PARAMETRIC PROGRAM SLICING

John Field G. Ramalingam
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598, USA
{jfield,rama}@watson.ibm.com

Frank Tip*
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
tip@cwi.nl

Abstract

Program slicing is a technique for isolating computational threads in programs. In this paper, we show how to mechanically extract a family of practical algorithms for computing slices directly from semantic specifications. These algorithms are based on combining the notion of *dynamic dependence tracking* in term rewriting systems [13] with a program representation whose behavior is defined via an equational logic [12]. Our approach is distinguished by the fact that changes to the behavior of the slicing algorithm can be accomplished through simple changes in rewriting rules that define the semantics of the program representation. Thus, e.g., different notions of dependence may be specified, properties of language-specific datatypes can be exploited, and various time, space, and precision tradeoffs may be made. This flexibility enables us to generalize the traditional notions of static and dynamic slices to that of a *constrained* slice, where any subset of the inputs of a program may be supplied.

1 Introduction

Program *slicing* is an important technique for program understanding and program analysis. Informally, a program slice consists of the program parts that (potentially) affect the values of specified variables at some designated program point—the *slicing criterion*. Although originally proposed as a means for program debugging [33], it has subsequently been used for performing such diverse tasks as program integration and "differencing" [16], software maintenance and testing [15, 8], compiler tuning [23], and parallelization of sequential code [32].

In this paper, we describe how a family of practical slicing algorithms can be derived directly from semantic specifications. The title of this paper is a triple entendre, in the sense that our technique is "parameterized" in three respects:

- We generalize the traditional notions of static and dynamic slices to that of a *constrained* slice. Static and dynamic slices have previously been computed by different techniques. By contrast, our approach provides a generic algorithm for computing constrained slices.
- Given a well-defined specification of a translation from a programming language to a common intermediate representation called PIM [12], we automatically extract a semantically well-founded language-*specific* algorithm for computing constrained slices. An advantage of this approach is that only the PIM translation is language dependent; the mechanics of slicing itself are independent of the language.
- PIM's semantics (and thus that of the source language via translation) is defined by a set of *rewriting* rules. These rules implicitly carry out many techniques used in optimizing compilers, e.g., conditional constant propagation and dead code elimination. The slices we obtain are thus often more precise than those computed by previous algorithms. By choosing different subsets of rules or adding additional rules, the precision of the analysis, as well as its time and space complexity, may be readily varied. We illustrate the flexible nature of our approach by defining several extensions to PIM's core logic. These variants describe differing treatments of loop semantics, and consequently define differing slice behaviors.

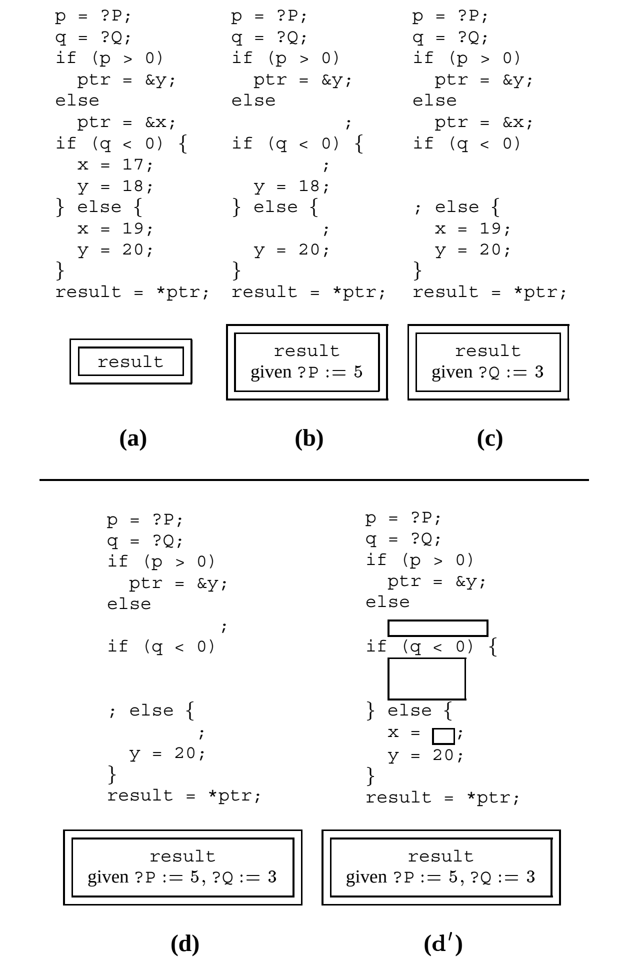
One of the primary contributions of this paper is an algorithm for computing *constrained slices*. Despite the myriad variations on the theme of slicing that can be found in the literature [28], almost all existing slicing algorithms fall into one of two classes: *static slicing algorithms*, which make no assumptions about the inputs to the program, and consequently compute slices that are valid for all possible input instances, and *dynamic slicing algorithms*, which accept a specific instantiation of *all* inputs, and compute slices valid only for that specific case. A constrained slice is valid for all instantiations of the inputs that satisfy a given set of constraints. In the sequel, we will primarily consider constraints that specify the values of some subset of the input parameters of the program.

The relation between constrained slicing, static slicing, and dynamic slicing is straightforward: a fully constrained slice (with every input a fixed constant) is a dynamic slice, and a fully *un*constrained slice is a static slice. We believe that constrained slicing can be more useful than static or dynamic slicing in helping programmers understand programs, by enabling the programmer to supply a variety of plausible input scenarios that the slicing system can exploit to simplify the slice obtained.

While Venkatesh has defined a notion of a *quasi-static* slice [30] similar to that of a constrained slice, we know of no previous work that describes how such slices may be *computed*. In a recent paper [24], Ning et al. describe a reverse engineering tool that permits users to specify constraints on variables and extract *conditional* slices, but they do not specify how these slices are computed or how powerful the constraints can be. One might consider combining partial evaluation of programs with static slicing to compute constrained slices, but, as will be explained later, this does not lead to satisfactory results.

[*] Supported in part by the European Union under ESPRIT project # 5399 (Compiler Generation for Parallel Machines—COMPARE). Part of this work was done while this author was at IBM.

To appear in the Proceedings of the 22nd ACM Symposium on
Principles of Programming Languages, San Francisco, January
1995



```
p = ?P;
q = ?Q;
if (p > 0)
  ptr = &y;
else
  ptr = &x;
if (q < 0) {
  x = 17;
  y = 18;
} else {
  x = 19;
  y = 20;
}
result = *ptr;
```

(a)

```
p = ?P;
q = ?Q;
if (p > 0)
  ptr = &y;
else
          ;
if (q < 0) {
        ;
  y = 18;
} else {
        ;
  y = 20;
}
result = *ptr;
```

(b)

```
p = ?P;
q = ?Q;
if (p > 0)
  ptr = &y;
else
  ptr = &x;
if (q < 0)


; else {
  x = 19;
  y = 20;
}
result = *ptr;
```

(c)

```
p = ?P;
q = ?Q;
if (p > 0)
  ptr = &y;
else
          ;
if (q < 0)


; else {
        ;
  y = 20;
}
result = *ptr;
```

(d)

```
p = ?P;
q = ?Q;
if (p > 0)
  ptr = &y;
else
  □
if (q < 0) {
  □
} else {
  x = □;
  y = 20;
}
result = *ptr;
```

(d′)

Figure 1: **(a)** Example program (= static slice). **(b)** Constrained slice with ?P := 5. **(c)** Constrained slice with ?Q := 3. **(d)** Constrained slice with ?P := 5, ?Q := 3 (= dynamic slice). **(d′)** Non-postprocessed term slice corresponding to **(d)**.

The feasibility of the ideas in this paper has been demonstrated by a successful prototype implementation of the PIM logic and translators for significant subsets of such disparate languages as C and Cobol using the ASF+SDF Meta-environment [19], a programming environment generator based on algebraic specifications.

2 Overview

In this section, we will give a brief overview of our approach using examples. Details will follow in subsequent sections.

2.1 Motivating Example

Fig. 1(a) shows an example program written in μC, a C subset that we will use for all the examples in this paper. μC has the standard C syntax and semantics, with one extension: *meta-variables* like ?P and ?Q are used to represent unknown values or inputs. All data in μC are assumed to be integers or pointers; we also assume that no address arithmetic is used. When we discuss loops in Section 5, we will for simplicity further restrict our analysis to programs containing only constant L-values.

The example of Fig. 1(a) is not entirely trivial, due to manipulation of pointers in a conditional statement. The static slice with respect to the final value of `result` consists of the entire program. The dynamic slice with respect to the final value of `result` for input `p = 5, q = 3` is shown in 1(d); note that it does not immediately reveal the effect of each input. The effect of input `p = 5` is illustrated by the constrained slice of 1(b); clearly it causes the aliasing of `*ptr` to `y`, and thereby makes both assignments to `x` obsolete. In 1(c), the effect of the other input, `q = 3` is shown: the statements in the first branch of the second `if` statement become irrelevant. Note that in general, it is not the case that a slice with respect to multiple constraints consists of the "intersection" of the slices with respect to each constraint.

In examples in the sequel, we will use the double box notation of Fig. 1 to denote a slicing criterion and the constraints, if any, on meta-variables. We will also use the terminology "slice of P *at* x [given C]" to denote the slice of P with respect to the final value of variable x [given meta-variable constraints C]. Slicing with respect to arbitrary expressions at intermediate program points will be discussed in Section. 4.6.

2.2 Slicing via Rewriting

PIM [12] consists of a rooted directed acyclic graph program representation[1] and an equational logic that operates on PIM graphs. These graphs can also be interpreted (or depicted) as *terms* after "flattening." A subsystem of the full PIM logic defines a *rewriting* semantics for a program's PIM representation. Rewriting rules can be used not only to execute programs, but also to perform various kinds of analysis by simplification of a program's PIM representation; each simplification step consists of the application of a rule of PIM's logic.

To compute the slice of a program with respect to the final value of a variable x, we begin with a term that "encodes" (i) the abstract syntax tree (AST) of the program, (ii) the variable x that represents the slicing criterion, and (iii) a (possibly empty) set of additional constraints. Next, we translate the AST to a graph comprising its PIM representation. This translation is assumed to be defined by a rewriting system (although it need not necessarily be implemented that way). The resulting graph is then simplified by repeated application of sets of rewriting rules derived from the PIM logic. This *reduction* process is carried out using the technique of *term graph rewriting* [4]. The graph that results from the reduction process represents the final value of variable x (in terms of the unconstrained meta-variables). During the reduction process, we maintain *dynamic dependence relations* [13] that relate nodes of the graph being manipulated to the AST. These relations are defined in a simple way directly from the structure of each rewriting rule, and will be discussed in more detail in Section 3. By tracing the dynamic dependence relations from the simplified PIM-graph back to the AST yields, we finally derive the slice of the AST with respect to x. The steps involved in the slicing process are depicted in Fig. 2.

This basic slicing algorithm is unusually flexible, in that it can be adapted to new languages simply by providing a source-to-PIM translator for the language. In addition, simple alterations to the rules or rewriting strategy can be used to affect the kinds of slice produced, as well as the time or space complexity of the reduction process. The ease with which we can handle constrained slices is due principally to the fact that the reduction process adapts itself to the presence or absence of information represented by constraints. As more information is available, more rules are applicable that have the potential to further simplify the slice.

[1] Although loops and recursive procedures admit a PIM graph representation with cycles, we will use a simpler DAG representation for such constructs in this paper.

