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 PtM [12], we automatically extract a semantically well-founded language-specific algorithm for computing constrained slices. An advantage of this approach is that only the PtM translation is language dependent; the mechanics of slicing itself are independent of the language.

- PtM’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 PtM’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 unconstrained 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.
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 and an equational logic that operates on PIM graphs.

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.
2.3 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.

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 \( \mathsf{N} := 5 \) and \( \mathsf{P} := 1 \). 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:

\[
\begin{align*}
\forall \langle \mathbf{T}, \mathbf{p} \rangle & \rightarrow \mathbf{T} \quad \text{(B10)} \\
\forall \langle \langle \mathbf{p}_1, \mathbf{p}_2 \rangle, \mathbf{p}_3 \rangle & \rightarrow \langle \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3 \rangle \quad \text{(B14)}
\end{align*}
\]

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, \( \mathbf{p}, \mathbf{p}_1, \mathbf{p}_2, \) and \( \mathbf{p}_3 \)) match any subterm; all other symbols must match exactly. By applying the rules above, the term

\[
\wedge \langle \forall \langle \mathbf{T}, \mathbf{F} \rangle, \wedge \langle \mathbf{F}, \mathbf{T} \rangle, \mathbf{F} \rangle
\]

may be rewritten as follows (subterms affected by rule applications...
Figure 4: An example program. (a) Constrained slice at \( x \) given \( ?A := 2 \). (b) Conditional constrained slice at \( x \) given \( ?A > 5 \).

Figure 5: Example of creation and residuation relations.

\[
\begin{align*}
T_0 &= \land \{ \lor \{ T, F \}, \land \{ F, T \} \} \\
T_1 &= \land \{ \lor \{ T, \lor \{ F, \land \{ F, T \} \} \} \} \\
T_2 &= \land \{ T, F \}
\end{align*}
\]

Observe in the example above that the outer context \( \land \{ \cdot, F \} \) (\( \cdot \) 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 \( p_1, p_3, \) and \( p_0 \) in both the left-hand side and the right-hand side of (B14) causes the subterms \( T, F, \) and \( \land \{ F, T \} \) 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 \( (T_1) \lor \{ F, \land \{ F, T \} \} \) 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 ‘\( \lor \)’ and ‘\( T \)’. This observation is the keystone of our reduction-based slicing technique: We “track” those subterms that are relevant to each reduction step; 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 \( T_i \) made possible by the multi-step reduction rule. The dynamic dependence relation for a multi-step reduction rule then consists, roughly speaking, 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 subduction of \( r \). These properties

\footnote{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 \[13\].}
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 $\vee(T, m, n)$.

### 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 $\vee(\Lambda(T, F), \Lambda(T, T))$.)

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:

$$\land(p_1, \lor(p_2, p_3)) = \lor(\land(p_1, p_2), \land(p_1, p_3)) \quad (B22)$$

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 $T_0 = \land(\lor(T, F), \lor(F, T))$.

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.

### 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 Slice($P_1$, $x$, {0}), 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 $x$. Slice($P_1$, $x$, {0}) is simply the PIM expression denoting the final value of the variable $x$. 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. The translation uses several sequent forms corresponding to the principal C syntactic components. The general form for these sequents is:

$$s \vdash e \Rightarrow t$$

Such a sequent may be read as "µC construct $e$ 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.
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, essentially an abstract term representation of memory. \( S_{P_1} \) is constructed from the sequential composition (using the `of` operator) of subtrees 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

\[
S_1 \equiv \{ \text{addr}(p) \rightarrow [T \triangleright P_0] \}
\]

A store cell associates an address expression (here \text{addr}(p)) with a store cell, (here \([T \triangleright P_0]\)). Constant addresses such as \text{addr}(p) represent ordinary variables. More generally, address

\[\text{addr}(x)\]

initial transitive dependence relations between C syntax tree and PIM graph (only a subset of edges are depicted)

\[\text{denotes root node of enclosed graph}\]

