A Complete Transformational Toolkit for Compilers*

J.A. Bergstra[1], T.B. Dinesh[2], J. Field[3], J. Heering[2]

[1] Faculty of Mathematics and Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands (janb@fwi.uva.nl)
[2] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
({T.B.Dinesh, Jan.Heering}@cwi.nl)
[3] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA (jfield@watson.ibm.com)

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.

1 Introduction

In an earlier paper [13], 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 [7] 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 PIM normal forms can be manipulated in a manner similar to intermediate representations commonly used in optimizing compilers.

A major question left open in [13] 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

* 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.

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 [13], 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).

We are aware of only a few prior completeness results for logics for imperative languages: Mason and Talcott [22] 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. [18] 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.

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.

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 [12], SSA form [9], GSA form [2], the PRG [28], the VDG [26], and the representation of Click [8]. 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 are "first-class" features of the formalism.
- 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 unstructured programs, the PIM analogue of a traditional IR may be constructed either by first restructuring the program's control flow graph (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 [26]).
- PIM is an *equational logic*, rather than, e.g., a sequent calculus such as that of Mason and Talcott [22]. 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.
- Unlike work on calculi for reasoning about imperative features in otherwise functional languages [11, 25, 24], 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 [27] 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. [18] and Boehm [6] treat Algol-class languages, [18] does not accommodate *computed* addresses arising from pointers and arrays, and neither [6] nor [18] 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).

In this paper, we will concentrate on the formal properties of first order systems derived from PIM's core algebraic component, PIM$_t$. For further details on the correspondence between PIM and traditional IR's, see [13]. For an example of a practical application of PIM, see [14], 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 PIM Terms and Graphs

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. The only non-standard addition to μC is the notion of a *meta-variable*, e.g., '?P' or '?X'. 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. The only deviation from standard C semantics in μC is the assumption that no address arithmetic is used.

```
{ p = ?P;      { p = 0;      { p = ?P;      { p = 0;      { p = ?P;
  x = ?X;        x = 1;        y = p;         y = p;        y = ?P;
  y = p;         y = p;        x = p;         x = p;        x = ?P;
  x = p;         x = p;        if (p)         if (p)      }
  if (p)         if (p)          x = y;         x = y;
    x = y;         x = y;    }              }
}              }
```

P_1 P_2 P_3 P_4 P_5

Fig. 1. Some simple μC programs.

A directed *term graph* [3] form of the PIM representation of P_1, S_{P_1}, is depicted in Fig. 4. S_{P_1} is generated by a simple syntax-directed translation, complete details of which may be found in [4]. 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. Shared PIM subgraphs are constructed systematically as a consequence of the translation process, or as a "side-effect" of the natural extension of term rewriting to term graphs [3]. Parent nodes in PIM term graphs will be depicted *below* their children to emphasize the correspondence between program constructs and corresponding PIM subgraphs. This 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.

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, since the size of the term form of a program's PIM representation may be exponentially larger than the graph form.

3.2 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 [13] and slightly revised in [14] augments PIM$_t$ with lambda expressions, an induction rule, and certain additional higher-order *merge distribution* rules that propagate conditional "contexts" inside expressions computing base values or addresses. As shown in [14], PIM's higher-order constructs allow loops (among other things) to be modeled in a straightforward way. 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.

The signature[4] of PIM$_t$ terms is given in Fig. 2. 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 system PIM$_t^=$ remain valid even when the term formation restrictions are dropped[5].

Fig. 3 depicts the equations of the system PIM$_t^0$. The equations labeled (Ln) are generic to merge or store structures, i.e., in each case 'p' should be interpreted as one of either s or m. Equations (A1) and (A2) are schemes for an infinite set of equations. 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. In examples in the sequel, we will augment PIM with a small number of function symbols to model addresses and integer data in μC programs. From the point of view of our formal results, these additional functions are simply treated as uninterpreted "inert" constructors.

3.3 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. 4. 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 $S_1 \equiv \{\mathtt{addr(p)} \mapsto [\mathtt{meta(P)}]\}$. 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.

[4] This signature differs slightly from the corresponding signature in [13]; the differences principally relate to a simplification in the structure of merge expressions.

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

[6] For clarity, Fig. 4 does not depict certain *empty* stores created by the translation process; this elision will be irrelevant in the sequel.

