Semantic Foundations of Binding-Time Analysis for Imperative Programs

Manuvir Das,[1] *Thomas Reps,*[1] *and Pascal Van Hentenryck* [2]
1: University of Wisconsin-Madison; 2: Brown University

This paper examines the role of dependence analysis in defining binding-time analyses (BTAs) for imperative programs and in establishing that such BTAs are safe. In particular, we are concerned with characterizing safety conditions under which a program specializer that uses the results of a BTA is guaranteed to terminate. Our safety conditions are formalized via semantic characterizations of the statements in a program along two dimensions: *static* versus *dynamic*, and *finite* versus *infinite*. This permits us to give a semantic definition of "static-infinite computation", a concept that has not been previously formalized. To illustrate the concepts, we present three different BTAs for an imperative language; we show that two of them are safe in the absence of "static-infinite computations".

In developing these notions, we make use of *program representation graphs*, which are a program representation similar to the dependence graphs used in parallelizing and vectorizing compilers. In operational terms, our BTAs are related to the operation of *program slicing*, which can be implemented using such graphs.

1. Introduction

This paper explores the role of dependence analysis in defining binding-time analyses (BTAs) for the two-phase, off-line specialization of imperative programs [6] and in establishing that such BTAs are safe. The motivation for this work stems from a well-known danger that arises in such program specializers, namely that the binding-time information obtained in the first phase may cause the second phase of specialization to fall into an infinite loop. This problem is illustrated by the following example, adapted from [6, pp. 265-266] (see also [13, pp. 501-502], [9, pp. 337], and [7, pp. 299]):

```
P1:   read (x1);
      x2 := 0;
   w: while ( x1 ≠ 0 ) do
        u: x1 := x1 − 1;
        v: x2 := x2 + 1
      od
```

At program point *v*, variable x_1 should clearly be classified as "dynamic"; the issue is whether x_2 should be classified as "static" or "dynamic". Both choices lead to "(uniform) congruent divisions" in the terminology of [6]. The BTAs given by Jones, Sestoft, and Mogensen would label x_2 "static". This choice is unfortunate because it causes the specialization phase to enter an infinite loop, creating specialized program points of the form $\langle w,x_2\rangle$ and $\langle u,x_2\rangle$ for the infinitely many values x_2 that x_2 may take on.

Although this problem has been addressed via the "termination analyses" of Holst [4] and Jones *et al.* [7, Chapter 14], the methods developed are targeted for data domains that are bounded (*i.e.*, data domains for which there is an ordering on values such that, for each value *v*, there is a finite number of values less than *v*). Natural numbers and list structures are examples of bounded data domains, but integers are an unbounded data domain. This is one indication that some central aspect of the problem has been overlooked.

Jones calls the process of classifying a variable occurrence (such as x_2 at *v*) as dynamic when congruence would allow it to be classified as static a form of *generalization* [7]. Our work takes a different approach: rather than focusing on intensional concepts, such as congruence, we introduce semantic (*i.e.*, extensional) definitions for concepts such as "staticness", "dynamicness", "finiteness", and "infiniteness". This allows us to give a firm semantic foundation to some heretofore only informally defined concepts, such as "static-infinite computation" and "bounded static variation". (In contrast with previous work, by our definitions x_2 at *v* would never be classified as "static".) We then give intensional definitions (in the form of binding-time analyses) that safely approximate the extensional definitions.

The contributions of the paper can be summarized as follows:

- We give a ***semantic characterization*** of when a BTA is ***safe***.
  - Safety is formalized via semantic characterizations of the statements in a program *P* along two dimensions: ***static*** versus ***dynamic***, and ***finite*** versus ***infinite***. (The sets of *P*'s program points that meet these conditions are denoted by Static(*P*), Dynamic(*P*), Finite(*P*), and Infinite(*P*), respectively.)
  - Three different kinds of static vertices are defined: ***strongly*** static, ***weakly*** static, and ***boundedly varying***. All strongly static vertices are weakly static, and all weakly static vertices are boundedly varying.
  - A BTA is safe when S(*P*), the set of *P*'s program points that are identified by the BTA as being ***specializable***, is a subset of Static(*P*) ∩ Finite(*P*).

This work was supported in part by the National Science Foundation under grant CCR-9100424 and under a National Young Investigator Award, by a David and Lucile Packard Fellowship for Science and Engineering, and by the Defense Advanced Research Projects Agency under ARPA Orders No. 8856 and No. 8225 (monitored by the Office of Naval Research under contracts N00014-92-J-1937 and N00014-91-J-4052, respectively).

Authors' addresses: Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton St., Madison, WI 53706; Computer Sciences Department, Brown University, 115 Waterman St., Providence, RI 02906.
Electronic mail: {manuvir, reps}@cs.wisc.edu, pvh@cs.brown.edu.

- We give a ***semantic characterization*** of when a BTA is ***conditionally safe***. This formalizes the previously informal notion of "a BTA for which the specialization phase terminates, assuming that the program contains no static-infinite computations".
  - With a conditionally safe BTA, S(P) $\subseteq$ Static(P). Thus, on every program Q for which Static(P) $\cap$ Infinite(P) $= \emptyset$, a conditionally safe BTA will be safe.
  - We show that program slicing [15,10] can be used to define a conditionally safe BTA (the Strong-Staticness BTA) that identifies strongly static behaviour. Since this leads to an unsatisfactory result for many programs, we develop two other BTAs based on modified slicing algorithms.

Our results are based on two insights:

- It is appropriate to use ***control dependences*** along with data dependences to trace the effect of dynamic input through a program. Furthermore, control dependences that do not affect the actual values computed at the point of dependence can be ignored when tracing dynamic behaviour (see the Weak-Staticness and Bounded-Variation BTAs).
- The notion of a "static computation" and other related concepts can be formalized using a ***value-sequence***-oriented semantics for a program [11], rather than a state-oriented semantics. The value-sequence semantics is defined in terms of the program's program representation graph (PRG) [16], which is a form of the "program dependence graph" used in vectorizing and parallelizing compilers [3] extended with some of the features of static single-assignment form [1]. Rather than treating each program point as a state-to-state transformer, the value-sequence semantics treats each program point as a ***value-sequence transformer*** that takes (possibly infinite) argument sequences from dependence predecessors to a (possibly infinite) output sequence, which represents the ***sequence of values*** computed at that point during program execution.

The rest of this paper is organized as follows: In Section 2, we present an overview of the structure and semantics of program representation graphs. In Section 3, we define the properties of staticness and finiteness based on the PRG semantics. In Section 4, we use these properties to characterize BTAs as safe, conditionally safe, and unsafe. In Section 5, we present three BTAs based on program slicing. Section 6 discusses related work.

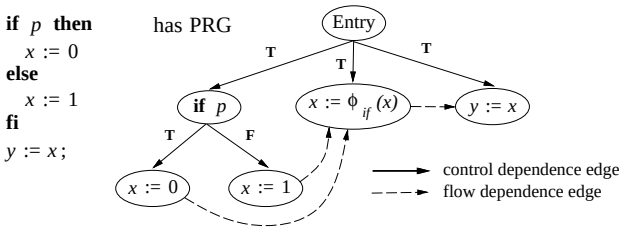
2. The PRG: A Representation that Formalizes Dependences

In this section we present the program representation graph (PRG), an intermediate form in which control dependences are represented explicitly. The structure of PRGs is discussed in Section 2.1; a semantics for PRGs is presented in Section 2.2.