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.

expressions are used when addresses are computed, e.g., \( \text{in pointer references.} \) \( \emptyset \) 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 \text{merge cell} such as \( [T \triangleright P_0] \), in which some boolean predicate (here, \( T \)) guards a value (here, the free PIM variable \( P_0 \) representing the \( \mu C \) meta-variable \( ?p \)). 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 \text{ordered lists} of the form \( m_1 ; m_2 \), in which the \text{rightmost} guarded cell takes precedence. Such lists correspond roughly to Lisp \text{cond} expressions, and represent information similar to SSA-form \( \phi \) nodes [7], particularly the \text{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 \text{selection} operation, \( \text{!'} \). 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
and merge constructs when no confusion will arise.

In addition to guards in merge cells, stores such as \( S_t \) (which corresponds to the "i \( \epsilon \)" statement as a whole) may also be guarded. The guard expression \( V_t \) corresponds to the \( i \epsilon \)’s predicate expression. Consistent with standard C semantics, the guard \( V_t \) 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

\[
\forall x_1 := \text{Exp}_1, \ldots, \forall x_n := \text{Exp}_n
\]

is

\[
\text{Slice} (P, x, \{ \forall x_1 := \text{Exp}_1, \ldots, \forall x_n := \text{Exp}_n \}) \equiv \left( (S_P \@ \text{addr}(x)) ! \right) [X_{1 \text{Y}} := v_1, \ldots, X_{n \text{Y}} := v_n]
\]

where \( S_P \) is the PIM store to which \( P \) compiles, the \( X_{1 \text{Y}} \) are free variables corresponding to the meta-variables and the \( v_i \) are PIM graphs corresponding to the value of the \( \text{Exp}_i \) (ignoring side effects). \( \text{Slice} (P, x, \{ \cdot \cdot \cdot \}) \) 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\(^-\), plus a set of additional non-oriented equational rules for reasoning about operational equivalences in PIM\(^-\), instances of which can also be oriented for use in analysis. PIM\(^-\) 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\(^-\) that defines the semantics of programs without loops or procedures, PIM\(^-\), is canonical, that is, strongly normalizing as well as confluent. PIM\(^-\) can be enriched with certain oriented instances of rules in (PIM\(^-\) PIM\(^-\)) 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 \( \text{Slice} (P, x, \{ \cdot \cdot \cdot \}) \) 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 \( \text{Slice} (P, x, \{ \cdot \cdot \cdot \}) \) 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 \( \text{Slice} (P_0, x, \{ \check{\gamma} P := 0 \}) \) and \( \text{Slice} (P, x, \{ \cdot \cdot \cdot \}) \), the slices of \( P_0 \) 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 \( \text{Slice} (P_0, x, \{ \cdot \cdot \cdot \}) \). Graph (2) results from multiple applications of the rule

\[
(s_1 \circ_s s_2) @ v \rightarrow (s_1 \circ_v) o_m (s_2 \circ v) \quad \text{(S4)}
\]

which have the effect of distributing the reference to the variable \( x \), \( \text{addr}(x) \), to the sequence of substores \( S_1, S_2, S_3, S_4 \). Graph (3) results from applications of the rule

\[
\{v_1 \mapsto m\} @ v_2 \rightarrow \{v_1, v_2\} \triangleright_m m \quad \text{(S1)}
\]

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 \( S_P \). Graph (4) results from evaluation of address comparisons. The comparison fails for assignments represented by \( S_i x \) and \( S_j x \) (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 \( M_j \), yielding a merge cell containing the free meta-variable \( ?P \), then combining the PIM graph representing the predicate guarding the \( i \epsilon \) statement, \( V_i \), with a predicate derived from the address comparison for the nested store for the assignment in store \( S_k \) (representing the result of the assignment inside the \( i \epsilon \)).

