Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A complete transformational toolkit for compilers

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

Computer Science/Department of Software Technology

CS-R9601 1996

Report CS-R9601
ISSN 0169-118X

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

A COMPLETE TRANSFORMATIONAL TOOLKIT FOR COMPILERS

J.A. Bergstra
Faculty of Mathematics and Computer Science, University of Amsterdam[*]

T.B. Dinesh
Department of Software Technology, CWI[†]

J. Field
IBM T.J. Watson Research Center[‡]

J. Heering
Department of Software Technology, CWI[§]

Abstract

In an earlier paper, one of the present authors presented a preliminary account of an equational logic called PIM. PIM is intended to function as a "transformational toolkit" to be used by compilers and analysis tools for imperative languages, and 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 rewriting system that characterizes the behavior of lazy stores and generalized conditionals. A major question left open in the earlier paper was whether there existed a *complete* equational axiomatization of PIM's semantics. In this paper, we answer this question in the affirmative for PIM's core algebraic component, PIM_t, under the assumption of certain reasonable restrictions on term formation. We systematically derive the complete PIM logic as the culmination of a sequence of increasingly powerful equational systems starting from a straightforward "interpreter" for closed PIM terms.

CR Subject Classification (1991): D.3.4, I.2.2, F.3.2.
Keywords & Phrases (1991): code optimization, program transformation, algebraic semantics.
Note: An abridged version of this paper has been accepted for the 1996 European Symposium on Programming, and will be published in the ESOP '96 proceedings (which will appear as a volume of Springer-Verlag's Lecture Notes in Computer Science series). This work was supported in part by the European Communities under ESPRIT Basic Research Action 7166 (CONCUR II) and the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project.

[*] Kruislaan 403, 1098 SJ Amsterdam, The Netherlands; e-mail: janb@fwi.uva.nl
[†] Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; e-mail: T.B.Dinesh@cwi.nl
[‡] P.O. Box 704, Yorktown Heights, NY 10598, USA; e-mail: jfield@watson.ibm.com
[§] Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; e-mail: Jan.Heering@cwi.nl

1 Introduction

In an earlier paper [18], one of the present authors presented a preliminary account of an equational logic called PIM. PIM is intended to function as a "transformational toolkit" to be used by compilers and analysis tools for imperative languages. In a nutshell, PIM consists of the untyped lambda calculus extended with an algebraic rewriting system that characterizes the behavior of lazy stores [9] and generalized conditionals. Together, these constructs are sufficient to model the principal dynamic semantic elements of most Algol-class languages.

Translation of programs in most such languages to PIM is straightforward; programs can then be formally manipulated by reasoning about their PIM analogues. Moreover, the graph representations of various PIM normal forms can be manipulated in a manner similar to intermediate representations commonly used in optimizing compilers. As with other intermediate representations, the PIM form of the program can serve as a more suitable starting point for subsequent analysis than the program text itself. Unlike other such representations, however, PIM has a strong formal foundation and permits a number of different semantics-preserving manipulations to be carried out in the same framework.

A major question left open in [18] was whether there existed a *complete* equational axiomatization of PIM's semantics. In this paper, we answer this question in the affirmative for PIM's core algebraic component, PIM_t, under the assumption of certain reasonable restrictions on term formation. Formally, we show that there exists an ω-complete equational axiomatization of PIM's final algebra semantics. Obtaining a positive answer to the completeness question is, we believe, quite important, since it means that we can be assured that our transformational toolkit has an adequate supply of tools. In [18], 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 arbitrary programs were present (with or without restrictions on term formation). Obtaining a complete equational axiomatization for PIM is also an important prerequisite to designing a decision procedure.

We are aware of only a few prior completeness results for logics for imperative languages: Mason and Talcott [32] show that their logic for reasoning about equivalence in a Lisp-like (rather than Algol-class) language is complete; however, unlike PIM, their logic is a sequent calculus, rather than an equational system. Hoare et al. [24] present a partial completeness result for an equational logic; however, their result does not hold for the cases where addresses or stores are unknowns, i.e., can be represented by variables. For program transformations commonly required in optimizing compilers, it is extremely important that these latter cases be addressed.

Although the merits of equational reasoning for programs have been widely touted [2, 26, 35, 24, 27], results characterizing the power of formal systems required for such reasoning appear to be few and far between. Similarly, while partial evaluation has often been advocated as a means for optimizing programs [14, 15, 10, 7, 38], it is unusual to find a precise characterization of the transformational capabilities that a particular partial evaluator implements.

In the sequel, we systematically derive the complete PIM logic as the culmination of a sequence of increasingly powerful equational systems starting from a straightforward "interpreter" for PIM's term language. While this approach to developing a logic is not new, it appears to be undertaken only

rarely. We therefore believe that the exposition of the process, especially for a nontrivial program logic such as PIM, is itself a useful contribution.

The remainder of this paper proceeds as follows: In Section 2, we give an overview of PIM and discuss related work. In Section 3, we introduce the system PIM_t, which axiomatizes the 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 discuss the behavior of PIM_t's functions using examples written in a simple programming language. Section 4 gives an overview using examples of the relative power of the equational axiomatization of PIM_t's initial algebra semantics (PIM_t^0), the initial algebra axiomatization of PIM_t's final algebra semantics (PIM_t^+), and an ω-complete enrichment of PIM_t^+, $\text{PIM}_t^=$. Section 5 discusses the relationship between ω-complete equational systems and partial evaluation. In Section 6, we make 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 7, 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 8, we produce our main result: an *ω-complete* axiomatization $\text{PIM}_t^=$ for 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 9, 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 properties. We then develop several rewriting systems based on $\text{PIM}_t^=$ and prove that they are confluent and terminating. Finally, Section 10 covers possible extensions and open problems.

2 PIM in Perspective

While there has been considerable work on calculi and logics of program equivalence for imperative languages, our work has the following points of departure:

- A graph form of PIM is by design closely related to popular intermediate representations (IRs) used in optimizing compilers, such as the PDG [17], SSA form [12], GSA form [3], the PRG [44], the VDG [41], and the representation of Click [11]. Indeed, PIM can be regarded as a rational reconstruction of elements of the earlier IRs. With the exception of the VDG and Click's representation, PIM differs from the other IRs in that procedures, functions, and computations on addresses (e.g., as required for arrays or pointers) are "first-class" features of the formalism. What sets PIM apart from other IR work, however, is not so much the form of its graphs, but rather the existence of a logic for reasoning about equivalences, carrying out partial evaluation, and performing analysis.

- 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 [3], various purely 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. We know of no other formal system that has this property.

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 [1]), or by using a continuation-passing transformation (e.g., one similar to that used in [41]) in which unstructured transfers of control such as `gotos` or `breaks` 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.

- PIM is an *equational logic*, rather than, e.g., a sequent calculus such as that of Mason and Talcott [32]. This is not simply an immaterial stylistic difference. A purely equational logic has the advantage that it can be used 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). Equational logics are also particularly amenable to mechanical implementation, using such techniques as graph rewriting [4], unification, and equational unification.

- Unlike work on calculi for reasoning about imperative features in otherwise functional languages [16, 40, 39], PIM has a particular affinity for constructs in Algol-class (as opposed to Lisp-like) languages, since it 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.

- Yang, Horwitz, and Reps [43] have presented an algorithm that determines when some pairs of programs are behaviorally equivalent. However, their approach is limited by its reliance on structural properties of the fixed PRG graphs used to represent the programs, and they make no claims of completeness.

- Although the logics of Hoare et al. [24] and Boehm [8] treat Algol-class languages, [24] does not accommodate *computed* addresses arising from pointers and arrays, and neither [8] nor [24] cleanly separates store operations from operations on pure values. The separation of these concerns in PIM means that it is easy to represent a language in which expressions with and without side effects are intermixed in complicated ways (e.g., C). This also means that 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.

In this paper, we will concentrate on the formal properties of first order systems derived from PIM's core algebraic component, PIM_t. In particular, we will be concerned with the relative power of these systems to reason about PIM graphs and the source programs to which they correspond. For further

details on the correspondence between PIM and traditional IR's, see [18]. For an example of a practical application of PIM, see [19], which describes a novel algorithm for program *slicing*. The latter paper also makes use of the full higher-order version of PIM, in which looping and recursive constructs are treated by embedding the core first order algebraic system PIM_t (treated here) in an untyped lambda calculus.