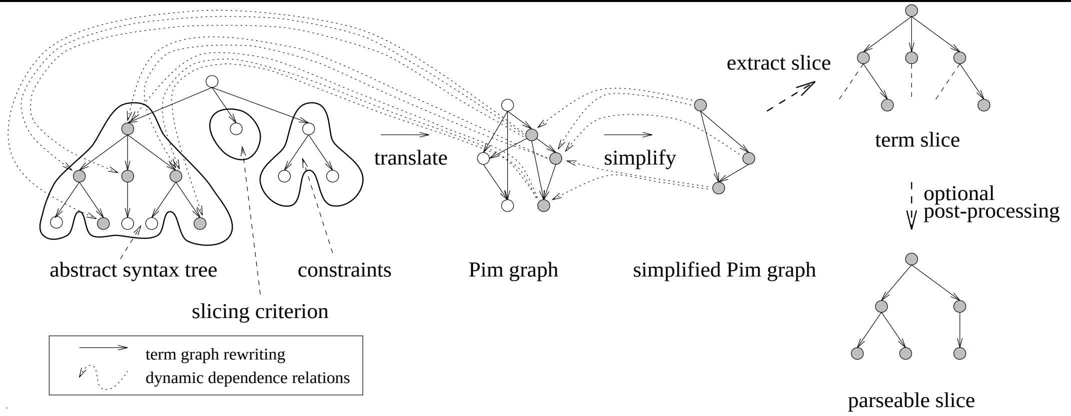


Figure 2: Overview of our approach.

2.3 Term Slices and Parseable Slices

Formally, our slices are *contexts* derived from the program's AST, i.e., a connected set of AST nodes in which certain subtrees are omitted, leaving "holes" behind. By interpreting these contexts as *open* terms, all of the slices we compute are "executable" via the PIM rewriting semantics, in the sense that any syntactically valid substitution for the holes in a term slice yields a program with the same behavior with respect to the slicing criterion[2].

It is often the case, however, that one wishes obtain a *parseable* representation of the slice (i.e., a syntactically well-formed AST without missing subtrees). Therefore, term slices may be optionally postprocessed in various ways to obtain parseable programs with identical behavior.

Fig. 1(d$'$) depicts the term slice corresponding to 1(d) before postprocessing. Certain fine details are present in this term slice that do not appear in Fig. 1(d), e.g., the L-values but not the R-values of certain assignment statements appear in the term slice.

The advantage of term slices is that they have a consistent semantic interpretation, and are oblivious to a language's syntactic quirks. This is particularly important in a language like C, where virtually any expression can have a side effect, and thus for which some parts of an expression can be relevant to a slice while others are not.

Unfortunately, term slices often introduce a certain amount of "clutter" not present in more ad-hoc algorithms; thus for the sake of clarity, most of the example slices we use in the sequel will be minimally postprocessed, primarily by replacing assignments with a hole in the right-hand side by empty statements. We will distinguish parseable slices from term slices by using boxes in the latter to represent holes.

2.4 More Examples

The example in Fig. 3 illustrates the flexibility of our technique by showing some of the differing treatments of loops that are possible (loops will be further studied in Sec. 5). Fig. 3(b) depicts what we will call a *pure dynamic slice* at `result`, given `?N := 5` and `?P := 1`. Note that this slice includes the `while` loop though it computes no value relevant to the criterion. This is the case because the underlying slicing algorithm faithfully reflects the standard semantics, under which there *is* a dependence between the `while` loop and the subsequent assignment to `result`. This phenomenon is noted in Cartwright and Felleisen's discussion of demand and control dependence [5]. This notion of dependence is also closely related to the notion of *weak control dependence* discussed by Podgurski and Clarke [26]. The slice in Fig. 3(c), similar to the kind computed by Agrawal and Horgan [1], results from adding some simple equational rules to be discussed later. The same variant of the slicing algorithm produces the result in Fig. 3(d), though the program is non-terminating for the constraints specified under the standard semantics. Previous dynamic slicing algorithms [1, 21] will not terminate for this input constraint. In this sense, our dynamic slicing algorithm is "more consistently lazy".

As a final example, consider the program in Fig. 4. Although absurdly contrived, the example illustrates several important points. By not insisting that the slice be parseable, we can make distinctions between assignment statements whose R-values are included but whose L-values are excluded and vice versa, as Fig. 4(b) shows. We also see that it is possible to determine that the values tested in a conditional are irrelevant to the slice, even though the body is relevant. In general, our approach can make a variety of fine distinctions that other algorithms cannot.

Fig. 4(c) gives an example of a *conditional constraint*. Such constraints can be handled by straightforward extensions to our basic algorithm. A detailed treatment of such constraints is outside the scope of this paper, but we will discuss them briefly in Section 4.7.

3 Term Rewriting and Dynamic Dependence Tracking

Our approach to slicing is based on extending the generic notion of *dynamic dependence tracking* in term rewriting systems [13] to realistic programming languages. In this section, we review dynamic dependence tracking and the basic ideas behind term and graph rewriting. For further details on term rewriting, the reader is referred to the excellent tutorial survey of Klop [20].

We begin by considering two PIM rewriting rules that define simple boolean identities:

$$\begin{array}{rcll} \vee\langle \mathbf{T}\ ,\ p\rangle & \longrightarrow & \mathbf{T} & \text{(B10)} \\ \vee\langle\vee\langle p_1\ ,\ p_2\rangle\ ,\ p_3\rangle & \longrightarrow & \vee\langle p_1\ ,\ \vee\langle p_2\ ,\ p_3\rangle\rangle & \text{(B14)} \end{array}$$

A rewriting rule is used to replace a subterm of a term that *matches* the rule's left hand side by the rule's right hand side. Variables (here, p, p_1, p_2, and p_3) match any subterm; all other symbols must match exactly. By applying the rules above, the term

$$\wedge\langle\vee\langle\vee\langle\mathbf{T}\ ,\ \mathbf{F}\rangle\ ,\ \wedge\langle\mathbf{F}\ ,\ \mathbf{T}\rangle\rangle\ ,\ \mathbf{F}\rangle$$

may be rewritten as follows (subterms affected by rule applications

[2] More precisely, the term "encoding" the original program and the slicing criterion and the term "encoding" the slice (with any syntactically valid substitution for the holes) and the slicing criterion both reduce to the same term/value.

```
n = ?N;
i = 1;
sum = 0;
while (i != n) {
  sum = sum + i;
  i = i + 1;
}
if (?P)
  result = n*(n-1)/2;
else
  result = sum;
```

(a)

```
n = ?N;
i = 1;
      ;
while (i != n) {
            ;
  i = i + 1;
}
if (?P)
  result = n*(n-1)/2;
else
              ;
```

result
given ?N := 5, ?P := 1

(b)

```
n = ?N;
    ;
      ;




;
if (?P)
  result = n*(n-1)/2;
else
             ;
```

result
given ?N := 5, ?P := 1

(c)

```
n = ?N;
    ;
      ;




;
if (?P)
  result = n*(n-1)/2;
else
             ;
```

result
given ?N := 0, ?P := 1

(d)

Figure 3: **(a)** An example program. **(b)** Pure dynamic slice at `result` given ?N := 5, ?P := 1. **(c)** Lazy dynamic slice at `result` given ?N := 5, ?P := 1. **(d)** Lazy dynamic slice at `result` given ?N := 0, ?P := 1.

```
*(ptr = &a) = ?A;
b = ?B;
x = a;
if (a < 3)
  ptr = &y;
else
  ptr = &x;
if (b < 2)
  x = a;
(*ptr) = 20;
```

(a)

```
*(□ = &a) = ?A;
b = □;
x = a;
if (a < 3)
  ptr = &y;
else
  □
if (□ < □)
  x = a;
(*ptr) = □;
```

x
given ?A := 2

(b)

```
*(□ = &a) = ?A;
b = □;
x = □;
if (a < 3)
  □
else
  ptr = &x;
if (□ < □)
  x = □;
(*ptr) = 20;
```

x
given ?A > 5

(c)

Figure 4: **(a)** An example program. **(b)** Constrained slice at x given ?A := 2. **(c)** Conditional constrained slice at x given ?A > 5.

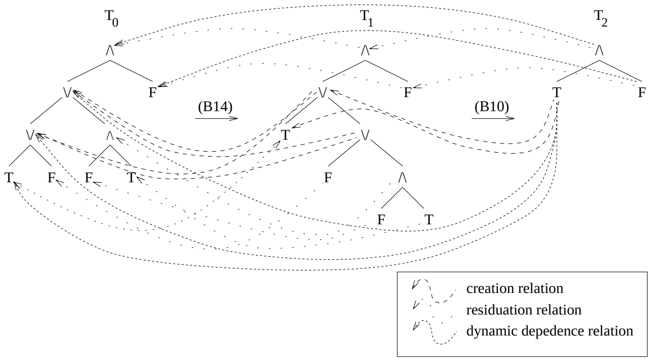


Figure 5: Example of creation and residuation relations.

are underlined):

$$\begin{aligned} T_0 &= \wedge\langle \underline{\vee\langle\vee\langle \mathbf{T}\,,\ \mathbf{F}\rangle\,,\ \wedge\langle\mathbf{F}\,,\ \mathbf{T}\rangle\rangle}\,,\ \mathbf{F}\rangle \longrightarrow^{(B14)} \\ T_1 &= \wedge\langle \underline{\vee\langle\mathbf{T}\,,\ \vee\langle\mathbf{F}\,,\ \wedge\langle\mathbf{F}\,,\ \mathbf{T}\rangle\rangle\rangle}\,,\ \mathbf{F}\rangle \longrightarrow^{(B10)} \\ T_2 &= \wedge\langle\mathbf{T}\,,\ \mathbf{F}\rangle \end{aligned}$$

Observe in the example above that the outer context $\wedge\langle\bullet\ ,\ \mathbf{F}\rangle$ ('$\bullet$' denotes a missing subterm) is not affected at all, and therefore occurs in T_0, T_1, and T_2. Furthermore, the occurrence of variables p_1, p_2, and p_3 in both the left-hand side and the right-hand side of (B14) causes the subterms **T**, **F**, and $\wedge\langle\mathbf{F}\ ,\ \mathbf{T}\rangle$ of the underlined subterm of T_0 to reappear in T_1. Also note that variable p occurs only in the left-hand side of (B10): consequently, the subterm (of T_1) $\vee\langle\mathbf{F}\ ,\ \wedge\langle\mathbf{F}\ ,\ \mathbf{T}\rangle\rangle$ matched against p does not reappear in T_2. Thus, the subterm matched against p is *irrelevant* for producing the constant **T** in T_2: the 'creation' of this subterm **T** only requires the presence of the matched symbols "$\vee$" and "**T**". This observation is the keystone of our reduction-based slicing technique: We "track" those subterms that are relevant to each reduction steps; subterms that are relevant to *no* reduction step can then be eliminated from the slice.

The tracking process determines not only which subterms are relevant to a given reduction step, but also how subterms are combined and propagated by the reduction as a whole. To accomplish this task, we define for each reduction step that takes a term T_i and yields a new term T_{i+1} the notions of *creation* and *residuation*. These are binary relations between the nodes of T_i and the nodes of T_{i+1}. The creation relation relates the new symbols in T_{i+1} produced by the rewriting step to the nodes of T_i that matched the symbols in the left-hand side of the rewriting rule (making the rewriting step possible). The residuation relation relates every other node in T_{i+1} to the corresponding occurrence of the same node in the T_{i+1}. The dynamic dependence relation for a multi-step reduction r then consists, roughly speaking[3], of the transitive closure of creation and residuation relations for the rewriting steps in r. Fig. 5 shows all the relations for the example reduction discussed above.

For any reduction r which transforms a term T into a term T', a *term slice* with respect to some subcontext C of T' is defined as the subcontext S of T that is found by tracing back the dynamic dependence relations from C. The term slice S satisfies the following properties: (i) S reduces to a term C' containing context C via a reduction r', and (ii) r' is a subreduction of r. These properties

[3] The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. The exact problems posed by these rules are outside the scope of this paper, but are extensively discussed in [13].

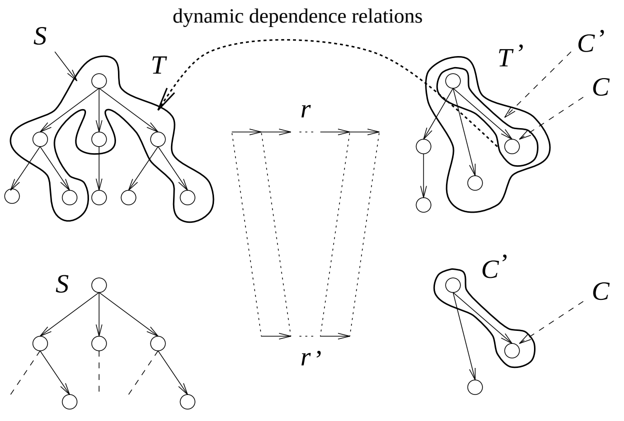


Figure 6: The concept of dynamic dependence.

are rendered pictorially in Fig. 6, and have the important implication that all the slices computed by our technique are effectively "executable" with respect to the rewriting semantics.

