PARTIAL EVALUATION USING DEPENDENCE GRAPHS

By

Manuvir Das

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN – MADISON

1998

Abstract

This thesis describes the use of program dependence graphs, as opposed to control flow graphs, as the basis for the partial evaluation of imperative programs. Partial evaluation is a program specialization operation in which programs with multiple inputs are specialized to take into account known values for some of their inputs. Thus, the result of partially evaluating a program given a division of its inputs into known or "static" inputs and unknown or "dynamic" inputs is a specialized version of the program, in which computations that require only the static inputs are absent. The specialized program produces exactly the same output, when run on the dynamic inputs, as that produced by the original program when run on all of its inputs.

A known problem with partial evaluators has been that in attempting to aggressively identify static code, they may fail to terminate on some subject programs. However, partial evaluators are increasingly being developed for heavily used languages, for which the goal is to allow an arbitrary user to improve the performance of his or her program by using a partial evaluator as a "black box" that optimizes code, in much the same way that optimizing compilers have been used for years. Therefore, it is necessary to ensure that partial evaluators provide a semantic guarantee of termination, while optimizing the common case.

A partial evaluator may fail to terminate on a subject program if its analysis phase (termed "binding-time analysis" or BTA) fails to identify variables whose values are built up in dynamic loops (*i.e.*, loops whose predicates use dynamic data) as dynamic. We use program dependence graphs, which make both data dependences and control

dependences explicit in their structure, as the basis for BTA algorithms that tackle this problem. As a result, our algorithms provide a termination guarantee for partial evaluation in the absence of "static-infinite computations" (roughly, infinite computations that use only static data). We argue that this is an appropriate termination guarantee for partial evaluation of imperative programs. In order to handle a real imperative language such as C, with complex features including arbitrary control flow and pointer variables, we identify a new form of program dependence, called "loop dependence." Our BTA algorithm is able to use these dependences to provide a termination guarantee, without compromising the ability of the partial evaluator to aggressively identify static code and execute it at compile time.

In the case of functional programs that manipulate only list data, it is possible to design analysis algorithms that provide a termination guarantee for partial evaluation on all programs (including programs that contain static-infinite computations), without unduly compromising the ability of the analysis to identify static code. We present an analysis algorithm that uses generalized reachability properties on a dependence graph representation to provide such a termination guarantee.

Dependence graphs provide a natural basis for the analysis actions of partial evaluation because they make program dependences explicit in their structure. Another question that arises is whether dependence graphs provide a basis for the specialization actions of partial evaluation as well. We address this question by extending the specialization operation to dependence graphs.

Finally, we present experimental results obtained using our implementation of the ideas presented in this thesis.

Acknowledgements

I would first like to thank my advisor, Thomas Reps, for his constant help and guidance with this doctoral thesis. He has shown me the value of clear thinking and working with concepts rather than detail. I would also like to thank Charles Fischer and Susan Horwitz for their comments on this thesis, and Raghu Ramakrishnan and Kewal Saluja for sitting on the thesis committee.

In addition, I would like to thank all the people who made my days working on this dissertation an enjoyable experience. My officemate Michael Siff, Navin Kabra, members of "The Family", and everyone else who I cannot mention here.

I would also like to thank my parents, Premvir and Nalini Das. This thesis is the result of all their guidance, love, wishes, and advice over the years.

And finally, I would like to thank my wife, Payal. Every word of this thesis was written with thoughts of her running through my mind. Her love has made it all possible, and meaningful. This thesis is for her.

List of Figures

Contents

Chapter 1

Introduction

This thesis describes the use of program dependence graphs, as opposed to control flow graphs, as the basis for the off-line partial evaluation of imperative programs. The primary result of this research is a new form of binding-time analysis that ensures the termination of off-line partial evaluation without compromising the ability of the analysis to identify static behaviour.