2.1. The Structure of PRGs

The PRG is a dependence graph that represents a standard imperative language without procedures, in which programs consist of the following statements: assignments, conditionals (**if**), loops (**while**), input (**read**), and output (**write**). The language provides only scalar variables, which may be of type integer, real, or boolean.

The PRG of program P is a directed graph $G(P) = (V, E)$ where V is a set of vertices and E is a set of edges. $V(G)$ includes a unique **Entry** vertex, zero or more **Initialize** vertices, and vertices that represent the statements and predicates of the program. $E(G)$ consists of data and control dependence edges defined in the usual manner [3],[1] except that in cases where multiple definitions of a variable reach the same use, $V(G)$ is augmented with ϕ vertices that "mediate" between the different definition points. For example,



The $x := \phi_{if}(x)$ vertex is placed between the definitions of x at $x := 0$ and $x := 1$ and the use of x at $y := x$. The sense in which it "mediates" between $x := 0$ and $x := 1$ is explained in Section 2.2.

Other ϕ vertices are added to the PRG as follows:

- ϕ_{if} vertices : for variables defined within an **if** statement that are used before being defined after the **if**;
- ϕ_{enter} vertices : for variables defined within a loop and used before being defined within the loop;
- ϕ_{exit} vertices : for variables defined within a loop and used before being defined after the loop;
- ϕ_{T} vertices : for variables used before being defined within the true branch of an **if** statement;
- ϕ_{F} vertices : for variables used before being defined within the false branch of an **if** statement;
- ϕ_{copy} vertices : for variables used within a loop and not defined within it.
- ϕ_{while} vertices : for variables used within a loop and redefined within it.

With each kind of vertex, we assume there is an appropriate set of access functions to predecessor vertices. In the example above, the ϕ_{enter} vertex has two data

[1] A control dependence edge from vertex u to vertex v with label $L \in \{ T, F \}$ in the PRG represents the condition that whenever u evaluates to L, v is guaranteed to execute assuming (i) that all paths in the control flow graph are executable and (ii) that the program terminates normally.

predecessors, denoted by *innerDef* (v) and *outerDef* (v). □

Example 2.1. Figure 1 shows program P_1 from Section 1 and its program representation graph $G(P_1)$, which contains several ϕ vertices. Figure 1 will be explained in detail shortly. (See Example 2.2.)

2.2. Concrete Semantics of PRGs

In the formal semantics of the PRG, dependence edges transmit the results of computations through the PRG. Every vertex v produces a value sequence that is the sequence of values computed at the corresponding program point, and every outgoing edge $v \longrightarrow w$ propagates the value sequence produced at v to w. Thus, every vertex is a function from its input sequences (the output sequences of its dependence predecessors) to its output sequence. Full details of the semantics of PRGs can be found in [11]; in this section, we summarize the relevant concepts.

Formally, the PRG semantics is defined in terms of the semantic domains given below:

$$Val = Booleans + Integers + Reals + \cdots$$
$$Sequence = (\{nil, err\} + (Val \times Sequence))_{\perp}$$
$$Stream = (Val + (Val \times Stream))$$
$$VertexFunc = Stream \longrightarrow Vertex \longrightarrow Sequence$$

Val is a standard domain of values related by the discrete partial order. *Sequence* is the domain of value sequences described in [12, pp. 252-266], members of which are partially ordered as follows:

(i) $\perp \sqsubseteq s \qquad \forall\ s \in Sequence$
(ii) $s \sqsubseteq s \qquad \forall\ s \in Sequence$
(iii) $v \cdot s_1 \sqsubseteq v \cdot s_2 \Leftrightarrow s_1 \sqsubseteq s_2 \qquad \forall\ s_1, s_2 \in Sequence,\ v \in Val$

Sequences terminated by *err* indicate computational errors (such as division by zero).

Stream, the domain of program inputs, is the set of finite and infinite sequences formed from members of *Val*. *VertexFunc* is the domain of mappings to which the meaning of a PRG belongs. For a given PRG G, the meaning is the least mapping $f \in VertexFunc$ that satisfies the following recursive equation (see Figure 2):

$$f = \lambda i.\lambda v.\ \mathbf{E_G}(i, v, f)$$

where $\mathbf{E_G}$ is the conditional expression of the form given in Figure 2 that is appropriate for G. (Note that the given PRG G of interest is encoded in the predecessor-access functions used in $\mathbf{E_G}$, such as *whileNode*(v), *innerDef*(v), *etc*.) All of the sequence-transformation functions (*replace*, *select*, *whileMerge*, *etc.*) are continuous.

Definition. The meaning function **M** over the domain of PRGs is:

$$\mathbf{M} : PRG \longrightarrow VertexFunc$$
$$\mathbf{M}[G] = \mathbf{fix}\,\mathbf{F} \quad \text{where} \quad \mathbf{F} : VertexFunc \longrightarrow VertexFunc$$
$$\mathbf{F} = \lambda f.\lambda i.\lambda v.\ \mathbf{E_G}(i, v, f)$$
□

Example 2.2. Figure 1 shows program P_1 from Section 1 and the semantic equations at each vertex in its PRG. In particular:

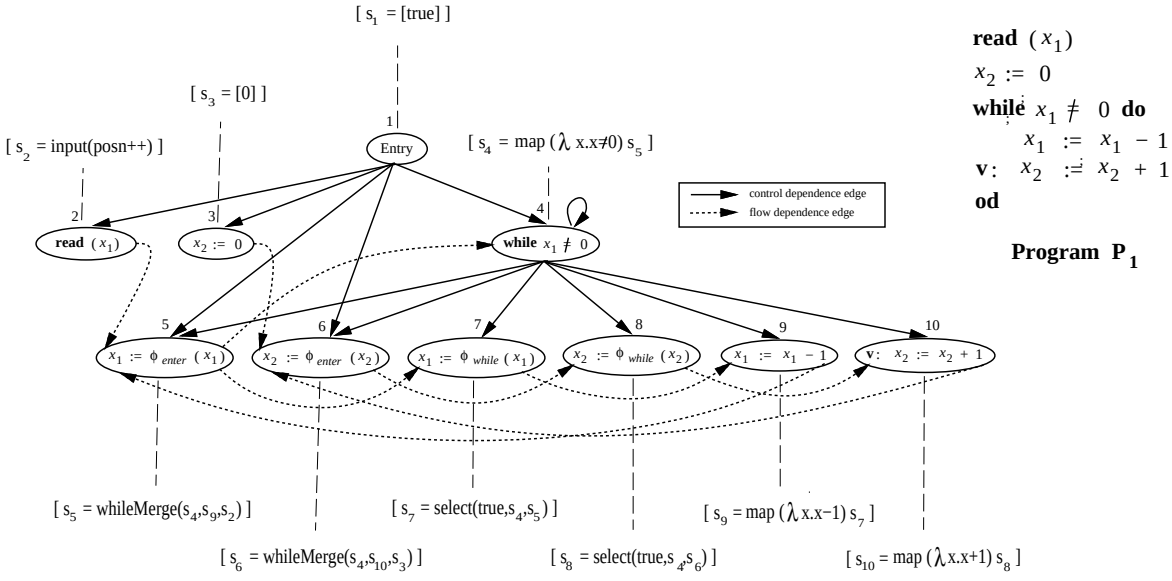


Figure 1. Example program P1 and its program representation graph $G(P_1)$, annotated with its semantic equations. The dashed lines indicate the semantic equation associated with each vertex.

