text{STACK} + (1)$ as left-to-right rewrite rules is easily seen to be confluent and terminating.[10] The corresponding ground normal forms of sort $\mathcal{S}$ are

$$push(a_{i_1}, push(a_{i_2}, \ldots, push(a_{i_p}, \emptyset) \ldots)) \quad (p \geq 0, i_p \neq 1) \tag{2}$$

Equation (1) happens to be the only additional equation we need, and STACK^+ is the initial algebra specification we are looking for.

PROOF. It is sufficient to check that two distinct terms in normal form (2) are observationally distinct. Let

$$t_1 = push(a_{i_1}, \ldots, push(a_{i_p}, \emptyset) \ldots)$$
$$t_2 = push(a_{j_1}, \ldots, push(a_{j_q}, \emptyset) \ldots)$$

be two distinct normal forms, i.e., $p \neq q$ or $a_{i_k} \neq a_{j_k}$ for some $k \geq 1$. In the first case ($p > q$ say), t_1 and t_2 are distinguished by the context $C_p = top(pop^{p-1}(\Box))$ since

$$C_p(t_1) = a_{i_p} \neq a_1 = C_p(t_2).$$

In the second case they are distinguished by C_k since

$$C_k(t_1) = a_{i_k} \neq a_{j_k} = C_k(t_2).$$

$\Box$

[10] Such rewrite systems are usually called *complete* or *canonical*. To avoid undue overloading of both adjectives in this paper we prefer the more cumbersome terminology.

$$m \circ_m [v] = [v] \qquad \text{(M2}'\text{)}$$

$$\{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\} = (a_1 \asymp a_2) \rhd_s \{a_1 \mapsto (m_1 \circ_m m_2)\} \circ_s \neg(a_1 \asymp a_2) \rhd_s (\{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\}) \qquad \text{(S8)}$$

Figure 11. Additional Equations of PIM_t^+.

Step (B)

We give an ω-complete enrichment $\text{STACK}^=$ of STACK^+. The equation

$$push(top(s), pop(s)) = s \qquad (3)$$

is easily seen to be valid in $I(\text{STACK}^+)$ by verifying it for terms in normal form (2). It is not equationally derivable from STACK^+ since the corresponding rewrite system is not applicable. Again, it happens to be the only additional equation we need. We show that $\text{STACK}^= = \text{STACK}^+ + (3)$ is ω-complete.

PROOF. Normal forms of sort $\mathcal{V}$ are (i) constants $a_1, \ldots, a_N$ $(N > 1)$; (ii) variables $v, \ldots$; and (iii) terms $C_n(t) = top(pop^{n-1}(t))$ with t a variable of sort $\mathcal{S}$. We have to check all combinations of normal forms. These are easily seen to be different in $I(\text{STACK}^=)$. For instance, the normal forms $C_1(s)$ and $C_2(s)$ are distinguished by the substitution $s = push(a_2, \emptyset)$. Note the importance of $N > 1$.

Normal forms of sort $\mathcal{S}$ are (i) ground normal forms (2); (ii) variables $s, \ldots$; (iii) terms $pop^n(t)$ with t a variable of sort $\mathcal{S}$ and $n \geq 1$; (iv) terms $push(a_i, t)$ with t a normal form of sort $\mathcal{S}$ containing at least one variable; (v) terms $push(t_1, t_2)$ with t_1 a variable of sort $\mathcal{V}$ and t_2 a normal form of sort $\mathcal{S}$; and (vi) terms $push(C_n(t_1), t_2)$ with t_1 a variable of sort $\mathcal{S}$, and t_2 a normal form of sort $\mathcal{S} \neq pop^n(t_1)$ in view of (3).

As before, we have to check all combinations of normal forms. The only nontrivial cases are (iv)–(vi). Let the length of an equation be the number of nonlogical symbols in it. For instance, (3) has length 6. We proceed by induction on the length of equations. (a) The equations of length ≤ 6 valid in $I(\text{STACK}^+)$ are provable from (St1)–(St4), (1), (3). (b) Assume the valid equations of length $< n$ are provable. Let $push(t_1, t_2) = t_3$ be a valid equation of length n. Then $top(t_3) = t_1$ and $pop(t_3) = t_2$ are valid equations of length $< n$, and hence provable by assumption. Hence, $push(t_1, t_2) = t_3$ itself is provable since $push(t_1, t_2) = push(top(t_3), pop(t_3)) = t_3$ by (3). □

8. Step (A)—The Final Algebra

We give an initial algebra specification PIM_t^+ of the final model $F_{\mathcal{V}}(\text{PIM}_t^0)$. PIM_t^0 is shown in Figures 3 and 4. The additional equations of PIM_t^+ are shown in Figure 11.[11]

PROPOSITION 1. $F_{\mathcal{V}}(\text{PIM}_t^0) \models$ (M2′), (S8).

[11] Equation (M2′) in Figure 11 is a special case of (M6) in the preliminary version of $\text{PIM}_t^=$ given in [Field 1992, Figure 6]. (S8) subsumes (S6) and (S7) in the preliminary version.

PROOF. We prove (M2′). The proof of (S8) is similar.
The normalized contexts of type $\mathcal{M} \to \mathcal{V}$ are

$$C_{k,n} = ([c_{i_1}] \circ_m \cdots \circ_m [c_{i_{k-1}}] \circ_m \Box \circ_m [c_{i_{k+1}}] \circ_m \cdots \circ_m [c_{i_n}])!$$

$(1 \le k \le n)$. By (M2)

$$\begin{aligned} C_{k,n}(m \circ_m [v]) &= c_{i_n} = C_{k,n}([v]) \quad (k < n) \\ C_{n,n}(m \circ_m [v]) &= v = C_{n,n}([v]). \end{aligned}$$

□

(M2) is rendered superfluous by (M2′). Let $\text{PIM}_t^+ = \text{PIM}_t^0 - (\text{M2}) + (\text{M2}') + (\text{S8})$. We have

PROPOSITION 2. $I(\text{PIM}_t^+) = F_{\mathcal{V}}(\text{PIM}_t^0)$.

PROOF. We show that two distinct ground normal forms are observationally distinct. (i) Ground normal forms of sort $\mathcal{M}$ are

$$\emptyset_m,\ [?],\ [c_i] \quad (i \ge 1). \tag{4}$$

$\emptyset_m$ and $[?]$ are distinguished by the context $([c_1] \circ_m \Box)!$, the others by $\Box!$.
(ii) Ground normal forms of sort $\mathcal{S}$ are

$$\emptyset_s,\ \{\alpha_{i_1} \mapsto M_1\} \circ_s \cdots \circ_s \{\alpha_{i_n} \mapsto M_n\} \quad \begin{array}{l}(n \ge 1,\ i_1 < \cdots < i_n, \\ M_j \text{ in normal form (4)}, \\ M_j \ne \emptyset_m \text{ in view of (S2))}.\end{array} \tag{5}$$

Two distinct normal forms of sort $\mathcal{S}$ can be distinguished with respect to $\mathcal{M}$ by a suitable store dereference of the form $\Box\ @\ \alpha_k$ for some k. Hence, they can be distinguished with respect to $\mathcal{V}$ according to (i).
(iii) Sorts $\mathcal{A}$ and $\mathcal{B}$ are not affected. Any identification of elements of these sorts would immediately lead to collapse of the base values. □

We note that repeating the above step for PIM_t^+ does not yield further equations since $F_{\mathcal{V}}(\text{PIM}_t^+) = I(\text{PIM}_t^+)$.

9. Step (B)—ω-Complete Enrichment

We give an ω-complete enrichment $\text{PIM}_t^=$ of PIM_t^+. The additional equations of $\text{PIM}_t^=$ are shown in Figure 12. As before, ρ in equations (Ln) is assumed to be one of m or s. The reader will have no difficulty verifying the validity of the additional equations of $\text{PIM}_t^=$ in the initial algebra $I(\text{PIM}_t^+)$ by structural induction. Somewhat unexpectedly, their automatic inductive verification succeeded only with considerable difficulty [Naidich and Dinesh 1996].

The ω-completeness proof uses both proof methods mentioned in Section 7.1. It basically proceeds by considering increasingly complex open terms and their canonical forms. The latter are determined up to some explicitly given set of equations and are considered distinct only if they are not equal modulo these equations. The fact that two distinct canonical forms can be instantiated to ground terms that cannot be proved equal from $\text{PIM}_t^=$ is not explicitly shown in each case, but has been verified and the reader will have no difficulty repeating this.

We successively consider the following terms:

—boolean terms without $\asymp$;
—boolean terms with $\asymp$;
—open merge structures with $\asymp$ but without @ or !;
—unrestricted open merge structures;
—open terms of sort $\mathcal{V}$;
—open store structures without @ or $\asymp$ and without variables of sort $\mathcal{S}$;
—open store structures with @ and $\asymp$ and with variables of sort $\mathcal{S}$, but without variables of sort $\mathcal{A}$;
—unrestricted open store structures.

In two cases (boolean terms with $\asymp$ and unrestricted open store structures) the proof is not based on canonical forms, but proceeds by induction on the number of different address variables in an equation (its "length"). The same method (with a different definition of length) is used in Section 7.4 to show the ω-completeness of $\text{STACK}^=$. In the case of boolean terms with $\asymp$ we also obtained a canonical form (included below), but we failed to obtain one for unrestricted open store structures. It would certainly be preferable to have one, but the ω-completeness proof does not depend on it. The proof would only get a more constructive character.

In the following we concentrate on the various canonical forms. The (rather lengthy) proofs that they can actually be reached by equational reasoning from $\text{PIM}_t^=$ as well as the proofs of the two inductive cases are available in Appendix B.

Boolean terms without $\asymp$

The only booleans are **T** and **F**. Suitable canonical forms are the well-known disjunctive normal forms without nonessential variables (variables whose value does not matter). See, e.g., [Bergstra and Heering 1994, Theorem 3.1].

Boolean terms with $\asymp$

These require (A3)–(A6) in addition to (A1–2). (A5) and (A6) are substitution laws. (S9) and (S10) are similar laws for guarded store and merge structures which will be needed later on. The transitivity of $\asymp$ is given by the equation

$$(a_1 \asymp a_2) \wedge (a_2 \asymp a_3) \wedge \neg(a_1 \asymp a_3) = \mathbf{F},$$

which is an immediate consequence of (A5) or (A6) in conjunction with (B11). Note that the number of address constants α_i is infinite. Otherwise an equation $\bigvee_{i=1}^{K}(a \asymp \alpha_i) = \mathbf{T}$ would have been needed.