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. 15 of Appendix A. μC has standard C syntax and semantics, with the following exceptions:

- *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 a (read-only) function parameter.
- All data in μC are assumed to be integers or pointers (to integer variables or other pointers).
- It is assumed that no address arithmetic is used[1].

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 *per se*. We use "`const`" in the declarations of meta-variables to emphasize their non-assignable character.

3.2 PIM Terms and Graphs

A directed *term graph* [4] 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 translation, further details of which are given in Appendix A. A term graph may be viewed as a term by traversing it from its root and replacing all shared subgraphs by separate copies of their term representations[2]. 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 [4], shared subgraphs are constructed "dynamically" during the rewriting process.

Parent nodes in PIM term graphs will be depicted *below* their children to emphasize the correspondence between μC program constructs and corresponding PIM subgraphs. This "upside down" orientation also corresponds to the manner in which compiler IR graphs are commonly rendered. In the sequel, only a small number of graph edges will be depicted explicitly, primarily those that are shared. Most other subgraphs will be "flattened" for clarity.

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

[2] Cycles are admissible in term graphs, and correspond to infinite terms. While such cycles arise naturally in conjunction with loops and recursive functions, an acyclic representation suffices for the program constructs considered in this paper.

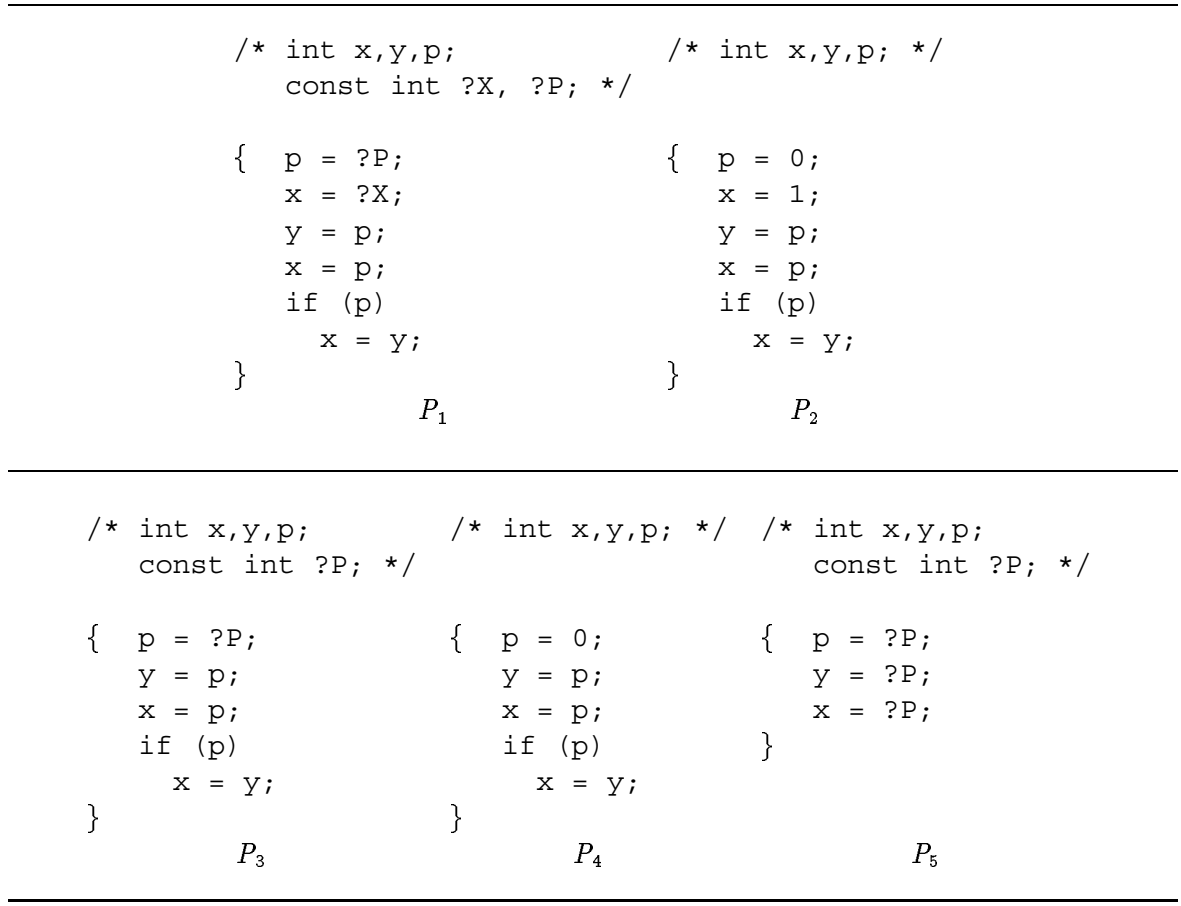

Figure 1: Some simple μC programs.

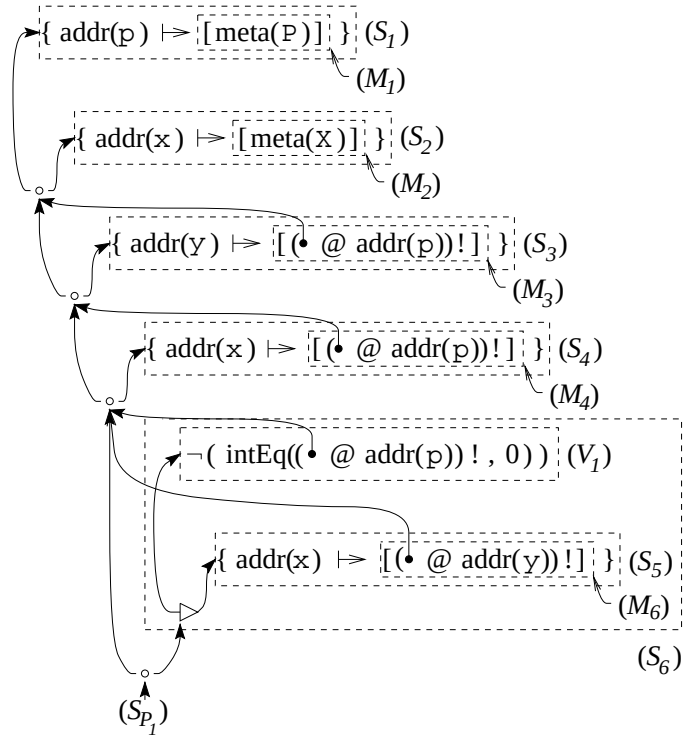

Figure 2: S_{P_1}: The PIM representation of program P_1. Empty stores and subscripts on store composition operators are omitted for clarity.

The properties of the equational systems we consider in this paper are completely independent of whether a tree or graph representation is used for PIM terms. Nonetheless, sharing is quite important in practice. For instance, while the graph representation of a μC program constructed by the translation in Appendix A always has size linear in the size of the program's parse tree, the corresponding *tree* form of the term may have an *exponentially* greater number of nodes (consider, e.g., the size of the tree form of graph S_{P_1} in Fig. 2).

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 [18] and slightly revised in [19] 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 [19] propagate conditional "contexts" inside expressions computing base values or addresses. As shown in [19], PIM's higher-order constructs allow loops (among other things) to be modeled in a straightforward way. In addition, [19] shows 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, the constructs in PIM$_t$ alone are sufficient to model the control- and data-flow aspects of finite programs in Algol-class languages. As we shall see, these constructs have a non-trivial equational axiomatization, and understanding their properties is a necessary prerequisite to studying the properties of higher-order variants.

The signature[3] of PIM$_t$ terms is given in Fig. 3. 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 PIM$_t^=$ remain valid even when the term formation restrictions are dropped[4]. 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[5] 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 $\mathcal{A}$. 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

[3] This signature differs slightly from the corresponding signature in [18]; the differences principally relate to a simplification in the structure of merge expressions.

[4] 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.

[5] 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 [18].

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)

Figure 3: Signature of PIM$_t$ Terms.

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

Figure 4: Equations of PIM_t^0. The equations labeled (L*n*) 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.

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

Figure 5: Signature of C-Specific PIM Extensions.

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.

3.4 PIM's Parts

In the remainder of this section, we briefly 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.