$$
\begin{aligned}
\mathbf{E_G}(i,v,f) \triangleq\ & \mathbf{type}(v) = \mathbf{Entry} \longrightarrow \text{true} \cdot nil \\
& \mathbf{type}(v) = \mathbf{read} \longrightarrow \text{input}(i) \\
& \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\} \longrightarrow \begin{cases} \text{replace}(\mathit{controlLabel}(v),\ \mathit{funcOf}(v),\ f\ i\ \mathit{parent}(v)) & \text{if } \#\mathit{dataPreds}(v) = 0 \\ \mathit{map}\ \mathit{funcOf}(v)\ (f\ i\ \mathit{dataPred}_1(v),\ f\ i\ \mathit{dataPred}_2(v),\ \cdots) & \text{otherwise} \end{cases} \\
& \mathbf{type}(v) = \phi_{\mathbf{enter}} \longrightarrow \text{whileMerge}(f\ i\ \mathit{whileNode}(v),\ f\ i\ \mathit{innerDef}(v),\ f\ i\ \mathit{outerDef}(v)) \\
& \mathbf{type}(v) = \phi_{\mathbf{exit}} \longrightarrow \text{select}(\text{false},\ f\ i\ \mathit{whileNode}(v),\ f\ i\ \mathit{dataPred}(v)) \\
& \mathbf{type}(v) = \phi_{\mathbf{while}} \longrightarrow \text{select}(\text{true},\ f\ i\ \mathit{whileNode}(v),\ f\ i\ \mathit{dataPred}(v)) \\
& \mathbf{type}(v) = \phi_{\mathbf{T}} \longrightarrow \text{select}(\text{true},\ f\ i\ \mathit{parent}(v),\ f\ i\ \mathit{dataPred}(v)) \\
& \mathbf{type}(v) = \phi_{\mathbf{F}} \longrightarrow \text{select}(\text{false},\ f\ i\ \mathit{parent}(v),\ f\ i\ \mathit{dataPred}(v)) \\
& \mathbf{type}(v) = \phi_{\mathbf{if}} \longrightarrow \text{merge}(f\ i\ \mathit{ifNode}(v),\ f\ i\ \mathit{trueDef}(v),\ f\ i\ \mathit{falseDef}(v))
\end{aligned}
$$

where *replace*, *whileMerge*, *select*, and *merge* are defined as follows:

replace :
$$\text{replace}(x, y, \bot) = \bot \qquad \text{replace}(x, y, nil) = nil$$
$$\text{replace}(x, y, z \cdot tail) = \mathbf{if}\ (x = z)\ \mathbf{then}\ y \cdot \text{replace}(x, y, tail)\ \mathbf{else}\ \text{replace}(x, y, tail)$$

whileMerge :
$$\text{whileMerge}(s_1, s_2, \bot) = \bot \qquad \text{whileMerge}(s_1, s_2, nil) = nil$$
$$\text{whileMerge}(s_1, s_2, x \cdot tail) = x \cdot \text{merge}(s_1, s_2, tail)$$

merge :
$$\text{merge}(\bot, s_1, s_2) = \bot \qquad \text{merge}(nil, s_1, s_2) = nil$$
$$\text{merge}(true \cdot tail_1, \bot, s_2) = \bot \qquad \text{merge}(true \cdot tail_1, nil, s) = nil$$
$$\text{merge}(false \cdot tail_1, s_1, \bot) = \bot \qquad \text{merge}(false \cdot tail_1, s, nil) = nil$$
$$\text{merge}(true \cdot tail_1, x \cdot tail_2, s) = x \cdot \text{merge}(tail_1, tail_2, s)$$
$$\text{merge}(false \cdot tail_1, s, x \cdot tail_2) = x \cdot \text{merge}(tail_1, s, tail_2)$$

select :
$$\text{select}(x, \bot, z) = \bot \qquad \text{select}(x, nil, nil) = nil$$
$$\text{select}(x, y, \bot) = \bot$$
$$\text{select}(x, y \cdot tail_1, z \cdot tail_2) = \mathbf{if}\ (x = y)\ \mathbf{then}\ z \cdot \text{select}(x, tail_1, tail_2)\ \mathbf{else}\ \text{select}(x, tail_1, tail_2)$$

Figure 2. The semantic equations associated with PRG vertices. Some vertex types are omitted for brevity (see [11] for a complete definition of $\mathbf{E_G}$).

– At vertex 2, the function **input** uses the implicit input stream, indexed by *posn*, the position in the input stream, to obtain its values. Also implicit at a **read** is an assignment of the form $posn := posn + 1$;

– At vertex 3, the function **replace** uses the sequence from the control predecessor (vertex 1) to produce the singleton sequence [0]:

$$f\ i\ v_3 = \text{replace}(\text{true}, 0, f\ i\ v_1)$$

In general, **replace** generates a copy of a constant value for each time the vertex executes.

– At vertex 5, the function **whileMerge** produces a value sequence s_5 for variable x_1 by merging the sequences for x_1 from vertex 2 and vertex 9 (sequences s_2 and s_9, respectively). It uses the Boolean value sequence from its control-dependence predecessor (s_4) to determine how the two sequences for x_1 should be merged:

$$f\ i\ v_5 = \text{whileMerge}(f\ i\ v_4, f\ i\ v_9, f\ i\ v_2)$$

– At vertex 7, the function **select** filters out values from the value sequence at the ϕ_{enter} vertex (vertex 5) that correspond to instances when the loop predicate evaluates to *false*:

$$f\ i\ v_7 = \text{select}(\text{true}, f\ i\ v_4, f\ i\ v_5)$$

– The functions at the remaining non-ϕ vertices (vertices 4, 9, and 10) are **map** functions. Thus $\mathbf{M}[G]$ associates vertex 10 with output sequences as follows:

$$\mathbf{M}[G]\ 1 \cdot nil\ v_{10} = 1 \cdot nil$$
$$\mathbf{M}[G]\ 2 \cdot nil\ v_{10} = 1 \cdot 2 \cdot nil$$ □

It should be pointed out that the PRG semantics are non-standard in one respect: they are *more defined* than the standard semantics in the case of inputs on which the program does not terminate. On such inputs, the sequence of values computed at a program point according to the standard operational semantics has been shown to be a prefix of the value sequence associated with the program point in the PRG semantics. (Roughly, value sequences transmitted along dependence edges can bypass non-terminating loops.) For inputs on which the program terminates normally, it has been shown that the two sequences are identical [11].

As we show in Section 3, the value-sequence approach provides a clean way to formalize the notions needed to characterize safety conditions for BTAs, namely, “static”, “dynamic”, “finite”, and “infinite” behaviours.

3. Semantically Static and Semantically Finite Behaviour

As noted in the introduction, the usual notion of a "congruent division" is unsatisfactory in the case of program P_1 in Example 2.1, since a division that classifies variable x_2 at v as static is congruent. Although various methods have been proposed for a reclassification based on some form of termination or finiteness analysis, in our formulation of these issues v would not be classified as static. Furthermore, the notion of staticness is orthogonal to that of finiteness or boundedness.

We now use the concepts that were introduced in Section 2 to give semantic definitions of static, dynamic, finite, and infinite behaviours.