Partial evaluation is a program specialization operation in which programs with multiple inputs are specialized to take into account known values for some of their inputs. Thus, the result of partially evaluating a program given an "initial division" of its inputs into known or "static" inputs and unknown or "dynamic" inputs is a specialized version of the program, which when run on the dynamic inputs produces exactly the same output as that produced by the original program when run on all of its inputs [JGS93]. The intent of partial evaluation is that the specialized program will require less time to execute than the original program, as some computations that can be completed using only the known inputs will be absent in the specialized program. In an off-line partial evaluator for imperative programs, the process of partial evaluation is separated into two phases: The first phase is an analysis phase, termed "binding-time analysis" (BTA), in which all of the variables in the program whose values are computable using only the known inputs (static variables) are identified. Alternatively, a BTA algorithm may first identify program statements as either static or dynamic, and then convert this information into static or dynamic markings for program variables. The second phase is the specialization phase, in which statements involving only static variables are executed and statements involving dynamic variables are emitted, once for each combination of values taken by the static variables in the program. The emitted statements are combined to produce the specialized program.

The traditional approach to partial evaluation has been to aggressively identify static program variables; this has resulted in the development of partial evaluators that produce efficient code wherever possible, but that may not terminate on some subject programs. This approach has sufficed in the past because partial evaluators have typically been used by programmers who have written these tools themselves. However, partial evaluators are increasingly being developed for heavily used languages, for which the goal is to allow an arbitrary user to improve the performance of his or her program by using a partial evaluator as a "black box" that optimizes code, in much the same way that optimizing compilers have been used for years. Therefore, the old aggressive model of partial evaluation is no longer suitable. In particular, this thesis advocates a model similar to that of traditional optimizing compilers, which attempt to optimize the common case while providing a semantic guarantee of safety for all programs.

As pointed out by Jones in [Jon88], a partial evaluator will not terminate on a subject program if it attempts to enumerate an unbounded set of values taken on by a program variable that has been identified as static. A variable that may take on an unbounded number of different values over all possible values of the unknown inputs is said to fail the "bounded-static-varying" (BSV) property. Therefore, a partial evaluator will fail to terminate if it identifies as static a variable that is not BSV. Traditional BTA algorithms identify static and dynamic statements in a program by tracing the flow of dynamic values forward from the dynamic inputs of a program through the statements in the program using "flow dependences." Informally, a flow dependence from a statement u to a statement v indicates that a variable that is assigned a value at statement u may be used in the computation at statement v. Alternatively, the values computed at u may affect the values computed at v. A BTA that uses flow dependences to trace dynamic behaviour is said to be correct if it satisfies the "congruence" condition: A BTA is congruent if for every statement in the program that is identified as static, all of the flow predecessors of the statement are also identified as static.

Consider the power function shown in Figure 1, an example commonly found in early papers explaining the concept of partial evaluation. If function *power* from Figure 1 is specialized with an initial division in which the base x is dynamic and the index n is static, the loop header in the function is identified as static and the loop can therefore be unrolled, as expected. On the other hand, if *power* is specialized with static base x and dynamic index n, the assignment to a within the dynamic loop controlled by n is identified as static by a congruence-based BTA. This is because there is no path of flow dependences from the dynamic parameter n to the assignment to a within the loop. However, the values taken by a cannot be bounded if the value of n is not known. Hence, variable a is not BSV. As a result, a partial evaluator using the results of a congruence-based BTA would not terminate on the power function, given an input division in which base x is static and index n is dynamic. The primary focus of the research described in this thesis is a new form of BTA that treats variables such as a

as dynamic, thus ensuring the termination of partial evaluation.

Looking at variable a in function *power* from Figure 1 again, the values taken by a are unbounded because even though all of its values are built by operations on initially static values, these values are built up under dynamic control, as pointed out by Jones in [Jon88]. Therefore, variable a should not be treated as static. In this thesis, we show that congruence-based BTAs are unable to identify variables that take values built up under dynamic control as dynamic for the following reason: At the semantic level, congruence-based BTAs are based on a control-flow based structure, in which every statement is treated as a state-to-state transformer. Under this model, a statement such as the assignment to a within the loop in *power* from Figure 1 is static because every value computed at the statement is known, as indicated by the state transformation function associated with the statement. The statement should in fact be treated as dynamic, because the complete set of values computed at the statement is not bounded, but this information is not captured by the state transformation function. At the algorithmic level, congruence-based BTAs use flow dependences to trace dynamic behaviour, and there is no flow dependence from a dynamic loop predicate to the assignments within its body.