The graph S_{P_1} is a PIM *store structure*[6], an abstract representation of memory. S_{P_1} is constructed from the sequential composition (using the operator '$\circ_s$') of substores corresponding to the statements comprising P_1. 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

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

A store cell associates an *address expression* (here '`addr(p)`') with a *merge structure*, (here [`meta(P)`], where '`meta(P)`' is the translation of the μC meta-variable '`?P`'). 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 empty store. Equations (L1) and (L2) of PIM_t^0 indicate that null stores disappear when composed 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

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

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).

An expression of the form

$$s \mathbin{@} a$$

represents the result of *dereferencing* store s at address a. Examples of such expressions are those contained in the subgraphs labeled (M_3) and (M_4) in S_{P_1}. The result of the dereferencing operation is a merge structure. Unlike an ordinary "lookup" operation which retrieves a single value given some "key", the PIM store dereferencing operator can be thought of as retrieving *all* of the values ever associated with the address at which the store is dereferenced, and amalgamating those results into a merge structure. This retrieval behavior is codified by equations (S1)–(S4), (A1), and (A2), and can be thought of as computing a very conservative initial approximation to all the definitions of a given address that "reach" a particular use. Further simplification of merge expressions that result from a store dereferencing operation can yield a more accurate (and conventional) set of definitions reaching a given use.

The simplest nonempty form of merge expression 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.

Merge structures used in conjunction with the *selection* operator, '!', yield values. When the selection operator is applied to a 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 '?'.

Note in Fig. 2 that the '!' operator is used in the translation of every reference to the value of a variable in program P_1. When the retrieval semantics of the '@' operator are combined with the selection semantics of the '!' operator in an expression of the form

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

the net effect is to

1. Retrieve all the values in store s associated with address a (i.e., assignments to the variable associated with a).

2. Yield the rightmost (i.e., "most recently assigned") value associated with a.

The merge structures depicted in Fig. 2 are all either simple merge cells or dereferenced stores. However, more interesting merge structures are often produced by equational simplification. For

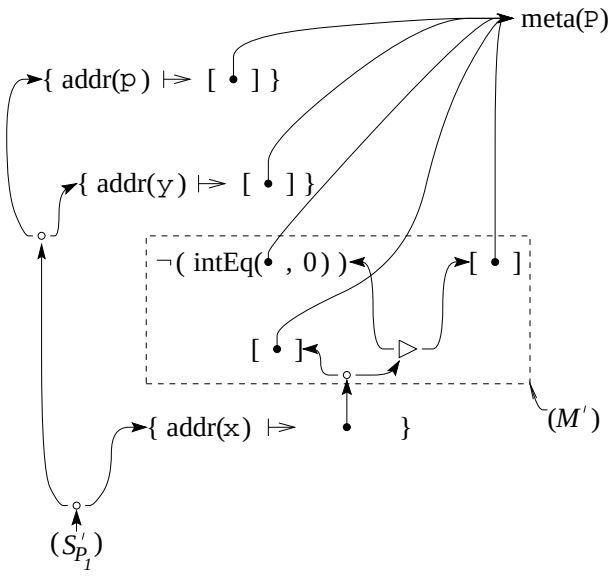

Figure 6: S'_{P_1}: A simplified form of S_{P_1}.

example, it turns out that the store structure S'_{P_1} depicted in Fig. 6 is equivalent to S_{P_1}, a fact provable in the ω-complete system PIM$_t^=$ we develop in the sequel. The merge structure denoted by M' in S'_{P_1} represents all the assignments made to variable x in P_1. M' 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." It should be easy to see that M' can be further simplified to the expression [?P], although the resulting graph does not correspond as closely to a compiler IR as does S'_{P_1}.

4 Reasoning with PIM Terms and Graphs

4.1 Evaluating value-producing PIM expressions

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. Given S_{P_2}, the expression

$$V_{P_2}^{\mathtt{x}} = (S_{P_2} \ @ \ \mathtt{addr(x)})!$$

represents 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. A similar expression can be constructed to compute the final value of any variable in the program (including, if desired, a variable which never receives an initial assignment!).

Since $V_{P_2}^{\mathtt{x}}$ is a closed expression of sort $\mathcal{V}$, we can use the equations of Fig. 4 to evaluate it. A simple interpreter for such expressions may be constructed 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*). The result of normalizing $V_{P_2}^{\mathbf{X}}$ is the constant '0'.

4.2 Reasoning about arbitrary PIM expressions

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 to axiomatize the *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 7.

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 8.

In Section 9, 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 be used to yield normal forms that function in a manner similar to that of a traditional compiler IR, and can also be used to implement partial evaluation for optimization purposes.

5 Partial Evaluation and ω-Completeness

It is often assumed that an operational semantics forms an adequate basis for program optimization and transformation. Unfortunately, many valid program transformations do not result from the application of evaluation rules alone. 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.

It is our view that transformations such as the equation above are best viewed as instances of *partial evaluation*. Unlike some others, we are not concerned with binding-time analysis or self-application [28], but, following [23], simply assert that

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

However, how do we know that we have *enough* rules for performing partial evaluation?

The open equations (equations containing variables, such as the one above) valid in the initial algebra of a specification are not in general equationally derivable, but require stronger rules of inference (such as structural induction) for their proofs. An ω*-complete* specification [23] is one in which all valid open equations may be deduced using only equational reasoning. In our setting, then, finding such 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.

6 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 [33], Meseguer and Goguen's survey [34], and Wirsing's survey [42]. We assume some familiarity on the part of the reader with equational logic and initial algebra semantics.

6.1 ω-Completeness

Definition 1 *An algebraic specification $\mathbf{S} = (\Sigma, E)$ with non-void many-sorted signature Σ, finite 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, especially if it has an A-, AC-, or general E-rewriting capability.
- 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 [23, 31, 5].

In the foregoing we assumed initial algebra semantics. In the case of PIM$_t^0$, which uses final algebra semantics, we give an equivalent initial algebra specification first (Step (A), Section 7).

6.2 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 $\mathcal{S}, \mathcal{T} \in \mathrm{sorts}(\Sigma)$. 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, finite 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 [34, p. 488]. Observable sorts are called *visible* in [34] and *primitive* in [42, Section 5.4].

6.3 Steps Towards a Completeness Result

Our completeness result will require two basic technical steps:

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

(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 $\textsc{Pim}_t^0$ itself.

6.4 A Simple Example

We perform steps (A) and (B) (Section 6.3) for the specification of a stack data type shown in Figure 7.

Step (A) Consider the final model $F_{\mathcal{V}}(\textsc{Stack})$ in the sense of Definition 3 with $O = \{\mathcal{V}\}$. We give an initial algebra specification of it [22, 6]. 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

$$push(a_1, \emptyset) = \emptyset \tag{1}$$

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)
equations	$top(\emptyset)$	$=$	a_1	(St1)
	$top(push(v, s))$	$=$	v	(St2)
	$pop(\emptyset)$	$=$	$\emptyset$	(St3)
	$pop(push(v, s))$	$=$	s	(St4)

Figure 7: 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.[7] 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. This is shown in Appendix B.1.

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

$$push(top(s), pop(s)) = s \tag{3}$$

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. It is shown in Appendix B.2 that $\text{STACK}^= = \text{STACK}^+ + (3)$ is ω-complete.

7 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 8.[8]

[7] 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.

[8] Equation (M2′) in Figure 8 is a special case of (M6) in the preliminary version of $\text{PIM}_t^=$ given in [18, Figure 6]. (S8) subsumes (S6) and (S7) in the preliminary version.

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

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

Figure 8: Additional Equations of PIM_t^+.

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

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

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

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

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

□

(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

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

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

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

Two distinct normal forms of sort $\mathcal{S}$ can be distinguished with respect to $\mathcal{M}$ by a suitable store dereference of the form $\Box$ @ α_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. □

8 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 9. 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.

The ω-completeness proof uses both proof methods mentioned in Section 6.1. 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 in the corresponding theory. 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 is easily verified.

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 Appendix B.2 to show the ω-completeness of $\text{STACK}^=$.

Boolean terms without $\asymp$ The only booleans are **T** and **F**. To see that (B1)–(B19) constitute an ω-complete specification of the booleans, take $n = 2$ in [5, Theorem 3.1]. Suitable canonical forms are the disjunctive normal forms without nonessential variables (variables whose value does not matter) used in the proof.

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.