Definition 3.1. Vertex v in PRG G is *strongly* (*semantically*) *static* iff $\forall\, i_1, i_2 \in \mathit{Stream}$ the following property holds:

$$\text{(a) } \mathbf{M}\,[G]\,i_1\,v \;=\; \mathbf{M}\,[G]\,i_2\,v$$

Vertex v is *weakly* (*semantically*) *dynamic* iff it is not strongly static. □

Property (a) above says that a vertex is strongly static[2] provided its behaviour (the sequence it produces) is unaffected by changes in the run-time input. For instance, vertex v in program P_1 from Example 2.1 is semantically dynamic because $\mathbf{M}\,[G]\,1\cdot \mathit{nil}\;v \neq \mathbf{M}\,[G]\,2\cdot \mathit{nil}\;v$.

In Section 5.1, when proving the conditional safety of the Strong-Staticness BTA, we will use an abstraction function that identifies vertices that satisfy a generalization of property (a): for a given approximation m to a program's meaning function $\mathbf{M}\,[G]$, vertex v approximates the strong-staticness property if the sequences produced at v by m form a chain. Because $\mathbf{M}\,[G]$ does not produce any $\perp$-terminated sequences—it is a member of *VertexFunc* that corresponds to a *program*—for $\mathbf{M}\,[G]$ the generalized property coincides with property (a).

We have also identified a second semantic notion of staticness that generalizes Definition 3.1. The motivation for this alternative definition comes from considering the behaviour at program points v and w in the programs below:

```
P2:  read (x1);
     if ( x1 ≠ 0 ) then
        x2 := 0 ;
        while ( x2 < 3 ) do
           v: x2 := x2 + 1
        od
     fi
```

```
P3:  read (x1);
     while ( x1 ≠ 0 ) do
        x2 := 0 ;
        while ( x2 < 3 ) do
           w: x2 := x2 + 1
        od;
        x1 := x1 − 1;
     od
```

[2]We term such vertices *strongly* static as there are weaker notions of staticness that are also useful for binding-time analysis (see Definitions 3.2 and 3.3).

$$\mathbf{M}\,[G]\,i\,v \in \{\, \mathit{nil}\,,\, 1\cdot 2\cdot 3\cdot \mathit{nil}\,\} \quad \forall\, i \in \mathit{Stream}$$

$$\mathbf{M}\,[G]\,i\,w \in \{\, \mathit{nil}\,,\, 1\cdot 2\cdot 3\cdot \mathit{nil}\,,\, 1\cdot 2\cdot 3\cdot 1\cdot 2\cdot 3\cdot \mathit{nil}\,,\, ...\,\} \quad \forall\, i \in \mathit{Stream}$$

Under Definition 3.1, vertices v in P_2 and w in P_3 are both dynamic. The key observation behind a generalized notion of staticness is that, at both of these vertices, every output sequence is formed by zero or more repetitions of a common base sequence $(1\cdot 2\cdot 3)$. Although this notion may not seem intuitive, it says that while run-time data may control *how many times* the vertex executes, it does not control the actual *values* it computes. In program $\mathbf{P_2}$ ($\mathbf{P_3}$) above, the control dependence from the if predicate (outer loop predicate) to the inner loop predicate represents the effect of run-time data on how many times v (w) executes; under our generalized notion of staticness, both these dependences are irrelevant. In program P_1 from Example 2.1, however, vertex v is semantically dynamic even under the generalized definition because there is no common base sequence from which the sequences in $\{\, 1\cdot \mathit{nil}\,,\, 1\cdot 2\cdot \mathit{nil}\,,\, 1\cdot 2\cdot 3\cdot \mathit{nil}\;\cdots\,\}$ are formed.

Definition 3.2. Vertex v in PRG G is *weakly* (*semantically*) *static* iff at least one of the following holds:

(a) $\exists\, s \in \mathit{Val}^*$ s.t. $\forall i \in \mathit{Stream}$,
$\mathbf{M}\,[G]\,i\,v \;\in\; \{\mathit{nil}\} \cup \{\, s^n\cdot \mathit{nil} \mid n \in \mathbf{Nat}\,\} \cup \{\, s^\infty\,\}$

or

(b) $\exists\, s \in \mathit{Val}^\omega$ s.t. $\forall i \in \mathit{Stream}$, $\mathbf{M}\,[G]\,i\,v \;\in\; \{\mathit{nil}\,,\, s\,\}$

Vertex v is *strongly* (*semantically*) *dynamic* iff it is not semantically static. □

We call sets of the form $\{\mathit{nil}\} \cup \{\, s^n\cdot \mathit{nil} \mid n \in \mathbf{Nat}\,\} \cup \{\, s^\infty\,\}$ or $\{\mathit{nil}\,,\, s\,\}$ from the properties above *rational repetitions*. Property (b) above accounts for a situation where the base sequence is infinitely long. It is included so that the class of weakly static vertices includes all the strongly static vertices. Again, we use a more general property in proving the conditional safety of the Weak-Staticness BTA: vertex v approximates the weak-staticness property if the sequences produced at v belong to the downwards closure of a rational repetition (subsets of such downwards closures are termed *approximate rational repetitions*).

Note that Definitions 3.1 and 3.2 permit vertices that produce infinitely many different values to be considered "static". A third, more general, form of static behaviour that does involve boundedness conditions is "bounded static variation" [7, pp. 300]). Consider the behaviour at program point v in the program below:

```
P4:  read (x1);
     if ( x1 ≠ 0 ) then
        x2 := 0
     else
        x2 := 10
     fi;
     v: x3 := x2
```

$\mathbf{M}[G]\, i\, v \in \{0 \cdot nil, 10 \cdot nil\} \quad \forall i \in Stream$

Under both Definition 3.1 and Definition 3.2, vertex v in P_4 is dynamic. In particular, property (a) from Definition 3.2 is not satisfied at v as there is no common base sequence in $\{0 \cdot nil, 10 \cdot nil\}$. However, there is a bounded set of base *values* from which these sequences are formed, namely $\{0, 10\}$.

We capture this behaviour by generalizing weak staticness to *bounded variation*:

Definition 3.3. Vertex v in PRG G is *boundedly varying* iff at least one of the following holds:

(a) $\exists B \subset Val$, $|B|$ finite, such that $\forall i \in Stream$,
$\mathbf{M}[G]\, i\, v \in \{nil\} \cup \{v_1 \cdot .. \cdot v_k \cdot nil \mid v_1,..,v_k \in B\} \cup B^{\omega}$

or

(b) $\exists s \in Val^{\omega}$ s.t. $\forall i \in Stream$, $\mathbf{M}[G]\, i\, v \in \{nil, s\}$

Vertex v is *unboundedly varying* iff it is not boundedly varying. □

Sets of the form $\{nil\} \cup \{v_1 \cdot .. \cdot v_k \cdot nil \mid v_1,..,v_k \in B\} \cup B^{\omega}$ or $\{nil, s\}$ from the properties above are termed *bounded variations*. Property (a) above ensures that all sequences at the vertex are constructed from a finite set of base values. Property (b) is introduced in order to ensure that boundedly varying behaviour generalizes weakly static behaviour, in the sense that every weakly static vertex is boundedly varying. Once again we use a more general property later in the paper: vertex v approximates the bounded variation property if the sequences produced at v belong to the downwards closure of a bounded variation (subsets of such downwards closures are termed *approximate bounded variations*).