		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)
$\mathcal{B} \triangleright_s \mathcal{S}$	$\rightarrow$	$\mathcal{S}$	(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)
$\mathcal{B} \triangleright_m \mathcal{M}$	$\rightarrow$	$\mathcal{M}$	(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)
T, 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)

Fig. 2. Signature of PIM$_t$ Terms.

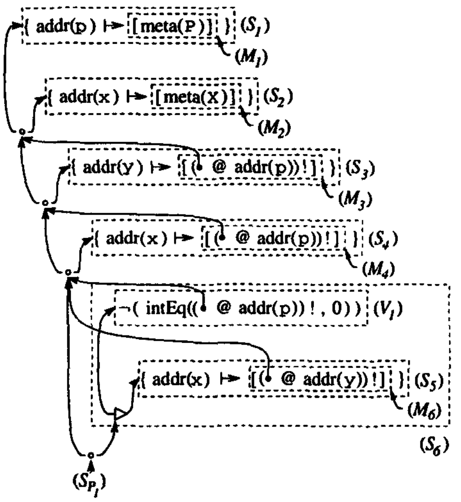
$$
\begin{array}{rcll}
\emptyset_p \circ_p l & = & l & \text{(L1)} \\
l \circ_p \emptyset_p & = & l & \text{(L2)} \\
l_1 \circ_p (l_2 \circ_p l_3) & = & (l_1 \circ_p l_2) \circ_p l_3 & \text{(L3)} \\
\mathrm{T} \triangleright_p l & = & l & \text{(L5)} \\
\mathrm{F} \triangleright_p l & = & \emptyset_p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

Fig. 3. Equations of PIM$_t^0$.

Stores may be guarded, i.e., executed conditionally. The subgraph labeled S_6 in Fig. 4 is such a store, and corresponds to the '`if`' statement in P_1. The guard expression denoted by V_1 corresponds to the `if`'s predicate expression. Consistent with standard C semantics, the guard V_1 tests whether the value of the variable p is nonzero. When guarded by the true predicate ('T'), a store structure evaluates to itself. If a store structure is guarded by the false predicate ('F'), it evaluates to the null store structure. These behaviors are axiomatized by equations (L5) and (L6).

An expression of the form $s \,@\, a$ 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. 4 are all merge cells. As with store structures, nontrivial merge structures may be built by prepending guard expressions, or by composing merge substructures using '$\circ_m$'. $\emptyset_m$ 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.



{ addr(p) ↦ [meta(P)] } (S_1)
(M_1)
{ addr(x) ↦ [meta(X)] } (S_2)
(M_2)
{ addr(y) ↦ [(• @ addr(p))!] } (S_3)
(M_3)
{ addr(x) ↦ [(• @ addr(p))!] } (S_4)
(M_4)
¬(intEq((• @ addr(p))!, 0)) (V_1)
{ addr(x) ↦ [(• @ addr(y))!] } (S_5)
(M_6)
(S_6)
(S_{P_1})

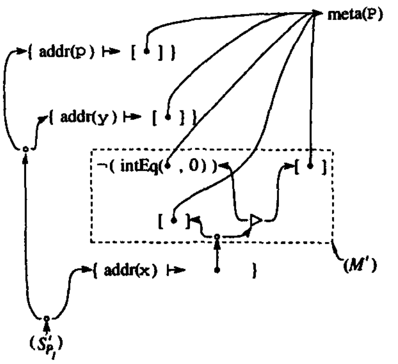


Fig. 4. S_{P_1} : PIM representation of program P_1. **Fig. 5.** S'_{P_1} : A simplified form of S_{P_1}.

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. 4 that the '!' operator is used in the translation of every reference to the value of a variable. When the retrieval semantics of the '@' operator are combined with the selection semantics of the '!' operator in an expression of the form $(s \ @ \ a)!$, the net effect is first to retrieve all the values in s associated with address a (i.e., assignments to the variable associated with a), then to yield the rightmost (i.e., most recently assigned) value associated with a.

4 Reasoning with PIM Terms and Graphs

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 $V^{\mathrm{X}}_{P_2} = (S_{P_2} \ @ \ \mathrm{addr(x)})!$ 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 $V^{\mathrm{X}}_{P_2}$ is a closed expression of sort V, we can use the equations of Fig. 3 to evaluate it. A simple interpreter for such expressions may be constructed by orienting the equations in Fig. 3 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 $V^{\mathrm{X}}_{P_2}$ is the constant '0'.

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^0_t 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 6.

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 7.