A suitable canonical form is the disjunctive normal form without nonessential variables mentioned before with the additional condition that the corresponding multiset of address constants and variables is minimal with respect to the multiset extension of the strict partial ordering

$$\cdots \succ a_2 \succ a_1 \succ \alpha_i \quad (i \geq 1). \tag{6}$$

A multiset gets smaller in the extended ordering by replacing an element in it by arbitrarily many (possibly 0) elements which are less in the original ordering [Klop 1992, p. 38]. The canonical form is determined up to symmetry of $\asymp$ and up to associativity and commutativity of $\vee$ and $\wedge$ as before.

$$
\begin{aligned}
p \rhd_\rho \emptyset_\rho &= \emptyset_\rho && \text{(L4)}\\
p \rhd_\rho (l_1 \circ_\rho l_2) &= (p \rhd_\rho l_1) \circ_\rho (p \rhd_\rho l_2) && \text{(L7)}\\
p_1 \rhd_\rho (p_2 \rhd_\rho l) &= (p_1 \wedge p_2) \rhd_\rho l && \text{(L8)}\\
l \circ_\rho l_1 \circ_\rho l &= l_1 \circ_\rho l && \text{(L9)}\\
(p \rhd_\rho l_1) \circ_\rho (\neg p \rhd_\rho l_2) &= (\neg p \rhd_\rho l_2) \circ_\rho (p \rhd_\rho l_1) && \text{(L10)}\\
(p_1 \rhd_\rho l) \circ_\rho (p_2 \rhd_\rho l) &= (p_1 \vee p_2) \rhd_\rho l && \text{(L11)}
\end{aligned}
$$

$$
\begin{aligned}
(a \asymp a) &= \mathbf{T} && \text{(A3)}\\
(a_1 \asymp a_2) &= (a_2 \asymp a_1) && \text{(A4)}\\
(a_1 \asymp a_2) \wedge (a_1 \asymp a_3) &= (a_1 \asymp a_2) \wedge (a_2 \asymp a_3) && \text{(A5)}\\
(a_1 \asymp a_2) \wedge \neg(a_1 \asymp a_3) &= (a_1 \asymp a_2) \wedge \neg(a_2 \asymp a_3) && \text{(A6)}
\end{aligned}
$$

$$
\begin{aligned}
[m!] &= [?] \circ_m m && \text{(M7)}\\
((p \rhd_m [?]) \circ_m m)! &= m! && \text{(M8)}
\end{aligned}
$$

$$
\begin{aligned}
p \rhd_s \{a \mapsto m\} &= \{a \mapsto (p \rhd_m m)\} && \text{(S5)}\\
(a_1 \asymp a_2) \rhd_s \{a_1 \mapsto m\} &= (a_1 \asymp a_2) \rhd_s \{a_2 \mapsto m\} && \text{(S9)}\\
(a_1 \asymp a_2) \rhd_m (s \mathbin{@} a_1) &= (a_1 \asymp a_2) \rhd_m (s \mathbin{@} a_2) && \text{(S10)}\\
(p \rhd_s s) \mathbin{@} a &= p \rhd_m (s \mathbin{@} a) && \text{(S11)}\\
\{a \mapsto m\} \circ_s s &= s \circ_s \{a \mapsto m \circ_m (s \mathbin{@} a)\} && \text{(S12)}
\end{aligned}
$$

$$
\begin{aligned}
\neg\neg p &= p && \text{(B7)}\\
(p_1 \wedge p_2) \wedge p_3 &= p_1 \wedge (p_2 \wedge p_3) && \text{(B8)}\\
p_1 \wedge p_2 &= p_2 \wedge p_1 && \text{(B9)}\\
p \wedge p &= p && \text{(B10)}\\
p \wedge \neg p &= \mathbf{F} && \text{(B11)}\\
(p_1 \vee p_2) \vee p_3 &= p_1 \vee (p_2 \vee p_3) && \text{(B12)}\\
p_1 \vee p_2 &= p_2 \vee p_1 && \text{(B13)}\\
p \vee p &= p && \text{(B14)}\\
p \vee \neg p &= \mathbf{T} && \text{(B15)}\\
p_1 \wedge (p_2 \vee p_3) &= (p_1 \wedge p_2) \vee (p_1 \wedge p_3) && \text{(B16)}\\
p_1 \vee (p_2 \wedge p_3) &= (p_1 \vee p_2) \wedge (p_1 \vee p_3) && \text{(B17)}\\
\neg(p_1 \wedge p_2) &= \neg p_1 \vee \neg p_2 && \text{(B18)}\\
\neg(p_1 \vee p_2) &= \neg p_1 \wedge \neg p_2 && \text{(B19)}
\end{aligned}
$$

Figure 12. Additional Equations of $\textsc{Pim}_t^=$.

Open merge structures with $\asymp$ but without @ or !

These are similar to the **if**-expressions treated in [Heering 1986, Section 3.3], but there are some additional complications. First, we have

$$m \circ_m (p \rhd_m [v]) = (\neg p \rhd_m m) \circ_m (p \rhd_m [v]), \qquad (7)$$

since

$$\begin{aligned} m \circ_m (p \rhd_m [v]) &\overset{(B15)(L11)}{=} (\neg p \rhd_m m) \circ_m (p \rhd_m m) \circ_m (p \rhd_m [v]) \\ &\overset{(L7)}{=} (\neg p \rhd_m m) \circ_m (p \rhd_m (m \circ_m [v])) \\ &\overset{(M2')}{=} (\neg p \rhd_m m) \circ_m (p \rhd_m [v]). \end{aligned}$$

(7) is a generalization of (M2′). Unfortunately, the even more general equation

$$m_1 \circ_m (p \rhd_m m_2) = (\neg p \rhd_m m_1) \circ_m (p \rhd_m m_2)$$

is not valid for $p = \mathbf{T}$ and $m_2 = \emptyset_m$. Instead we have the weaker analogue

$$(p_1 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) = ((\neg p_2 \wedge p_1) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l), \qquad (8)$$

since

$$\begin{aligned} &(p_1 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) = \\ &\overset{(L9)}{=} (p_1 \rhd_m l) \circ_m (p_2 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) \\ &\overset{(L11)}{=} ((p_1 \vee p_2) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) \\ &= (((\neg p_2 \wedge p_1) \vee p_2) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) \\ &\overset{(L11)}{=} ((\neg p_2 \wedge p_1) \rhd_m l) \circ_m (p_2 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) \\ &\overset{(L9)}{=} ((\neg p_2 \wedge p_1) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l). \end{aligned}$$

This affects the canonical forms of subterms involving variables of sort $\mathcal{M}$, making them somewhat more complicated then would otherwise be the case.

(L10) has the equivalent conditional form

$$p_1 \wedge p_2 = \mathbf{F} \implies (p_1 \rhd_\rho l_1) \circ_\rho (p_2 \rhd_\rho l_2) = (p_2 \rhd_\rho l_2) \circ_\rho (p_1 \rhd_\rho l_1), \qquad (9)$$

since, assuming $p_1 \wedge p_2 = \mathbf{F}$, we have

$$\begin{aligned} (p_1 \rhd_\rho l_1) \circ_\rho (p_2 \rhd_\rho l_2) &= (p_1 \rhd_\rho l_1) \circ_\rho ((\neg p_1 \wedge p_2) \rhd_\rho l_2) \\ &\overset{(L8)}{=} (p_1 \rhd_\rho l_1) \circ_\rho (\neg p_1 \rhd_\rho (p_2 \rhd_\rho l_2)) \\ &\overset{(L10)}{=} (\neg p_1 \rhd_\rho (p_2 \rhd_\rho l_2)) \circ_\rho (p_1 \rhd_\rho l_1) \\ &= (p_2 \rhd_\rho l_2) \circ_\rho (p_1 \rhd_\rho l_1). \end{aligned}$$

(9) is often more readily applicable than (L10).

A suitable canonical form (OMCF) for open merge structures without @ or ! is shown in Fig. 13.

Unrestricted open merge structures

A suitable canonical form (OMCFgen), very similar to (OMCF), is shown in Fig. 14.

$$\emptyset_m$$

and

$$(P_1 \rhd_m [V_1]) \circ_m \cdots \circ_m (P_k \rhd_m [V_k]) \circ_m (Q_1 \rhd_m M_1) \circ_m \cdots \circ_m (Q_n \rhd_m M_n)$$

with

(i) P_i in boolean canonical form $\neq \mathbf{F}$, $P_i \wedge P_j = \mathbf{F}$ $(i \neq j)$
(ii) V_i a variable or constant of sort $\mathcal{V}$, $V_i \neq V_j$ $(i \neq j)$
(iii) Q_i in boolean canonical form $\neq \mathbf{F}$, $Q_i \wedge Q_j = \mathbf{F}$ $(i \neq j)$
(iv) M_i an open merge structure $m_{i_1} \circ_m m_{i_2} \circ_m \cdots$ consisting of ≥ 1 different variables $m_{i_1}, m_{i_2}, \ldots$ of sort $\mathcal{M}$, and $M_i \neq M_j$ $(i \neq j)$.

Figure 13: (OMCF) Canonical form for open merge structures without @ or !. The canonical form is determined up to associativity and commutativity of $\vee$ and $\wedge$, symmetry of $\asymp$, and associativity and conditional commutativity of $\circ_m$ (equations (L3) and (9) with $\rho = m$).

$$\emptyset_m$$

and

$$(P_1 \rhd_m [V_1]) \circ_m \cdots \circ_m (P_k \rhd_m [V_k]) \circ_m (Q_1 \rhd_m M_1) \circ_m \cdots \circ_m (Q_n \rhd_m M_n)$$

with

(i)–(iii) As for canonical form (OMCF)
(iv) M_i an open merge structure consisting of ≥ 1 different variables, which may be either ordinary variables of sort $\mathcal{M}$ or compound variables s @ A, and $M_i \neq M_j$ $(i \neq j)$
(v) The corresponding multiset of address constants and variables is minimal with respect to the ordering (6).

Figure 14: (OMCFgen) Canonical form for unrestricted open merge structures. Apart from requirements (iv) and (v), it is the same as canonical form (OMCF).

$$M!$$

with M in canonical form (OMCFgen) without subterm $P \rhd_m [?]$.

Figure 15. (OVCF) Canonical form for open terms of sort $\mathcal{V}$

Open terms of sort $\mathcal{V}$

A suitable canonical form (OVCF) is shown in Fig. 15.

Open store structures without @ or $\asymp$ and without variables of sort $\mathcal{S}$

We first note the following immediate consequences of (S8):

$$\{a \mapsto m_1\} \circ_s \{a \mapsto m_2\} = \{a \mapsto (m_1 \circ_m m_2)\} \tag{S6}$$

$$(a_1 \asymp a_2) = \mathbf{F} \implies \{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\} = \{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\} \tag{S7}$$

$$\begin{aligned} &\neg(a_1 \asymp a_2) \rhd_s (\{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\}) = \\ &= \neg(a_1 \asymp a_2) \rhd_s (\{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\}) \end{aligned} \tag{10}$$

(S7) is a conditional commutative law.[12] (10) is similar but with an appropriate guard rather than a condition.

A suitable canonical form (OSCF) is shown in Fig. 16. An important special case (OSCFsimple) is shown in Fig. 17.

Open store structures with @ and $\asymp$ and with variables of sort $\mathcal{S}$, but without variables of sort $\mathcal{A}$

The main equation we need is (S12). Note that in case of a finite number K of address constants α_i, the stronger equation $s = (\{\alpha_1 \mapsto s \mathbin{@} \alpha_1\}) \circ_s \cdots \circ_s (\{\alpha_K \mapsto s \mathbin{@} \alpha_K\})$ would hold.

Since there are no address variables, any occurrences of $\asymp$ can be eliminated by (A1–2) and the extension (OSCFvar) of the simple canonical form (OSCFsimple) shown in Figure 18 applies.