Clearly, flow dependences do not contain adequate information for tracing dynamic behaviour through programs. However, previous work in the areas of program security [DD77], automatic parallelization [FOW87], and program slicing [Wei84, HRB90] has relied on another form of dependence between program statements in addition to flow dependence, termed "control dependence." Informally, a control dependence from a statement u to a statement v indicates that the computation at u may determine the number of times the computation at v is carried out during program execution.

(a)

```
float power ( float x, int n ) {
  float a = 1.0;
  while ( n > 0 ) {
    n = n - 1;
    a = a * x;
  }
  return a;
}
```

(b)

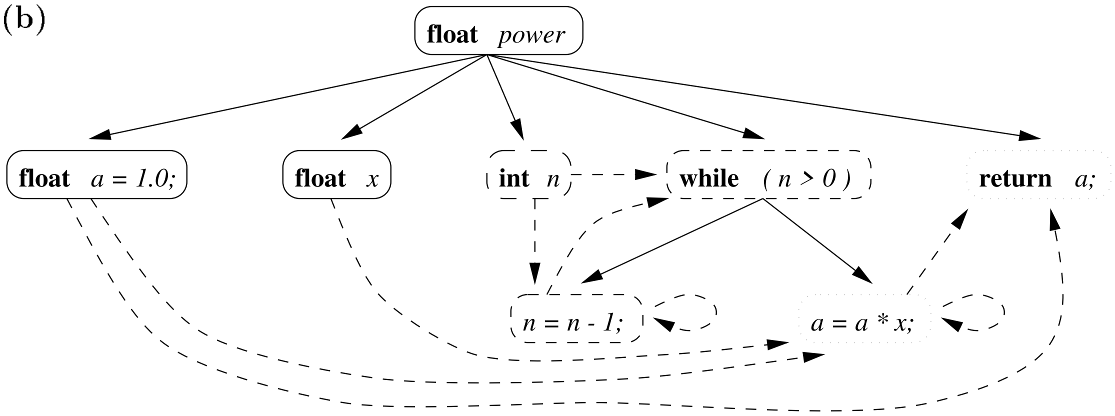


Figure 1: The power function.

In figure (a) above, function *power* assigns the value of x^n to a and returns this value. The dependence graph for *power* is shown in figure (b) above. Every program point is represented by a vertex in the dependence graph. Vertices in the graph are connected by two kinds of edges: Flow dependence edges, shown as dashed lines, and control dependence edges, shown as solid lines. Vertices shown as dashed boxes are identified as dynamic by a congruence-based BTA that follows only flow dependence edges, given an initial division in which base x is static and index n is dynamic, while vertices shown as dotted boxes are identified as dynamic by a BTA that also follows control dependences.

As pointed out by Denning and Denning in [DD77], control dependences may reflect hidden data flow through a program, as is the case in the power function from Figure 1, in which, although there are no flow dependences from parameter n to the assignment to a, the variable is assigned a value that is a mathematical function (in particular, the exponentiation function) of parameters x and n.

Therefore, this thesis advocates BTA algorithms based on following both flow and control dependences through a program. For this purpose, our work relies on program dependence graphs [FOW87, HRB90], representations that make both flow and control dependences explicit in their structure and semantics, as opposed to the control-flow representations used by congruence-based BTAs.

At the semantic level, dependence graphs have a sequence semantics in which every program point is characterized by the sequence of values produced at the program point during execution. Therefore, by using dependence graphs as the basis for BTA, we are able to provide semantic characterizations of static behaviour that properly account for the effect of dynamic control. At the algorithmic level, dependence graphs contain control dependences, which can be used to correctly account for the effect of dynamic control on the values taken by program variables.