The finiteness of a computation at a vertex is determined by the number of distinct elements in its output sequences:

Definition 3.4. Vertex v in PRG G is *semantically finite* iff

$\exists B \subset Val$, $|B|$ finite, such that $\forall i \in Stream$,
$\mathbf{M}[G]\, i\, v \in \{nil\} \cup \{v_1 \cdot .. \cdot v_k \cdot nil \mid v_1,..,v_k \in B\} \cup B^{\omega}$

Vertex v is *semantically infinite* iff it is not semantically finite. □

Definitions 3.1–3.3, our three progressively more inclusive definitions of static behaviour, all allow the vertices that satisfy their conditions to produce infinitely many different values in their output sequences. Definition 3.4 differs from Definition 3.3 by dropping property (b), thereby ensuring that only a finite set of different values is produced.

4. Safe and Conditionally Safe BTAs

In the previous section we defined the properties of staticness and finiteness in terms of the PRG semantics; we now use these definitions to establish a framework for determining the safety of binding-time analyses.

4.1. Specializable Vertices and Static-Infinite Computations

We group vertices in the PRG of a program with similar properties into sets as follows:

$Static(G) = \{ v \in V(G) \mid v \text{ is semantically static} \}$
$Finite(G) = \{ v \in V(G) \mid v \text{ is semantically finite} \}$
$Specializable(G) = Static(G) \cap Finite(G)$.

Vertices that belong to $Static(G)$ do not require any run-time inputs to compute their values. Some of these vertices are also finite; a specializer can perform the computation at these vertices, which are termed *specializable vertices*, without entering into non-terminating computation.

With the sets defined above, we are able to provide a formalization of the term "static-infinite computation".

Definition 4.1. PRG G is *static-infinite* iff the following holds:

$$Static(G) - Finite(G) \neq \emptyset.$$ □

In contrast with Jones et al. who give an intensional definition of an "infinite static loop" as "a loop not involving any dynamic tests" [7, pp. 118], Definition 4.1 is an extensional definition.

Given this formal notion of static-infinite computation, we can now define the notions of safety and conditional safety for binding-time analyses.

4.2. BTA characterizations

A binding-time analysis *bta* of program P (or its PRG G) is a function that maps vertices in G to the set $\{'S', 'D'\}$. We divide $V(G)$ into two sets $S(G)$ and $D(G)$ on this basis:

$$S_{bta}(G) = \{ v \in V(G) \mid bta\ G\ v = 'S' \}$$
$$D_{bta}(G) = V(G) - S_{bta}(G)$$

By mapping vertices to $'S'$, a binding-time analysis identifies them as vertices that are specializable. The binding-time analysis is safe only if these vertices are semantically specializable.

Definition. Binding-time analysis *bta* is *safe* on *Gset* iff $\forall G \in Gset$, $S_{bta}(G) \subseteq Specializable(G)$. □

A safe bta results in two-phase specialization that is guaranteed to terminate for all programs, including those that contain static-infinite computations. A natural way of weakening the condition on safety is to restrict the set of input programs to those that do not contain static-infinite computations:

Definition. Binding-time analysis *bta* is *conditionally safe* on *Gset* iff $\forall G \in Gset$, $S_{bta}(G) \subseteq Static(G)$. □

This definition is the tool with which one can formalize the notion of "a BTA for which the specialization phase terminates, assuming that the program contains no static-infinite computations":

Lemma. For a set of PRGs *Gset* that contains no static-infinite PRG,

bta is conditionally safe on *Gset* $\Rightarrow$ *bta* is safe on *Gset*.

5. Three Binding-Time Analyses for Imperative Programs via Program Slicing

In this section, we are interested in defining BTAs for imperative programs by using dependence analysis to identify dynamic vertices in their PRGs. We define three such BTAs as abstract interpretations of the PRG semantics; the first follows control dependences blindly and marks only strongly static vertices with $'S'$; the second follows control dependences selectively, and thus marks some weakly static vertices with $'S'$ as well. The third BTA marks some boundedly varying vertices $'S'$ by ignoring control dependences to vertices which have multiple static data dependence predecessors. We use the framework developed in the previous sections to prove the conditional safety of these analyses. All three BTAs can be viewed operationally as variants of operations for program slicing [15] and consequently can be performed as straightforward (and efficient) reachability operations on the PRG.

5.1. The Strong-Staticness BTA

A forward program slice [5] from vertex v in the PRG marks all vertices in the PRG that can be reached through dependence edges from v. Operationally, the Strong-Staticness BTA consists of marking with $'D'$ all vertices in the forward program slice from the set of **read** vertices in the PRG. Vertices that are not in this forward slice are marked with $'S'$.

Our task is now to justify this from a semantic standpoint—in particular, to show that this is a conditionally safe BTA. We do this by presenting the Strong-Staticness BTA as the fixed point of an abstract interpretation that is consistent with the PRG semantics defined in Section 2.2. This interpretation is defined by the following recursive equation (see Figure 3) which resembles the PRG equation from Section 2.2:

$$VertexAbs = Vertex \longrightarrow \{\ 'S', 'D'\ \} \quad \text{with } 'S' \sqsubset 'D'$$
$$f_a : VertexAbs \quad ; \quad f_a = \lambda v.\ \mathbf{E^a_G}(v, f_a)$$

All the *abs_** functions in $\mathbf{E^a_G}$ are continuous and propagate the value $'D'$ if any of their inputs is the value $'D'$.

The abstract semantics is defined as the least $f_a \in VertexAbs$ that satisfies the equation above:

$$\mathbf{M_a} : PRG \longrightarrow VertexAbs$$
$$\mathbf{M_a}\,[G] = \mathbf{fix}\,\mathbf{F_a} \quad \text{where} \quad \mathbf{F_a} : VertexAbs \longrightarrow VertexAbs$$
$$\mathbf{F_a} = \lambda f_a.\lambda v.\ \mathbf{E^a_G}(v, f_a)$$

□

$\mathbf{F_a}$ is continuous on a finite domain (a given G has a finite number of vertices). Hence, the fixed point is always reached in a finite number of steps. In fact, the abstract semantics merely encodes a reachability problem on the PRG whose solution can be obtained in time linear in the size of G.

In order to demonstrate that the Strong-Staticness BTA is conditionally safe (*i.e.*, that a vertex is marked $'S'$ at the fixed point only if it is strongly static), we compare the results of $\mathbf{F}$ and $\mathbf{F_a}$ using an abstraction function *abs*, as shown in Figure 4. *abs* takes an element of type *VertexFunc* from the concrete domain, determines whether that maps a vertex to a chain of sequences (possibly uncompleted) over all inputs, and abstracts the vertex output to $'S'$ or $'D'$ accordingly.

The conditional safety of the Strong-Staticness BTA is established by the following sequence of lemmas. (Some of the proofs are omitted for the sake of brevity.)

Lemma 5.1. *abs* is continuous on *VertexFunc*.