It is sufficient to show that the tautologies are provably equal to **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 (A5–6) we obtain

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

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

$$(a_{N+1} \asymp a_i) \wedge t = (a_{N+1} \asymp a_i) \quad (1 \leq i \leq N) \tag{6}$$

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

$$(a_{N+1} \asymp \alpha_i) \wedge t = (a_{N+1} \asymp \alpha_i) \quad (1 \leq i \leq m) \tag{7}$$

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

Figure 9: Additional Equations of $\textsc{Pim}_t^=$.

is provable. Let

$$\Pi = \bigvee_{i=1}^{m} (a_{N+1} \asymp \alpha_i) \vee \bigvee_{i=1}^{N} (a_{N+1} \asymp a_i). \tag{8}$$

By (B16), (6), and (7)

$$\Pi \wedge t = \Pi. \tag{9}$$

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

$$\neg\Pi = \bigwedge_{i=1}^{m} \neg(a_{N+1} \asymp \alpha_i) \wedge \bigwedge_{i=1}^{N} \neg(a_{N+1} \asymp a_i) \tag{10}$$

we obtain

$$\neg\Pi \wedge t = \neg\Pi \wedge t', \tag{11}$$

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 (11). Hence, t' is a tautology and since it contains only N address variables it is provably equal to $\mathbf{T}$ by assumption. Hence

$$\neg\Pi \wedge t = \neg\Pi \tag{12}$$

is provable. By combining (9) and (12) 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. 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

$$\cdots \succ a_2 \succ a_1 \succ \alpha_i \quad (i \geq 1). \tag{13}$$

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 [30, p. 38]. The canonical form is determined up to symmetry of $\asymp$ and up to associativity and commutativity of $\vee$ and $\wedge$ as before.

Open merge structures with $\asymp$ but without @ or ! These are similar to the **if**-expressions treated in [23, Section 3.3], but there are some additional complications.

First, we have

$$m \circ_m (p \rhd_m [v]) = (\neg p \rhd_m m) \circ_m (p \rhd_m [v]), \tag{14}$$

since

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

(14) 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

$$(p_1 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) = ((\neg p_2 \wedge p_1) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l), \tag{15}$$

since

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

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

$$p_1 \wedge p_2 = \mathbf{F} \implies (p_1 \rhd_\rho l_1) \circ_\rho (p_2 \rhd_\rho l_2) = (p_2 \rhd_\rho l_2) \circ_\rho (p_1 \rhd_\rho l_1), \tag{16}$$

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

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

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

Suitable canonical forms for open merge structures without @ or ! are $\emptyset_m$ and

$$(P_1 \rhd_m [V_1]) \circ_m \cdots \circ_m (P_k \rhd_m [V_k]) \circ_m (Q_1 \rhd_m M_1) \circ_m \cdots \circ_m (Q_n \rhd_m M_n) \tag{17}$$

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 $M_i \neq M_j$ $(i \neq j)$.

It is easily verified that two such canonical forms are equal in $I(\text{PIM}_i^+)$ if and only if they are syntactically equal modulo associativity and commutativity of $\vee$ and $\wedge$, modulo symmetry of $\asymp$, and modulo associativity and conditional commutativity of $\circ_m$ (equations (L3) and (16) with $\rho = m$). The canonical form is derived 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 (14) followed by as many applications of (16) as needed. Further applications of (14), (16) 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 (17). 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

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

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 (16). 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 (15) 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 (16). 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 (16) and (L11).

Open merge structures with $\asymp$ and @ but without ! These can be flattened by using the distributive law (S4) and replacing any dereferenced store cells $(P \rhd_s \{A \mapsto M\}) \mathbin{@} A'$ with $P \wedge (A \asymp A') \rhd_m M$ by (S11), (S1), (L8). Dereferenced variables $(P \rhd_s s) \mathbin{@} A$ with s a variable of sort $\mathcal{S}$ and A an address constant or variable, can be replaced by $P \rhd_m (s \mathbin{@} A)$ by (S11). Any remaining compound variables $s \mathbin{@} A$ 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

$$\begin{aligned}&(a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \mathbin{@} a_1)) \circ_m m_2) = \\ &\quad = \quad (a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \mathbin{@} a_2)) \circ_m m_2),\end{aligned} \tag{18}$$

which is a consequence of (L7–8), (S10). Two compound variables $s \mathbin{@} A$ and $s' \mathbin{@} A'$ are different if $s \not\equiv s'$ or $A \neq A'$ (modulo (18)). Canonical form (17) is still applicable if requirement (iv) is replaced by

(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 $s \mathbin{@} A$, 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 (13).

Hence, an open merge structure with @ but without ! can be brought in

$$\text{canonical form (17) with (iv′) and (v) instead of (iv).} \tag{19}$$

Open merge structures with ! This is the general case of merge structures. Subterms containing ! are of the form $[M!]$ for some merge structure M. These subterms can be eliminated by means of (M7), which is valid in $I(\text{PIM}_t^+)$ since

$$\begin{aligned}&[\emptyset_m!] = [?] = [?] \circ_m \emptyset_m \\ &[[v]!] = [v] \overset{(\text{M2}')}{=} [?] \circ_m [v].\end{aligned}$$

Hence, merge structures with ! can be brought in canonical form (19).

Open terms of sort $\mathcal{V}$ These can be brought in the form $M!$ with M in canonical form (19). If M has a subterm $P \rhd_m [?]$ move it to the leftmost position by repeated application of (16), and eliminate it with (M8), which is valid in $I(\text{PIM}_t^+)$ since

$$\begin{aligned}&((\mathbf{T} \rhd_m [?]) \circ_m \emptyset_m)! = (\mathbf{T} \rhd_m [?])! = ? = \emptyset_m! \\ &((\mathbf{T} \rhd_m [?]) \circ_m (\mathbf{T} \rhd_m [v]))! \overset{(\text{M2}')}{=} [v]! \\ &((\mathbf{F} \rhd_m [?]) \circ_m m)! = (\emptyset_m \circ_m m)! = m!.\end{aligned}$$

Hence, open terms of sort $\mathcal{V}$ have canonical form

$$M! \quad (M \text{ in canonical form (19) without subterm } P \rhd_m [?]). \tag{20}$$

Open store structures without @ or $\asymp$ and without variables of sort $\mathcal{S}$ We first note the following immediate consequences of (S8):

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

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

$$\begin{aligned} &\neg(a_1 \asymp a_2) \rhd_s (\{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\}) = \\ &\quad = \neg(a_1 \asymp a_2) \rhd_s (\{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\}) \end{aligned} \tag{21}$$

(S7) is a conditional commutative law.[9] (21) is similar but with an appropriate guard rather than a condition.

Suitable canonical forms in this case are $\emptyset_s$ and

$$(\Pi_1 \rhd_s \{A_1 \mapsto M_1\}) \circ_s \cdots \circ_s (\Pi_n \rhd_s \{A_n \mapsto M_n\}) \tag{22}$$

with

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

(ii) M_i a merge structure without $\asymp$ in canonical form (17) $\neq \emptyset_m$

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

$$\bigwedge_{k=1}^{n} \pm(A_i \asymp A_k) \tag{23}$$

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

(vi) The corresponding multiset of address constants and variables is minimal with respect to the ordering (13).

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 (16), (S7), (21) 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. It is derived as follows:

[9] (S6) and (S7) correspond to (S6) and (S7) in the preliminary version of PIM$_t^=$ given in [18, Figure 6].

(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 (22) reduces to

$$(\mathbf{T} \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \rhd_s \{\alpha_{i_n} \mapsto M_n\}) \tag{24}$$

with all α_{i_j} different in view of (iv) and the M_i in canonical form (17) $\neq \emptyset_m$ in view of (ii). Apart from the normalization of the merge structure components, this canonical form can be reached by (S7) and (S6).