The role of control dependences in binding-time analysis has been noted previously in the literature, in particular by Ershov in [Ers82] and by Jones in [Jon88]. However, previous work on defining or implementing BTA algorithms has ignored control dependences. This is because it has been argued that following control dependences leads to conservative BTA algorithms that identify too few statements as static. In particular, it has been believed that important objectives such as self application, which refers to the application of a partial evaluator to itself, and the ability to handle a common modification to user code known as "The Trick" cannot be met when using control dependences.

However, in later chapters of this thesis, we show that it is possible to define BTA algorithms for realistic programming languages (in particular, C) that use control dependences to ensure termination without compromising either the ability of the partial evaluator to carry out non-trivial self application or its ability to handle code that represents The Trick. This is because of two reasons: Firstly, it is possible to define analyses that follow control dependences selectively, in particular, in situations where control dependences reflect looping behaviour. Secondly, results of a BTA that uses control dependences can be improved using a precision analysis that eliminates some control dependence edges from the dependence graph based on conservative termination criteria.

Thus, the BTA algorithms defined and implemented as part of this thesis research use flow and control dependences to provide a termination guarantee for partial evaluation, without compromising performance in the common case. There are two issues that require clarification with regard to our algorithm and its relationship to previous work in this area:

- We do not insist that partial evaluation must terminate under all conditions. In particular, some programs contain variables that fail the BSV property even though they are built up from static values and are not built up under dynamic control. One such program is shown in Figure 2. Intuitively, the values taken by variable n in function *st-inf* from Figure 2 are unbounded even though they are independent of the dynamic input. This behaviour has been referred to by Jones as "static-infinite computation" in [Jon88]. He has argued that a partial evaluator need not provide a termination guarantee on programs that contain static-infinite computation because they reflect poor programming: If a program containing a static-infinite loop is executed and control reaches such a loop during execution, the program diverges. Further, a partial evaluator that guarantees termination on programs with static-infinite computation must examine every static loop for possible unbounded behaviour. Because precisely determining the bounded nature of a program loop reduces to solving the halting problem, an analysis that guarantees the termination of every static loop must produce conservative results. Therefore, we adopt Jones' approach by defining BTA algorithms that fail to provide a termination guarantee in the presence of static-infinite computation. In this thesis we have used the semantics of dependence graphs to provide the first semantic characterization of static-infinite computation. This allows us to be precise about what our BTA algorithms do and do not ensure. In particular, the BTA algorithms defined in this thesis provide a semantic guarantee of termination for partial evaluation in the absence of static-infinite computation.

- Other authors have also tackled the termination problem, and have defined termination analyses that can be combined with congruence-based BTA to produce analyses that provide a termination guarantee for all programs [Hol91, GJ96, AH96]. The chief differences between their work and ours are:
  - Other termination analyses are applicable only to restricted languages, and can analyze only simple kinds of termination criteria. A later chapter of

```
int st-inf ( int x ) {
  int n = 1;
  while ( n > 0 ) {
    n = n + 1;
    x = x − 1;
  }
  return n;
}
```

Figure 2: Static-infinite computation.

If function *st-inf* above is specialized given an initial division in which parameter x is dynamic, the values taken by variable n cannot be bounded. Hence, n is not BSV. However, the values taken by n are built up from static values under purely static control. Therefore, function *st-inf* contains static-infinite computation. The BTA algorithms described in this thesis would identify n as a static variable.

this thesis describes a new termination analysis that improves upon previous analyses by using an optimistic approach and context-free language reachability (CFL-Reachability), an extended form of graph reachability. This algorithm provides a termination guarantee even in the presence of static-infinite computation.

– More importantly, other termination analyses are based on conservatively identifying BSV variables, and do not consider control dependences explicitly. As a result, such analyses cannot distinguish between unbounded behaviour arising from dynamic control and unbounded behaviour arising from static-infinite computation. Therefore, every static loop must be treated conservatively, limiting the ability of the BTA to identify static variables. In contrast, the approach advocated in this thesis can be explained as follows: We selectively use control dependences to distinguish potential unbounded behaviour arising from dynamic control from unbounded behaviour resulting from static-infinite computation. We then use a precision analysis to conservatively treat as static some of the variables identified as unbounded due to dynamic control.