Proof. We prove the lemma in two parts:

(a) *abs* is monotonic on *VertexFunc*:

Consider $c, c' \in VertexFunc$ s.t. $c \sqsubseteq c'$. Then it must be that $c\ i\ v \sqsubseteq c'\ i\ v\ \ \forall i \in Stream, \forall v \in Vertex$.

if $abs(c)\ v = 'S'$ **then** $abs(c)\ v \sqsubseteq abs(c')\ v$ since $'S' \sqsubset 'D'$

else $abs(c)\ v = 'D'$. From Definition 3.1,
$\exists\ i_1, i_2 \in Stream$ s.t. $c\ i_1\ v$ and $c\ i_2\ v$ are incomparable.
Since $c\ i_1\ v \sqsubseteq c'\ i_1\ v$ and $c\ i_2\ v \sqsubseteq c'\ i_2\ v$, it follows that $c'\ i_1\ v$ and $c'\ i_2\ v$ are incomparable. Hence, $abs(c')\ v = 'D'$.

(b) for any chain $c_1, c_2, \cdots c_j, \cdots c_n$ in *VertexFunc*,

$$abs(\bigsqcup_{j=1}^{n} c_j)\ v = \bigsqcup_{j=1}^{n} abs(c_j)\ v:$$

if $abs(\bigsqcup_{j=1}^{n} c_j)\ v = 'S'$ **then** $abs(c_j)\ v = 'S',\ j = 1..n$ since *abs* is monotonic. Hence, $\bigsqcup_{j=1}^{n} abs(c_j)\ v = 'S'$

else $abs(\bigsqcup_{j=1}^{n} c_j)\ v = 'D'$. Then $\exists\ i_1, i_2 \in Stream$ s.t. $(\bigsqcup_{j=1}^{n} c_j)\ i_1\ v$ and $(\bigsqcup_{j=1}^{n} c_j)\ i_2\ v$ are incomparable. Let $k \in \mathbf{Nat}$ be the first position at which these sequences have different non-$\perp$ values. Then:

(i) $\exists\ m_1 \in [1..n]$ s.t. $|c_j\ i_1\ v| \geq k$ for all $j \geq m_1$

(ii) $\exists\ m_2 \in [1..n]$ s.t. $|c_j\ i_2\ v| \geq k$ for all $j \geq m_2$

Hence $|c_j\ i_1\ v| \geq k$ and $|c_j\ i_2\ v| \geq k$, for all $j \geq \mathbf{max}(m_1, m_2)$. Since $c_1,..,c_n$ is a chain, it follows that $c_j\ i_1\ v$ and $c_j\ i_2\ v$ differ at position k for all $j \geq \mathbf{max}(m_1, m_2)$. As a result, $abs(c_j)\ v = 'D'$ for all $j \geq \mathbf{max}(m_1, m_2)$ and $\bigsqcup_{j=1}^{n} abs(c_j)\ v = 'D'$. □

The next lemma is a statement of the property "chains beget chains".

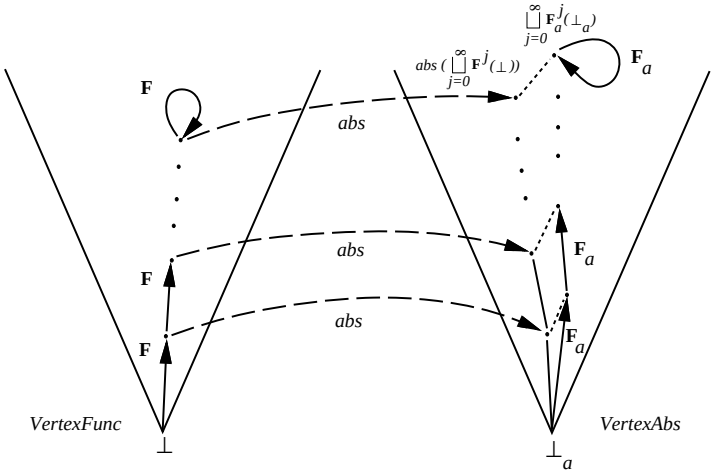
Lemma 5.2. For any PRG vertex v that is not a **read** vertex, $\{\ \mathbf{F}^{j+1}\ \perp\ i\ v \mid i \in Stream\ \}$ is a chain in *Sequence* if:

$$
\begin{aligned}
\mathbf{E}^{\mathbf{a}}_{\mathbf{G}}(v, f_a) \triangleq\ & \mathbf{type}(v) = \mathbf{Entry} \longrightarrow \text{'}S\text{'} \\
& \mathbf{type}(v) = \mathbf{read} \longrightarrow \text{'}D\text{'} \\
& \mathbf{type}(v) \in \{\, \mathbf{assign}, \mathbf{if}, \mathbf{while} \,\} \longrightarrow \begin{cases} \text{abs\_replace}(\ f_a\ parent\,(v)\ ) & \text{if } \#dataPreds\,(v) = 0 \\ abs\_map\ (\ f_a\ dataPred_1(v)\ ,\ f_a\ dataPred_2(v)\ ,\ \cdots\ ) & \text{otherwise} \end{cases} \\
& \mathbf{type}(v) = \phi_{\mathbf{enter}} \longrightarrow \text{abs\_whileMerge}(\ f_a\ whileNode\,(v)\ ,\ f_a\ innerDef\ (v)\ ,\ f_a\ outerDef\ (v)\ ) \\
& \mathbf{type}(v) = \phi_{\mathbf{exit}} \longrightarrow \text{abs\_select}(\ f_a\ whileNode\,(v)\ ,\ f_a\ dataPred\,(v)\ ) \\
& \mathbf{type}(v) = \phi_{\mathbf{while}} \longrightarrow \text{abs\_select}(\ f_a\ whileNode\,(v)\ ,\ f_a\ dataPred\,(v)\ ) \\
& \mathbf{type}(v) = \phi_{\mathbf{T}} \longrightarrow \text{abs\_select}(\ f_a\ parent\,(v)\ ,\ f_a\ dataPred\,(v)\ ) \\
& \mathbf{type}(v) = \phi_{\mathbf{F}} \longrightarrow \text{abs\_select}(\ f_a\ parent\,(v)\ ,\ f_a\ dataPred\,(v)\ ) \\
& \mathbf{type}(v) = \phi_{\mathbf{if}} \longrightarrow \text{abs\_merge}(\ f_a\ ifNode\,(v)\ ,\ f_a\ trueDef\ (v)\ ,\ f_a\ falseDef\ (v)\ )
\end{aligned}
$$

where *abs_replace*, *abs_map*, *abs_whileMerge*, *abs_select* and *abs_merge* are defined as follows:

$$abs\_replace \triangleq \lambda a.a\ ,\ abs\_select \triangleq \lambda a.\lambda b.a \sqcup b\ ,\ abs\_whileMerge \triangleq abs\_merge \triangleq \lambda a.\lambda b.\lambda c.a \sqcup b \sqcup c\ ,\ abs\_map \triangleq \lambda a_1..\lambda a_n.a_1 \sqcup .. \sqcup a_n$$

Figure 3. The abstract equations representing the Strong-Staticness BTA.



$$abs : VertexFunc \longrightarrow VertexAbs$$

$$abs\,(c) = \begin{cases} \lambda v.\text{'}S\text{'} & \text{if } \{\, c\ i\ v \mid i \in Stream \,\} \\ & \text{is a chain in } Sequence \\ \lambda v.\text{'}D\text{'} & \text{otherwise} \end{cases}$$

Figure 4. *abs* is an abstraction function used to compare the results of **F** and $\mathbf{F_a}$.

$$\forall\ u \in preds\,(v),\ \{\, \mathbf{F}^j\ \perp\ i\ u \mid i \in Stream \,\} \text{ is a chain in } Sequence.$$

The proof of this property involves a case analysis on the PRG equations.