Assuming store structures with $\leq N$ address variables can be brought in canonical form (22), 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 (8) and (10). We have

$$\begin{array}{ll} S \circ_s \{a_{N+1} \mapsto M_{N+1}\} = & \\ \quad = & S \circ_s ((\Pi \vee \neg\Pi) \rhd_s \{a_{N+1} \mapsto M_{N+1}\}) \\ \quad \overset{(L11)}{=} & S \circ_s (\Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\}) \circ_s (\neg\Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\}). \end{array} \tag{25}$$

In view of (10), $\neg\Pi$ is an instance of (23), so the last store cell of (25) 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

$$\begin{array}{ll} \Pi \rhd_s \{a_{N+1} \mapsto M_{N+1}\} = & \\ \quad \overset{(8)(L11)}{=} & (a_{N+1} \asymp \alpha_1) \rhd_s \{a_{N+1} \mapsto M_{N+1}\} \circ_s \cdots \\ & \cdots \circ_s (a_{N+1} \asymp a_N) \rhd_s \{a_{N+1} \mapsto M_{N+1}\} \\ \quad \overset{(S9)(A4)}{=} & (\alpha_1 \asymp a_{N+1}) \rhd_s \{\alpha_1 \mapsto M_{N+1}\} \circ_s \cdots \\ & \cdots \circ_s (a_N \asymp a_{N+1}) \rhd_s \{a_N \mapsto M_{N+1}\}. \end{array} \tag{26}$$

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 (26). Since S is in canonical form, unconditional commutativity applies and $S = S_1 \circ_s S_2$, where

$$S_2 = (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \tag{27}$$

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,

$$\begin{array}{ll} S \circ_s ((a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{N+1}\}) = & \\ \quad = & S_1 \circ_s S_2 \circ_s ((a_i \asymp a_{N+1}) \rhd_s \{a_i \mapsto M_{N+1}\}) \\ \quad = & S_1 \circ_s \\ & (\neg(a_i \asymp a_{N+1}) \rhd_s S_2) \circ_s \\ & ((a_i \asymp a_{N+1}) \rhd_s (S_2 \circ_s \{a_i \mapsto M_{N+1}\})). \end{array} \tag{28}$$

For the last element of (28) we obtain

$$
\begin{array}{ll}
S_2 \circ_s \{a_i \mapsto M_{N+1}\} = & \\
\overset{(27)(\mathrm{L5})}{=} & (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \circ_s \\
& (\mathbf{T} \rhd_s \{a_i \mapsto M_{N+1}\}) \\
\overset{(\mathrm{v})(\mathrm{L11})}{=} & (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{j_1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{j_k}\}) \circ_s \\
& (\Pi_{j_1} \rhd_s \{a_i \mapsto M_{N+1}\}) \circ_s \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto M_{N+1}\}) \\
\overset{(\mathrm{iv})(\mathrm{S6})}{=} & (\Pi_{j_1} \rhd_s \{a_i \mapsto (M_{j_1} \circ_s M_{N+1})\}) \circ_s \cdots \\
& \cdots \circ_s (\Pi_{j_k} \rhd_s \{a_i \mapsto (M_{j_k} \circ_s M_{N+1})\}).
\end{array}
$$

Hence, by (28) 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 (25) yields the required canonical form (22).

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 following extension of the simple canonical form (24) applies:

$$\Sigma \circ_s ((\mathbf{T} \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \rhd_s \{\alpha_{i_n} \mapsto M_n\})) \tag{29}$$

with

(i) Σ in canonical form (17) with $k = 0$, or rather its equivalent for sort $\mathcal{S}$

(ii) the rightmost part in canonical form (24), but with merge structure components M_i in canonical form (19) $\neq \emptyset_m$ rather than (17)

(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.

The canonical form is derived 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 (17) with $k = 0$ by using equations (L*n*) and (15) 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 (16) with $\rho = s$ for the store variable part, and (S7) and (16) with $\rho = m$ for the store cell part. Apply

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

(4) Repeat steps (1)–(3) until both parts are in the required form and (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 (8) and (10). First, consider the equation $\Pi \rhd_s t_1 = \Pi \rhd_s t_2$. It is valid in $I(\text{PIM}_t^+)$ and

$$\begin{aligned}
\Pi \rhd_s t_i \quad &\overset{(8)(L11)}{=} \quad ((a_{N+1} \asymp \alpha_1) \rhd_s t_i) \circ_s \cdots \circ_s ((a_{N+1} \asymp a_N) \rhd_s t_i) \\
&= \quad ((a_{N+1} \asymp \alpha_1) \rhd_s t_{i,1}) \circ_s \cdots \circ_s ((a_{N+1} \asymp a_N) \rhd_s t_{i,m+N}),
\end{aligned}$$

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 (10) 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 $s \mathbin{@} a_{N+1}$ 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. We have not obtained a canonical form in this case. Note, however, that there is some overlap between the present case and the case based on canonical form (22).

Let $\text{PIM}_t^= = \text{PIM}_t^+ +$ the equations of Figure 9. In view of the foregoing we have

Proposition 3 $\text{PIM}_t^=$ *is ω-complete.*

9 PIM in Practice

9.1 Rewriting PIM Graphs

By orienting equation instances of $\text{PIM}_t^=$ and implementing the resulting rules on graphs, we obtain a *term graph rewriting* system [4]. 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. 6 is obtained by first normalizing the graph S_{P_1} (Fig. 2) with respect to the system $\text{PIM}_t^{\rightarrow}$ developed in Section 9.2, then using instances of equation (S8) of PIM_t^+ to permute addresses with respect to a fixed ordering. S_{P_1} is the normal form of the PIM representations 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 [18]. 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 [3]. If we ignore the guard, we can also interpret the composition operator in M' as an SSA form ϕ node [12]. 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 [41] also have this property).

Fig. 10 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. 11. 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. 12 (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 systems 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,

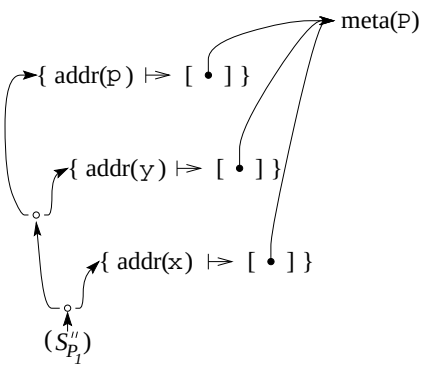

Figure 10: S''_{P_1}: Simplest PIM representation of programs P_1, P_3, and P_5.

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.

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

In this section, we concentrate on obtaining confluent and terminating rewriting systems derived from $\text{PIM}_t^=$. Much of this work was carried out with the assistance of the TIP inductive theorem proving system [20], which was used to perform Knuth-Bendix completion [13], 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, the remaining equations had to be verified by hand since we were unable to successfully use (S8) as a lemma during semi-automatic proofs (other theorem provers available to us had similar limitations)—details will be discussed elsewhere [37]. Here, we indicate the progress with respect to 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. Unfortunately, the default mode of TIP's completion procedure uses the lexicographic term ordering, which overrides the left-to-right orientation and orients (L3) in the right-associative direction. With rule (M2), this causes the completion procedure to add an infinite number of rules

$$(m_1 \circ_m (m_2 \circ_m \cdots (m_i \circ_m [v]) \cdots))! \rightarrow v \quad (i \geq 2). \tag{30}$$