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 "**if** (p) **then** e **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 [19], but, following [17], 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 [17] 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.

To formalize these ideas, we require some definitions:

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* $t_1 = t_2$.

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. See also [17, 21, 5].

In the foregoing we assumed initial algebra semantics; however, as was pointed out in Section 4, a final algebra semantics is required to capture behavioral equivalence. To this end, we need the following:

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

The instantiation $C(\square := t)$ of a Σ-context C of type $\mathcal{S} \rightarrow \mathcal{T}$ 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 $\mathcal{S}' \rightarrow \mathcal{S}$, $C(t)$ is a Σ-context of type $\mathcal{S}' \rightarrow \mathcal{T}$.

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

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 [23, p. 488].

From the foregoing, we see that our completeness result will require two basic technical steps:

(A) Finding an initial algebra specification of the final model $F_{\mathcal{V}}(\text{PIM}_t^0)$. $F_{\mathcal{V}}(\text{PIM}_t^0)$ is the quotient of the initial algebra $I(\text{PIM}_t^0)$ by behavioral equivalence with respect to the observable sort $\mathcal{V}$ of base values. We add an equational definition of the behavioral equivalence to PIM_t^0, resulting in an initial algebra specification of $F_{\mathcal{V}}(\text{PIM}_t^0)$.
(B) Making the specification obtained in step (A) ω-complete to improve its ability to cope with program transformation and partial evaluation.

6 Step (A)—The Final Algebra

In this section, 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 2 and 3. The additional equations of PIM_t^+ are shown in Figure 6.

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

$$\begin{aligned}\{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\} = {} & (a_1 \asymp a_2) \rhd_s \{a_1 \mapsto (m_1 \circ_m m_2)\} \circ_s \\ & \neg(a_1 \asymp a_2) \rhd_s (\{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\}) \quad \text{(S8)}\end{aligned}$$

Fig. 6. Additional Equations of PIM_t^+

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

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

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

$\Box$

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

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

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

$$\emptyset_s,\ \{\alpha_{i_1} \mapsto M_1\} \circ_s \cdots \circ_s \{\alpha_{i_n} \mapsto M_n\} \quad \begin{array}{l}(n \geq 1,\ i_1 < \cdots < i_n,\\ M_j \text{ in normal form (1)},\\ M_j \neq \emptyset_m \text{ in view of (S2)}).\end{array} \tag{2}$$

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

7 Step (B)—ω-Complete Enrichment

We are now in a position to derive $\text{PIM}_t^=$, the ω-complete enrichment 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 5. 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"). Although the details of the canonical forms are included here, the (rather lengthy) proofs that they can actually be reached by equational reasoning from $\text{PIM}_t^=$ as well as the two inductive cases are omitted for reasons of space. The full proof is available in [4].

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.

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

$$\cdots \succ a_2 \succ a_1 \succ \alpha_i \quad (i \geq 1). \tag{3}$$

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 [20, 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 [17, Section 3.3], but there are some additional complications. First, we have

$$m \circ_m (p \rhd_m [v]) = (\neg p \rhd_m m) \circ_m (p \rhd_m [v]), \tag{4}$$

which 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

$$(p_1 \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l) = ((\neg p_2 \wedge p_1) \rhd_m l) \circ_m l_1 \circ_m (p_2 \rhd_m l). \tag{5}$$

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

$$p_1 \wedge p_2 = \mathbf{F} \implies (p_1 \rhd_\rho l_1) \circ_\rho (p_2 \rhd_\rho l_2) = (p_2 \rhd_\rho l_2) \circ_\rho (p_1 \rhd_\rho l_1), \tag{6}$$

which is often more readily applicable than (L10). Suitable canonical forms for open merge structures without @ or ! are $\emptyset_m$ and

$$(P_1 \rhd_m [V_1]) \circ_m \cdots \circ_m (P_k \rhd_m [V_k]) \circ_m (Q_1 \rhd_m M_1) \circ_m \cdots \circ_m (Q_n \rhd_m M_n) \tag{7}$$

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 $\mathcal{V}$, $V_i \not\equiv V_j$ $(i \neq j)$
(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 $M_i \not\equiv M_j$ $(i \neq j)$.

It is easily verified that two such canonical forms are equal in $I(\text{PIM}_t^+)$ 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 (6) with $\rho = m$).