Let $\textsc{Pim}_t^= = \textsc{Pim}_t^+ +$ the equations of Figure 12. In view of the foregoing we have

Proposition 3. *$\textsc{Pim}_t^=$ is ω-complete.*

10. Pim in Practice

10.1 Rewriting Pim Graphs

By orienting equation instances of $\textsc{Pim}_t^=$ and implementing the resulting rules on graphs, we obtain a *term graph rewriting* system [Barendregt et al. 1987]. Such systems can be designed to produce normal forms with a variety of interesting properties. For example, the graph S'_{P_1} depicted in Fig. 7 is obtained by first normalizing the graph S_{P_1} (Fig. 2) with respect to the system $\textsc{Pim}_t^\rightarrow$ developed in Section 10.2, then using instances of equation (S8) of $\textsc{Pim}_t^+$ to permute addresses with respect to a fixed ordering. S_{P_1} is the normal form of the Pim representations

[12](S6) and (S7) correspond to (S6) and (S7) in the preliminary version of $\textsc{Pim}_t^=$ given in [Field 1992, Figure 6].

$$\emptyset_s$$

and

$$(\Pi_1 \ \rhd_s \{A_1 \mapsto M_1\}) \circ_s \cdots \circ_s (\Pi_n \ \rhd_s \{A_n \mapsto M_n\})$$

with

(i) A_i a constant or variable of sort $\mathcal{A}$

(ii) M_i a merge structure without $\asymp$ in canonical form (OMCF) $\neq \emptyset_m$

(iii) Π_i the canonical form $\neq \mathbf{F}$ of

$$\bigwedge_{k=1}^{n} \pm(A_i \asymp A_k)$$

with $\pm(A_i \asymp A_k)$ denoting one of $A_i \asymp A_k$ or $\neg(A_i \asymp A_k)$

(iv) $\Pi_i \wedge \Pi_j = \mathbf{F}$ ($A_i = A_j$ modulo (S9), $i \neq j$)

(v) $\displaystyle\bigvee_{\substack{A_j = A_i \\ \text{modulo (S9)}}} \Pi_j \ = \ \mathbf{T} \quad (1 \leq i \leq n)$

(vi) The corresponding multiset of address constants and variables is minimal with respect to the ordering (6).

Figure 16: (OSCF) Canonical form for open store structures without @ or $\asymp$ and without variables of sort $\mathcal{S}$. The canonical form is determined up to associativity of $\circ_s$ (equation (L3) with $\rho = s$). Furthermore, as a consequence of requirements (iii) and (iv) at least one of the conditional commutative laws (9), (S7), (10) applies to any pair of adjacent store cells, and the canonical form is unconditionally commutative. Unlike the original term, the canonical form is not $\asymp$-free.

$$(\mathbf{T} \ \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \ \rhd_s \{\alpha_{i_n} \mapsto M_n\})$$

with

(i) All α_{i_j} different in view of requirement (iv) of canonical form (OSCF)

(ii) As requirement (ii) of canonical form (OSCF).

Figure 17. (OSCFsimple) Special case of (OSCF) if all addresses are known.

of both P_1 (i.e., S_{P_1}) and P_3 (Fig. 1); therefore, it is immediate that they are behaviorally equivalent.

The normalization process can be used not only to discover equivalences not apparent from the initial PIM representations, but also to "build" useful graph-based compiler IRs as a side effect [Field 1992]. For example, the composition operator in the subgraph M' of S'_{P_1} is very similar to an instance of the γ node of GSA form [Ballance, MacCabe, and Ottenstein 1990]. If we ignore the guard, we can also interpret the composition operator in M' as an SSA form ϕ node [Cytron et al. 1991]. The principal difference between these IRs and the class of normal forms exemplified by S'_{P_1} is that variable uses are linked *directly* to the expressions that define their value, even when, e.g., a chain of copying assignments intervenes (VDGs [Weise et al. 1994] also have this property).

Fig. 19 depicts the graph S''_{P_1}, a further simplification of S'_{P_1}. S''_{P_1} is also a simplified form of the PIM translations of programs P_3 and P_5 in Fig. 1. As with S'_{P_1}, S''_{P_1} is produced by a rewriting system, namely, $\text{PIM}_t^{\rightarrow}$ augmented with an oriented instance of $\text{PIM}_t^{=}$'s equation (L11), followed as before by address permutation using (S8). Depending on the application, it may be more appropriate to use systems that produce normal forms similar to compiler IRs, such as S'_{P_1}, rather than simplifying further to forms such as S''_{P_1}

Consider finally the μC programs depicted in Fig. 20. All of these programs are behaviorally equivalent; this fact may be deduced by inspection of the normal form graph S'_{P_6} shown in Fig. 21 (produced by augmenting the system used to produce S'_{P_1} with an oriented instance of equation (L11)). We know of no intermediate representation in the compiler literature for which the representations of P_6–P_9 would be the same.

In general, the design of a PIM-based rewriting system will be governed by the tradeoffs between properties desired of normal forms and the time complexity of normalization strategies that yield those normal forms. As several PIM operators are commutative and associative, one cannot rely entirely on structural properties of normalized graphs to detect all valid program equivalences. However, rewriting systems are an important stepping-stone to more powerful decision procedures, and allow structural identity to be used to detect many more equivalences than would be possible otherwise.

In the next section, we obtain *confluent* and *terminating* rewriting systems from $\text{PIM}_t^{=}$. While such systems constitute partial decision procedures for term equivalence, they are also good starting points for designing (possibly non-confluent) systems that derive terms or graphs with useful structural properties (e.g., in which variable definitions are linked directly to uses). The development of confluent and terminating systems can be mechanized to a certain extent by Knuth-Bendix completion [Dershowitz and Jouannaud 1990]. However, producing systems that yield graphs with particular structural properties requires a more ad-hoc process of introducing (or deleting) rules derived from $\text{PIM}_t^{=}$ until the desired property is achieved.

10.2 Confluent Subsystems of $\text{PIM}_t^{=}$

Our goal in this section will be to derive the strongest possible confluent and terminating subsystems of $\text{PIM}_t^{=}$. Much of this work described in this section was carried out with the assistance of the TIP inductive theorem proving system [Fraus

$$\Sigma \circ_s ((\mathbf{T} \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \rhd_s \{\alpha_{i_n} \mapsto M_n\}))$$

with

(i) Σ in canonical form (OMCF) with $k = 0$, or rather its equivalent for sort $\mathcal{S}$

(ii) the rightmost part in canonical form (OSCFsimple), but with merge structure components M_i in canonical form (OMCFgen) $\neq \emptyset_m$ rather than (OMCF)

(iii) $p \wedge q = \mathbf{F}$ for any $p \rhd_s (\cdots \circ_s s)$ in Σ and $q \rhd_m ((s \mathbin{@} \alpha_{i_j}) \circ_m \cdots)$ in M_j.

Figure 18: (OSCFvar) Canonical form for open store structures with @ and $\asymp$ and with variables of sort $\mathcal{S}$, but without variables of sort $\mathcal{A}$.

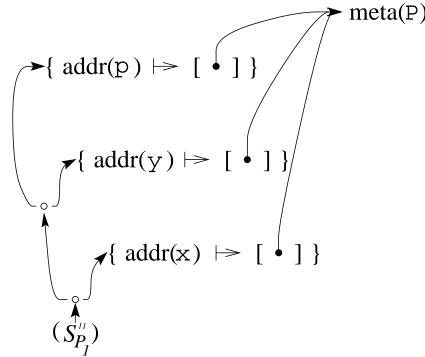

Figure 19. S''_{P_1}: Simplest PIM representation of programs P_1, P_3, and P_5.

```
/* int x,y,z;
   int *ptr;
   const int ?P; */

{  ptr = &z;
   x = 12;
   y = 17;
   if (!(?P))
     x = 13;
   z = y + x;
}
```

P_6

```
/* int x,y,z;
   int *ptr;
   const int ?P; */

{  ptr = &z;
   if (!(?P))
     x = 13;
   else
     x = 12;
   if (?P)
     y = 17;
   else
     y = 17;
   z = y + x;
}
```

P_7

```
/* int x,y,z;
   int *ptr;
   const int ?P; */

{  ptr = &z;
   if (?P) {
     x = 12;
     y = 17;
   }
   else {
     x = 13;
     y = 17;
   }
   z = y + x;
}
```

P_8

```
/* int x,y,z;
   int *ptr;
   const int ?P, ?Q; */

{  ptr = &x;
   if (?P) {
     (*ptr) = 12;
     y = 17;
   }
   ptr = &y;
   z = 17;
   if (!(?P)) {
     (*ptr) = 19;
     x = 13;
     ptr = &z;
   }
   if (?P || ?Q)
     ptr = &z;
   (*ptr) = y + x;
}
```

P_9

Figure 20. Semantically equivalent μC programs.

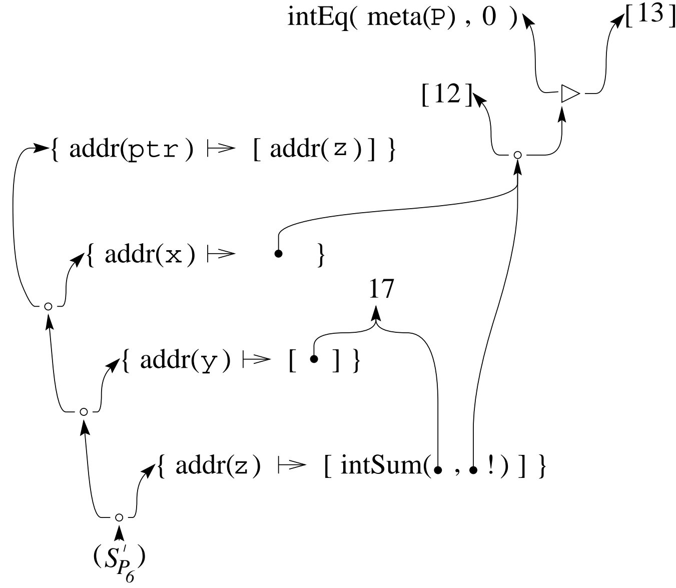

Figure 21. S'_{P_6}: Common representation for P_6, P_7, P_8, and P_9.

1994], which was used to perform Knuth-Bendix completion and to aid in inductive verification of equations incorporated in $\text{PIM}_t^=$. While TIP was able to inductively verify many $\text{PIM}_t^=$ equations, it was unable to successfully use (S8) as a lemma during semi-automatic proofs (other theorem provers available to us had similar limitations), and the full mechanical verification succeeded only with considerable difficulty [Naidich and Dinesh 1996]. Here, we indicate progress in converting part of $\text{PIM}_t^=$ into a term rewriting system. We first consider the completion of PIM_t^0, then treat the additional equations of PIM_t^+, and finally those of $\text{PIM}_t^=$.

Knuth-Bendix completion of PIM_t^0. The rewriting system obtained by interpreting the equations of PIM_t^0 as left-to-right rewriting rules and with AC-declarations for $\wedge$ and $\vee$ is confluent and terminating with the addition of the rule

$$(a_1 \asymp a_2) \rhd_m \emptyset_m \quad \rightarrow \quad \emptyset_m, \tag{MA0}$$

which originates from a critical pair generated from the rules (S1) and (S2). PIM_t^0 is ground confluent even without this rule. Left-associative orientation of (L3) (i.e., left-to-right orientation of the equation) is significant for PIM_t^0, as a right-associative orientation of (L3) in conjunction with rule (M2) causes the completion procedure to add an infinite number of rules

$$(m_1 \circ_m (m_2 \circ_m \cdots (m_i \circ_m [v]) \cdots))! \rightarrow \ v \quad (i \geq 2). \tag{11}$$

Knuth-Bendix completion of PIM_t^+. When (M2′) is substituted for (M2), the orientation of (L3) becomes irrelevant, since the context in which the pattern

$m \circ_m [v]$ could be matched is now immaterial. Using a left-associative orientation of (L3) for the merge case, the completion procedure adds only the rule (MA0). We note that this is a special case of (L4) below.