Our implementation maintains the transitive dependence relations between the nodes of the initial term and the nodes of the current term of the reduction by storing with each node n in the current term its term slice, which is the set of nodes in the initial term to which n is related. (The dependence relations associated with individual rewriting steps are not stored.) The term slice with respect to a subgraph S of T is then defined as the union of term slices with respect to the nodes in S.

Returning to the example of Fig. 5, we can determine the term slice with respect to the constant **T** in T_2 by tracing back all creation and residuation relations to T_0. By following the transitive relations in Fig. 5; the reader may verify that this slice consists of the subcontext $\vee\langle\vee\langle\mathbf{T}\ ,\ \bullet\rangle\ ,\ \bullet\rangle$.

3.1 Efficient Implementation of Term Rewriting

We implement term rewriting using the technique of *term graph rewriting* [4]. This technique extends the basic idea of term rewriting from labeled trees to rooted, labeled graphs, or *term graphs*. 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. For clarity, we will frequently depict PIM term graphs or subgraphs in "flattened" form as terms. (The flattened representation of the graph T_2 in Fig. 7, for instance, is $\vee\langle\wedge\langle\mathbf{T}\ ,\ \mathbf{F}\rangle\ ,\ \wedge\langle\mathbf{T}\ ,\ \mathbf{T}\rangle\rangle$.)

For certain kinds of rewriting rules, term graph rewriting has the effect of creating *shared* subgraphs where none existed previously. Consider following PIM boolean rule:

$$\wedge\langle p_1\ ,\ \vee\langle p_2\ ,\ p_3\rangle\rangle \quad = \quad \vee\langle\wedge\langle p_1\ ,\ p_2\rangle\ ,\ \wedge\langle p_1\ ,\ p_3\rangle\rangle \qquad \text{(B22)}$$

In rule (B22), the variable p_1 appears *twice* on the right-hand side. Although the left-hand side instance of p_1 in (B22) matches only a single subterm, the result of the rule application must contain two instances of the subterm matched by p_1. Rather than duplicating such a term, it can be shared, as illustrated by the example in Fig. 7, in which rule (B22) is applied to term $T_0 \equiv \wedge\langle\vee\langle\mathbf{T}\ ,\ \mathbf{F}\rangle\ ,\ \vee\langle\mathbf{F}\ ,\ \mathbf{T}\rangle\rangle$. We see also from Fig. 7 that the result of a single application of reduction rule (here, rule (B10)) *inside* a shared subterm can also be shared, thus giving the effect of multiple reductions for the price of one.

In general, graph rewriting is performed by *replacing* the subgraph matched by a rule with the graph corresponding to the rule's right hand side. The nodes in a replaced subterm that are not accessible from elsewhere in the graph are reclaimed by a memory manager. Since the PIM representation of programs contains many shared subgraphs, a graph rewriting implementation is critical to acceptable performance of the algorithm in practice.

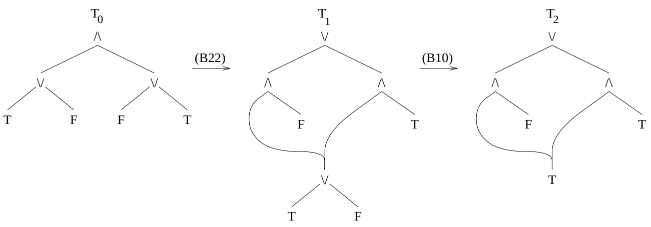


Figure 7: Creation of shared subgraphs and a shared reduction step using a graph rewriting implementation.

4 PIM $+$ Dynamic Dependence Tracking $=$ Slicing

PIM was designed to generalize and rationalize many of the properties of commonly used graphical representations for imperative programs such as SSA-form [7] and PDGs [10], and to provide a semantically sound but mechanizable framework for performing program analysis and optimization. PIM's formal progenitor is Cartwright and Felleisen's notion of *lazy store* [5], interpreted operationally rather than denotationally. Unlike SSA-form and PDGs, computations on addresses required for arrays or pointers are "first-class citizens," and procedures and functions are integral parts of the formalism.

4.1 μC-To-PIM Translation

Fig. 8 depicts a very simple μC program, P_1, its corresponding PIM representation, and several slicing-related structures.

The graph depicted in Fig. 8, denoted by $\mathit{Slice}(P_1, \mathtt{x}, \langle\rangle)$, is generated by translating P_1 to its corresponding PIM representation and embedding the resulting graph (labeled S_{P_1}) in a graph corresponding to the slicing criterion $\mathtt{x}$. $\mathit{Slice}(P_1, \mathtt{x}, \langle\rangle)$ is simply the PIM expression denoting the final value of the variable $\mathtt{x}$. Only a small number of graph edges, primarily those connecting shared subgraphs to multiple parents are shown explicitly in Fig. 8; we have flattened most other subgraphs for clarity. Parent nodes in the graph are depicted *below* their children to emphasize the correspondence between program constructs and corresponding PIM subgraphs.

S_{P_1} is generated by a simple syntax-directed translation. A representative subset of the translation rules appears in Fig. 9. The translation is specified in the Natural Semantics style [17] for clarity; however, the translation is implemented by a pure rewriting system[4]. The translation uses several sequent forms corresponding to the principal C syntactic components. The general form for these sequents is:

$$s \vdash\ c\ \Rightarrow\ t$$

Such a sequent may be read as "μC construct c translates to PIM term t, given initial (PIM) store s. '$\Rightarrow$' is subscripted by 'Pgm', 'Exp', or 'LValue', depending on whether a statement, expression, or L-value (address), respectively, is being translated. Pure expressions (those

[4] A rewriting system can be derived from simple classes of Natural Semantics specifications such as the one in Fig. 8 in a purely mechanical fashion.

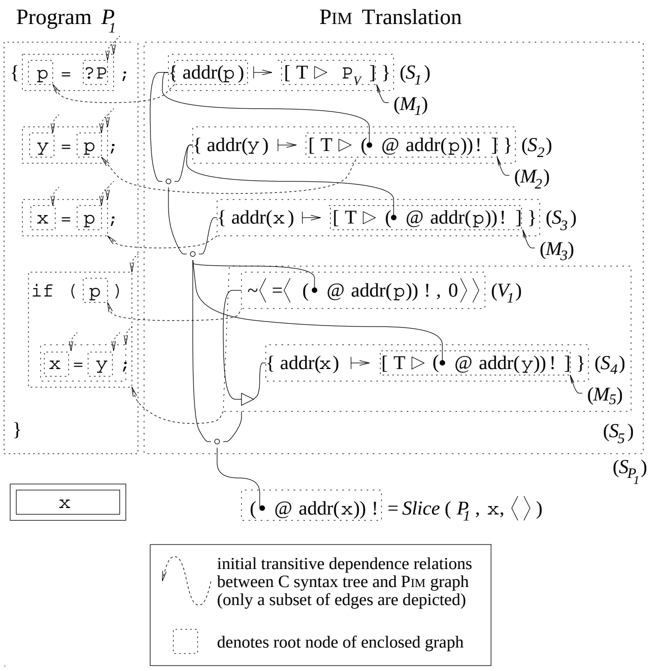

Program P_1
PIM Translation
{ p = ?P ;
y = p ;
x = p ;
if (p)
x = y ;
}
{ addr(p) ↦ [T ▷ $P_\mathcal{V}$] } (S_1)
(M_1)
{ addr(y) ↦ [T ▷ (• @ addr(p))!] } (S_2)
(M_2)
{ addr(x) ↦ [T ▷ (• @ addr(p))!] } (S_3)
(M_3)
~⟨ =⟨ (• @ addr(p)) ! , 0⟩ ⟩ (V_1)
{ addr(x) ↦ [T ▷ (• @ addr(y))!] } (S_4)
(M_5)
(S_5)
(S_{P_1})
x
(• @ addr(x)) ! = Slice (P_1 , x, ⟨ ⟩)
initial transitive dependence relations between C syntax tree and PIM graph (only a subset of edges are depicted)
denotes root node of enclosed graph

Figure 8: P_1 and its PIM representation, S_{P_1}. Major corresponding structures in P_1 and S_{P_1} are located side-by-side.

having no side-effects) and unpure expressions are distinguished in the translation process; subscripts p and u are used to denote the two types. The shared subgraphs in S_{P_1} arise from repeated instances of store variables in the antecedents of the translation rules in Fig. 9, as illustrated in Fig. 7.

The translation process establishes transitive dependence relations between nodes of the program's AST and the PIM graph S_{P_1}, as described in Section 3. Fig. 8 depicts a representative subset of these relations for the root nodes of certain subtrees of the syntax tree of P_1. We have used vestigial arrows in the syntax tree to indicate that nodes are referred to by *some* set of nodes in the PIM graph. We have also depicted statements of P_1 and their corresponding PIM subgraphs side-by-side.

4.2 Overview of PIM

In this section, we briefly outline the function of various PIM substructures using program P_1 and its PIM translation, S_{P_1}.

The graph S_{P_1} as a whole is a PIM *store* structure[5], essentially an abstract term representation of memory. S_{P_1} is constructed from the sequential composition (using the '$\circ$' operator) of substores corresponding to the statements comprising P_1. The subgraphs accessible from boxes labeled S_1–S_4 in Fig. 8 correspond to the four assignment statements in P_1. The simplest form of store is a *cell* such as

$$S_1 \equiv \{\mathtt{addr(p)} \mapsto [\mathbf{T} \rhd \mathtt{P}_\mathcal{V}]\}$$

A store cell associates an *address expression* (here $\mathtt{addr(p)}$) with a *merge structure*, (here $[\mathbf{T} \rhd \mathtt{P}_\mathcal{V}]$). *Constant* addresses such as $\mathtt{addr(p)}$ represent ordinary variables. More generally, address

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

$$(S_1)\qquad \frac{\begin{array}{c} s \vdash \{\text{ StmtList }\} \Rightarrow_{\text{Stmt}} u, \\ s \circ u \vdash \quad \text{Stmt} \quad \Rightarrow_{\text{Stmt}} u' \end{array}}{s \vdash \{\text{ StmtList Stmt }\} \Rightarrow_{\text{Stmt}} u \circ u'}$$

$$(S_2)\qquad \frac{s \vdash \text{ Exp } \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash \text{ Exp}\,\mathtt{;}\ \Rightarrow_{\text{Stmt}} u}$$

$$(S_3)\qquad \frac{\begin{array}{c} s \vdash \text{ Exp } \Rightarrow_{\text{Exp}} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u_S \end{array}}{\begin{array}{c} s \vdash \mathtt{if}\ \mathtt{(}\ \text{Exp}\ \mathtt{)}\ \text{Stmt} \Rightarrow_{\text{Stmt}} \\ u_E \circ (v'_E \rhd u_S) \end{array}} \qquad v'_E = \neg\langle =\langle v_E\ ,\ 0\rangle\,\rangle$$

$$(S_4)\qquad \frac{\begin{array}{c} x_S \vdash \text{ Exp } \Rightarrow_{\text{Exp}} \langle v_E,\ u_E\rangle, \\ s' \vdash \text{Stmt} \Rightarrow_{\text{Stmt}} u_S \end{array}}{\begin{array}{c} s \vdash \mathtt{while}\ \mathtt{(}\ \text{Exp}\mathtt{)}\ \text{Stmt} \Rightarrow_{\text{Stmt}} \\ Loop(\lambda x_S.body(u_E, v'_E, u_S), s) \end{array}} \qquad \begin{array}{l} s' = x_S \circ u_E \\ v'_E = \neg\langle =\langle v_E\ ,\ 0\rangle\,\rangle \end{array}$$