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: \( \text{Slice}(P_1, x, \{ ?P := 0 \}) \)

In the constrained case, \( P_0 \) is bound to 0, i.e., is false. In step (8a), the highlighted application of the rule

\[
F \triangleright_p I \rightarrow 0_p \quad \text{(L6)}
\]

has the effect of eliminating the body of the \( i \epsilon \) 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 \( i \epsilon \) statement and the identifier \( p \) in the assignment \( y = p \) have their origin in the subgraph \( S_k \). 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 \( P_0 \). 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: \( \text{Slice}(P, x, \{ \cdot \cdot \cdot \} \}

The unconstrained case is somewhat more interesting than the constrained case: Although we do not know the value of \( P_0 \) and thus cannot effectively evaluate the \( i \epsilon \) statement in \( P_1 \), we discover that the two reaching definitions for \( x \) both assign the same value to \( x \), namely, \( P_1 \). 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 resolving term (12b) (or the final term (13b)). Consequently, the unconstrained slice does not contain the predicate of the \( i \epsilon \) statement, though it does contain assignment statement within the \( i \epsilon \) 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
Figure 10: Reduction of \textit{Slice}(P_1, x, \{\}) and \textit{Slice}(P_1, x, \{ ?P := 0 \}). 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

\[ \text{if } (P) \ f(\text{foo}); \text{ else } f(\text{bar}); \]

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 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 \( \mu \)C 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

\[ v = op(v, e); \]

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 to denote those PIM rules that are not loop related and are common to the slicing variants we will present. While the COREPIM 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

\[ \text{Loop}(\lambda \varphi. \text{body}(u_E, v_E, u_S), s) \]

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 \( z_E \) at the beginning of a loop iteration. The second argument \( s \) is the incoming store. The term
Loop(\lambda x_S.body(u_E, v_E, u_S), s) \rightarrow S(\lambda x_f \rightarrow \lambda x_S.
\begin{array}{ll}
    \{ & u_E \circ (v_E \circ_s (u_S \circ_s s)) \\
    | & S(\lambda x_f \rightarrow \lambda x_S. \\
    | & (f_E \circ (x_E \circ_s (v_E \circ_s u_E \circ_s u_S)))) \\
\end{array}
) [\text{loop}]
\]

(where \( f_E \not\in \{FV(u_E) \cup FV(u_S) \cup FV(v_E)\} \))

[\lambda x_f \rightarrow g \rightarrow f[x := g] \quad [\beta]

[\lambda f \rightarrow f \rightarrow f \quad [\text{recursion}]

Figure 11: Loop execution rules

Loop(\lambda x_S.body(u_E, v_E, u_S), s) \rightarrow \text{Project}(\lambda x_S.body(u_E, v_E, u_S), s),
\text{AssignedVars}(u_E \circ_s u_S)) \quad [\text{LA1}]

Project(s, \{\}) \rightarrow 0_s \quad [\text{LA2}]

Project(s, \{v\} \cup r) \rightarrow \{v \mapsto s \circ v \circ \text{Project}(s, r) \quad [\text{LA3}]

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

\]

Figure 12: \( \phi \) rules

Loop(\lambda x_S.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, \( x_S \circ_s u_E \circ_s u_S \). The rest of the term serves to express the recursion using the recursion combinator \( Y \). \( f[x := g] \) represents the result of substituting \( g \) for free occurrences of \( x \) in \( f \) (with the usual provisos about variable capture and renaming). \( FV(r) \) is the set of free variables in a term \( r \), and \( \text{S}' \) is a technicality—a "sort coercion" operator that has no semantic content. [11] shows how the \( \beta \) rule and substitution can be encoded as pure rewriting rules.

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

5.2 \( \phi \) Rules: Lazy Dynamic Slicing

Fig. 12 depicts a set of rules, the \( \phi \) rules, that statically simplify \text{PIM} stores generated by loops. The effect of \( \phi \) rules on the \text{PIM} representation is essentially to introduce an SSA-form \( \phi \) 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 \( \phi \) rules in conjunction with \text{COREPIM} as a \text{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 than to the slices produced by the Karel-Laski algorithm [21]. The Karel-Laski slices tend to be larger than the Agrawal-Horgan slides 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.