However, we can use TIP's "set permutation" command and allow a left-associative ordering for (L3), which then yields a complete system without generating any rules in addition to (MA0). Alternatively,

```
/* 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 11: Semantically equivalent μC programs.

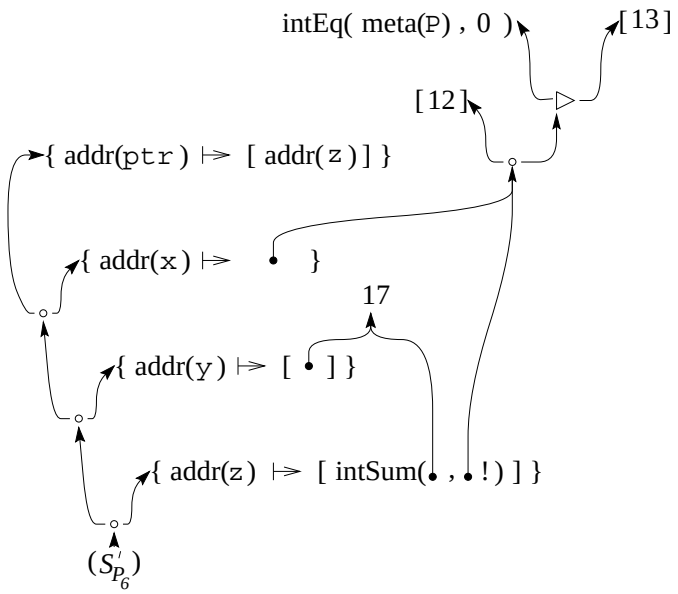

Figure 12: S'_{P_6}: Common representation for P_6, P_7, P_8, and P_9.

we can also observe that the infinite number of rules (30) generated above is due to the absence of the desired unification modulo associativity [10] during completion, although we only need matching modulo associativity during rewriting. The system is complete without the rules (30), in the presence of rewriting modulo associativity of operator '$\circ_m$'. By rewriting modulo associativity we mean, as usual, that a rule containing this operator is matched against all possible associative variants.

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. As a result, TIP's completion procedure terminates automatically under the default lexicographic term ordering, giving (L3) for the merge case a right-associative orientation and generating the additional rules (MA0) and

$$m_1 \circ_m ([v] \circ_m m_2) \quad \rightarrow \quad [v] \circ_m m_2. \tag{MA1}$$

If we now force a left-associative orientation of (L3) for the merge case, TIP's 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. 25). (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

[10] Associative unification is infinitary.

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 [36].

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 $\Box!$ 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 13. $\text{PIM}_t^{\rightarrow}$ is confluent, terminating, and left-linear. If we assume rewriting modulo associativity, we do not have to consider explicit versions of (L3) and thus (MA1) and (MA5) can be dispensed with (see Table 1).

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. 14. If we then add (S6), we need the rule (SB1), also shown in Fig. 14. Adding all the rules in Fig. 14 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 [21] 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). Rewriting modulo associativity is commonly available since it can be efficiently implemented. We can obtain several variants of these systems by choosing rewriting modulo associativity, or modulo associativity and commutativity. For example, we can treat (L3) and thus (MA1), (MA5), (MB3), (MB4) and (SB1) using rewriting modulo associativity. Table 1 describes some of them. 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 $\vee$.

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

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), ($\widetilde{\text{B3}}$)–(B6) − (L3) + rewriting modulo A	confluent (modulo A), 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), ($\widetilde{\text{B3}}$)–(B6) − (L3) + rewriting modulo A	confluent (modulo A), terminating, left-linear
PIM_t^+ − (S8), equations oriented left-to-right + (MA0)	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-linear
$\text{PIM}_t^{\rightarrow}$ + (B8), (B12), ($\widetilde{\text{B3}}$)–(B6)	confluent, terminating, left-linear
$\text{PIM}_t^{\rightarrow}$ + ($\widetilde{\text{B3}}$)–(B6) − (L3), (MA1), (MA5) + mod A rewriting	confluent ($\circ_\rho$: modulo A), terminating, left-linear
$\text{PIM}_t^{\rightarrow}$	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-linear
$\text{PIM}_t^{\rightarrow}$ + (B10), (B14), (B16), (BA1)	confluent ($\wedge$, $\vee$: modulo AC), terminating, left-*non*linear
$\text{PIM}_t'^{\rightarrow}$	confluent ($\wedge$, $\vee$: modulo AC) terminating, left-*non*linear
$\text{PIM}_t'^{\rightarrow}$ − (L3), (MA1), (MA5), (MB3), (MB4), (SB1) + mod A rewriting	confluent ($\wedge$, $\vee$: modulo AC; $\circ_\rho$: modulo A), terminating, left-*non*linear

Mod A rewriting: Rewriting using associative matching for associative operators.
($\widetilde{\text{B3}}$)–(B6): Symmetric versions of the rules (B3), (B4), (B5) and (B6).

Table 1: Properties of some of the PIM_t systems.

$\emptyset_\rho \circ_\rho l$	$\rightarrow$	l	(L1)
$l \circ_\rho \emptyset_\rho$	$\rightarrow$	l	(L2)
$(l_1 \circ_\rho l_2) \circ_\rho l_3$	$\rightarrow$	$l_1 \circ_\rho (l_2 \circ_\rho l_3)$	(L3)
$p \rhd_\rho \emptyset_\rho$	$\rightarrow$	$\emptyset_\rho$	(L4)
$\mathbf{T} \rhd_\rho l$	$\rightarrow$	l	(L5)
$\mathbf{F} \rhd_\rho l$	$\rightarrow$	$\emptyset_\rho$	(L6)
$p_1 \rhd_\rho (p_2 \rhd_\rho l)$	$\rightarrow$	$(p_1 \wedge p_2) \rhd_\rho l$	(L8)
$\{a_1 \mapsto m\} @ a_2$	$\rightarrow$	$(a_1 \asymp a_2) \rhd_m m$	(S1)
$\{a \mapsto \emptyset_m\}$	$\rightarrow$	$\emptyset_s$	(S2)
$\emptyset_s @ a$	$\rightarrow$	$\emptyset_m$	(S3)
$(s_1 \circ_s s_2) @ a$	$\rightarrow$	$(s_1 @ a) \circ_m (s_2 @ a)$	(S4)
$p \rhd_s \{a \mapsto m\}$	$\rightarrow$	$\{a \mapsto (p \rhd_m m)\}$	(S5)
$(p \rhd_s s) @ a$	$\rightarrow$	$p \rhd_m (s @ a)$	(S11)
$(\alpha_i \asymp \alpha_i)$	$\rightarrow$	$\mathbf{T} \quad (i \geq 1)$	(A1)
$(\alpha_i \asymp \alpha_j)$	$\rightarrow$	$\mathbf{F} \quad (i \neq j)$	(A2)
$m \circ_m [v]$	$\rightarrow$	$[v]$	(M2′)
$[v]!$	$\rightarrow$	v	(M3)
$\emptyset_m!$	$\rightarrow$	$?$	(M4)
$[m!]$	$\rightarrow$	$[?] \circ_m m$	(M7)
$((p \rhd_m [?]) \circ_m m)!$	$\rightarrow$	$m!$	(M8)
$\neg\mathbf{T}$	$\rightarrow$	$\mathbf{F}$	(B1)
$\neg\mathbf{F}$	$\rightarrow$	$\mathbf{T}$	(B2)
$\mathbf{T} \wedge p$	$\rightarrow$	p	(B3)
$\mathbf{F} \wedge p$	$\rightarrow$	$\mathbf{F}$	(B4)
$\mathbf{T} \vee p$	$\rightarrow$	$\mathbf{T}$	(B5)
$\mathbf{F} \vee p$	$\rightarrow$	p	(B6)
$\neg\neg p$	$\rightarrow$	p	(B7)
$\neg(p_1 \wedge p_2)$	$\rightarrow$	$\neg p_1 \vee \neg p_2$	(B18)
$\neg(p_1 \vee p_2)$	$\rightarrow$	$\neg p_1 \wedge \neg p_2$	(B19)
$m_1 \circ_m ([v] \circ_m m_2)$	$\rightarrow$	$[v] \circ_m m_2$	(MA1)
$([?] \circ_m m)!$	$\rightarrow$	$m!$	(MA2)
$(p \rhd_m [?])!$	$\rightarrow$	$?$	(MA3)
$[?] \circ_m (p \rhd_m [?])$	$\rightarrow$	$[?]$	(MA4)
$[?] \circ_m ((p \rhd_m [?]) \circ_m m)$	$\rightarrow$	$[?] \circ_m m$	(MA5)

Figure 13: Rewriting rules of $\textsc{Pim}_t^{\rightarrow}$.

$$
\begin{array}{rcll}
p \rhd_\rho (l_1 \circ_\rho l_2) & \rightarrow & (p \rhd_\rho l_1) \circ_\rho (p \rhd_\rho l_2) & \text{(L7)} \\
\{a \mapsto m_1\} \circ_s \{a \mapsto m_2\} & \rightarrow & \{a \mapsto (m_1 \circ_m m_2)\} & \text{(S6)} \\
(p \rhd_m m) \circ_m (p \rhd_m [v]) & \rightarrow & p \rhd_m [v] & \text{(MB1)} \\
((p \wedge p_1) \rhd_m m) \circ_m (p \rhd_m [v]) & \rightarrow & p \rhd_m [v] & \text{(MB2)} \\
(p \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) & \rightarrow & (p \rhd_m [v]) \circ_m m & \text{(MB3)} \\
((p \wedge p_1) \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) & \rightarrow & (p \rhd_m [v]) \circ_m m & \text{(MB4)} \\
\{a \mapsto m_1\} \circ_s (\{a \mapsto m_2\} \circ_s s) & \rightarrow & \{a \mapsto (m_1 \circ_m m_2)\} \circ_s s & \text{(SB1)}
\end{array}
$$

Figure 14: $\text{PIM}_t'^{\rightarrow} = \text{PIM}_t^{\rightarrow} +$ rules above.

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 KB-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 confluent ω-complete specification of the booleans [25], 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.

10 Extensions and Future Work

There are four major areas in which we would like to see additional work:

- Using the canonical forms discussed in this paper to develop a decision procedure for PIM_t.