Adding (S8) is, however, a difficult problem since the equation is (conditionally) commutative. We therefore proceed by first splitting (S8) into (S6) and (S7) (see p. 30). (S7) is difficult to orient, but (S6) has an obvious orientation and is in acceptable form for mechanical analyzers. After attempting TIP's completion procedure on the system with (S6) and (M2′), we see immediately that the critical pairs that result from (S6) and (S4), using (S1), give rise to a special case of (L7) for $\rho = m$. Unfortunately, both (S6) and (L7) are left-*non*linear rules (when oriented left-to-right). Obtaining a left-linear completion is often preferable to a left-nonlinear completion, since

—a left-linear system admits an efficient implementation, without the need for equality tests during matching;

—when a left-linear system is embedded in the untyped lambda calculus (as is necessary to extend PIM to arbitrary source programs), it is straightforward to show that the combined system remains confluent [Müller 1992].

We therefore consider left-nonlinear equations separately, and proceed for the moment without (S6) and (L7).

Adding the boolean equations (B7), (B18) and (B19), along with the oriented versions of the equations (L4) and (L8) results in a confluent and terminating system. (L8) requires us to use multiset ordering, due to the permutability of the guards. To also accommodate (L3), we use the generalized recursive path ordering with status, which in TIP is called the "multiset ordering based on the lexicographic ordering".

Adding (M7) or (M8) requires that (L3) be oriented in the right-associative direction. This is caused by the generation of the rule (MA2), which is similar to (M2) (see the completion of PIM_t^0 above) but with a pattern [?] in the context $\square!$ appearing on the left. Also, adding (M7) and (M8) generates the rules (MA1) to (MA5). (MA1) and (MA5) are due to the right-associative ordering of (L3). The resulting system $\text{PIM}_t^{\rightarrow}$ is shown in Figure 22. $\text{PIM}_t^{\rightarrow}$ is confluent, terminating, and left-linear.

Enhancing the rewriting systems. Further enrichments to $\text{PIM}_t^{\rightarrow}$ seem to require left-nonlinear rules in order to achieve confluence. Adding (L7), we require the additional rules (MB1)–(MB4) shown in Fig. 23. If we then add (S6), we need the rule (SB1), also shown in Fig. 23. Adding all the rules in Fig. 23 to those of $\text{PIM}_t^{\rightarrow}$, we get the system $\text{PIM}_t'^{\rightarrow}$.

If we enrich $\text{PIM}_t^{\rightarrow}$ with the equations (B10), (B14) and (B16), oriented left-to-right, the completion procedure of the LP system [Garland and Guttag 1991] adds the absorption law

$$p \vee (p \wedge p_1) \quad \rightarrow \quad p. \tag{BA1}$$

Finally, both $\text{PIM}_t^{\rightarrow}$ and $\text{PIM}_t'^{\rightarrow}$ produce normal forms modulo associativity and commutativity of $\wedge$ and $\vee$, i.e., with respect to (B8), (B9), (B12) and (B13). Note that $\text{PIM}_t^{\rightarrow}$ does not require rewriting modulo associativity and commutativity, since it can be enhanced with the symmetric variants of the rules (B3)–(B6) and the two associativity rules for $\wedge$ and $\vee$ (see Table 1).

SYSTEMS	PROPERTIES
PIM$_t^0$, equations oriented left-to-right + (B8), (B12), $\widetilde{\text{(B3)–(B6)}}$	ground-confluent, terminating, left-linear
PIM$_t^0$, equations oriented left-to-right + (MA0), (B8), (B12), $\widetilde{\text{(B3)–(B6)}}$	confluent, terminating, left-linear
PIM$_t^0$, equations oriented left-to-right + (MA0)	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-linear
PIM$_t^+ -$ (S8), equations oriented left-to-right + (B8), (B12), $\widetilde{\text{(B3)–(B6)}}$	ground-confluent, terminating, left-linear
PIM$_t^+ -$ (S8), equations oriented left-to-right + (MA0), (B8), (B12), $\widetilde{\text{(B3)–(B6)}}$	confluent, terminating, left-linear
PIM$_t^+ -$ (S8), equations oriented left-to-right + (MA0)	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-linear
PIM$_t^{\rightarrow}$ + (B8), (B12), $\widetilde{\text{(B3)–(B6)}}$	confluent, terminating, left-linear
PIM$_t^{\rightarrow}$	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-linear
PIM$_t^{\rightarrow}$ + (B10), (B14), (B16), (BA1)	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-*non*linear
PIM$_t'^{\rightarrow}$	confluent ($\wedge$, $\vee$: modulo AC) terminating, left-*non*linear

Table 1: Properties of some of the PIM$_t$ systems. $\widetilde{\text{(B3)–(B6)}}$ indicates symmetric versions of the rules (B3), (B4), (B5) and (B6).

$$
\begin{array}{rcll}
\emptyset_\rho \circ_\rho l & \to & l & \text{(L1)} \\
l \circ_\rho \emptyset_\rho & \to & l & \text{(L2)} \\
(l_1 \circ_\rho l_2) \circ_\rho l_3 & \to & l_1 \circ_\rho (l_2 \circ_\rho l_3) & \text{(L3)} \\
p \rhd_\rho \emptyset_\rho & \to & \emptyset_\rho & \text{(L4)} \\
\mathbf{T} \rhd_\rho l & \to & l & \text{(L5)} \\
\mathbf{F} \rhd_\rho l & \to & \emptyset_\rho & \text{(L6)} \\
p_1 \rhd_\rho (p_2 \rhd_\rho l) & \to & (p_1 \wedge p_2) \rhd_\rho l & \text{(L8)} \\
\\
\{a_1 \mapsto m\} \mathbin{@} a_2 & \to & (a_1 \asymp a_2) \rhd_m m & \text{(S1)} \\
\{a \mapsto \emptyset_m\} & \to & \emptyset_s & \text{(S2)} \\
\emptyset_s \mathbin{@} a & \to & \emptyset_m & \text{(S3)} \\
(s_1 \circ_s s_2) \mathbin{@} a & \to & (s_1 \mathbin{@} a) \circ_m (s_2 \mathbin{@} a) & \text{(S4)} \\
p \rhd_s \{a \mapsto m\} & \to & \{a \mapsto (p \rhd_m m)\} & \text{(S5)} \\
(p \rhd_s s) \mathbin{@} a & \to & p \rhd_m (s \mathbin{@} a) & \text{(S11)} \\
\\
(\alpha_i \asymp \alpha_i) & \to & \mathbf{T} \quad (i \geq 1) & \text{(A1)} \\
(\alpha_i \asymp \alpha_j) & \to & \mathbf{F} \quad (i \neq j) & \text{(A2)} \\
\\
m \circ_m [v] & \to & [v] & \text{(M2}'\text{)} \\
[v]! & \to & v & \text{(M3)} \\
\emptyset_m! & \to & ? & \text{(M4)} \\
[m!] & \to & [?] \circ_m m & \text{(M7)} \\
((p \rhd_m [?]) \circ_m m)! & \to & m! & \text{(M8)} \\
\\
\neg\mathbf{T} & \to & \mathbf{F} & \text{(B1)} \\
\neg\mathbf{F} & \to & \mathbf{T} & \text{(B2)} \\
\mathbf{T} \wedge p & \to & p & \text{(B3)} \\
\mathbf{F} \wedge p & \to & \mathbf{F} & \text{(B4)} \\
\mathbf{T} \vee p & \to & \mathbf{T} & \text{(B5)} \\
\mathbf{F} \vee p & \to & p & \text{(B6)} \\
\neg\neg p & \to & p & \text{(B7)} \\
\neg(p_1 \wedge p_2) & \to & \neg p_1 \vee \neg p_2 & \text{(B18)} \\
\neg(p_1 \vee p_2) & \to & \neg p_1 \wedge \neg p_2 & \text{(B19)} \\
\\
m_1 \circ_m ([v] \circ_m m_2) & \to & [v] \circ_m m_2 & \text{(MA1)} \\
([?] \circ_m m)! & \to & m! & \text{(MA2)} \\
(p \rhd_m [?])! & \to & ? & \text{(MA3)} \\
[?] \circ_m (p \rhd_m [?]) & \to & [?] & \text{(MA4)} \\
[?] \circ_m ((p \rhd_m [?]) \circ_m m) & \to & [?] \circ_m m & \text{(MA5)}
\end{array}
$$

Figure 22. Rewriting rules of $\text{PIM}_t^{\to}$.

$$p \rhd_\rho (l_1 \circ_\rho l_2) \;\rightarrow\; (p \rhd_\rho l_1) \circ_\rho (p \rhd_\rho l_2) \qquad \text{(L7)}$$
$$\{a \mapsto m_1\} \circ_s \{a \mapsto m_2\} \;\rightarrow\; \{a \mapsto (m_1 \circ_m m_2)\} \qquad \text{(S6)}$$

$$(p \rhd_m m) \circ_m (p \rhd_m [v]) \;\rightarrow\; p \rhd_m [v] \qquad \text{(MB1)}$$
$$((p \wedge p_1) \rhd_m m) \circ_m (p \rhd_m [v]) \;\rightarrow\; p \rhd_m [v] \qquad \text{(MB2)}$$
$$(p \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) \;\rightarrow\; (p \rhd_m [v]) \circ_m m \qquad \text{(MB3)}$$
$$((p \wedge p_1) \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) \;\rightarrow\; (p \rhd_m [v]) \circ_m m \qquad \text{(MB4)}$$

$$\{a \mapsto m_1\} \circ_s (\{a \mapsto m_2\} \circ_s s) \;\rightarrow\; \{a \mapsto (m_1 \circ_m m_2)\} \circ_s s \qquad \text{(SB1)}$$

Figure 23. $\text{PIM}_t'^{\rightarrow} = \text{PIM}_t^{\rightarrow} +$ rules above.

Problematic equations. Attempts to obtain further enriched confluent and terminating rewriting systems have been unsuccessful thus far. Adding both (B16) and (B17) results in a non-terminating system. Even adding one of them causes problems for the TIP system. The left-nonlinear rules resulting from (B10), (B11), (B14) and (B15) cause problems with TIP's treatment of AC declarations. Unlike TIP, LP was able to add (B10), (B14) and (B16) to $\text{PIM}_t^{\rightarrow}$. (A4), (A5), (A6), (S9), (S10) are good candidates to be put in the set of "modulo" equations but we are not aware of any available Knuth-Bendix completion system that allows it. (S12) and the general form of (S8) cannot be ordered properly and thus lead to non-terminating term rewriting systems. (L9), (L10) and (L11) lead to left-nonlinear rules, which again cause problems for completion modulo AC. Despite these difficulties, we conjecture that larger confluent subsystems of $\text{PIM}_t^{=}$ exist, particularly if we consider confluence modulo associativity, idempotence, identity, and commutativity. Finding such systems is left as future work. One approach might be to incorporate Hsiang and Dershowitz's [1983] confluent ω-complete specification of the booleans, since the well-known disjunctive and conjunctive boolean normal forms are not produced by any rewriting system. We have not been more successful here, again due to the interference of the left-nonlinear rules in these booleans with the other left-nonlinear rules of PIM.