Open merge structures with $\asymp$ and @ but without !. These can be flattened by using the distributive law (S4) and replacing any dereferenced store cells $(P \rhd_s \{A \mapsto M\})$ @ A' with $P \wedge (A \asymp A') \rhd_m M$ by (S11), (S1), (L8). Dereferenced variables $(P \rhd_s s)$ @ A with s a variable of sort $\mathcal{S}$ and A an address constant or variable, can be replaced by $P \rhd_m (s \text{ @ } A)$ by (S11). Any remaining compound variables s @ A cannot be eliminated but are similar to ordinary variables of sort $\mathcal{M}$ except that the address component A is subject to the substitution law

$$\begin{aligned}(a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \text{ @ } a_1)) \circ_m m_2) = \\ = (a_1 \asymp a_2) \rhd_m (m_1 \circ_m (p \rhd_m (s \text{ @ } a_2)) \circ_m m_2),\end{aligned} \tag{8}$$

which is a consequence of (L7–8), (S10). Two compound variables s @ A and s' @ A' are different if $s \not\equiv s'$ or $A \not\equiv A'$ (modulo (8)). Canonical form (7) 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 @ A, and $M_i \not\equiv M_j$ $(i \neq j)$
(v) The corresponding multiset of address constants and variables is minimal with respect to the ordering (3).

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

$$\text{canonical form (7) with (iv}'\text{) and (v) instead of (iv).} \tag{9}$$

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). Hence, merge structures with ! can be brought in canonical form (9).

Open terms of sort $\mathcal{V}$. These can be brought in the form $M!$ with M in canonical form (9). If M has a subterm $P \rhd_m [?]$ move it to the leftmost position by repeated application of (6), and eliminate it with (M8). Hence, open terms of sort $\mathcal{V}$ have canonical form

$$M! \quad (M \text{ in canonical form (9) without subterm } P \rhd_m [?]). \tag{10}$$

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

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

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

$$\begin{aligned} &\neg(a_1 \asymp a_2) \rhd_s (\{a_1 \mapsto m_1\} \circ_s \{a_2 \mapsto m_2\}) = \\ &= \neg(a_1 \asymp a_2) \rhd_s (\{a_2 \mapsto m_2\} \circ_s \{a_1 \mapsto m_1\}) \end{aligned} \tag{11}$$

(S7) is a conditional commutative law. (11) is similar but with an appropriate guard rather than a condition. Suitable canonical forms in this case are $\emptyset_s$ and

$$(\Pi_1 \rhd_s \{A_1 \mapsto M_1\}) \circ_s \cdots \circ_s (\Pi_n \rhd_s \{A_n \mapsto M_n\}) \tag{12}$$

with

(i) A_i a constant or variable of sort $\mathcal{A}$
(ii) M_i a merge structure without $\asymp$ in canonical form (7) $\neq \emptyset_m$
(iii) Π_i the canonical form $\neq \mathbf{F}$ of

$$\bigwedge_{k=1}^{n} \pm(A_i \asymp A_k) \tag{13}$$

with $\pm(A_i \asymp A_k)$ denoting one of $A_i \asymp A_k$ or $\neg(A_i \asymp A_k)$
(iv) $\Pi_i \wedge \Pi_j = \mathbf{F}$ ($A_i = A_j$ modulo (S9), $i \neq j$)
(v) $\displaystyle\bigvee_{\substack{A_j = A_i \\ \text{modulo (S9)}}} \Pi_j = \mathbf{T} \quad (1 \leq i \leq n)$
(vi) The corresponding multiset of address constants and variables is minimal with respect to the ordering (3).

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 (6), (S7), (11) 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. If all addresses are known, (12) reduces to

$$(\mathbf{T} \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \rhd_s \{\alpha_{i_n} \mapsto M_n\}) \tag{14}$$

with all α_{i_j} different in view of (iv) and the M_i in canonical form (7) $\neq \emptyset_m$ in view of (ii). Apart from the normalization of the merge structure components, this canonical form can be reached by (S7) and (S6).

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 (14) applies:

$$\Sigma \circ_s ((\mathbf{T} \rhd_s \{\alpha_{i_1} \mapsto M_1\}) \circ_s \cdots \circ_s (\mathbf{T} \rhd_s \{\alpha_{i_n} \mapsto M_n\})) \qquad (15)$$

with

(i) Σ in canonical form (7) with $k = 0$, or rather its equivalent for sort $\mathcal{S}$
(ii) the rightmost part in canonical form (14), but with merge structure components M_i in canonical form (9) $\neq \emptyset_m$ rather than (7)
(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.