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

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

$$(E_{p\,1})\qquad s \vdash \text{ Id } \Rightarrow_{\text{Exp}_p} (s\ @\ \text{addr(Id)})\ !$$

$$(E_{p\,2})\qquad s \vdash \text{ ?Id } \Rightarrow_{\text{Exp}_p} \text{Id}_\mathcal{V}$$

$$(E_{u\,1})\qquad \frac{s \vdash \text{ Exp } \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash\ \star\ \text{Exp } \Rightarrow_{\text{Exp}_u} \langle((s \circ u)\ @\ v)\ !,\ u\rangle}$$

$$(E_{u\,2})\qquad \frac{s \vdash \text{ LValue } \Rightarrow_{\text{LValue}} \langle v,\ u\rangle}{s \vdash\ \&\,\text{LValue } \Rightarrow_{\text{Exp}_u} \langle v,\ u\rangle}$$

$$(E_{u\,3})\qquad \frac{\begin{array}{c} s \vdash \text{ Exp } \Rightarrow_{\text{Exp}_u} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{LValue} \Rightarrow_{\text{LValue}} \langle v_L,\ u_L\rangle \end{array}}{\begin{array}{c} s \vdash \text{LValue} = \text{Exp } \Rightarrow_{\text{Exp}_u} \\ \langle v_E,\ u_E \circ u_L \circ \{v_L \mapsto [\mathbf{T} \rhd v_E]\}\rangle \end{array}}$$

$$(E_{u\,4})\qquad \frac{\begin{array}{c} s \vdash \text{Exp}_1 \Rightarrow_{\text{Exp}_u} \langle v_1,\ u_1\rangle, \\ s \circ u_1 \vdash \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \langle v_2,\ u_2\rangle \end{array}}{\begin{array}{c} s \vdash \text{Exp}_1 + \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \\ \langle +\langle v_1\ ,\ v_2\rangle,\ u_1 \circ u_2\rangle \end{array}}$$

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

$$(L_u)\qquad \frac{s \vdash \text{ Exp } \Rightarrow_{\text{Exp}_u} \langle v,\ u\rangle}{s \vdash\ \star\ \text{Exp } \Rightarrow_{\text{LValue}_u} \langle v,\ u\rangle}$$

Figure 9: Representative translation rules for μC.

expressions are used when addresses are computed, e.g., in pointer references. '$\emptyset_s$' is used to denote the empty store.

Merge structures are a special kind of conditional construct containing ordered guarded expressions. The simplest form of merge expression is a *merge cell* such as $[\mathbf{T} \rhd \mathtt{P}_\mathcal{V}]$, in which some boolean predicate (here, $\mathbf{T}$) guards a value (here, the free PIM variable $\mathtt{P}_\mathcal{V}$ representing the μC meta-variable $\mathtt{?P}$). The formal consequence of the presence of a free variable is that any subsequent rewriting-based analysis is valid for *any* instantiation of the free variable.

Merge expressions m_1 and m_2 may be composed into *ordered lists* of the form $m_1 \circ_m m_2$, in which the *rightmost* guarded cell takes precedence. Such lists correspond roughly to Lisp **cond** expressions, and represent information similar to SSA-form ϕ nodes [7], particularly the *gated* SSA variant of [3]. Unlike normal conditional expressions, however, merges cannot evaluate to values unless they are referred to in a special context represented by the *selection* operation, '!'. Among other places, this operator is used in the translation of every variable reference. S_{P_1} contains no non-trivial merge structures, but such structures will arise in the simplification process. $\emptyset_m$ denotes the null merge structure. In the sequel, we will often drop subscripts distinguishing related store

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

and merge constructs when no confusion will arise.

In addition to guards in merge cells, stores such as S_5 (which corresponds to the '`if`' statement as a whole) may also be guarded. The guard expression 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.

The general form for the PIM graph constructed for a slice of program P at `x` given constraints

$$?\mathtt{X}_1 := \mathrm{Exp}_1, \ldots, ?\mathtt{X}_n := \mathrm{Exp}_n$$

is

$$\begin{aligned} &\mathit{Slice}(P, \mathtt{x}, \langle ?\mathtt{X}_1 := \mathrm{Exp}_1, \ldots, ?\mathtt{X}_n := \mathrm{Exp}_n \rangle) \\ &\quad \equiv ((S_P @ \mathrm{addr}(\mathtt{x}))!)\ [\mathtt{X}_{1\mathcal{V}} := v_1, \ldots, \mathtt{X}_{n\mathcal{V}} := v_n] \end{aligned}$$

where S_P is the PIM store to which P compiles, the $\mathtt{X}_{1\mathcal{V}}$ are free variables corresponding to the meta-variables and the v_i are PIM graphs corresponding to the value of the Exp_i (ignoring side effects). $\mathit{Slice}(P, \mathtt{x}, \langle \cdots \rangle)$ is the PIM representation of the value of `x` after execution of P, with substitutions for free variables defined by the constraints.

4.3 PIM Rewriting and Elimination of Dependences

PIM's equational logic consists of an "operational" subsystem, PIM$^{\rightarrow}$, plus a set of additional non-oriented equational rules for reasoning about operational equivalences in PIM$^{\rightarrow}$, instances of which can also be oriented for use in analysis. PIM$^{\rightarrow}$ is confluent and normalizing (assuming an appropriate strategy), thus it can be viewed as defining an operational semantics or interpreter for PIM terms. An important subsystem of PIM$^{\rightarrow}$ that defines the semantics of programs without loops or procedures, PIM$_t^{\rightarrow}$, is *canonical*, that is, *strongly* normalizing as well as confluent. PIM$^{\rightarrow}$ can be enriched with certain oriented instances of rules in (PIM$-$ PIM$^{\rightarrow}$) in such a way that confluence is preserved on closed terms, and such that unique normal forms for open terms exist up to certain trivial permutations. PIM's rules and subsystem structure are described in detail in [12]; key subsystems are reviewed in Appendix A.

Given a program P and a slicing criterion `x`, we use normalizing sets of oriented PIM equations to *simplify* $\mathit{Slice}(P, \mathtt{x}, \langle \cdots \rangle)$ graphs by reducing them to normal (i.e., irreducible) forms. From the point of view of slicing, the goal of this simplification process is *to eliminate in a sound, systematic way, as many subgraphs of* $\mathit{Slice}(P, \mathtt{x}, \langle \cdots \rangle)$ *as possible that do* not *affect its behavior*.

4.4 Reduction of Unconstrained and Constrained Slices

Fig. 10 depicts key steps in the reductions of $\mathit{Slice}(P_1, \mathtt{x}, \langle ?\mathtt{P} := 0 \rangle)$ and $\mathit{Slice}(P_1, \mathtt{x}, \langle \rangle)$, the slices of P_1 that result from these reductions, and certain dependence relations for reduction steps that are critical to producing the slices. These reductions share a common initial subsequence that is independent of the substitution generated in the constrained case. We have numbered certain important intermediate graphs in the reductions. The interpretation of several of these graphs (depicted in flattened form) is as follows:

Graph (1) is the flattened and abbreviated form of $\mathit{Slice}\,(P_1, \mathtt{x}, \langle \rangle)$. Graph (2) results from multiple applications of the rule

$$(s_1 \circ_s s_2) @ v \quad \longrightarrow \quad (s_1 @ v) \circ_m (s_2 @ v) \quad \text{(S4)}$$

which have the effect of distributing the reference to the variable `x`, `addr(x)`, to the sequence of substores S_1, S_2, S_3, S_5. Graph (3) results from applications of the rule

$$\{v_1 \mapsto m\} @ v_2 \quad \longrightarrow \quad =\langle v_1\ ,\ v_2 \rangle \rhd_m m \quad \text{(S1)}$$

to all but the rightmost subgraph. (S1) has the effect of converting references to store cells into conditional tests comparing the cell and dereferencing addresses; these predicates guard the merge cells M_1, M_2, M_3, and M_4, which are part of the original PIM graph S_{P_1}. Graph (4) results from evaluation of address comparisons. The comparison fails for assignments represented by S_1 and S_2 (which are irrelevant to `x`) and succeeds in the case of S_2 and S_4 (which both contain assignments to `x`). At (5), references to irrelevant assignments have been reduced to null merges. At (6), after eliminating null stores, the remaining expressions essentially represent the two definitions of `x` that "reach" its final value. Graph (7) is derived by first simplifying the expression containing merge structure M_3, yielding a merge cell containing the free meta-variable `?P`, then combining the PIM expression representing the predicate guarding the `if` statement, V_1, with a predicate derived from the address comparison for the nested store for the assignment in store S_4 (representing the result of the assignment inside the `if`).

The reduction thus far has the effect of eliminating all assignments irrelevant to the final value of `x`. At this point, the reductions in the constrained and unconstrained cases diverge:

4.4.1 Constrained Case: $\mathit{Slice}(P_1, \mathtt{x}, \langle ?\mathtt{P} := 0 \rangle)$

In the constrained case, $P_{\mathcal{V}}$ is bound to 0, i.e., is false. In step (8a), the highlighted application of the rule

$$\mathbf{F} \rhd_\rho l \quad \longrightarrow \quad \emptyset_\rho \quad \text{(L6)}$$

has the effect of eliminating the body of the `if` from the final slice. This can be seen in detail in the "exploded" (L6) rule application in Fig. 10. In this case, the only transitive dependence edges linking the constructs in the body of the `if` statement and the identifier `p` in the assignment `y = p;` have their origin in the subgraph S_4. When this subgraph is eliminated by the application of (L6), the constructs effectively disappear from the slice.

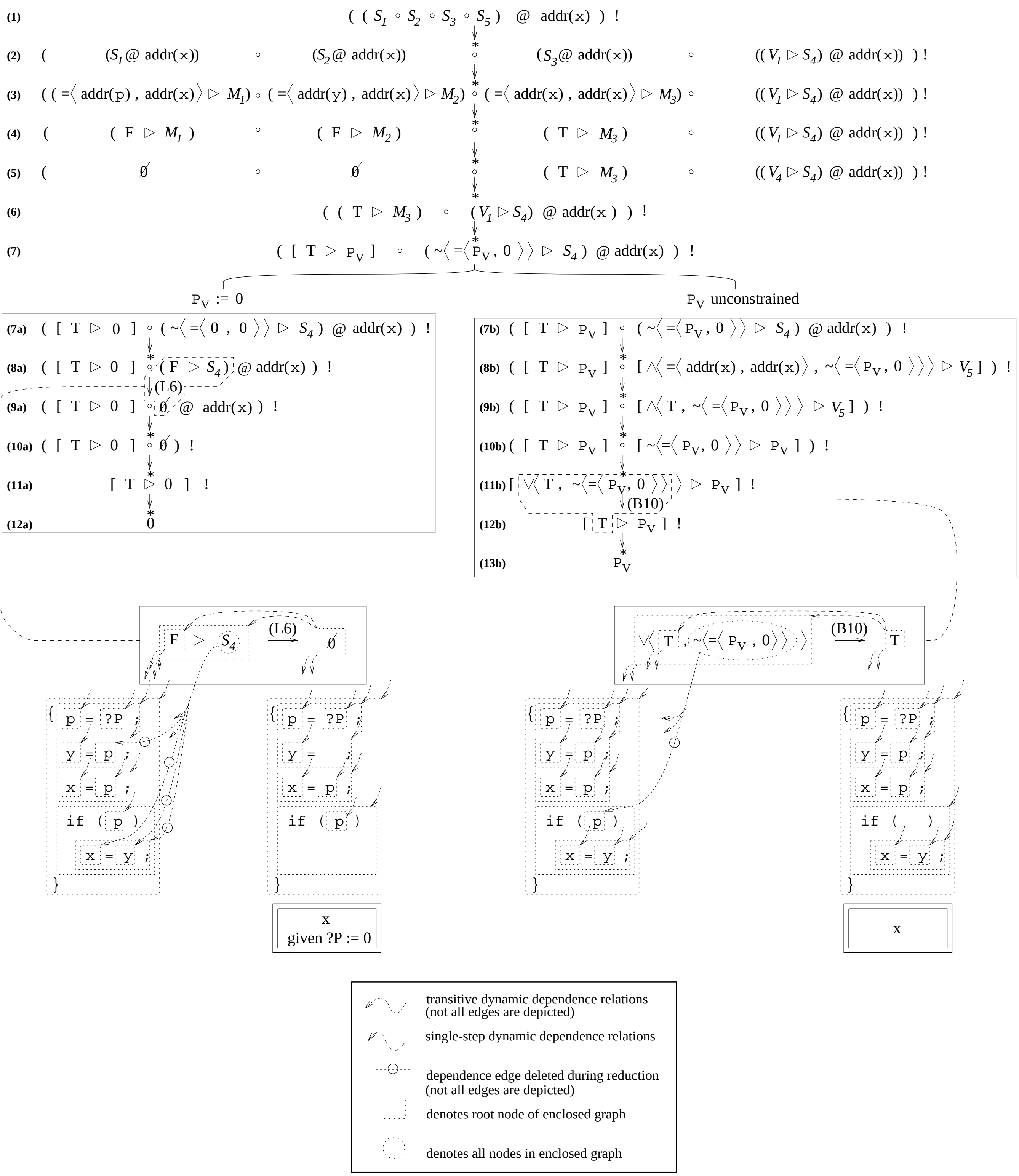
While it may appear that the slice results entirely from the application of a single rule, this rule is only the last of several rules that eliminate transitive edges from PIM nodes to the omitted constructs in P_1. Only when the last edges are eliminated does the construct disappear from the slice. Other rules have the effect of combining dependence edges emanating from several intermediate nodes into a single node (as the two single-step dependence edges in the depiction of rule (L6) illustrate).