11. Extensions and Future Work

Although we have treated the algebraic core of PIM quite extensively in this paper, more work remains to be done to study the higher-order versions of PIM touched on in Section 5. In particular, we would like to develop this work along the following lines:

Provide an operational semantics for higher-order PIM. This would proceed by defining a reduction strategy for PIM_t + the β rule, then proving a standardization theorem. We expect such a result to be routine, given similar results by other authors [Felleisen and Hieb 1992; Odersky, Rabin, and Hudak 1993], the lack of overlap in the structure of pure lambda terms and PIM_t terms, and the fact that an oriented version of PIM_t^0 is *sequential* [Boudol 1985]. Confluence follows trivially from the result of Müller [1992].

Study the effect of nontermination on $\text{PIM}_t^=$. It is not difficult to show that in the presence of nontermination, the additional equations introduced by $\text{PIM}_t^=$ are unsound in a technical sense, since they may cause a nonterminating term to be equated with a terminating one. However, we expect to be able to show that equations between any two terms are valid *provided* both of their standard reductions terminate. Such a result would provide a weak form of soundness consistent with the way most compiler transformations are implemented in practice. Alternatively, one could attempt to completely reconcile formal semantics and practice by reformulating certain equations of $\text{PIM}_t^=$ in conditional form, predicating their consequents on proving nontermination for appropriate subterms.

Consider typed higher-order variants of PIM. Due to its simplicity and computational power, the untyped lambda calculus is attractive for encoding a variety of higher-order program constructs. However, the absence of typing makes it more difficult to define strong inference rules for commonly occurring higher-order constructs. It would therefore be useful to investigate typed higher-order versions of PIM. PIM's stores pose a challenge to defining appropriate type systems, since (like real machine memory) they may hold a variety of values of different types—including, e.g., functional values.

Study induction principles. While we believe that the induction rules defined in Field [1992] and Field, Ramalingam, and Tip [1995] are a good starting point for reasoning about loops and recursion, other forms of induction may also be useful. Rules for which inductive premises can be established mechanically, e.g., by fixpoint iteration, are especially attractive.

Obtain completeness results for restricted higher-order systems. Given the lack of success in finding an ω-complete axiomatization for the pure untyped lambda calculus [Plotkin 1974], it appears unlikely that a completeness result for a logic axiomatizing the operational properties of PIM_t + the β rule could be obtained. However, one might nonetheless attempt to formulate completeness results for restricted systems, e.g., for typed subsystems, or systems that eschew lambda terms in favor of more restricted forms of recursion or iteration.

In addition to further study of the properties of higher-order variants of PIM, we would also like to extend our results in the following directions:

—Using the canonical forms discussed in this paper to develop a decision procedure for PIM_t. Obtaining a canonical form for unrestricted open store structures would be a useful complement to this.

—Obtaining completeness results for variants of PIM_t, including versions with no restrictions on the formation of address or predicate expressions, variants incorporating the *merge* distribution rules, as used for addresses in [Field 1992] and generalized in [Field, Ramalingam, and Tip 1995], and extensions with non-trivial value operations.

—Constructing confluent and/or terminating rewriting subsystems of $\text{PIM}_t^=$ stronger than $\text{PIM}_t^{\prime\rightarrow}$.

ACKNOWLEDGMENTS

We are grateful to G. Ramalingam for his assistance in implementing various versions of PIM, as well as for valuable technical suggestions. We also wish to thank

the referees and associate editor for suggestions that helped to improve the presentation of the paper. Without the help of Dimitri Naidich the mechanical verification of $\text{PIM}_t^=$ would not have succeeded.

References

Ammarguellat, Z. 1992. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering 18,* 3 (March), 237–251.

Ballance, R. A., MacCabe, A. B., and Ottenstein, K. J. 1990. The program dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY, pp. 257–271.

Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., and Sleep, M. 1987. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages*, Volume 259 of *Lecture Notes in Computer Science*, pp. 141–158. Springer-Verlag.

Bergstra, J. A. and Heering, J. 1994. Which data types have ω-complete initial algebra specifications? *Theoretical Computer Science 124*, 149–168.

Bergstra, J. A. and Tucker, J. V. 1995. The data type variety of stack algebras. *Annals of Pure and Applied Logic 73,* 1 (May), 11–36.

Berlin, A. and Weise, D. 1990. Compiling scientific code using partial evaluation. *IEEE Computer 23,* 12 (December), 25–36.

Boehm, H.-J. 1985. Side effects and aliasing can have simple axiomatic descriptions. *ACM Trans. on Programming Languages and Systems 7,* 4 (October), 637–655.

Boudol, G. 1985. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds (Eds.), *Algebraic Methods in Semantics*, Chapter 5, pp. 169–236. Cambridge University Press.

Cartwright, R. and Felleisen, M. 1989. The semantics of program dependence. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, pp. 13–27.

Chambers, C. and Ungar, C. 1989. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 146–160.

Click, C. 1995. Global code motion, global value numbering. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, La Jolla, CA, pp. 246–257. Published as ACM SIGPLAN Notices 30(6).

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems 13,* 4 (October), 451–490.

Dershowitz, N. and Jouannaud, J.-P. 1990. Rewrite systems. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, pp. 243–320. Elsevier/The MIT Press.

Ershov, A. P. 1982. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science 18*, 41–67.

Ershov, A. P. and Ostrovski, B. N. 1987. Controlled mixed computation and its appplication to systematic development of language-oriented parsers. In L. G. L. T. Meertens (Ed.), *Program Specification and Transformation*, pp. 31–48. North-Holland.

Felleisen, M. and Friedman, D. P. 1989. A syntactic theory of sequential state. *Theoretical Computer Science 69*, 243–287.

Felleisen, M. and Hieb, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 235–271.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9,* 3 (July), 319–349.

Field, J. 1992. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Francisco, pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

Field, J., Ramalingam, G., and Tip, F. 1995. Parametric program slicing. In *Proc. Twenty-second ACM Symp. on Principles of Programming Languages*, San Francisco, pp. 379–392.

Fraus, U. 1994. Inductive theorem proving for algebraic specifications—TIP system user's manual. Technical Report MIP 9401, University of Passau. The TIP system is available at URL: ftp://forwiss.uni-passau.de/pub/local/tip.

Garland, S. and Guttag, J. 1991. A Guide to LP, The Larch Prover. Technical Report 82 (December), Systems Research Center, DEC.

Heering, J. 1985. Variaties op het thema 'stack'. Technical Report CS-N8502, CWI, Amsterdam. (In Dutch).

Heering, J. 1986. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science 43*, 149–167.

Hoare, C., Hayes, I., Jifeng, H., Morgan, C., Roscoe, A., Sanders, J., Sorensen, I., Spivey, J., and Sufrin, B. 1987. Laws of programming. *Communications of the ACM 30,* 8 (August), 672–686. Corrigenda, ibid., p. 770.

Horwitz, S., Prins, J., and Reps, T. 1988. On the adequacy of program dependence graphs for representing programs. In *Proc. Fifteenth ACM Symp. on Principles of Programming Languages*, San Diego, CA, pp. 146–157.

Hsiang, J. and Dershowitz, N. 1983. Rewrite methods for clausal and non-clausal theorem proving. In J. Diaz (Ed.), *Automata, Languages and Programming (10th ICALP)*, Volume 154 of *Lecture Notes in Computer Science*, pp. 331–346. Springer-Verlag.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Kahn, G. 1987. Natural semantics. In *Fourth Annual Symp. on Theoretical Aspects of Computer Science*, Volume 247 of *Lecture Notes in Computer Science*, pp. 22–39. Springer-Verlag.

Kennaway, J. R., Klop, J. W., Sleep, M. R., and de Vries, F. J. 1994. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. on Programming Languages and Systems 16,* 3, 493–523.

Klop, J. W. 1992. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum (Eds.), *Handbook of Logic in Computer Science, Vol. II*, pp. 1–116. Oxford University Press.

Lazrek, A., Lescanne, P., and Thiel, J.-J. 1990. Tools for proving inductive equalities, relative completeness, and ω-completeness. *Information and Computation 84*, 47–70.

Mason, I. A. and Talcott, C. 1989. Axiomatizing operational equivalence in the presence of side effects. In *Proc. Fourth IEEE Symp. on Logic in Computer Science*, Cambridge, MA, pp. 284–293.

Meinke, K. and Tucker, J. V. 1992. Universal algebra. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science, Vol. I*, pp. 189–411. Oxford University Press.

Meseguer, J. and Goguen, J. A. 1985. Initiality, induction and computability. In M. Nivat and J. C. Reynolds (Eds.), *Algebraic Methods in Semantics*, pp. 459–541. Cambridge University Press.

Müller, F. 1992. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters 41*, 293–299. Amended proof available from Inge Bethke (inge@cwi.nl).

Naidich, D. and Dinesh, T. B. 1996. An automated induction method for verification of Pim—a transformational toolkit for compilers. Technical Report CS-R9630, CWI, Amsterdam, The Netherlands.

Nirkhe, V. and Pugh, W. 1992. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *Proc. Nineteenth ACM Symp. on Principles of Programming Languages*, Albuquerque, NM, pp. 269–280.

ODERSKY, M., RABIN, D., AND HUDAK, P. 1993. Call by name, assignment, and the lambda calculus. In *Proc. Twentieth ACM Symp. on Principles of Programming Languages*, Charleston, SC, pp. 43–56.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice Hall International, Englewood Cliffs, NJ.

PLOTKIN, G. D. 1974. The λ-calculus is ω-incomplete. *J. Symbolic Logic 39,* 2 (June), 313–317.

RAMALINGAM, G. AND REPS, T. 1989. Semantics of program representation graphs. Technical Report 900 (December), Computer Sciences Department, University of Wisconsin, Madison, 1210 W. Dayton St., Madison WI 53706.

SELKE, R. P. 1989a. A rewriting semantics for program dependence graphs. In *Proc. Sixteenth ACM Symp. on Principles of Programming Languages*, Austin, TX, pp. 12–24.

SELKE, R. P. 1989b. Transforming program dependence graphs. Technical Report TR90-131 (July), Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892.

SWARUP, V., REDDY, U. S., AND IRELAND, E. 1991. Assignments for applicative languages. In *Proc. Fifth ACM Conf. on Functional Programming Languages and Computer Architecture*, Volume 523 of *Lecture Notes in Computer Science*, pp. 192–214. Springer-Verlag.

VAN DEURSEN, A., HEERING, J., AND KLINT, P. (Eds.) 1996. *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing. World Scientific.

WAND, M. AND WANG, Z.-Y. 1990. Conditional lambda-theories and the verification of static properties of programs. In *Proc. Fifth IEEE Symp. on Logic in Computer Science*, Philadelphia, PA.

WEISE, D., CREW, R. F., ERNST, M., AND STEENSGAARD, B. 1994. Value dependence graphs: Representation without taxation. In *Proc. Twenty-First ACM Symp. on Principles of Programming Languages*, Portland, OR, pp. 297–310.

WIRSING, M. 1990. Algebraic specification. In J. VAN LEEUWEN (Ed.), *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, pp. 675–788. Elsevier/The MIT Press.

YANG, W., HORWITZ, S., AND REPS, T. 1989. Detecting program components with equivalent behaviors. Technical Report 840 (April), University of Wiconsin-Madison.

YANG, W., HORWITZ, S., AND REPS, T. 1990. A program integration algorithm that accommodates semantics-preserving transformations. In *Proc. Fourth ACM SIGSOFT Symp. on Software Development Environments*, Irvine, CA, pp. 133–143.