In summary, control dependences, or the dependence graphs that make them explicit, serve two purposes in providing a termination guarantee for partial evaluation: They represent hidden data flow through data transfer loops, such as in the power function from Figure 1, and they provide the ability to distinguish between the two forms of unbounded behaviour, as shown in Figure 2, so that conservative analyses can be performed on only the programs that require a termination guarantee.

Program dependence graphs provide a natural basis for binding-time analysis because they make both data and control dependences explicit in their structure and semantics. In addition, the BTA algorithms defined on dependence graphs in this thesis are variations on the operation of program slicing [Wei84, HRB90] that has previously been defined on dependence graphs. The natural question that therefore arises is whether dependence graphs provide a suitable basis for the specialization phase of partial evaluation as well. In this thesis, we have designed a specialization algorithm that operates on dependence graphs, and we have built a specializer for program representation graphs (PRGs) [RR89], variants of program dependence graphs that have a data-flow semantics. This experience has shown that whereas dependence graph specialization has the advantage that certain elements of the specialization process are more easily defined, the use of dependence graphs in the specialization phase introduces problems as well. In particular, the lack of a state in dependence graphs makes reconstitution of the specialized program from the specialized dependence graph non-trivial.

The major contributions of this thesis can be summarized as follows:

- A semantic foundation for safe binding-time analysis on imperative programs, including a semantic characterization of static-infinite computation.

- Three BTA algorithms on dependence graphs that provide a guarantee of termination for partial evaluation in the absence of static-infinite computation.

- The definition of "loop dependence," a property similar to control dependence, for handling programs with arbitrary control flow, together with a BTA algorithm for C programs that uses loop dependences to provide a termination guarantee for partial evaluation in the absence of static-infinite computation.

- A termination analysis for functional programs that uses an optimistic approach and more complex termination criteria to improve upon the results of previous analyses. This analysis provides a termination guarantee for all programs.

- Extension of the program specialization operation to dependence graphs, and an implementation of specialization on dependence graphs and reconstitution of program text from dependence graphs.

- A safe binding-time analysis for C programs that uses loop dependences and a precision analysis to provide a termination guarantee for partial evaluation. We claim that this approach does not significantly compromise the range of programs that can be specialized in practice, and present experimental evidence gathered from our implementation of this method to back up this claim.

The contributions of this thesis in terms of handling the termination problem are summarized graphically in Figure 3.

We have chosen an imperative language with side effects, in particular C, as the basis for much of the work described in this thesis because the motivation for this work is to make partial evaluation accessible to ordinary programmers (as opposed to experts in partial evaluation). However, the concepts developed in this thesis can be applied to programs written in functional languages as well.

The chapters of this thesis are organized as follows:

In Chapter 2 we present background material on the key concepts on which this thesis is based, including partial evaluation, dependence graphs, termination analysis, and CFL-Reachability.

In Chapter 3 we present the program representation graph (PRG), and we use the semantics of PRGs to develop a semantic foundation for binding-time analysis. We use this foundation to characterize static-infinite computation, and the notion of conditionally safe BTA. We define three conditionally safe BTA algorithms that are reachability operations on the PRG.

In Chapter 4 we extend PRGs to system representation graphs (SRGs), which represent programs with procedures. We extend the notion of conditional safety and the BTA algorithms defined in Chapter 3 to SRGs.

In Chapter 5 we define the notion of loop dependences, which are similar to control dependences. We use loop dependences to define a conditionally safe BTA for programs with arbitrary control flow. We also define a precision analysis that we use in conjunction with conditionally safe BTA in order to provide a termination guarantee for partial evaluation of C programs without compromising the ability of the analysis to identify

Safety Level	Functional programs (S-expressions)	Single-procedure imperative programs	Multi-procedure imperative programs	Programs w/ arbitrary control flow and aliasing
Unsafe	Previous Work	Previous Work	Previous Work	Previous Work
Conditionally safe	SRG BTA (Chapter 4)	PRG BTA (Chapter 3)	SRG BTA (Chapter 4)	Loop-Dep BTA (Chapter 5)
Safe	Previous Work CFL-Reachability (Chapter 6)	Previous Work	No Work	No Work