- Providing a more extensive formal treatment of PIM's embedding into the untyped λ-calculus than that of [18] and [19], addressing in particular nontermination issues and the induction rule used in [18].
- 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 [18] and generalized in [19], and extensions with non-trivial value operations.
- Constructing confluent and/or terminating rewriting subsystems of $\text{PIM}_t^{=}$ stronger than $\text{PIM}_t^{\rightarrow}$.

References

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

[2] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8 (August 1978), 613–641.

[3] BALLANCE, R. A., MACCABE, A. B., AND OTTENSTEIN, K. J. 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, June 1990), pp. 257–271.

[4] BARENDREGT, H., VAN EEKELEN, M., GLAUERT, J., KENNAWAY, J., PLASMEIJER, M., AND SLEEP, M. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages* (Eindhoven, The Netherlands, 1987), vol. 259 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141–158.

[5] BERGSTRA, J., AND HEERING, J. Which data types have ω-complete initial algebra specifications? *Theoretical Computer Science 124* (1994), 149–168.

[6] BERGSTRA, J., AND TUCKER, J. The data type variety of stack algebras. *Annals of Pure and Applied Logic 73*, 1 (May 1995), 11–36.

[7] BERLIN, A., AND WEISE, D. Compiling scientific code using partial evaluation. *IEEE Computer 23*, 12 (December 1990), 25–36.

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

[9] CARTWRIGHT, R., AND FELLEISEN, M. The semantics of program dependence. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, June 1989), pp. 13–27.

[10] CHAMBERS, C., AND UNGAR, C. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (1989), pp. 146–160.

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

[12] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems 13*, 4 (October 1991), 451–490.

[13] DERSHOWITZ, N., AND JOUANNAUD, J.-P. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier/The MIT Press, 1990, pp. 243–320.

[14] ERSHOV, A. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science 18* (1982), 41–67.

[15] ERSHOV, A. P., AND OSTROVSKI, B. N. Controlled mixed computation and its appplication to systematic development of language-oriented parsers. In *Program Specification and Transformation*, L. Meertens, Ed. North-Holland, 1987, pp. 31–48.

[16] FELLEISEN, M., AND FRIEDMAN, D. P. A syntactic theory of sequential state. *Theoretical Computer Science 69* (1989), 243–287.

[17] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9*, 3 (July 1987), 319–349.

[18] FIELD, J. 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, June 1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

[19] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Proc. Twenty-second ACM Symp. on Principles of Programming Languages* (San Francisco, January 1995), pp. 379–392.

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

[21] GARLAND, S., AND GUTTAG, J. A Guide to LP, The Larch Prover. Tech. Rep. 82, Systems Research Center, DEC, Dec 1991.

[22] HEERING, J. Variaties op het thema 'stack'. Tech. Rep. CS-N8502, CWI, Amsterdam, 1985. (In Dutch).

[23] HEERING, J. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science 43* (1986), 149–167.

[24] HOARE, C., HAYES, I., JIFENG, H., MORGAN, C., ROSCOE, A., SANDERS, J., SORENSEN, I., SPIVEY, J., AND SUFRIN, B. Laws of programming. *Communications of the ACM 30*, 8 (August 1987), 672–686.

[25] HSIANG, J., AND DERSHOWITZ, N. Rewrite methods for clausal and non-clausal theorem proving. In *Automata, Languages and Programming (10th ICALP)* (1983), J. Diaz, Ed., vol. 154 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 331–346.

[26] HUGHES, J. Why functional programming matters. Report 16, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden, 1984.

[27] JIFENG, H., AND HOARE, C. From algebra to operational semantics. *Information Processing Letters 45* (February 1993), 75–80.

[28] JONES, N., GOMARD, C., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[29] KAHN, G. Natural semantics. In *Fourth Annual Symp. on Theoretical Aspects of Computer Science* (1987), vol. 247 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–39.

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

[31] LAZREK, A., LESCANNE, P., AND THIEL, J.-J. Tools for proving inductive equalities, relative completeness, and ω-completeness. *Information and Computation 84* (1990), 47–70.

[32] MASON, I. A., AND TALCOTT, C. Axiomatizing operational equivalence in the presence of side effects. In *Proc. Fourth IEEE Symp. on Logic in Computer Science* (Cambridge, MA, March 1989), pp. 284–293.

[33] MEINKE, K., AND TUCKER, J. Universal algebra. In *Handbook of Logic in Computer Science, Vol. I*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 189–411.

[34] MESEGUER, J., AND GOGUEN, J. Initiality, induction and computability. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds. Cambridge University Press, 1985, pp. 459–541.

[35] MOSSES, P. A basic abstract semantic algebra. In *Proc. Semantics of Data Types* (1984), vol. 173 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 87–107.

[36] MÜLLER, F. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters 41* (1992), 293–299.

[37] NAIDICH, D., AND DINESH, T. B. An automated induction method for verification of PIM—a transformational toolkit for compilers. Tech. rep., CWI, Amsterdam, The Netherlands, 1996. Upcoming report.

[38] NIRKHE, V., AND PUGH, W. 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, January 1992), pp. 269–280.

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

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

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

[42] WIRSING, M. Algebraic specification. In *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier/The MIT Press, 1990, pp. 675–788.

[43] YANG, W., HORWITZ, S., AND REPS, T. Detecting program components with equivalent behaviors. Tech. Rep. 840, University of Wiconsin-Madison, April 1989.

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

A μC-To-PIM Translation

A.1 μC Programs to PIM Terms

The syntax of μC is given in Fig. 15. A formal description of the translation of μC programs to PIM terms is given in Figures 16 and 17. The translation is written in the style of Natural Semantics [29], 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 μC syntactic components. These sequent forms are as follows:

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

Each of the sequents above takes a μC construct c and an incoming PIM store s, and yields a PIM term or a pair[11] of PIM terms, depending on the nature of the μC construct being translated. Pure expressions (those having no side-effects) and unpure expressions are distinguished in the translation process; subscripts p and u are used to denote the two types. Details of the sequent types are as follows:

- Sequents with '$\Rightarrow_{\mathtt{Stmt}}$' are used to translate statements. Sequents with '$\Rightarrow_{\mathtt{Pgm}}$' 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 '$\Rightarrow_{\mathtt{Exp}_p}$' are used to translate *pure* expressions computing ordinary values. Sequents with '$\Rightarrow_{\mathtt{LValue}_p}$' 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 '$\Rightarrow_{\mathtt{Exp}_u}$' are used to translate *unpure* expressions computing ordinary values. Sequents with '$\Rightarrow_{\mathtt{LValue}_u}$' are used to translate unpure 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.

[11] 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.