4.4.2 Unconstrained Case: $\mathit{Slice}(P_1, \mathtt{x}, \langle \rangle)$

The unconstrained case is somewhat more interesting than the constrained case: Although we do not know the value of $P_{\mathcal{V}}$ and thus cannot effectively evaluate the `if` statement in P_1, we discover that the two reaching definitions for `x` both assign the *same* value to `x`, namely, $P_{\mathcal{V}}$. Application of several rules allows us to combine guards of merge cells with the same guarded value into the disjunctive expression shown in (11b).

The next step, the reduction of (11b) to (12b), discovers that the predicate's value itself is irrelevant to the final value of `x`. As the exploded view this rewrite step illustrates, there is *no* transitive dependence between the predicate `p` of the source AST and any of the nodes in the resulting term (12b) (or the final term (13b)). Consequently, the unconstrained slice does not contain the predicate of the `if` statement, though it does contain assignment statement within the `if` statement.

Slices which contain statements from the arms of a conditional statement but not its predicate, are unusual enough to deserve some discussion. Such slices indicate that the value of the predicate itself is irrelevant, even though the conditional statement contains



(1) ((S1 ∘ S2 ∘ S3 ∘ S5) @ addr(x)) !
(2) ((S1 @ addr(x)) ∘ (S2 @ addr(x)) ∘ (S3 @ addr(x)) ∘ ((V1 ▷ S4) @ addr(x))) !
(3) ((=⟨ addr(p) , addr(x) ⟩ ▷ M1) ∘ (=⟨ addr(y) , addr(x) ⟩ ▷ M2) ∘ (=⟨ addr(x) , addr(x) ⟩ ▷ M3) ∘ ((V1 ▷ S4) @ addr(x))) !
(4) ((F ▷ M1) ∘ (F ▷ M2) ∘ (T ▷ M3) ∘ ((V1 ▷ S4) @ addr(x))) !
(5) (∅ ∘ ∅ ∘ (T ▷ M3) ∘ ((V4 ▷ S4) @ addr(x))) !
(6) ((T ▷ M3) ∘ (V1 ▷ S4) @ addr(x)) !
(7) ([T ▷ PV] ∘ (~⟨ =⟨ PV , 0 ⟩⟩ ▷ S4) @ addr(x)) !
PV := 0
PV unconstrained
(7a) ([T ▷ 0] ∘ (~⟨ =⟨ 0 , 0 ⟩⟩ ▷ S4) @ addr(x)) !
(8a) ([T ▷ 0] ∘ (F ▷ S4) @ addr(x)) !
(L6)
(9a) ([T ▷ 0] ∘ ∅ @ addr(x)) !
(10a) ([T ▷ 0] ∘ ∅) !
(11a) [T ▷ 0] !
(12a) 0
(7b) ([T ▷ PV] ∘ (~⟨ =⟨ PV , 0 ⟩⟩ ▷ S4) @ addr(x)) !
(8b) ([T ▷ PV] ∘ [∧⟨ =⟨ addr(x) , addr(x) ⟩ , ~⟨ =⟨ PV , 0 ⟩⟩⟩ ▷ V5]) !
(9b) ([T ▷ PV] ∘ [∧⟨ T , ~⟨ =⟨ PV , 0 ⟩⟩⟩ ▷ V5]) !
(10b) ([T ▷ PV] ∘ [~⟨ =⟨ PV , 0 ⟩⟩ ▷ PV]) !
(11b) [∨⟨ T , ~⟨ =⟨ PV , 0 ⟩⟩⟩ ▷ PV] !
(B10)
(12b) [T ▷ PV] !
(13b) PV
F ▷ S4 (L6) ∅
∨⟨ T , ~⟨ =⟨ PV , 0 ⟩⟩⟩ (B10) T
{ p = ?P ; y = p ; x = p ; if (p) x = y ; }
{ p = ?P ; y = ; x = p ; if (p) }
x given ?P := 0
{ p = ?P ; y = p ; x = p ; if (p) x = y ; }
{ p = ?P ; y = p ; x = p ; if ( ) x = y ; }
x
transitive dynamic dependence relations (not all edges are depicted)
single-step dynamic dependence relations
dependence edge deleted during reduction (not all edges are depicted)
denotes root node of enclosed graph
denotes all nodes in enclosed graph

Figure 10: Reduction of $\mathit{Slice}(P_1, \mathtt{x}, \langle\rangle)$ and $\mathit{Slice}(P_1, \mathtt{x}, \langle \mathtt{?P} := 0 \rangle)$. Steps in which removal of dependence edges eliminates constructs from slices are highlighted along with the resulting slices.

some relevant statement, e.g., an assignment to some relevant variable. Such situations *can* arise in realistic programs. Consider, for example, the statement

```
if (P) f(foo); else f(bar);
```

where the procedure `f` has some side-effect on some variable `x` of interest, and where this side-effect itself is independent of the argument to the procedure. Here, the two call statements are relevant to the final value of `x`, though the predicate itself is irrelevant. This reflects a kind of reasoning that programmers do use when analyzing a program backwards, and can result in substantially smaller slices because of the elimination of the statements that the predicate itself depends on. The possibilities for computing more precise slices in this fashion are even greater in the case of constrained slicing.

4.5 Slicing and Reduction Strategies

As PIM$^{\rightarrow}$ is a confluent rewriting system, reductions may be performed anywhere in a graph without affecting the final term produced (assuming the reduction terminates at all). This "stateless" property of reduction systems accommodates a variety of performance tradeoffs derived from varying the reduction *strategy*. We use an outermost or "lazy" strategy, which ensures that only steps that contribute to a final result are performed. (Note, however, that the reduction depicted in Fig. 10 uses a strategy that is not strictly outermost to better illustrate the properties of certain intermediate terms). Alternatively, the PIM representation of the entire program could be normalized "eagerly" prior to the specification of any slicing criterion; those steps specific to the criterion or constraints could be performed later.

Reduction strategies can also have an effect on slices. In the constrained case, both of the reductions depicted in Fig. 10 are valid, and, consequently *both* of the slices depicted are also valid. Slices are therefore not necessarily unique, even when the underlying reduction system is confluent. However, our reduction strategy favors the left reduction over the right one in the constrained case. Intuitively, this favors a "standard" execution semantics that corresponds most closely with results of traditional program slicing algorithms.

4.6 Slicing at Intermediate Program Points

Although our discussion thus far has concentrated on computing slices with respect to the final values of variables, our approach is capable of computing slices with respect to any expression at any program point. Conceptually, a slice with respect to a μC expression e (assumed to be side-effect free) at some specific program point can be computed as follows: First, introduce a new variable `v` and an assignment of the form

$$\texttt{v} = op\langle \texttt{v}\,,\, e\rangle\,;$$

at the program point of interest, where *op* is an abstract, uninterpreted operator. Then, compute the slice with respect to the final value of `v`. Variable `v` has the effect of accumulating the sequence of values the expression takes on at the desired program point.

In practice, it is not necessary to alter the program in order to compute slices at intermediate points. An implementation can instead construct and maintain a reference to the PIM store subgraph s_p corresponding to every program point p (note that the graphs representing these stores will generally have many nodes in common). The slice with respect to the program point of interest is then computed by normalizing the PIM expression corresponding to the translation of e in initial store s_p.

4.7 Conditional Constraints

A slice with respect to a *conditional constraint* such as that depicted in Fig. 4(c) can be computed by constructing a PIM graph roughly equivalent to that which would be produced by inserting the body of the program in a conditional statement where the predicate is the conjunction of all such constraints.

The effectiveness of our slicing algorithm in handling conditional constraints depends primarily on its ability to reason about the operations allowed in such constraints. The extensible nature of our approach makes it easy to augment the slicing algorithm by incorporating sophisticated reasoning capabilities about particular domains into the slicing algorithm, as it only involves adding rewrite rules characterizing the appropriate domains. For example, in the case of Fig. 4(c), rudimentary rules for reasoning about arithmetic inequalities suffice to compute the slice shown.

4.8 Complexity Tradeoffs

Use of different normalizing subsets of PIM equations allows various accuracy/time tradeoffs in the analysis process. For instance, pointer-induced alias analysis is NP-complete even in the absence of loops and procedures [22], although such analysis is usually tractable in practice. By including or excluding appropriate PIM rules, one can effectively choose more precise (but potentially slow) or more conservative (but guaranteed fast) pointer analysis. For instance, eliminating rule (M3) (see Fig. 18) effectively inhibits propagation of symbolic addresses representing pointer values, thus preventing these expressions involving these addresses from being resolved or simplified. Rule (L11) has the effect of joining common results of common expression propagation (including address expressions) in different branches of a conditional, and can be enabled, disabled, or restricted to prevent or allow such propagation.

The result of more accurate pointer analysis in slicing is manifested by elimination of more subgraphs of the program representation that are irrelevant to the slicing criterion. A similar phenomenon occurs with simplification of boolean predicates involved in conditionals.

5 Variations on a Looping Theme

This section discusses a number of PIM variants suitable for computing slices in loops. These sets of rules may be used as building blocks for generating a variety of different slicing algorithms without changing the underlying algorithmic framework. In the sequel, we will use COREPIM$^{\rightarrow}$ to denote those PIM rules that are not loop related and are common to the slicing variants we will present. While the COREPIM$^{\rightarrow}$ rules allow addresses to be stored as values and manipulated, the analysis rules presented in this section assume for simplicity that no pointers are used. The ideas in this section can be adapted easily to produce conservative slices in the presence of pointers; more precise pointer analysis is also possible, but requires more sophisticated rules for reasoning about address equivalence.