Figure 3: Solutions to the termination problem presented in this thesis.
The contributions of this thesis to the termination problem in partial evaluation are summarized above. Analyses are categorized as either unsafe (no termination guarantee for partial evaluation), conditionally safe (termination guarantee in the absence of static-infinite computation), or safe (termination guarantee for all programs). Unsafe BTA algorithms that use only flow dependences have been designed for all of the language categories. The BTA algorithms designed in this thesis handle programs in all of the language categories. They use control dependences or loop dependences in addition to flow dependences, and are therefore conditionally safe. Safe BTA algorithms have been designed for functional programs that manipulate only S-expression data. Our termination analysis algorithm extends this work. A termination analysis has been developed for single-procedure programs with only downwards-closed integral data (for instance, natural numbers). No such analyses exist for either multi-procedure programs or programs with pointer variables.

static behaviour. We present experimental evidence based on our implementation.

In Chapter 6 we define a termination analysis for functional programs that extends the work of Holst in [Hol91] and Glenstrup and Jones in [GJ96]. The algorithm, which conservatively identifies BSV parameters, uses an optimistic approach and CFL-Reachability, an extended form of graph reachability, to obtain more precise results than previous algorithms.

In Chapter 7 we extend the operation of specialization to dependence graphs. We define a partial semantics for PRGs, and we describe an implementation of specialization on PRGs that uses C++ classes to mimic the partial dataflow execution of PRGs that represents the specialization operation. We discuss the reconstitution problem and outline our solution.

Chapter 8 presents some conclusions.

Chapter 2

Partial evaluation, dependence graphs, and termination analysis

The main focus of this thesis is the use of dependence graphs in defining BTA algorithms that ensure the termination of partial evaluation. In this chapter we provide background information on the two primary but previously unrelated concepts that are central to this work, partial evaluation and dependence graphs. In Section 2.1 we provide a brief overview of the traditional approach to partial evaluation and the role of binding-time analysis. We review the standard control-flow representation of programs and explain dependence relationships (and the dependence graphs that are formed from these dependences) in Section 2.2. In this thesis we also use dependence graphs to develop a termination analysis for functional programs that can be combined with a congruence-based BTA algorithm to produce an analysis that provides a termination guarantee for all programs. Our analysis improves upon previous termination analyses by using an optimistic approach and more complex termination criteria based on a generalized from of graph reachability referred to as context-free language reachability ("CFL-Reachability"). In Section 2.3 we review previous work on termination analysis, and we also review CFL-Reachability and its role in program analysis.

2.1 Partial evaluation

Partial evaluation is a program specialization operation in which programs with multiple inputs are specialized to take into account known values for some of their inputs. Thus, the result of partially evaluating a program given an "initial division" of its inputs into known or "static" inputs and unknown or "dynamic" inputs is a specialized version of the program, which when run on the dynamic inputs produces exactly the same output as that produced by the original program when run on all of its inputs [JGS93]. More precisely, partial evaluation of program P, given a known subset i_1 of its input $< i_1, i_2 >$, results in a specialized program P', such that the behaviour of P' on i_2 is identical to the behaviour of P on $< i_1, i_2 >$. Partial evaluation is carried out by performing those parts of P's computations that depend only on i_1 and generating code for those computations that depend on i_2. A partial evaluator therefore performs a mixture of execution and code-generation actions. As a result, partial evaluation is often referred to as "mixed computation" [Ers82].

The operation of partial evaluation has been known to be possible ever since Kleene showed that every function can be specialized with respect to a subset of its parameters (Kleene's S-m-n theorem, [Kle52]). The S-m-n theorem established the equivalent functionality of a program P with multiple arguments and the program P' obtained by absorbing an argument of P into the body of P, representing trivial specialization. The technique has been considered as a method to actually *improve* program performance ever since Futamura formulated his equations for compilation and compiler generation using partial evaluation (the so-called "Futamura projections" [Fut71, Ers82]). Since then partial evaluators have been built for a variety of programming languages and paradigms, including flowchart languages [Bul88, GJ89], imperative languages [Jac90, And92], functional languages [Har78, GJ91, Mog92], logic-programming languages [Kom81, BFR90], and object-oriented languages [MS92].