Our next task is to show that, at every step, the vertex function produced by **F** abstracts to a lower value than that produced by $\mathbf{F_a}$ at the corresponding step (see Figure 4).

Lemma 5.3. $abs\,(\mathbf{F}^j\ \perp) \sqsubseteq \mathbf{F}_{\mathbf{a}}^j\ \perp_a\ \forall\ j \in \mathbf{Nat}$

Proof. We prove the lemma by induction on *j*:

Base case ($j = 0$): $abs\,(\perp)\ v = \text{'}S\text{'} \sqsubseteq \perp_a\ v$

Induction step: assume $abs\,(\mathbf{F}^j\ \perp) \sqsubseteq \mathbf{F}_{\mathbf{a}}^j\ \perp_a$

if $abs\,(\mathbf{F}^{j+1}\ \perp)\ v = \text{'}S\text{'}$ **then** $abs\,(\mathbf{F}^{j+1}\ \perp)\ v \sqsubseteq \mathbf{F}_{\mathbf{a}}^{j+1}\ \perp_a\ v$

else $abs\,(\mathbf{F}^{j+1}\ \perp)\ v = \text{'}D\text{'}$. From Lemma 5.2, either

(*i*) *v* is a **read** vertex. Then $\mathbf{F}_{\mathbf{a}}^{j+1}\ \perp_a\ v = \text{'}D\text{'}$ by definition.

(*ii*) $\exists\ u \in preds\,(v)$ s.t. $abs\,(\mathbf{F}^j\ \perp)\ u = \text{'}D\text{'}$. Hence by assumption, $\mathbf{F}_{\mathbf{a}}^j\ \perp_a\ u = \text{'}D\text{'}$. Then $\mathbf{F}_{\mathbf{a}}^{j+1}\ \perp_a\ v = \text{'}D\text{'}$ by definition of $\mathbf{F_a}$. □

Phrased differently, Lemma 5.3 says that at every step, if the value produced by $\mathbf{F_a}$ at a vertex is 'S' then **F** produces a chain of sequences over all inputs at the given vertex.

This result, when extended to the fixed points of $\mathbf{F_a}$ and **F**, demonstrates that the Strong-Staticness BTA is conditionally safe for all PRGs:

Theorem 5.4. For every vertex *v* in PRG *G*,

$$\mathbf{M_a}\ [G]\ v = \text{'}S\text{'} \Rightarrow v \in Static\,(G).$$

Proof. From Lemma 5.3, for all *j*, $abs\,(\mathbf{F}^j\ \perp)\ v \sqsubseteq \mathbf{F}_{\mathbf{a}}^j\ \perp_a\ v$.

Hence, $\bigsqcup_{j=0}^{\infty} abs(\mathbf{F}^j \perp) v \sqsubseteq \bigsqcup_{j=0}^{\infty} \mathbf{F}_{\mathbf{a}}^j \perp_a v$. Because *abs* is continuous (Lemma 5.1), it follows that:

$$abs(\bigsqcup_{j=0}^{\infty} \mathbf{F}^j \perp) v \sqsubseteq \bigsqcup_{j=0}^{\infty} \mathbf{F}_{\mathbf{a}}^j \perp_a v$$

or $abs(\mathbf{M}[G]) v \sqsubseteq \mathbf{M_a}[G] v$. In particular, if $\mathbf{M_a}[G] v = 'S'$, $abs(\mathbf{M}[G]) v = 'S'$. Hence $\{ \mathbf{M}[G] i v \mid i \in Stream \}$ is a chain in *Sequence*. Because $\mathbf{M}[G]$ does not produce any $\perp$-terminated sequences, it must be that $\mathbf{M}[G] i v$ is the *same* value for all $i \in Stream$, from which it follows that $v \in Static(G)$. □

To summarize, we have shown that the forward-slice operation, a natural algorithm for tracing dynamic behaviour in terms of dependences, produces a conditionally safe binding-time-analysis algorithm. To define a safe BTA, the algorithm would need to be extended with an auxiliary analysis to detect static-infinite computations.

Because program slicing can be solved as a reachability problem on the PRG, the computational complexity of the Strong-Staticness BTA is linear in the size of the PRG.

5.2. The Weak-Staticness BTA

The Strong-Staticness BTA is a rather restrictive analysis because it always transmits dynamic behaviour through control dependences. This is undesirable in situations where static computations may be nested beneath dynamic predicates, as in programs P_2 and P_3 from Section 3. We define the Weak-Staticness BTA, an analysis that is identical to the Strong-Staticness BTA except at constant assignment vertices, to tackle this problem. The sequence produced at a constant assignment vertex is given by (Figure 2):

$$f\ i\ v = \text{replace}(controlLabel(v),\ funcOf(v),\ f\ i\ parent(v))$$

where *funcOf* (v) is the constant expression and *parent* (v) is the control predecessor. In the corresponding abstract semantic function used in the Strong-Staticness BTA a $'D'$ value is produced if *parent* (v) has a $'D'$ value, since $f\ i\ parent(v)$ determines the length of $f\ i\ v$. In the Weak-Staticness BTA an $'S'$ value is produced regardless, the idea being that although $f\ i\ parent(v)$ determines the length of $f\ i\ v$, it does not determine the actual values in it (since the same value is produced multiple times).

Example. In program P_3 from Section 3, the constant assignment $x_2 := 0$ within the dynamic outer loop is marked $'S'$ by the Weak-Staticness BTA. As a result, the entire inner loop is marked $'S'$, and specialization produces the following residual program:

```
P3': read (x1);
     while ( x1 ≠ 0 ) do
       x2 := 3 ;
       x1 := x1 − 1;
     od
```

The initialization of x_2 in P_3 has the effect of blocking the dependence from the outer loop to the inner. If the initialization were moved outside the outer loop, the inner loop would no longer be invariant with respect to the outer; it would be marked $'D'$ by the Weak-Staticness BTA. □

The proof that this BTA is conditionally safe mimics the one for the Strong-Staticness BTA, with two modifications:

(a) *abs* is modified to capture weakly static behaviour:

$$abs(c) = \begin{cases} \lambda v.'S' & \text{if } \{ c\ i\ v \mid i \in Stream \} \\ & \text{is an approximate rational repetition} \\ \lambda v.'D' & \text{otherwise} \end{cases}$$

(b) Lemma 5.2 is modified to account for weakly static behaviour:

Lemma 5.5. For a PRG vertex v that is not a **read** vertex, $\{ \mathbf{F}^{j+1} \perp i\ v \mid i \in Stream \}$ is an approximate rational repetition if:

$$\forall u \in preds(v),\ \{ \mathbf{F}^j \perp i\ u \mid i \in Stream \} \text{ is an approximate rational repetition.}$$

The functions at PRG vertices are all structured so that when predecessor sequences $u_1, u_2, \cdots, u_k$ at vertex v are all rational repetitions, the output sequence at v is a rational repetition whose base repeating sequence is at most as long as the least common multiple of the lengths of the base repeating sequences in $u_1, u_2, \cdots, u_k$.

Proceeding as before, we use this property to show that the Weak-Staticness BTA is conditionally safe on all PRGs (that is, we can show the analogue of Theorem 5.4).

5.3. The Bounded-Variation BTA

The Weak-Staticness BTA is also a somewhat restricted analysis because it assumes that the result of using a dynamic condition to choose between static values is dynamic. This is undesirable in situations where static computations nested beneath different branches of a dynamic predicate are used in later computations, as in program P_4 from Section 3.