5.1 Loop Execution Rules: Pure Dynamic Slicing

Loops are represented in PIM by terms of the form

$$Loop(\lambda x_S . body(u_E, v_E, u_S), s)$$

Informally, u_E is a store representing the side-effects of evaluating the loop predicate, v_E represents the value of the predicate, u_S is a store representing the side-effects of the loop body, all as functions of the store x_S at the beginning of a loop iteration. The second argument s is the incoming store. The term

$$
\begin{array}{l}
\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s) \quad \longrightarrow \\
\qquad S((\ \mathbf{Y}\ \lambda f_{\mathcal{L}} \lambda x_{\mathcal{S}}. \\
\qquad\qquad \underline{(u_E \circ_s (v_E \rhd_s (u_S \circ_s} \\
\qquad\qquad\quad \underline{S(f_{\mathcal{L}}\ (x_{\mathcal{S}} \circ_s u_E \circ_s u_S)))))}\ ) \quad s) \quad (\text{loop}) \\
\quad (\text{where } f_{\mathcal{L}} \notin (FV(u_E) \cup FV(u_S) \cup FV(v_E)))
\end{array}
$$

$$
\begin{array}{rcll}
(\lambda x.f)\ \ g & \longrightarrow & f[x := g] & (\beta) \\
(\mathbf{Y}\ \ f) & \longrightarrow & f\ \ (\mathbf{Y}\ \ f) & (\text{recursion})
\end{array}
$$

Figure 11: Loop execution rules

$$
\begin{array}{l}
\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s) \quad \longrightarrow \\
\qquad\qquad \mathit{Project}(\ \mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s), \\
\qquad\qquad\qquad \mathit{AssignedVars}(u_E \circ_s u_S)\ ) \qquad\qquad (\text{LA1})
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Project}(s, \{\}) & \longrightarrow & \emptyset_s & (\text{LA2}) \\
\mathit{Project}(s, \{v\} \cup r) & \longrightarrow & (v \mapsto s\ @\ v) \circ_s \mathit{Project}(s, r) & (\text{LA3})
\end{array}
$$

$$\vdots$$

rules for computing $\mathit{AssignedVars}(s)$,
the set of variables assigned to in the store s

$$\vdots$$

Figure 12: ϕ rules

$\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s)$ itself denotes the store representing the side-effects of executing the loop until the predicate evaluates to false. The rewriting rules in Fig. 11, which we will refer to as loop execution rules, specify this behavior formally. The underlined subterm of the right-hand side of the rule (loop) may be read as: (the side-effects of executing the loop consists of) the side-effects of evaluating the loop predicate and, if the predicate evaluates to true, the side-effects of executing the loop body once, composed with the side-effects of executing the same loop with an appropriately updated store, namely, $x_{\mathcal{S}} \circ_s u_E \circ_s u_S$. The rest of the term serves to express the recursion using the recursion combinator $\mathbf{Y}$. $f[x := g]$ represents the result of substituting g for free occurrences of x in f (with the usual provisos about variable capture and renaming), $FV(\tau)$ is the set of free variables in a term τ, and 'S' is a technicality—a "sort coercion" operator that has no semantic content. [11] shows how the β rule and substitution can be encoded as pure rewriting rules.

Utilizing these rules and CorePim$^{\rightarrow}$ during the simplification phase leads to a straightforward dynamic slicing algorithm that we call a *pure dynamic slicing algorithm*. Sec. 2.4 discusses an example (Fig. 3(b)) of this sort of slice.

5.2 ϕ Rules: Lazy Dynamic Slicing

Fig. 12 depicts a set of rules, the ϕ rules, that statically simplify Pim stores generated by loops. The effect of ϕ rules on the Pim representation is essentially to introduce an SSA-form ϕ node [7] for every variable that might be assigned a value inside the loop. In terms of slicing, these rules have the effect of permitting loops to be removed from slices if it can be determined (statically) that the loop cannot assign to any "variable of interest".

We will refer to the slicing algorithm obtained by using both the loop execution rules and the ϕ rules in conjunction with CorePim$^{\rightarrow}$ as a *lazy dynamic slicing algorithm*. This slicing algorithm computes traditional dynamic slices, such as Fig. 3(c), as well as the somewhat more unusual result in Fig. 3(d).

The slices produced by our lazy dynamic slicing algorithm are closer to the slices produced by the Agrawal-Horgan algorithm [1] than to the slices produced by the Korel-Laski algorithm [21]. The Korel-Laski slices tend to be larger than the Agrawal-Horgan slices since they, unlike the Agrawal-Horgan slices, are executable. Our dynamic slices, though not executable under the "standard semantics", *are* executable with respect to the semantics specified by the rewriting rules.

```
x = ?X;            x = ?X;
y = x;                 ;
if (x < 0)         if (x < 0)
  x = -x;                ;
z = x;             z = x;
```

```
  z
given ?X := 5
```

(a) **(b)**

Figure 13: **(a)** Example program. **(b)** Dynamic slice at z given ?X := 5.

Fig. 13(a) illustrates an important difference between our algorithm and previous dynamic slicing algorithms. Since the `if` predicate evaluates to false, previous dynamic slicing algorithms exclude the predicate (and any of the statements the predicate evaluation is dependent upon) from the dynamic slice with respect to z. However, as has been observed before [21, 25, 2], the predicate does "affect" the final value of z, and in applications such as debugging it is useful to include these statements in the slice. In this sense, our slicing algorithm produces a slice that is semantically more consistent than existing dynamic slicing algorithms.

5.3 Loop Splitting Rules: Static Loop Slicing

Fig. 14 contains the essential subset of a collection of rules that we will refer to as loop splitting rules, which can be used in conjunction with CorePim$^{\rightarrow}$ to compute a classical static slice. The goal of these rules is quite simple. Consider the Pim term

$$\mathit{Slice}(\text{while}(\text{i} < 10)\{\text{j} = \text{j} + 2; \text{i} = \text{i} + 1;\}, \text{i}, \langle\rangle)$$

This reduces to a term of the form

$$\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s)\ @\ \text{addr}(i)$$

where u_S, representing the loop body, is the store

$$
\begin{array}{l}
\{\text{addr}(j) \mapsto \mathbf{T} \rhd_v +\langle (x_{\mathcal{S}}\ @\ \text{addr}(j))!\ ,\ 2\rangle\} \\
\quad \circ_s \{\text{addr}(i) \mapsto \mathbf{T} \rhd_v +\langle (x_{\mathcal{S}}\ @\ \text{addr}(i))!\ ,\ 1\rangle\}
\end{array}
$$

Intuition suggests that this term should be reducible to

$$\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, {u_S}'), s)\ @\ \text{addr}(i)$$

where ${u_S}'$ is the store

$$\{\text{addr}(i) \mapsto \mathbf{T} \rhd_v +\langle (x_{\mathcal{S}}\ @\ \text{addr}(i))!\ ,\ 1\rangle\}$$

Such reductions are crucial to computing static (and constrained) slices. In general, we would like to reduce a term of the form

$$\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(s_1, p, s_2), s_3)\ @\ a$$

to a term

$$\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}({s_1}', p, {s_2}'), {s_3}')\ @\ a$$

$$\begin{array}{rcll} \multicolumn{3}{l}{Loop(\lambda x_S.body(u_E, v_E, u_S), s) \mathbin{@} v \longrightarrow} & \\ \multicolumn{3}{r}{(Loop(\lambda x_S.body(u_E, v_E, u_S), s) \mathbin{@@} \{v\}) \mathbin{@} v} & \text{(SA1)} \\ (a \mapsto m) \mathbin{@@} l & \longrightarrow & (a \in l) \rhd_s (a \mapsto m) & \text{(SA2)} \\ \emptyset_s \mathbin{@@} l & \longrightarrow & \emptyset_s & \text{(SA3)} \\ (s_1 \circ_s s_2) \mathbin{@@} l & \longrightarrow & (s_1 \mathbin{@@} l) \circ_s (s_2 \mathbin{@@} l) & \text{(SA4)} \\ (g \rhd_s s) \mathbin{@@} l & \longrightarrow & g \rhd_s (s \mathbin{@@} l) & \text{(SA5)} \end{array}$$

$$\frac{\begin{array}{l} r \supseteq (\ l \cup Demand(v_E, x_S) \cup Demand(u_E \mathbin{@@} r, x_S) \\ \qquad \cup\, Demand(u_S \mathbin{@@} r, x_S)\ ) \ = \ \mathbf{T} \end{array}}{\begin{array}{l} Loop(\lambda x_S.body(u_E, v_E, u_S), s) \mathbin{@@} l \longrightarrow \\ \qquad Loop(\lambda x_S.body(u_E \mathbin{@@} r, v_E, u_S \mathbin{@@} r), s \mathbin{@@} r) \end{array}} \quad \text{(SA6)}$$

$$\vdots$$

rules for computing $Demand(s, x_S)$, the set of
addresses dereferencing free instances of x_S in s

$$\vdots$$

Figure 14: Loop splitting rules

where each ${s_i}'$ is a "restriction" of the original store s_i to the addresses that are relevant, given that we are interested only in the final value at address a. We need to do two things here. First, we need to identify the set r of relevant addresses (variables), second, we need to perform the actual restriction of the stores to the relevant variables.

The rules in Fig. 14 show the essence of what we need to do. Rule (SA1) simply transforms a dereference operation on a store computed by a loop into a corresponding dereference operation on a *restriction* of the loop-computed store. This leaves the bulk of the work to the operator @@, whose purpose is to restrict a store to a set of addresses of interest. Rule (SA2) is the key rule defining the behavior of this operator. (The operator $\in$ may be interpreted as denoting the usual set-theoretic member function.)

The rule of primary interest is (SA6), which performs the restriction operation for a store generated by a loop. Restricting a loop-computed store with respect to a set l of addresses requires restricting the loop body store and the initial incoming store with respect to a set of addresses r. The set r is a superset of the set l, and effectively accounts for loop-carried dependences. The antecedent of rule (SA6) specifies the condition this set r has to satisfy, namely that the set of variables r should include the set l, the set of variables required to compute the loop predicate, and the set of variables necessary to compute values assigned to the variables in r within the loop. Put another way, the antecedent of (SA6) ensures that the set of variables r is transitively closed with respect to loop-carried dependences. The auxiliary function $Demand(t, x_S)$, roughly, identifies "upwards exposed" variables in t. More formally, given a store or merge t, $Demand(t, x_S)$ identifies dereferences of the free store variable x_S in t and collects the address operands of such dereferences.

Rule (SA6) is not a pure rewriting rule, since the variable r in the antecedent of the rule is not bound in the left-hand side of the rewrite rule. Applying rule (SA6) thus requires computing some solution r to the constraint expressed by the antecedent. The computation of the least solution of this constraint can be performed easily using rewriting rules that compute an iterative computation of the constraint's least fixed point.

Rules expressed in a "constraint" style such as (SA6) have the advantage that they can accommodate analysis algorithms implemented by non-rewriting means (and thus for which dependence tracking cannot be performed). Observe that (SA6) is valid for every possible instantiation of the variable r—thus, one may view (SA6) as a rule schema describing infinitely many rewriting rules. One may then use any mechanism whatsoever to choose an instantiation for the rule, treating the instantiation as an ordinary rewriting rule with respect to dependence tracking. This approach ensures that the dependence information is computed correctly, notwithstanding the use of an external analysis algorithm.

5.4 Loop Invariance Rules: Invariance-Sensitive Slicing

We now turn our attention to the final set of rules, which we will refer to as loop invariance rules. We will refer to the slicing algorithm obtained by using these rules, the CorePim$^{\rightarrow}$ rules, and the loop splitting rules, as an *invariance sensitive* slicing algorithm. If the loop execution rules are used as well, we obtain a β *invariance-sensitive* algorithm. The primary difference between the two algorithms is that the latter will execute (i.e., unfold) a loop as long as its predicate evaluates to a constant. Fig. 15 illustrates this behavior. The β invariance-sensitive slicing algorithm, by executing the loop, discovers that *two* of the three assignments to `y` in the loop are irrelevant for the given input constraint `?N := 5`. The simpler algorithm avoids unfolding the loop; however, by effectively performing constant propagation, it discovers that *one* of the three assignment statements is irrelevant.

We describe the goals of the loop invariance rules below. Consider a store of the form

$$Loop(\lambda x_S.body(u_E, v_E, u_S), s) \mathbin{@} a$$

The store u_S represents the loop body, and free occurrences of x_S in u_S denote the store at the beginning of a specific loop iteration. In the presence of loop invariants, one can simplify the store u_S further. For instance, consider the example in Fig. 15(c). The store u_S compiled for the loop body will contain subterms of the form $(x_S \mathbin{@} \mathrm{addr}(x))!$, denoting the value of variable `x` in a particular iteration. Since the value of `x` is a loop-invariant constant `6` (given the constraint `?N := 5`), we would like to replace this term by `6`. This replacement, in turn, will allow further simplifications of the store, and ultimately lead to the slice depicted in the figure.