Fig. 13(a) illustrates an important difference between our algorithm and previous dynamic slicing algorithms. Since the \( \text{if f} \) 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 \text{COREPIM} to compute a classical static slice. The goal of these rules is quite simple. Consider the \text{PIM} term

\[
\text{Slice(while}(i < 10)\{j = j + 2; i = i + 1; \}, i, \{\})
\]

This reduces to a term of the form

\[
\text{Loop}(\lambda x_S.body(u_E, v_E, u_S), s) @ \text{addr}(i)
\]

where \( u_S \), representing the loop body, is the store

\[
\{\text{addr}(j) \mapsto T \cup +\{\langle x_S @ \text{addr}(j) \rangle, 2\} \}
\]

Intuition suggests that this term should be reducible to

\[
\text{Loop}(\lambda x_S.body(u_E, v_E, u_S'), s) @ \text{addr}(i)
\]

where \( u_S' \) is the store

\[
\{\text{addr}(i) \mapsto T \cup +\{\langle x_S @ \text{addr}(i) \rangle, 1\} \}
\]

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

\[
\text{Loop}(\lambda x_S.body(s_1, p, s_2), s_3) @ a
\]

to a term

\[
\text{Loop}(\lambda x_S.body(s_1', p, s_2'), s_3') @ a
\]
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 CorePm" 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

\[ \text{Loop}(\lambda x_{S}. \text{body}(u_{E}, v_{E}, u_{S}), s) \otimes a \]

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 \( x_{S} \otimes \text{addr}(v) \), denoting the value of variable \( x \) in a particular iteration. Since the value of \( x \) is a loop-invariant constant \( e \) (given the constraint \( ?N := 5 \)), we would like to replace this term by \( e \). 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 Pm 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 application of this 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 \( s_{2} \), it 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 \( u_{S} \circ_{s} u_{S} \), 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 \( s_{inv} \).
The most reduction strategies are used. This leads us to believe that efficient when an automaton-based matching algorithm and out-

Term graph reduction is a simple technique that can be implemented implementing our approach.

from this prototype have been encouraging, and we are now engaged using the ASF+SDF Meta-environment [19]. The results obtained for identifying loop invariants, such as the conditional constant (IA1) cannot be used directly by the dependence tracking system.

6.1 Properties of Graph Reduction

Propagating algorithm of Wegman et al [31].

6 Pragmatics

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 Ptm. Any algorithm to perform the translation suffices, provided that the dynamic dependence relations between the source AST and its Ptm 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

Ptm 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 Ptm’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 Ptm, see [12].

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.

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].
between most steps involved in PDG construction and certain PIM-based algorithms, it is apparent that there is a close correspondence. For instance, in comparing our work with PDG-semantics for a language, and that these techniques are useful for analysis that aspects of both intermediate representation construction and competitive with previous techniques. One characteristic of our approach that for comparable types of slices, our approach should be quite powerful. 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 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


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 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.

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 Ernest’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. These 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.
A Ptm Details

In this section, we briefly review the Ptm term structure and the most important subsets of Ptm 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, Ptm is augmented with rules defining the semantics of language-specific datatypes such as integers.

Ptm terms are constructed over an ordered signature. Ptm 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 Ptm Rules

The rules of Ptm 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 $p$ 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 $\langle S \cdot \rangle$ is of sort $S$. The rules of Ptm consist of those of Ptm along with the rules depicted in Fig. 11.

A.2 Ptm Equations

The rules of Ptm are those of Ptm along with those given in Fig. 19. As before, $p$ in rules (L9)–(L11) is assumed to be one of $s$ or $m$. In rule (M9), $C\{\cdot\}$ denotes an arbitrary context of value sort; this rule could also be less periphrastically rendered as a family of rules, one for each value-sorted function symbol.