A. μC-To-PIM Translation

A1 μC Programs to PIM Terms

The syntax of μC is given in Fig. 24. A formal description of the translation of μC programs to PIM terms is given in Figures 25 and 26. The translation is written in the style of Natural Semantics [Kahn 1987], and adheres very closely to standard C semantics, e.g., integers are used to represent boolean values.

The translation uses several different sequent forms corresponding to the principal

$$
\begin{array}{rcl}
\text{Pgm} & ::= & \{\ \text{StmtList}\ \} \\
\\
\text{StmtList} & ::= & \text{Stmt} \\
 & | & \text{StmtList Stmt} \\
\\
\text{Stmt} & ::= & \text{Exp}\ ; \\
 & | & \texttt{if}\ (\ \text{Exp}\ )\ \text{Stmt} \\
 & | & \texttt{if}\ (\ \text{Exp}\ )\ \text{Stmt}\ \texttt{else}\ \text{Stmt} \\
 & | & \texttt{while}\ (\ \text{Exp}\ )\ \text{Stmt} \\
 & | & \{\ \text{StmtList}\ \} \\
\\
\text{Exp} & ::= & \text{Exp}_p \\
 & | & \text{Exp}_i \\
\\
\text{Exp}_p & ::= & \text{Id} \\
 & | & \text{IntLiteral} \\
 & | & \texttt{?}\text{Id} \\
\\
\text{Exp}_i & ::= & \texttt{*}\ \text{Exp} \\
 & | & \texttt{\&}\ \text{LValue} \\
 & | & \text{LValue}\ \texttt{=}\ \text{Exp} \\
 & | & \text{LValue}\ \texttt{-=}\ \text{Exp} \\
 & | & \text{Exp}\ \texttt{+}\ \text{Exp} \\
 & | & \texttt{!}\ \text{Exp} \\
 & | & \text{Exp}\ \texttt{||}\ \text{Exp} \\
\\
\text{LValue} & ::= & \text{LValue}_p \\
 & | & \text{LValue}_i \\
\\
\text{LValue}_p & ::= & \text{Id} \\
\\
\text{LValue}_i & ::= & \texttt{*}\ \text{Exp}
\end{array}
$$

Figure 24. Syntax of μC.

$$(\mathrm{Pgm}_1)\quad \frac{\emptyset_s \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u}{\vdash \text{Stmt} \Rightarrow_{\text{Pgm}} u}$$

$$(\mathrm{Stmt}_1)\quad \frac{s \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u}{s \vdash \{\ \text{Stmt}\ \} \Rightarrow_{\text{Stmt}} u}$$

$$(\mathrm{Stmt}_2)\quad \frac{s \vdash \{\ \text{StmtList}\ \} \Rightarrow_{\text{Stmt}} u, \quad s \circ u \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u'}{s \vdash \{\ \text{StmtList Stmt}\ \} \Rightarrow_{\text{Stmt}} u \circ u'}$$

$$(\mathrm{Stmt}_3)\quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash \text{Exp}\texttt{;} \Rightarrow_{\text{Stmt}} u}$$

$$(\mathrm{Stmt}_4)\quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v_E,\ u_E\rangle, \quad s \circ u_E \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u_S}{s \vdash \texttt{if (} \text{Exp} \texttt{)} \text{ Stmt} \Rightarrow_{\text{Stmt}} u_E \circ (v'_E \triangleright u_S)} \qquad v'_E = ItoB(v_E)$$

$$(\mathrm{Stmt}_5)\quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v_E,\ u_E\rangle, \quad s' \vdash \text{Stmt}_1 \Rightarrow_{\text{Stmt}} u_{S_1}, \quad s' \vdash \text{Stmt}_2 \Rightarrow_{\text{Stmt}} u_{S_2}}{s \vdash \texttt{if (} \text{Exp} \texttt{)} \text{ Stmt}_1 \texttt{ else } \text{Stmt}_2 \Rightarrow_{\text{Stmt}} u_E \circ (v'_E \triangleright u_{S_1}) \circ (\neg v'_E \triangleright u_{S_2})} \qquad \begin{array}{l} s' = s \circ u_E \\ v'_E = ItoB(v_E) \end{array}$$

$$(\mathrm{Stmt}_6)\quad \frac{x_{\mathcal{S}} \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v_E,\ u_E\rangle, \quad x_{\mathcal{S}} \circ u_E \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u_S}{s \vdash \texttt{while (} \text{Exp}\texttt{)} \text{ Stmt} \Rightarrow_{\text{Stmt}} ((\mathbf{Y}\ (\lambda f.\lambda x_{\mathcal{S}}.\ (u_E \circ_s (v_E \triangleright_s (u_S \circ_s (f\ (x_{\mathcal{S}} \circ_s u_E \circ_s u_S)))))))\quad s)}$$

$$(\mathrm{Exp}_1)\quad \frac{s \vdash \text{Exp}_p \Rightarrow_{\text{Exp}_p} v}{s \vdash \text{Exp}_p \Rightarrow_{\text{Exp}} \langle v,\ \emptyset_s\rangle}$$

$$(\mathrm{Exp}_2)\quad \frac{s \vdash \text{Exp}_i \Rightarrow_{\text{Exp}_i} \langle v,\ u\rangle}{s \vdash \text{Exp}_i \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}$$

$$(\mathrm{Exp}_{p_1})\quad s \vdash \text{Id} \Rightarrow_{\text{Exp}_p} s\ @\ \text{addr(Id)}\ !$$

$$(\mathrm{Exp}_{p_2})\quad s \vdash \text{IntLiteral} \Rightarrow_{\text{Exp}_p} \text{IntLiteral}$$

$$(\mathrm{Exp}_{p_3})\quad s \vdash \texttt{?}\text{Id} \Rightarrow_{\text{Exp}_p} \text{meta(Id)}$$

$$(\mathrm{Exp}_{i_1})\quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash \texttt{*}\,\text{Exp} \Rightarrow_{\text{Exp}_i} \langle (s \circ u)\ @\ v\ !,\ u\rangle}$$

Figure 25. Translation rules for μC, Part I.

$$(\mathrm{Exp}_{i2}) \quad \frac{s \vdash \text{ LValue } \Rightarrow_{\mathrm{LValue}} \langle v,\ u\rangle}{s \vdash \texttt{\&}\,\text{LValue } \Rightarrow_{\mathrm{Exp}_i} \langle v,\ u\rangle}$$

$$(\mathrm{Exp}_{i3}) \quad \frac{s \vdash \text{ Exp } \Rightarrow_{\mathrm{Exp}_i} \langle v_E,\ u_E\rangle, \quad s \circ u_E \vdash \text{ LValue } \Rightarrow_{\mathrm{LValue}} \langle v_L,\ u_L\rangle}{s \vdash \text{LValue } \texttt{=} \text{ Exp } \Rightarrow_{\mathrm{Exp}_i} \langle v_E,\ u_E \circ u_L \circ \{v_L \mapsto [v_E]\}\rangle}$$

$$(\mathrm{Exp}_{i4}) \quad \frac{s \vdash \text{ Exp } \Rightarrow_{\mathrm{Exp}_i} \langle v_E,\ u_E\rangle, \quad s \circ u_E \vdash \text{ LValue } \Rightarrow_{\mathrm{LValue}} \langle v_L,\ u_L\rangle}{s \vdash \text{LValue } \texttt{-=} \text{ Exp } \Rightarrow_{\mathrm{Exp}_i} \langle v',\ u' \circ \{v_L \mapsto [v']\}\rangle} \quad \begin{array}{c} u' = u_E \circ u_L \\ v' = \mathrm{intDiff}((s \circ u')\ @\ v_L\ !\ ,\ v_E) \end{array}$$

$$(\mathrm{Exp}_{i5}) \quad \frac{s \vdash \text{ Exp}_1 \Rightarrow_{\mathrm{Exp}_i} \langle v_1,\ u_1\rangle, \quad s \circ u_1 \vdash \text{ Exp}_2 \Rightarrow_{\mathrm{Exp}_i} \langle v_2,\ u_2\rangle}{s \vdash \text{Exp}_1 \texttt{ + } \text{Exp}_2 \Rightarrow_{\mathrm{Exp}_i} \langle \mathrm{intSum}(v_1\ ,\ v_2),\ u_1 \circ u_2\rangle}$$

$$(\mathrm{Exp}_{i6}) \quad \frac{s \vdash \text{ Exp } \Rightarrow_{\mathrm{Exp}} \langle v,\ u\rangle}{s \vdash \texttt{ ! } \text{Exp } \Rightarrow_{\mathrm{Exp}_i} \langle BtoI(\neg ItoB(v)),\ u\rangle}$$

$$(\mathrm{Exp}_{i7}) \quad \frac{s \vdash \text{ Exp}_1 \Rightarrow_{\mathrm{Exp}_i} \langle v_1,\ u_1\rangle, \quad s \circ u_1 \vdash \text{ Exp}_2 \Rightarrow_{\mathrm{Exp}_i} \langle v_2,\ u_2\rangle}{s \vdash \text{Exp}_1 \texttt{ || } \text{Exp}_2 \Rightarrow_{\mathrm{Exp}_i} \langle BtoI(v_1' \vee v_2'),\ u_1 \circ (\neg v_1' \rhd u_2)\rangle} \quad \begin{array}{l} v_1' = ItoB(v_1) \\ v_2' = ItoB(v_2) \end{array}$$

$$(\mathrm{LValue}_1) \quad \frac{s \vdash \text{ LValue}_p \Rightarrow_{\mathrm{LValue}_p} a}{s \vdash \text{ LValue}_p \Rightarrow_{\mathrm{LValue}} \langle a,\ \emptyset_s\rangle}$$

$$(\mathrm{LValue}_2) \quad \frac{s \vdash \text{ LValue}_i \Rightarrow_{\mathrm{LValue}_i} \langle a,\ u\rangle}{s \vdash \text{ LValue}_i \Rightarrow_{\mathrm{LValue}} \langle a,\ u\rangle}$$

$$(\mathrm{LValue}_{p1}) \quad s \vdash \text{ Id } \Rightarrow_{\mathrm{LValue}_p} \mathrm{addr}(\mathrm{Id})$$

$$(\mathrm{LValue}_{i1}) \quad \frac{s \vdash \text{ Exp } \Rightarrow_{\mathrm{Exp}_i} \langle a,\ u\rangle}{s \vdash \texttt{ * } \text{Exp } \Rightarrow_{\mathrm{LValue}_i} \langle a,\ u\rangle}$$

where:

$$ItoB(v) \equiv \neg(\mathrm{intEq}(v\ ,\ 0))$$
$$BtoI(v) \equiv ([0] \circ (v \rhd [1]))!$$

Figure 26. Translation rules for μC, Part II.

μC syntactic components. These sequent forms are as follows:

$$
\begin{array}{llll}
s \vdash & c & \Rightarrow_{\text{Pgm}} & u \\
s \vdash & c & \Rightarrow_{\text{Stmt}} & u \\
\\
s \vdash & c & \Rightarrow_{\text{Exp}} & \langle v,\ u\rangle \\
s \vdash & c & \Rightarrow_{\text{Exp}_i} & \langle v,\ u\rangle \\
s \vdash & c & \Rightarrow_{\text{Exp}_p} & v \\
\\
s \vdash & c & \Rightarrow_{\text{LValue}} & \langle a,\ u\rangle \\
s \vdash & c & \Rightarrow_{\text{LValue}_i} & \langle a,\ u\rangle \\
s \vdash & c & \Rightarrow_{\text{LValue}_p} & a
\end{array}
$$

Each of the sequents above takes a μC construct c and an incoming PIM store s, and yields a PIM term or a pair[13] of PIM terms, depending on the nature of the μC construct being translated. Pure expressions (those having no side-effects) and impure expressions are distinguished in the translation process; subscripts p and i are used to denote the two types. Details of the sequent types are as follows:

—Sequents with '$\Rightarrow_{\text{Stmt}}$' are used to translate statements. Sequents with '$\Rightarrow_{\text{Pgm}}$' are used to translate entire programs. Both yield a PIM store term u corresponding to the cumulative effect of *updates* to the store made by the statement or program.