Achieving this kind of simplification requires us to do two things: We must identify the loop-invariant component of the store, and we must specialize the loop body (and the loop predicate) with respect to the loop invariant component of the store. The second task is relatively trivial. Once the loop invariant component s_{inv} of the store has been identified, we can replace the free occurrences of x_S in the loop body by $x_S \circ_s s_{inv}$. The rest of Pim will then take care of the specialization.

The rules in Fig. 16 formalize these intuitions. The most important rule is (IA1), a conditional rule in the style of rule (SA6) (Fig. 14). The consequent of the rule specializes the loop body and loop predicate of a loop-computed store with respect to the loop-invariant part of the store, namely s_{inv}. The antecedent guarding the applicability of the rule "defines" what it means for a part of the store to be loop invariant. This definition is stated in terms of a *subsumption* relationship $\succeq$ between program stores. A store s_1 subsumes a store s_2, if for every variable x assigned a value v in store s_2, x is also assigned the same value v in store s_1. The subsumption relation is concisely defined by the equational axiom (IA2). Rules (IA2.1) through (IA2.3) represent a conservative approximation to the $\succeq$ that is more "directly computable," since it is defined inductively. Less conservative approximations to (IA2) can also be defined that allow inference of more complex loop invariants.

Returning to the notion of a loop-invariant store, the store s_{inv} is considered to be loop invariant if (a) The incoming store s (the store before the loop begins its first iteration) subsumes s_{inv}, and (b) The loop body $u_E \circ_s u_S$, specialized for an incoming store $x_S \circ_s s_{inv}$ that subsumes s_{inv}, and then composed with s_{inv} results in a store that subsumes s_{inv}.

```
n = ?N; z = ?Z;    n = ?N; z = ?Z;    n = ?N; z = ?Z;
x = n + 1;         x = n + 1;         x = n + 1;
i = 1; y = 0;      i = 1; y = 0;      i = 1; y = 0;
while (i < n) {    while (i < n) {    while (i < n) {
  if (x > 100)       if (x > 100)       if (x > 100)
    y = y + 100;                  ;                  ;
  else if (y < 99)   else if (y < 99)   else if (y < 99)
    y = y + x;         y = y + x;         y = y + x;
  else               else               else
    y = y + 50;                  ;          y = y + 50;
  z = z + y;         z = z + y;         z = z + y;
  i = i + 1;         i = i + 1;         i = i + 1;
}                  }                  }

                     z                  z
                   given ?N := 5      given ?N := 5

      (a)                (b)                (c)
```

Figure 15: **(a)** Example program. **(b)** β invariance-sensitive slice at z given ?N $:=$ 5. **(c)** Simple invariance-sensitive slice at z given ?N $:=$ 5.

$$\frac{\begin{array}{c} s \succeq s_{inv} = \mathbf{T}, \\ s' = (x_{\mathcal{S}} \circ_s s_{inv}), \\ s_{inv} \circ_s ((u_E \circ_s u_S)[x_{\mathcal{S}} := s']) \succeq s_{inv} = \mathbf{T} \end{array}}{\begin{array}{l} Loop(\lambda x_{\mathcal{S}}.body(u_E, v_E, u_S), s) \longrightarrow \\ \quad Loop(\lambda x_{\mathcal{S}}.body(u_E[x_{\mathcal{S}} := s'], \\ \qquad v_E[x_{\mathcal{S}} := s'], \\ \qquad u_S[x_{\mathcal{S}} := s']), \; s) \end{array}} \qquad (IA1)$$

$$\frac{s_1 \circ_s s_2 = s_1}{s_1 \succeq s_2 \longrightarrow \mathbf{T}} \qquad (IA2)$$

$$s \succeq \emptyset_s \longrightarrow \mathbf{T} \qquad (IA2.1)$$

$$\frac{s \succeq s_1 = \mathbf{T}, s \succeq s_2 = \mathbf{T}}{s \succeq (s_1 \circ_s s_2) \longrightarrow \mathbf{T}} \qquad (IA2.2)$$

$$\frac{=\langle (s \;@\; a)! \,,\; m! \rangle = \mathbf{T}}{s \succeq \{a \mapsto m\} \longrightarrow \mathbf{T}} \qquad (IA2.3)$$

Figure 16: Loop invariance rules

As with rule (SA6) discussed in the section on static slicing, rule (IA1) cannot be used directly by the dependence tracking system. However, we can use the rule in conjunction with any algorithm for identifying loop invariants, such as the conditional constant propagation algorithm of Wegman et al [31].

6 Pragmatics

A prototype implementation of our methods has been completed using the ASF+SDF Meta-environment [19]. The results obtained from this prototype have been encouraging, and we are now engaged in implementing a "free-standing" reduction-based slicing system using the most efficient possible implementation techniques. In this section, we briefly touch on several pragmatic that arise in implementing our approach.

6.1 Properties of Graph Reduction

Term graph reduction is a simple technique that can be implemented efficiently when an automaton-based matching algorithm and outermost reduction strategies are used. This leads us to believe that it should scale well to relatively large programs. Graph reduction also has the advantage that results of reductions performed in shared subgraphs are immediately available to all supergraphs from which they are accessible. This means, e.g., that a slice can be computed in a subprogram (such as a procedure), and the results later used in computing the slice with respect to the entire program. It also means that reduction steps that are independent of a given criterion, but dependent on the program, can be shared and reused when new criteria are supplied.

6.2 Alternative Translation Algorithms

As alluded to in Section 2.2, it is not strictly necessary to use a rewriting system to translate a source program to PIM. Any algorithm to perform the translation suffices, *provided* that the dynamic dependence relations between the source AST and its PIM translation are correctly initialized. However, the correctness of these initial relations must be established by hand.

6.3 Chain Rules

Chain rules in a language's *abstract* syntax can be used to distinguish classes of syntactically related program constructs that have differing semantic properties. For instance, in our C grammar, we distinguish between "pure" expressions and those that may have side-effects. Dependences traced by CR-tracking to such nodes can be used to single out a particular property of a construct that causes it to be included or excluded from a slice.

7 Related Work

PIM was introduced as a semantically sound internal representation for program analysis in [12]. The theoretical underpinnings of the notion of dynamic dependence were developed in [13] for arbitrary term rewriting systems. In this paper we have augmented PIM's logic (particularly for loop analysis), and applied the notion of dynamic dependence to it to develop a family of extensible slicing algorithms for standard programming languages, exploiting in particular the possibility of computing slices with respect to constraints.

Some previous algorithms [6, 8, 18] combine both static and dynamic information to compute slices, but primarily to combine the efficiency of static slicing algorithms with the accuracy of dynamic slicing algorithms. The notion of constrained slices is not studied in these papers.

Constrained slicing and *partial evaluation* of programs are closely related, in a manner similar to the relationship between dynamic slicing and standard program evaluation. However, constrained slices cannot be obtained simply by partially evaluating a program, then computing a static slice from the residual program that results—one must also relate slices in the partially evaluated program to the source program; this is not necessarily a trivial task.

Consider the example in Fig. 17. Given the input constraint ?X $:=$ 5, the program in Fig. 17(a) can be simplified using constant propagation and dead code elimination to yield the program shown in Fig. 17(b). However, a static slice of this optimized program at z fails to provide the same information as our constrained slice of the original program with respect to the same criterion (Fig. 17(c)). This is due to the fact that the predicate of the if statement (which evaluates to false) is relevant to the computation of the final value of z, and should therefore be included in the slice. Slicing is intended to indicate *how* the value of a variable or expression is computed, not merely *what* its value may be. For further details on the relation between previous work on partial evaluation and PIM, see [12].

```
x = ?X;                ;       x = ?X;
y = ?Y;        y = ?Y;         y = ?Y;
if (x < 0)                     if (x < 0)
  y = -y;                ;              ;
z = y;         z = y;          z = y;

               +-------------+ +-------------+
               |      z      | |      z      |
               | given ?X := 5 | | given ?X := 5 |
               +-------------+ +-------------+

  (a)              (b)             (c)
```

Figure 17: **(a)** Example program. **(b)** Example program after optimization using constant propagation and dead code elimination, given ?X := 5. The static slice of the optimized program at z is the optimized program itself. **(c)** Our constrained slice at z given ?X := 5.

Ernst [9] presents an algorithm for static slicing that is similar to our algorithm in certain respects. In particular, Ernst describes how conventional program optimization techniques can be used to produce smaller and better slices. The internal representation that Ernst uses, the value dependence graph (VDG), has similarities with Pim, and the process of optimization itself consists of transforming the VDG. Ernst refers to the problem of maintaining a correspondence between the VDG and the source code graph throughout the optimization process, and mentions that this correspondence enables a history mechanism for explaining the transformations that were performed. No details are presented as to how this is done, but this aspect of Ernst's work appears to be the analogue of the notion of dynamic dependence used in our work to maintain a similar correspondence.

Tip shows that dynamic dependence tracking can be used to compute accurate *dynamic* slices from a simple "interpreter-style" semantics for a language, and that these techniques are useful for debugging [27]. [29] contains a comprehensive discussion of many issues related to slicing, dynamic dependence tracking, and use of algebraic specifications in generating program analysis tools.

While we have yet to undertake a formal comparison of the complexity of our approach with that of earlier methods based on dataflow analysis or dependence graphs, informal analysis indicates that for comparable types of slices, our approach should be quite competitive with previous techniques. One characteristic of our approach that must be kept in mind in any complexity analysis is that aspects of both intermediate representation construction and analysis *using* the intermediate representation are combined into reduction steps. For instance, in comparing our work with PDG-based algorithms, it is apparent that there is a close correspondence between most steps involved in PDG construction and certain Pim rewriting steps. The computation of the slice itself in PDG-based approaches requires a graph traversal that corresponds roughly to traversing the set of dynamic dependence relations in a reduced Pim term.

One advantage of our approach over PDG-based methods is that by using an outermost "lazy" graph reduction strategy, the analysis performed is effectively *demand-driven*. Thuse only those reduction steps directly relevant to the slicing criterion are performed. In this respect, our approach has the potential to outperform prior techniques that may eagerly compute dataflow information that is never used.

8 Future Work

There are several areas for future research: We are currently exploring the issues involved in extending our techniques to handle arbitrary control flow, arrays, address arithmetic, dynamic memory allocation, and procedures. We also intend to study various Pim subsystems and reduction strategies in isolation to determine their worst-case complexity versus their ability to make slices more precise. This study can assist in designing a set of stratified subsystems that let the user decide on an appropriate tradeoff between precision and speed. Finally, it would be would be interesting to attempt to extend the notion of dynamic dependence to more powerful theorem-proving techniques, such as those incorporating higher-order or equational unification or resolution.

References

[1] Agrawal, H., and Horgan, J. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (1990), pp. 246–256. *SIGPLAN Notices* 25(6).

[2] Ball, T. Personal communication.

[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), Springer-Verlag, pp. 141–158. Lecture Notes in Computer Science 259.

[5] 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.

[6] Choi, J.-D., Miller, B., and Netzer, R. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 491–530.

[7] 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. Program. Lang. Syst. 13*, 4 (October 1991), 451–490.

[8] Duesterwald, E., Gupta, R., and Soffa, M. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92* (California, 1992), pp. 131–145.

[9] Ernst, M. Practical fine-grained static slicing of optimized code. Tech. Rep. MSR-TR-94-14, Microsoft Research, Redmond, WA, 1994.

[10] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[11] Field, J. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages* (San Francisco, January 1990), pp. 1–15.

[12] 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.

[13] Field, J., and Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. Tech. Rep. RC 19???, IBM T.J. Watson Research Center, November 1994. (Corrected and expanded version of [14]).

[14] Field, J., and Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming* (September 1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.

[15] Gallagher, K., and Lyle, J. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (1991), 751–761.

[16] Horwitz, S., Prins, J., and Reps, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (1989), 345–387.

[17] Kahn, G. Natural semantics. In *Fourth Annual Symp. on Theoretical Aspects of Computer Science* (1987), vol. 247 of *LNCS*, Springer-Verlag, pp. 22–39.

[18] KAMKAR, M., FRITZSON, P., AND SHAHMEHRI, N. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming 38* (1993), 625–636.