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.

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.*

8 PIM in Practice

8.1 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 [3]. 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. 5 is obtained by first normalizing the graph S_{P_1} (Fig. 4) with respect to the system $\text{PIM}_t^{\rightarrow}$ developed in Section 8.2, 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. However, 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 [13]. 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 [2].

Consider finally the μC programs depicted in Fig. 7. Both of these programs are behaviorally equivalent; this fact may be deduced by inspection of the normal form graph S'_{P_6} shown in Fig. 8 (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 and P_7 would be the same.

8.2 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 [15], which was used to perform Knuth-Bendix completion [10] and to aid in inductive verification of equations incorporated in $\text{PIM}_t^{=}$. 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 $\text{PIM}_t^{=}$.

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

P_6

```
{ 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_7

Fig. 7. Semantically equivalent μC programs.

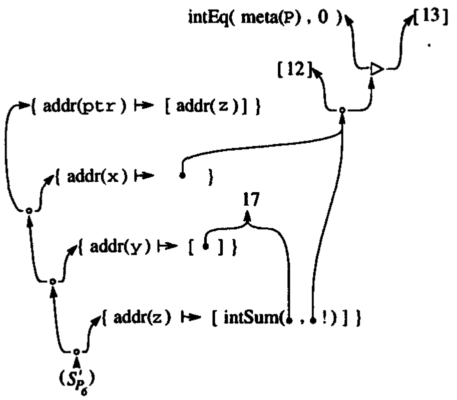


Fig. 8. S'_{P_6}: Normal form of PIM representations of P_6 and P_7.

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

$$(a_1 \asymp a_2) \rhd_m \emptyset_m \;\rightarrow\; \emptyset_m, \qquad \text{(MA0)}$$

which originates from a critical pair generated from the rules (S1) and (S2). PIM$_t^0$ is ground confluent even without this rule. Note that with rule (M2), a right-associative orientation of (L3) would cause the completion procedure to add an infinite number of rules

$$(m_1 \circ_m (m_2 \circ_m \cdots (m_i \circ_m [v]) \cdots))! \rightarrow v \quad (i \geq 2). \qquad (16)$$

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 by giving (L3) for the merge case a right-associative orientation and generating the additional rules (MA0) and (MA1) (see Figure 10). We note that (MA0) 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). (S7) is difficult to orient, but (S6) has an obvious orientation and is in acceptable form for mechanical analyzers. After attempting TIP's completion procedure on the system with (S6) and (M2′), we see immediately that the critical pairs that result from (S6) and (S4), using (S1), give rise to a special case of (L7) for $p = m$. 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. 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.

Adding (M7) or (M8) requires that (L3) be oriented in the right-associative direction. This is caused by the generation of the rule (MA2). 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 PIM$_t^{\to}$ is shown in Figure 10. PIM$_t^{\to}$ 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.

$$\begin{aligned}
p \rhd_p \emptyset_p &= \emptyset_p && \text{(L4)}\\
p \rhd_p (l_1 \circ_p l_2) &= (p \rhd_p l_1) \circ_p (p \rhd_p l_2) && \text{(L7)}\\
p_1 \rhd_p (p_2 \rhd_p l) &= (p_1 \wedge p_2) \rhd_p l && \text{(L8)}\\
l \circ_p l_1 \circ_p l &= l_1 \circ_p l && \text{(L9)}\\
(p \rhd_p l_1) \circ_p (\neg p \rhd_p l_2) &= (\neg p \rhd_p l_2) \circ_p (p \rhd_p l_1) && \text{(L10)}\\
(p_1 \rhd_p l) \circ_p (p_2 \rhd_p l) &= (p_1 \vee p_2) \rhd_p l && \text{(L11)}\\
(a \asymp a) &= \mathrm{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 @ a_1) &= (a_1 \asymp a_2) \rhd_m (s @ a_2) && \text{(S10)}\\
(p \rhd_s s) @ a &= p \rhd_m (s @ a) && \text{(S11)}\\
\{a \mapsto m\} \circ_s s &= s \circ_s \{a \mapsto m \circ_m (s @ 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 &= \mathrm{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 &= \mathrm{T} && \text{(B15)}\\
p_1 \wedge (p_2 \vee p_3) &= (p_1 \wedge p_2) \vee (p_1 \wedge p_3) && \text{(B16)}\\
p_1 \vee (p_2 \wedge p_3) &= (p_1 \vee p_2) \wedge (p_1 \vee p_3) && \text{(B17)}\\
\neg(p_1 \wedge p_2) &= \neg p_1 \vee \neg p_2 && \text{(B18)}\\
\neg(p_1 \vee p_2) &= \neg p_1 \wedge \neg p_2 && \text{(B19)}
\end{aligned}$$

Fig. 9. Additional Equations of PIM$_t^{=}$.