To tackle this problem, we define the Bounded-Variation BTA, an analysis that is identical to the Weak-Staticness BTA except at ϕ_{if} and ϕ_{exit} vertices. The sequence produced at a ϕ_{if} vertex is given by (Figure 2):

$$f\ i\ v = \text{merge}(f\ i\ ifNode(v),\ f\ i\ trueDef(v),\ f\ i\ falseDef(v))$$

where *ifNode* (v) is the corresponding predicate and *trueDef* (v) (*falseDef* (v)) is the definition within the *true* (*false*) branch of the conditional statement. In the corresponding abstract semantic function used in the Weak-Staticness BTA a $'D'$ value is produced if *ifNode* (v) has a $'D'$ value, since $f\ i\ ifNode(v)$ determines the values in $f\ i\ v$. In the Bounded-Variation BTA an $'S'$ value is produced regardless, the idea being that if the data predecessors produce bounded values, the ϕ_{if} produces bounded values as well, as it produces only values produced at either of its data predecessors.

Example. In program P_5 below, the assignment $x_3 := x_2$ is marked $'S'$ by the Bounded-Variation BTA.

```
P5:  read (x1);
     if ( x1 ≠ 0 ) then
        x2 := 0
     else
        x2 := 10
     fi;
     x3 := x2;
     if x3 < 10 then
        x4 :=0
     else
        read(x4)
     fi
```

As a result, the predicate following it is marked *'S'*, and specialization produces the following residual program:

```
P5':  read (x1);
      if ( x1 ≠ 0 ) then
         x4 := 0
      else
         read(x4)
      fi
```

□

The Bounded-Variation BTA seems plausible because of the following property: the functions at PRG vertices all have the property that when predecessor sequences u_1, u_2, $\cdots$, u_k at vertex v are all bounded variations, the output sequence at v is a bounded variation whose base set of values is at most as large as the product of the sizes of the base sets of values in u_1, u_2, $\cdots$, u_k.

Unfortunately, we have not been able to provide a semantic justification for the Bounded-Variation BTA. The difficulty lies in finding an abstraction function that captures Definition 3.3 and that is continuous over the domain *VertexFunc*. In particular, the following candidate abstraction function is not continuous because the successive approximants $\mathbf{F}^j \perp i\ v$ are always *'S'*, whereas $\mathbf{M}\ [G]\ i\ v$ might be *'D'*:

$$abs(c) = \begin{cases} \lambda v.'S' & \text{if } \{\, c\ i\ v \mid i \in \mathit{Stream} \,\} \\ & \text{is an approximate bounded variation} \\ \lambda v.'D' & \text{otherwise} \end{cases}$$

6. Related Work

One novelty of our treatment of BTAs lies in the use of control dependences along with data dependences to trace the flow of dynamic computations through a program. Control dependences were introduced by Denning and Denning to formalize the notion of information flow in programs in the context of computer-security issues [2]. Since then, they have played a fundamental role in vectorizing and parallelizing compilers (for instance, see [3].) The possibility of using control dependences during binding-time analysis was hinted at by Jones in a remark about "indirect dependences" caused by predicates of conditional statements [6, pp. 260], but this direction was not pursued.

In [7], Jones *et al.* informally present the notions of oblivious and weakly oblivious programs (in contrast with unoblivious programs), a distinction based on whether a program involves tests on dynamic data. While this is clearly related to control dependence (the test predicate is a control dependence predecessor of statements within the test structure), the notion of weakly oblivious is stronger than is necessary.

In the context of imperative programs, Meyer presents an approach that uses dynamic annotations rather than a separate BTA phase in order to obtain more efficient residual programs [8]. However, his analysis loses some precision as a result. Furthermore, he omits any discussion of termination by assuming that the program terminates for all inputs, which is a stronger restriction than "absence of static-infinite computation", the condition required for the results of our analyses to be used safely.

In [4], Holst uses the notion of in-situ increasing and decreasing parameters to argue about termination of specialization, and hence eliminates the need for any finiteness condition on programs. However he deals with data types (lists) that cannot decrease in an unbounded manner as our data types of interest (integers, reals) can.

Wand presents a correctness criterion for BTA-based partial evaluation of terms in the pure λ-calculus [14]. However, it is not clear to us whether the safety issue that we have examined in the present paper arises in the context of Wand's work.

A second novelty of our work is the use of a value-sequence-oriented semantics for imperative programs instead of a state-oriented semantics. With the value-sequence semantics, we identify *program points* as being static or dynamic, whereas state-oriented semantics have been used to identify which *variables* are static/dynamic at program points (*cf.* [6]). As we have shown, the value-sequence approach provides a clean way to formalize the notions needed to characterize safety conditions for BTAs, namely, "static", "dynamic", "finite", and "infinite".

We are not aware of any antecedents of the value-sequence approach in the partial-evaluation literature.

References

1. Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).
2. Denning, D.E. and Denning, P.J., "Certification of programs for secure information flow," *Commun. of the ACM* **20**(7) pp. 504-513 (July 1977).
3. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).
4. Holst, C.K., "Finiteness analysis," pp. 473-495 in *Functional Programming and Computer Architecture, Fifth ACM Conference*, (Cambridge, MA, Aug. 26-30, 1991), *Lecture Notes in Computer Science*, Vol. 523, ed. J.Hughes,Springer-Verlag, New York, NY (1991).
5. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

6. Jones, N.D., “Automatic program specialization: A reexamination from basic principles,” pp. 225-282 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, (Gammel Avernaes, Denmark, 18-24 October, 1987), ed. D. Bjϕrner, A.P. Ershov, N.D. Jones,North-Holland, New York, NY (1988).
7. Jones, N.D., Gomard, C.K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International, Englewood Cliffs, NJ (1993).
8. Meyer, U., “Techniques for partial evaluation of imperative languages,” *Proceedings of the SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 91)*, (New Haven, CT, June 17-19, 1991), *ACM SIGPLAN Notices* **26**(9) pp. 94-105 (September 1991).
9. Mogensen, T., “Partially static structures in a self-applicable partial evaluator,” pp. 325-347 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, (Gammel Avernaes, Denmark, 18-24 October, 1987), ed. D. Bjϕrner, A.P. Ershov, N.D. Jones,North-Holland, New York, NY (1988).
10. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
11. Ramalingam, G. and Reps, T., “Semantics of program representation graphs,” TR-900, Computer Sciences Department, University of Wisconsin, Madison, WI (December 1989).
12. Schmidt, D., *Denotational Semantics*, Allyn and Bacon, Inc., Boston, MA (1986).
13. Sestoft, P., “Automatic call unfolding in a partial evaluator,” pp. 485-506 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, (Gammel Avernaes, Denmark, 18-24 October, 1987), ed. D. Bjϕrner, A.P. Ershov, N.D. Jones,North-Holland, New York, NY (1988).
14. Wand, M., “Specifying the correctness of binding-time analysis,” pp. 137-143 in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, (Charleston, SC, January 10-13, 1993), ACM, New York, NY (1993).
15. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).
16. Yang, W., Horwitz, S., and Reps, T., “A program integration algorithm that accommodates semantics-preserving transformations,” *ACM Trans. Software Engineering and Methodology* **1**(3) pp. 310-354 (July 1992).