[19] KLINT, P. A meta-environment for generating programming environments. *ACM Trans. on Software Engineering and Methodology 2*, 2 (1993), 176–201.

[20] KLOP, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[21] KOREL, B., AND LASKI, J. Dynamic slicing of computer programs. *Journal of Systems and Software 13* (1990), 187–195.

[22] LANDI, W., AND RYDER, B. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation* (San Francisco, 1992), pp. 235–248. *SIGPLAN Notices* 27(7).

[23] LARUS, J., AND CHANDRA, S. Using tracing and dynamic slicing to tune compilers. Computer Science Technical Report 1174, University of Wisconsin-Madison, 1993.

[24] NING, J., ENGBERTS, A., AND KOZACZYNSKI, W. Automated support for legacy code understanding. *Communications of the ACM 37*, 5 (1994), 50–57.

[25] PAN, H. *Software Debugging with Dynamic Intrumentation and Test-Based Knowledge*. PhD thesis, Purdue University, 1993.

[26] PODGURSKI, A., AND CLARKE, L. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering 16*, 9 (1990), 965–979.

[27] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. Report CS-R9453, Centrum voor Wiskunde en Informatica (CWI), 1994.

[28] TIP, F. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[29] TIP, F. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995. Forthcoming.

[30] VENKATESH, G. The semantic approach to program slicing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Toronto, June 1991), pp. 80–91.

[31] WEGMAN, M. N., AND ZADECK, F. K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2 (April 1991), 181–210.

[32] WEISER, M. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters 17*, 3 (1983), 129–135.

[33] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

A PIM Details

In this section, we briefly review the PIM term structure and the most important subsets of PIM rules. Additional rules used primarily for performing induction are described in [12]; these rules are the foundation for the loop analysis rules presented in Section 5. In general, PIM is augmented with rules defining the semantics of language-specific datatypes such as integers.

PIM terms are constructed over an *order-sorted* signature. PIM sorts distinguish among fundamentally incompatible syntactic structures corresponding to observable values, merge structures, store structures, and lambda expressions; however, sorts should not be interpreted as types in the usual sense.

A.1 PIM$^{\rightarrow}$ Rules

The rules of PIM$_t^{\rightarrow}$ are given in Fig. 18. Variables v, m, s, and f will be used in the rules to refer to observable values, merge structures, store structures, and lambda-expressions, respectively. Equations (L1)–(L8) are generic to merge or store structures. Thus each of the operators labeled ρ is to be interpreted as one of s or m. (E1) and (E2) are schemes for an infinite set of equations. Equation (C1) only applies if the argument of '$S(\cdot)$' is of sort $\mathcal{S}$. The rules of PIM$^{\rightarrow}$ consist of those of PIM$_t^{\rightarrow}$, along with the rules depicted in Fig. 11.

A.2 PIM$_t^{=}$ Equations

The rules of PIM$_t^{=}$ are those of PIM$_t^{\rightarrow}$ along with those given in Fig. 19. As before, ρ in rules (L9)–(L11) is assumed to be one of m or s. In rule (M9), $C_{\mathcal{V}}[\ ]$ denotes an arbitrary *context* of value sort; this rule could also be less perspicuously rendered as a family of rules, one for each value-sorted function symbol.

$$
\begin{array}{rcll}
\emptyset_\rho \circ_\rho l & \longrightarrow & l & \text{(L1)} \\
l \circ_\rho \emptyset_\rho & \longrightarrow & l & \text{(L2)} \\
l_1 \circ_\rho (l_2 \circ_\rho l_3) & \longrightarrow & (l_1 \circ_\rho l_2) \circ_\rho l_3 & \text{(L3)} \\
p \rhd_\rho \emptyset_\rho & \longrightarrow & \emptyset_\rho & \text{(L4)} \\
\mathbf{T} \rhd_\rho l & \longrightarrow & l & \text{(L5)} \\
\mathbf{F} \rhd_\rho l & \longrightarrow & \emptyset_\rho & \text{(L6)} \\
p \rhd_\rho (l_1 \circ_\rho l_2) & \longrightarrow & (p \rhd_\rho l_1) \circ_\rho (p \rhd_\rho l_2) & \text{(L7)} \\
p_1 \rhd_\rho (p_2 \rhd_\rho l) & \longrightarrow & \wedge\langle p_1 , p_2\rangle \rhd_\rho l & \text{(L8)} \\
\{v_1 \mapsto m\} \ @\ v_2 & \longrightarrow & =\langle v_1 , v_2\rangle \rhd_m m & \text{(S1)} \\
\{v \mapsto \emptyset_m\} & \longrightarrow & \emptyset_s & \text{(S2)} \\
\emptyset_s \ @\ v & \longrightarrow & \emptyset_m & \text{(S3)} \\
(s_1 \circ_s s_2) \ @\ v & \longrightarrow & (s_1 \ @\ v) \circ_m (s_2 \ @\ v) & \text{(S4)} \\
p \rhd_s \{v \mapsto m\} & \longrightarrow & \{v \mapsto (p \rhd_m m)\} & \text{(S5)} \\
=\langle k_1 , k_2\rangle & \longrightarrow & \mathbf{T}, \quad k_i \text{ constants}, \quad k_1 \equiv k_2 & \text{(E1)} \\
=\langle k_1 , k_2\rangle & \longrightarrow & \mathbf{F}, \quad k_i \text{ constants}, \quad k_1 \not\equiv k_2 & \text{(E2)} \\
[\mathbf{F} \rhd v] & \longrightarrow & \emptyset_m & \text{(M1)} \\
(m \circ_m [\mathbf{T} \rhd v]) & \longrightarrow & [\mathbf{T} \rhd v] & \text{(M2)} \\
[\mathbf{T} \rhd v]! & \longrightarrow & v & \text{(M3)} \\
\emptyset_m! & \longrightarrow & ? & \text{(M4)} \\
p_1 \rhd_m [p_2 \rhd v] & \longrightarrow & [\wedge\langle p_1 , p_2\rangle \rhd v] & \text{(M5)} \\
\neg\langle \mathbf{T} \rangle & \longrightarrow & \mathbf{F} & \text{(B1)} \\
\neg\langle \mathbf{F} \rangle & \longrightarrow & \mathbf{T} & \text{(B2)} \\
\neg\langle \neg\langle p \rangle \rangle & \longrightarrow & p & \text{(B3)} \\
\wedge\langle \mathbf{T} , p\rangle & \longrightarrow & p & \text{(B4)} \\
\wedge\langle p , \mathbf{T}\rangle & \longrightarrow & p & \text{(B5)} \\
\wedge\langle \mathbf{F} , p\rangle & \longrightarrow & \mathbf{F} & \text{(B6)} \\
\wedge\langle p , \mathbf{F}\rangle & \longrightarrow & \mathbf{F} & \text{(B7)} \\
\wedge\langle \wedge\langle p_1 , p_2\rangle , p_3\rangle & \longrightarrow & \wedge\langle p_1 , \wedge\langle p_2 , p_3\rangle\rangle & \text{(B8)} \\
\neg\langle \wedge\langle p_1 , p_2\rangle \rangle & \longrightarrow & \vee\langle \neg\langle p_1 \rangle , \neg\langle p_2 \rangle\rangle & \text{(B9)} \\
\vee\langle \mathbf{T} , p\rangle & \longrightarrow & \mathbf{T} & \text{(B10)} \\
\vee\langle p , \mathbf{T}\rangle & \longrightarrow & \mathbf{T} & \text{(B11)} \\
\vee\langle \mathbf{F} , p\rangle & \longrightarrow & p & \text{(B12)} \\
\vee\langle p , \mathbf{F}\rangle & \longrightarrow & p & \text{(B13)} \\
\vee\langle \vee\langle p_1 , p_2\rangle , p_3\rangle & \longrightarrow & \vee\langle p_1 , \vee\langle p_2 , p_3\rangle\rangle & \text{(B14)} \\
\neg\langle \vee\langle p_1 , p_2\rangle \rangle & \longrightarrow & \wedge\langle \neg\langle p_1 \rangle , \neg\langle p_2 \rangle\rangle & \text{(B15)} \\
S(s) & \longrightarrow & s & \text{(C1)}
\end{array}
$$

Figure 18: Equations of PIM$_t^{\rightarrow}$

$$
\begin{array}{rcll}
=\langle v , v\rangle & = & \mathbf{T} & \text{(E3)} \\
l_2 \circ_\rho l_1 \circ_\rho l_2 & = & l_1 \circ_\rho l_2 & \text{(L9)} \\
\multicolumn{3}{c}{\dfrac{\wedge\langle p_1 , p_2\rangle = \mathbf{F}}{(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)}} & \text{(L10)} \\
(p_1 \rhd_\rho l) \circ_\rho (p_2 \rhd_\rho l) & = & (\vee\langle p_1 , p_2\rangle) \rhd_\rho l & \text{(L11)} \\
\{v \mapsto m_1\} \circ_s \{v \mapsto m_2\} & = & \{v \mapsto (m_1 \circ_m m_2)\} & \text{(S6)} \\
\multicolumn{3}{c}{\dfrac{=\langle v_1 , v_2\rangle = \mathbf{F}}{\{v_1 \mapsto m_1\} \circ_s \{v_2 \mapsto m_2\} = \{v_2 \mapsto m_2\} \circ_s \{v_1 \mapsto m_1\}}} & \text{(S7)} \\
\multicolumn{3}{l}{(\neg\langle p \rangle \rhd_m m_1 \circ_m} & \\
m_2 \circ_m [p \rhd v] & = & m_1 \circ_m m_2 \circ_m [p \rhd v] & \text{(M6)} \\
[p \rhd m!] & = & [p \rhd ?] \circ_m (p \rhd_m m) & \text{(M7)} \\
([\mathbf{T} \rhd ?] \circ_m m)! & = & m! & \text{(M8)} \\
C_{\mathcal{V}}[\ m!\ ] & = & (m \setminus (\lambda x_{\mathcal{V}} . C_{\mathcal{V}}[\ x_{\mathcal{V}}\ ]))\ !, & \\
 & & \qquad x_{\mathcal{V}} \notin FV(C_{\mathcal{V}}[\ ]) & \text{(M9)} \\
(m_1 \circ_m m_2) \setminus f & = & (m_1 \setminus f) \circ_m (m_2 \setminus f) & \text{(M10)} \\
[p \rhd v] \setminus f & = & [p \rhd (f \ \ v)] & \text{(M11)} \\
\emptyset_m \setminus f & = & \emptyset_m & \text{(M12)} \\
(p \rhd_m m) \setminus f & = & p \rhd_m (m \setminus f) & \text{(M13)} \\
(m \setminus \lambda x.v) \setminus f & = & m \setminus \lambda x.fv & \text{(M14)} \\
\wedge\langle p_1 , p_2\rangle & = & \wedge\langle p_2 , p_1\rangle & \text{(B16)} \\
\wedge\langle p , p\rangle & = & p & \text{(B17)} \\
\wedge\langle p , \neg\langle p \rangle\rangle & = & \mathbf{F} & \text{(B18)} \\
\vee\langle p_1 , p_2\rangle & = & \vee\langle p_2 , p_1\rangle & \text{(B19)} \\
\vee\langle p , p\rangle & = & p & \text{(B20)} \\
\vee\langle p , \neg\langle p \rangle\rangle & = & \mathbf{T} & \text{(B21)} \\
\wedge\langle p_1 , \vee\langle p_2 , p_3\rangle\rangle & = & \vee\langle \wedge\langle p_1 , p_2\rangle , \wedge\langle p_1 , p_3\rangle\rangle & \text{(B22)} \\
\vee\langle p_1 , \wedge\langle p_2 , p_3\rangle\rangle & = & \wedge\langle \vee\langle p_1 , p_2\rangle , \vee\langle p_1 , p_3\rangle\rangle & \text{(B23)} \\
[p \rhd (\wedge\langle p , q\rangle)] & = & [p \rhd q] & \text{(B24)}
\end{array}
$$

Figure 19: Additional Equations of PIM$_t^{=}$