—Sequents with '$\Rightarrow_{\text{Exp}_p}$' are used to translate *pure* expressions computing ordinary values. Sequents with '$\Rightarrow_{\text{LValue}_p}$' are used to translate pure expressions computing L-values (addresses). An instance of the former yields a PIM base value term v corresponding to the expression's value; an instance of the latter yields a PIM address term a corresponding to the expression's L-value.

—Sequents with '$\Rightarrow_{\text{Exp}_i}$' are used to translate *impure* expressions computing ordinary values. Sequents with '$\Rightarrow_{\text{LValue}_i}$' are used to translate impure expressions computing L-values. An instance of the former yields a pair $\langle v,\ u\rangle$ consisting of the PIM base value term v corresponding to the expression's value and the PIM store term u corresponding to the expression's side effects. An instance of the latter yields a pair $\langle a,\ u\rangle$ consisting of the PIM address term a corresponding to the expression's L-value and the store term u corresponding to the expression's side effects.

—Sequents with '$\Rightarrow_{\text{Exp}}$' or '$\Rightarrow_{\text{LValue}}$' are used to translate arbitrary ordinary or L-valued expressions. They yield pairs of the form $\langle v,\ u\rangle$ or $\langle a,\ u\rangle$, respectively. Rules using these sequents simply choose the appropriate pure or impure sequents, depending on the type of construct being translated.

As an example of how the translation process works, consider rule (E_{u5}) in Fig. 26. This rule may be read as follows: Given μC expression Exp_1 + Exp_2 and incoming PIM store s, first translate Exp_1 to the pair $\langle v_1,\ u_1\rangle$ using initial store s. Term v_1 represents the value of Exp_1, and term u_1 represents the side effects occurring in Exp_1. Next, translate Exp_2 in an initial store given by the *composition* of store s and store u_1, yielding pair $\langle v_2,\ u_2\rangle$. This means that any side effect occurring in Exp_1 is accounted for in the store used to translate Exp_2, thus effectively

[13]The pair constructor $\langle\cdot,\ \cdot\rangle$ is an auxiliary symbol used only during the translation process; it is not itself a function symbol of PIM.

encoding a left-to-right order of evaluation. Finally, the PIM term corresponding to the entire expression Exp_1 + Exp_2 is given by the pair $\langle \text{intSum}(v_1 , v_2),\ u_1 \circ u_2 \rangle$. The term $\text{intSum}(v_1 , v_2)$ corresponds to the sum of v_1 and v_2, and the term $u_1 \circ u_2$ is the PIM store corresponding to the cumulative side effects occurring in both Exp_1 and Exp_2.

A2 PIM Terms to PIM Graphs

The translation given in Figures 25 and 26 takes a μC program and produces a PIM term. To get the PIM *graphs* used in the examples in the main body of the paper, we simply adopt the convention that any term bound to a variable used in a translation rule is *shared* if that variable appears more than once in the rule. For example, in the case of rule (E_{u5}), the incoming store s appears twice in the rule's antecedent. If the term bound to s is used in the translation of both Exp_1 and Exp_2 (which would happen, e.g., if both Exp_1 and Exp_2 were identifiers, causing rule (E_{p2}) to be applicable to each), then the term bound to s may be shared.

In almost all cases where multiple instances of the same variable appear, the variable represents a PIM *store*. This should not be surprising, since the store must be "threaded" by the translation rules to every expression that could possibly use it—note, e.g., the extensive sharing of substores that occurs in the PIM graph S_{P_1} of Fig. 2. However, multiple instances of other kinds of variables also appear in the rules, e.g., in rules (S_5) and (E_{u4}).

Although shared subgraphs arise most naturally from the structure of the rules used in the translation, is often also useful to share identical subgraphs generated "serendipitously" during the translation of *unrelated* parts of the μC program. Such sharing is often referred to as *value numbering* in the program optimization literature, and *hash consing* in the functional and symbolic computation literature. However, unlike many IRs used in program optimization, it is always semantically valid to share identical PIM subgraphs, regardless of whether they represent statements or expressions, and regardless of the context in which they are used.

B. ω-Completeness of $\text{PIM}_t^=$—Proof details

In the following we show that the canonical forms given in Section 9 can actually be reached by equational reasoning from $\text{PIM}_t^=$. In two cases (boolean terms with $\asymp$ and unrestricted open store structures) the proof is not based on canonical forms, but proceeds by induction on the number of different address variables in an equation (its "length").

Boolean terms with $\asymp$

It is sufficient to show that the tautologies are provably equal to $\mathbf{T}$. We proceed by induction on the number of (different) address variables N. The case $N = 0$ reduces to that of tautologies not involving $\asymp$ by (A1–2). Assuming the statement holds for tautologies with $\leq N$ address variables, let t be a tautology with address variables

$a_1, \ldots, a_{N+1}$. By bringing it in disjunctive normal form and applying (B16) and (A4–6) we obtain

$$(a_{N+1} \asymp a_i) \wedge t = (a_{N+1} \asymp a_i) \wedge t',$$

where t' is t with a_i substituted for a_{N+1} everywhere ($1 \leq i \leq N$). Hence, since t' is a special case of t with N variables, it is provably equal to $\mathbf{T}$ by assumption, and

$$(a_{N+1} \asymp a_i) \wedge t = (a_{N+1} \asymp a_i) \quad (1 \leq i \leq N) \tag{12}$$

is provable. Without loss of generality we may assume the address constants in t to be $\alpha_1, \ldots, \alpha_m$ ($m \geq 0$). As in the previous case,

$$(a_{N+1} \asymp \alpha_i) \wedge t = (a_{N+1} \asymp \alpha_i) \quad (1 \leq i \leq m) \tag{13}$$

is provable. Let

$$\Pi = \bigvee_{i=1}^{m} (a_{N+1} \asymp \alpha_i) \vee \bigvee_{i=1}^{N} (a_{N+1} \asymp a_i). \tag{14}$$

By (B16), (12), and (13)

$$\Pi \wedge t = \Pi. \tag{15}$$

Next, consider $\neg\Pi \wedge t$. By bringing t in disjunctive normal form and using

$$\neg\Pi = \bigwedge_{i=1}^{m} \neg(a_{N+1} \asymp \alpha_i) \wedge \bigwedge_{i=1}^{N} \neg(a_{N+1} \asymp a_i) \tag{16}$$

we obtain

$$\neg\Pi \wedge t = \neg\Pi \wedge t', \tag{17}$$

where t' does not contain a_{N+1}. Suppose $\sigma(t') = \mathbf{F}$ for some valuation σ of $a_1, \ldots, a_N$. Since the number of different address constants is infinite, we can extend σ to a valuation σ^* of $a_1, \ldots, a_{N+1}$ such that $\sigma^*(\neg\Pi) = \mathbf{T}$. Since $\sigma^*(t) = \mathbf{T}$ by assumption, this contradicts (17). Hence, t' is a tautology and since it contains only N address variables it is provably equal to $\mathbf{T}$ by assumption. Hence

$$\neg\Pi \wedge t = \neg\Pi \tag{18}$$

is provable. By combining (15) and (18) we obtain

$$t = (\Pi \vee \neg\Pi) \wedge t = (\Pi \wedge t) \vee (\neg\Pi \wedge t) = \Pi \vee \neg\Pi = \mathbf{T}.$$

This completes the proof.

Open merge structures with $\asymp$ but without @ or !

These can be brought in canonical form (OMCF) (Fig. 13) as follows:

(1) Move guards inward by (L7), merge multiple guards by (L8), and add missing guards by (L5). The resulting merge structure has elements $P \rhd_m [V]$ and $P \rhd_m m$ with V a variable or constant of sort $\mathcal{V}$ and m a (single) variable of sort $\mathcal{M}$.

(2) Move elements $P \rhd_m [V]$ to the left by (7) followed by as many applications of (9) as needed. Further applications of (7), (9) and (L11) and normalization of the resulting mutually exclusive guards $P_1, \ldots, P_k$ ($k \geq 0$) yields a merge structure satisfying conditions (i) and (ii) of (OMCF). Its tail consists of elements $P \rhd_m m$ as before.

(3) We show that the tail can be brought in canonical form. For a single element $P \rhd_m m$ with m a single variable of sort $\mathcal{M}$ this is immediate. Assuming it is true for N elements, a merge structure $(P \rhd_m m) \circ_m \cdots$ with $N+1$ elements can be brought in the form

$$(P \rhd_m m) \circ_m (Q_1 \rhd_m M_1) \circ_m \cdots \circ_m (Q_n \rhd_m M_n),$$

where m is a single variable of sort $\mathcal{M}$ as before, and the tail $(Q_1 \rhd_m M_1) \circ_m \cdots$ is the canonical form of the last N elements. In particular, $Q_i \wedge Q_j = \mathbf{F}$ and $M_i \neq M_j$ $(i \neq j)$. There are two cases:
(a) m does not occur in any M_i. Let $P_1 = P \wedge \neg Q_1$, $Q_1' = Q_1 \wedge P$, $Q_1'' = Q_1 \wedge \neg P$. We have

$$\begin{aligned} &(P \rhd_m m) \circ_m (Q_1 \rhd_m M_1) = \\ &= \quad ((P_1 \vee Q_1') \rhd_m m) \circ_m ((Q_1' \vee Q_1'') \rhd_m M_1) \\ &\stackrel{(L11)}{=} (P_1 \rhd_m m) \circ_m (Q_1' \rhd_m m) \circ_m (Q_1' \rhd_m M_1) \circ_m (Q_1'' \rhd_m M_1) \\ &\stackrel{(L7)}{=} (P_1 \rhd_m m) \circ_m (Q_1' \rhd_m (m \circ_m M_1)) \circ_m (Q_1'' \rhd_m M_1). \end{aligned}$$

Since P_1, Q_1', Q_1'' are mutually exclusive, $P_1 \rhd_m m$ can be moved to the right of $Q_1'' \rhd_m M_1$ by (9). Next, repeat the above step for

$$(P_1 \rhd_m m) \circ_m (Q_2 \rhd_m M_2)$$

and so on until

$$(P_{n-1} \rhd_m m) \circ_m (Q_n \rhd_m M_n).$$

After normalizing the guards and dropping any element whose guard is $\mathbf{F}$, the result is in canonical form.
(b) m occurs in at least one M_i. First, use (L7) and (8) to replace $P \rhd_m m$ with $P' \rhd_m m$ where $P' = P \wedge \neg Q \wedge \neg Q' \wedge \ldots$ for all elements $Q \rhd_m M$, $Q' \rhd_m M', \ldots$ such that m occurs in M, $M', \ldots$. Move these elements to the left of $P' \rhd_m m$ by repeated application of (9). Next, apply (a) to bring the tail $(P' \rhd_m m) \circ_m \cdots$ in canonical form. Finally, merge any elements $Q \rhd_m M$ and $Q' \rhd_m M'$ with $M = M'$ using (9) and (L11).