```
Pgm       ::= { StmtList }

StmtList  ::= Stmt
            | StmtList Stmt

Stmt      ::= Exp ;
            | if ( Exp ) Stmt
            | if ( Exp ) Stmt else Stmt
            | { StmtList }

Exp       ::= Exp_p
            | Exp_u

Exp_p     ::= Id
            | IntLiteral
            | ?Id

Exp_u     ::= * Exp
            | & LValue
            | LValue = Exp
            | LValue += Exp
            | Exp + Exp
            | ! Exp
            | Exp || Exp

LValue    ::= LValue_p
            | LValue_u

LValue_p  ::= Id

LValue_u  ::= * Exp
```

Figure 15: Syntax of μC.

$$(P)\qquad \frac{\emptyset_s \vdash \text{Stmt} \Rightarrow_{\texttt{Stmt}} u}{\vdash \text{Stmt} \Rightarrow_{\texttt{Pgm}} u}$$

$$(S_1)\qquad \frac{s \vdash \text{Stmt} \Rightarrow_{\texttt{Stmt}} u}{s \vdash \{\ \text{Stmt}\ \} \Rightarrow_{\texttt{Stmt}} u}$$

$$(S_2)\qquad \frac{\begin{array}{c} s \vdash \{\ \text{StmtList}\ \} \Rightarrow_{\texttt{Stmt}} u, \\ s \circ u \vdash \text{Stmt} \Rightarrow_{\texttt{Stmt}} u' \end{array}}{s \vdash \{\ \text{StmtList Stmt}\ \} \Rightarrow_{\texttt{Stmt}} u \circ u'}$$

$$(S_3)\qquad \frac{s \vdash \text{Exp} \Rightarrow_{\texttt{Exp}} \langle v,\ u\rangle}{s \vdash \text{Exp}\,; \Rightarrow_{\texttt{Stmt}} u}$$

$$(S_4)\qquad \frac{\begin{array}{c} s \vdash \text{Exp} \Rightarrow_{\texttt{Exp}} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{Stmt} \Rightarrow_{\texttt{Stmt}} u_S \end{array} \qquad v'_E = ItoB(v_E)}{s \vdash \texttt{if (}\ \text{Exp}\ \texttt{)}\ \text{Stmt} \Rightarrow_{\texttt{Stmt}} u_E \circ (v'_E \rhd u_S)}$$

$$(S_5)\qquad \frac{\begin{array}{c} s \vdash \text{Exp} \Rightarrow_{\texttt{Exp}} \langle v_E,\ u_E\rangle, \\ s' \vdash \text{Stmt}_1 \Rightarrow_{\texttt{Stmt}} u_{S_1}, \\ s' \vdash \text{Stmt}_2 \Rightarrow_{\texttt{Stmt}} u_{S_2} \end{array} \qquad \begin{array}{l} s' = s \circ u_E \\ v'_E = ItoB(v_E) \end{array}}{\begin{array}{c} s \vdash \texttt{if (}\ \text{Exp}\ \texttt{)}\ \text{Stmt}_1\ \texttt{else}\ \text{Stmt}_2 \Rightarrow_{\texttt{Stmt}} \\ u_E \circ (v'_E \rhd u_{S_1}) \circ (\neg v'_E \rhd u_{S_2}) \end{array}}$$

$$(E_1)\qquad \frac{s \vdash \text{Exp}_p \Rightarrow_{\texttt{Exp}_p} v}{s \vdash \text{Exp}_p \Rightarrow_{\texttt{Exp}} \langle v,\ \emptyset_s\rangle}$$

$$(E_2)\qquad \frac{s \vdash \text{Exp}_u \Rightarrow_{\texttt{Exp}_u} \langle v,\ u\rangle}{s \vdash \text{Exp}_u \Rightarrow_{\texttt{Exp}} \langle v,\ u\rangle}$$

$$(E_{p_1})\qquad s \vdash \text{Id} \Rightarrow_{\texttt{Exp}_p} s\ @\ \text{addr(Id)}\ !$$

$$(E_{p_2})\qquad s \vdash \text{IntLiteral} \Rightarrow_{\texttt{Exp}_p} \text{IntLiteral}$$

$$(E_{p_3})\qquad s \vdash \texttt{?Id} \Rightarrow_{\texttt{Exp}_p} \text{meta(Id)}$$

$$(E_{u1})\qquad \frac{s \vdash \text{Exp} \Rightarrow_{\texttt{Exp}} \langle v,\ u\rangle}{s \vdash \texttt{*}\ \text{Exp} \Rightarrow_{\texttt{Exp}_u} \langle (s \circ u)\ @\ v\ !,\ u\rangle}$$

Figure 16: Translation rules for μC, Part I.

$$(E_{u2}) \quad \frac{s \vdash \text{LValue} \Rightarrow_{\text{LValue}} \langle v,\ u\rangle}{s \vdash \ \&\,\text{LValue} \Rightarrow_{\text{Exp}_u} \langle v,\ u\rangle}$$

$$(E_{u3}) \quad \frac{\begin{array}{c} s \vdash \text{Exp} \Rightarrow_{\text{Exp}_u} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{LValue} \Rightarrow_{\text{LValue}} \langle v_L,\ u_L\rangle \end{array}}{\begin{array}{l} s \vdash \text{LValue} = \text{Exp} \Rightarrow_{\text{Exp}_u} \\ \quad \langle v_E,\ u_E \circ u_L \circ \{v_L \mapsto [v_E]\}\rangle \end{array}}$$

$$(E_{u4}) \quad \frac{\begin{array}{c} s \vdash \text{Exp} \Rightarrow_{\text{Exp}_u} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{LValue} \Rightarrow_{\text{LValue}} \langle v_L,\ u_L\rangle \end{array}}{\begin{array}{l} s \vdash \text{LValue} \mathrel{+}= \text{Exp} \Rightarrow_{\text{Exp}_u} \\ \quad \langle v',\ u' \circ \{v_L \mapsto [v']\}\rangle \end{array}} \quad \begin{array}{c} u' = u_E \circ u_L \\ v' = \text{intSum}((s \circ u') \mathbin{@} v_L\,!\,,\ v_E) \end{array}$$

$$(E_{u5}) \quad \frac{\begin{array}{c} s \vdash \text{Exp}_1 \Rightarrow_{\text{Exp}_u} \langle v_1,\ u_1\rangle, \\ s \circ u_1 \vdash \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \langle v_2,\ u_2\rangle \end{array}}{\begin{array}{l} s \vdash \text{Exp}_1 + \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \\ \quad \langle \text{intSum}(v_1\ ,\ v_2),\ u_1 \circ u_2\rangle \end{array}}$$

$$(E_{u6}) \quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash \ !\ \text{Exp} \Rightarrow_{\text{Exp}_u} \langle BtoI(\neg ItoB(v)),\ u\rangle}$$

$$(E_{u7}) \quad \frac{\begin{array}{c} s \vdash \text{Exp}_1 \Rightarrow_{\text{Exp}_u} \langle v_1,\ u_1\rangle, \\ s \circ u_1 \vdash \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \langle v_2,\ u_2\rangle \end{array}}{\begin{array}{l} s \vdash \text{Exp}_1 \ ||\ \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \\ \quad \langle BtoI(ItoB(v_1) \vee ItoB(v_2)),\ u_1 \circ u_2\rangle \end{array}}$$

$$(L_1) \quad \frac{s \vdash \text{LValue}_p \Rightarrow_{\text{LValue}_p} a}{s \vdash \text{LValue}_p \Rightarrow_{\text{LValue}} \langle a,\ \emptyset_s\rangle}$$

$$(L_2) \quad \frac{s \vdash \text{LValue}_u \Rightarrow_{\text{LValue}_u} \langle a,\ u\rangle}{s \vdash \text{LValue}_u \Rightarrow_{\text{LValue}} \langle a,\ u\rangle}$$

$$(L_p) \quad s \vdash \text{Id} \Rightarrow_{\text{LValue}_p} \text{addr(Id)}$$

$$(L_u) \quad \frac{s \vdash \text{Exp} \Rightarrow_{\text{Exp}_u} \langle a,\ u\rangle}{s \vdash \ *\ \text{Exp} \Rightarrow_{\text{LValue}_u} \langle a,\ u\rangle}$$

where:

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

Figure 17: Translation rules for μC, Part II.

- Sequents with '$\Rightarrow_{\texttt{Exp}}$' or '$\Rightarrow_{\texttt{LValue}}$' 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 unpure sequents, depending on the type of construct being translated.

As an example of how the translation process works, consider rule (E_{u5}) in Fig. 17. This rule may be read as follows: Given μC expression $\texttt{Exp}_1$ + $\texttt{Exp}_2$ and incoming PIM store s, first translate $\texttt{Exp}_1$ to the pair $\langle v_1,\ u_1\rangle$ using initial store s. Term v_1 represents the value of $\texttt{Exp}_1$, and term u_1 represents the side effects occurring in $\texttt{Exp}_1$. Next, translate $\texttt{Exp}_2$ 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 $\texttt{Exp}_1$ is accounted for in the store used to translate $\texttt{Exp}_2$, thus effectively encoding a left-to-right order of evaluation. Finally, the PIM term corresponding to the entire expression $\texttt{Exp}_1$ + $\texttt{Exp}_2$ is given by the pair $\langle \texttt{intSum}(v_1\ ,\ v_2),\ u_1 \circ u_2\rangle$. The term $\texttt{intSum}(v_1\ ,\ v_2)$ 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 $\texttt{Exp}_1$ and $\texttt{Exp}_2$.

A.2 PIM Terms to PIM Graphs

The translation given in Figures 16 and 17 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 may be *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 $\texttt{Exp}_1$ and $\texttt{Exp}_2$ (which would happen, e.g., if both $\texttt{Exp}_1$ and $\texttt{Exp}_2$ 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 A Simple Example—Proofs

B.1 $I(\text{STACK}^+) = F_{\mathcal{V}}(\text{STACK})$

It is sufficient to show 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

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

B.2 $\text{STACK}^=$ is ω-Complete

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).