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

Fig. 10. Rewriting rules of PIM$_t^{\to}$.

Enhancing the rewriting systems. Further enrichments to PIM$_t^{\to}$ seem to require left-nonlinear rules in order to achieve confluence. Adding (L7), we require the additional rules

(MB1)–(MB4) shown in Fig. 11. If we then add (S6), we need the rule (SB1), also shown in Fig. 11. Adding all the rules in Fig. 11 to those of $\text{PIM}_t^{\rightarrow}$, we get the system $\text{PIM}_t^{\prime\rightarrow}$.

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 [16] adds the absorption law

$$p \vee (p \wedge p_1) \;\rightarrow\; p. \qquad \text{(BA1)}$$

$$\begin{array}{rcll}
p \rhd_p (l_1 \circ_p l_2) & \rightarrow & (p \rhd_p l_1) \circ_p (p \rhd_p l_2) & \text{(L7)}\\
\{a \mapsto m_1\} \circ_s \{a \mapsto m_2\} & \rightarrow & \{a \mapsto (m_1 \circ_m m_2)\} & \text{(S6)}\\[4pt]
(p \rhd_m m) \circ_m (p \rhd_m [v]) & \rightarrow & p \rhd_m [v] & \text{(MB1)}\\
((p \wedge p_1) \rhd_m m) \circ_m (p \rhd_m [v]) & \rightarrow & p \rhd_m [v] & \text{(MB2)}\\
(p \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) & \rightarrow & (p \rhd_m [v]) \circ_m m & \text{(MB3)}\\
((p \wedge p_1) \rhd_m m_1) \circ_m ((p \rhd_m [v]) \circ_m m) & \rightarrow & (p \rhd_m [v]) \circ_m m & \text{(MB4)}\\[4pt]
\{a \mapsto m_1\} \circ_s (\{a \mapsto m_2\} \circ_s s) & \rightarrow & \{a \mapsto (m_1 \circ_m m_2)\} \circ_s s & \text{(SB1)}
\end{array}$$

Fig. 11. $\text{PIM}_t^{\prime\rightarrow} = \text{PIM}_t^{\rightarrow} +$ rules above.

Finally, both $\text{PIM}_t^{\rightarrow}$ and $\text{PIM}_t^{\prime\rightarrow}$ produce normal forms modulo associativity and commutativity of $\wedge$ and $\vee$, i.e., with respect to (B8), (B9), (B12) and (B13). 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. 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 system. (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. (B11), (B15), (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.

9 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 [13] and [14], addressing in particular nontermination issues and the induction rule used in [13].
- Obtaining completeness results for variants of PIM_t, including versions with no restrictions on the formation of address or predicate expressions and variants incorporating the *merge* distribution rules, as used for addresses in [13] and generalized in [14].
- 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. 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.
3. 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.
4. BERGSTRA, J., DINESH, T., FIELD, J., AND HEERING, J. A complete transformational toolkit for compilers. Report CS-R9601, CWI, Amsterdam, January 1996; also Report RC 20342, IBM T.J. Watson Reseach Center, January 1996.
5. BERGSTRA, J., AND HEERING, J. Which data types have ω-complete initial algebra specifications? *Theoretical Computer Science 124* (1994), 149–168.
6. 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.
7. 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.
8. 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).
9. 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.
10. 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.
11. FELLEISEN, M., AND FRIEDMAN, D. P. A syntactic theory of sequential state. *Theoretical Computer Science 69* (1989), 243–287.
12. 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.
13. 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.
14. 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.
15. 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.
16. GARLAND, S., AND GUTTAG, J. A Guide to LP, The Larch Prover. Tech. Rep. 82, Systems Research Center, DEC, Dec 1991.
17. HEERING, J. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science 43* (1986), 149–167.
18. 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.
19. JONES, N., GOMARD, C., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. YANG, W., HORWITZ, S., AND REPS, T. Detecting program components with equivalent behaviors. Tech. Rep. 840, University of Wiconsin-Madison, April 1989.
28. 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.