Unrestricted open merge structures

These can be brought in canonical form (OMCFgen) (Fig. 14) as follows.

(1) Subterms containing ! are of the form $[M!]$ for some merge structure M. Eliminate them by means of (M7).

(2) The resulting term is flattened using the distributive law (S4) and replacing any dereferenced store cells $(P \rhd_s \{A \mapsto M\})$ @ A' with $P \wedge (A \asymp A') \rhd_m M$ by (S11), (S1), (L8). Dereferenced variables $(P \rhd_s s)$ @ A with s a variable of sort $\mathcal{S}$ and A an address constant or variable, can be replaced by $P \rhd_m (s \,@\, A)$ by (S11). Any remaining compound variables $s \,@\, A$ cannot be eliminated but are similar to ordinary variables of sort $\mathcal{M}$ except that the address component A is subject to the substitution law

$$\begin{aligned} &(a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \,@\, a_1)) \circ_m m_2) = \\ &= (a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \,@\, a_2)) \circ_m m_2), \end{aligned} \tag{19}$$

which is a consequence of (L7–8), (S10). Two compound variables s @ A and s' @ A' are different if $s \not\equiv s'$ or $A \neq A'$ (modulo (19)). Hence, the previous canonical form (OMCF) is still applicable in the slightly generalized form (OMCFgen).

Open terms of sort $\mathcal{V}$

These can be brought in canonical form (OVCF) (Fig. 15). First bring them in the form $M!$ with M in canonical form (OMCFgen). If M has a subterm $P \rhd_m [?]$ move it to the leftmost position by repeated application of (9), and eliminate it with (M8).

Open store structures without @ or $\asymp$ and without variables of sort $\mathcal{S}$

These can be brought in canonical form (OSCF) (Fig. 16) as follows:

(1) Eliminate any $\rhd_s$-operators by moving them into the store cells by (L8), (L7), (S5).

(2) We show that the resulting sequence of unguarded store cells can be brought in canonical form by induction on the number N of (different) address variables in it. If $N = 0$, all addresses are known and (OSCF) reduces to canonical form (OSCFsimple) (Fig. 17). Apart from the normalization of the merge structure components, the latter can be reached by (S7) and (S6).
Assuming store structures with $\leq N$ address variables can be brought in canonical form (OSCF), let S be a store structure in canonical form with address variables $a_1, \ldots, a_N$ and address constants $\alpha_1, \ldots, \alpha_m$. Let Π and $\neg\Pi$ be given by respectively (14) and (16). We have

$$\begin{aligned} & S \circ_s \{a_{N+1} \mapsto M_{N+1}\} = \\ & \quad = \quad S \circ_s ((\Pi \vee \neg\Pi) \rhd_s \{a_{N+1} \mapsto M_{N+1}\}) \\ & \overset{\text{(L11)}}{=} \quad S \circ_s (\Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\}) \circ_s (\neg\Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\}). \end{aligned} \tag{20}$$

In view of (16), $\neg\Pi$ is already in the conjunctive form required by (iii) of (OSCF), so the last store cell of (20) is already in the required form apart from straightforward normalization of $\neg\Pi$ and M_{N+1}. For the next to last cell we have

$$\begin{aligned} & \Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\} = \\ & \overset{(14)(\text{L11})}{=} \quad (a_{N+1} \asymp \alpha_1) \rhd_s \{a_{N+1} \mapsto M_{N+1}\} \circ_s \cdots \\ & \qquad\qquad \cdots \circ_s (a_{N+1} \asymp a_N) \rhd_s \{a_{N+1} \mapsto M_{N+1}\} \\ & \overset{(\text{S9})(\text{A4})}{=} \quad (\alpha_1 \asymp a_{N+1}) \rhd_s \{\alpha_1 \mapsto M_{N+1}\} \circ_s \cdots \\ & \qquad\qquad \cdots \circ_s (a_N \asymp a_{N+1}) \rhd_s \{a_N \mapsto M_{N+1}\}. \end{aligned} \tag{21}$$

Consider a single guarded store cell $(a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{N+1}\}$ in the right-hand side of (21). Since S is in canonical form, unconditional commutativity applies and $S = S_1 \circ_s S_2$, where

$$S_2 = (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \tag{22}$$

consists of all guarded store cells with address component a_i (modulo (S9)). By requirements (iv) and (v), $\Pi_{j_\lambda} \wedge \Pi_{j_\mu} = \mathbf{F}$ $(\lambda \neq \mu)$ and $\bigvee_{\lambda=1}^{k} \Pi_{j_\lambda} = \mathbf{T}$. Hence,

$$\begin{aligned} & S \circ_s ((a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{N+1}\}) = \\ & = S_1 \circ_s S_2 \circ_s ((a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{N+1}\}) \\ & = S_1 \circ_s \\ & \quad (\neg(a_i \asymp a_{N+1}) \rhd_s S_2) \circ_s \\ & \quad ((a_i \asymp a_{N+1}) \rhd_s (S_2 \circ_s \{a_i \mapsto M_{N+1}\})). \end{aligned} \tag{23}$$

For the last element of (23) we obtain

$$\begin{aligned} & S_2 \circ_s \{a_i \mapsto M_{N+1}\} = \\ & \overset{(22)(\text{L5})}{=} (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \circ_s \\ & \quad (\mathbf{T} \rhd_s \{a_i \mapsto M_{N+1}\}) \\ & \overset{(\text{v})(\text{L11})}{=} (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \circ_s \\ & \quad (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{N+1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{N+1}\}) \\ & \overset{(\text{iv})(\text{S6})}{=} (\Pi_{j_1} \rhd_s \{a_i \mapsto (M_{j_1} \circ_s M_{N+1})\}) \circ_s \cdots \\ & \quad \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto (M_{j_k} \circ_s M_{N+1})\}). \end{aligned}$$

Hence, by (23) and (L8) any guarded store cell $\Pi_{j_\lambda} \rhd_s \{a_i \mapsto M_{j_\lambda}\}$ $(1 \leq \lambda \leq k)$ of S_2 gives rise to two new ones, namely,

$$\Pi_{j_\lambda} \wedge \neg(a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{j_\lambda}\}$$

and

$$\Pi_{j_\lambda} \wedge (a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto (M_{j_\lambda} \circ_s M_{N+1})\}.$$

Repeating this for all elements of Π and substituting in the right-hand side of (20) yields the required canonical form (OSCF).

Open store structures with @ and $\asymp$ and with variables of sort $\mathcal{S}$, but without variables of sort $\mathcal{A}$

These can be brought in canonical form (OSCFvar) (Fig. 18) as follows:

(1) Move variables of sort $\mathcal{S}$ to the left by repeated application of (S12). The resulting store structure consists of a sequence Σ' of (possibly guarded) variables of sort $\mathcal{S}$ followed by a sequence of (possibly guarded) store cells Σ''. Bring Σ' in the equivalent of canonical form (OMCF) with $k = 0$ by using equations (Ln) and (8) with $\rho = s$ rather than $\rho = m$.

(2) Use (S11) to replace any instances of $\rhd_s$ with $\rhd_m$ in Σ'' and bring the resulting sequence of store cells in the form required by (ii).

(3) Suppose (iii) is not satisfied. Move any pair of offending items in adjacent positions using the commutative laws (9) with $\rho = s$ for the store variable part, and (S7) and (9) with $\rho = m$ for the store cell part. Apply

$$\begin{aligned} & (p \rhd_s s) \circ_s \{a \mapsto (q \rhd_m (s \mathbin{@} a))\} = \\ & = \quad ((p \wedge \neg q) \rhd_s s) \circ_s ((p \wedge q) \rhd_s s) \circ_s \end{aligned}$$

$$
\begin{aligned}
& \{a \mapsto (((p \wedge q) \rhd_s s) @ a)\} \circ_s \{a \mapsto ((\neg p \wedge q) \rhd_m (s @ a))\} \\
\stackrel{(S12)}{=} \; & ((p \wedge \neg q) \rhd_s s) \circ_s ((p \wedge q) \rhd_s s) \circ_s \{a \mapsto ((\neg p \wedge q) \rhd_m (s @ a))\} \\
= \; & (p \rhd_s s) \circ_s \{a \mapsto ((\neg p \wedge q) \rhd_m (s @ a))\}.
\end{aligned}
$$

(4) Repeat steps (1)–(3) until both parts are in the required form and requirement (iii) is satisfied.

Unrestricted open store structures

The proof is similar to that of boolean terms with $\asymp$, and proceeds by induction on the number N of (different) address variables. The case $N = 0$ (no address variables) corresponds to the previous case. Assuming the statement holds for equations with $\leq N$ address variables, let $t_1 = t_2$ be a valid equation with $N + 1$ address variables $a_1, \ldots, a_{N+1}$. Let Π and $\neg\Pi$ be given by respectively (14) and (16). First, consider the equation $\Pi \rhd_s t_1 = \Pi \rhd_s t_2$. It is valid in $I(\text{PIM}_t^+)$ and

$$
\begin{aligned}
\Pi \rhd_s t_i \stackrel{(14)(L11)}{=} \; & ((a_{N+1} \asymp \alpha_1) \rhd_s t_i) \circ_s \cdots \circ_s ((a_{N+1} \asymp a_N) \rhd_s t_i) \\
= \; & ((a_{N+1} \asymp \alpha_1) \rhd_s t_{i,1}) \circ_s \cdots \circ_s ((a_{N+1} \asymp a_N) \rhd_s t_{i,m+N}),
\end{aligned}
$$

such that α_{N+1} has been eliminated from $t_{i,j}$ ($i = 1, 2,\ 1 \leq j \leq m + N$) by the substitution laws (A5–6), (S9–10) in conjunction with the guard propagation laws (L7–8), (S5), (S11). Hence, the equations $t_{1,j} = t_{2,j}$ are provable by assumption and $\Pi \rhd_s t_1 = \Pi \rhd_s t_2$ is provable.

Next, consider the equation $\neg\Pi \rhd_s t_1 = \neg\Pi \rhd_s t_2$. It is valid in $I(\text{PIM}_t^+)$. By bringing t_1 and t_2 in flattened form (see above) and using (16) we obtain $\neg\Pi \rhd_s t_i = \neg\Pi \rhd_s t'_i$ ($i = 1, 2$), where the guards in t'_i no longer contain a_{N+1} although store cells $\{a_{N+1} \mapsto M\}$ and compound variables $s @ a_{N+1}$ are retained.

Suppose the equation $\neg\Pi \rhd_s t'_1 = \neg\Pi \rhd_s t'_2$ is not provable. Replace a_{N+1} by an address constant $\beta \neq \alpha_k$ ($1 \leq k \leq m$). Let

$$
\Gamma = (\neg\Pi)[a_{N+1} := \beta] = \bigwedge_{i=1}^{N} \neg(a_i \asymp \beta)
$$

and

$$
t''_i = t'_i[a_{N+1} := \beta].
$$

Then the equation $\Gamma \rhd_s t''_1 = \Gamma \rhd_s t''_2$ is not provable either since it does not allow additional derivation steps in comparison with $\neg\Pi \rhd_s t'_1 = \neg\Pi \rhd_s t'_2$. But since it is a valid equation with N variables this is a contradiction. Hence, $\neg\Pi \rhd_s t'_1 = \neg\Pi \rhd_s t'_2$ is provable, and $\neg\Pi \rhd_s t_1 = \neg\Pi \rhd_s t_2$ is provable as well. This completes the proof.