The intent of partial evaluation is that the specialized program will require less time to execute than the original program, as some computations that can be completed using only the known inputs will be absent in the specialized program. In an off-line partial evaluator for imperative programs, the process of partial evaluation is separated into two phases: The first phase is an analysis phase, termed "binding-time analysis" (BTA), in which all of the variables in the program whose values are computable using only the known inputs (static variables) are identified. The second phase is the specialization phase, in which statements or expressions involving only static variables are executed and those involving dynamic variables are emitted, once for each combination of values taken by the static variables in the program. The emitted statements are combined to produce the specialized program. As we mentioned in the introduction, a BTA algorithm may proceed by first identifying program statements as either static or dynamic, and then converting this information into static or dynamic markings for program variables.

Traditional BTA algorithms identify static and dynamic statements in a program by tracing the flow of dynamic values forward from the dynamic inputs of a program through the statements in the program using "flow dependences." Informally, a flow dependence from a statement u to a statement v indicates that a variable that is assigned a value at statement u may be used in the computation at statement v. Alternatively, the values computed at u may affect the values computed at v. An example of two-phase partial evaluation is shown in Figure 4.

		$x\ n\ a$		
(a)	**float** *power* (**float** x, **int** n) {	$[DSS]$	(b)	**float** $power_{n=2}$ (**float** x) {
	float $a = 1.0$;	$[DSS]$		**float** $a = 1.0$;
	while ($n > 0$) {	$[DSD]$		$a = a * x$;
	$n = n - 1$;	$[DSD]$		$a = a * x$;
	$a = a * x$;	$[DSD]$		**return** a;
	}			}
	return a;	$[DSD]$		
	}			

Figure 4: Two-phase partial evaluation of the power function.
The power function from Chapter 1 is shown in (a) above, along with the results of a BTA algorithm that uses flow dependences to identify dynamic behaviour. The figure shows a division in which every program variable is marked as static or dynamic at every program point, given an initial division in which base x is dynamic and exponent n is static. Figure (b) above shows the result of specializing the power function to the value 2 for static parameter n, using the BTA results from (a) above to selectively execute and residuate code. Code that uses only parameter n, which is static, is executed away, while code that uses either x or a, both of which are dynamic, is emitted.

In the context of imperative programs, self-applicable partial evaluators (explained in Section 2.1.1) for flowchart languages have been constructed by Bulyonkov and Ershov [Bul88], and by Gomard and Jones [GJ89]. Meyer has presented a specialization approach for a Pascal-like language that uses dynamic annotations rather than a separate BTA phase in order to obtain more efficient residual programs [Mey91]. However, his analysis loses some precision as a result. Furthermore, he sidesteps the issue of termination of the partial evaluator by assuming that the program terminates for all inputs, which is an overly strong restriction on program behaviour. Baier and Glück, among others, have developed a partial evaluator for Fortran [BGZ94], while Andersen's self-applicable partial evaluator for strictly-conforming Ansi C (*c-mix*, [And92]) handles most of the features of Ansi C, with heap allocated storage being a notable exception. Both of these specializers are based on control-flow representations of the subject program, whereas we are interested in using a dependence graph representation for reasons mentioned earlier. No speedup measurements have been reported for benchmark programs such as the Spec benchmark suite.

2.1.1 Self application and compiler generation

Historically, partial evaluation has been applied to restricted academic languages, most often variants of the functional paradigm. These languages have small, elegant interpreters that are written in the same language ("meta-circular interpreters"); partial evaluation has been used to improve the performance of these interpreters and to automatically generate compilers from them.

The effect of partial evaluation can be represented equationally by the pair of equations below. The first equation is the standard operational semantics of program p with two inputs i_1 and i_2, while the second equation represents the effect of partially evaluating p with respect to input i_1 (*mix* represents the partial evaluator.) The residual program p' produces the same output as p when supplied input i_2. Note that bracketed items, such as $[p]$, denote the meaning function (or semantics) of the item; unbracketed items denote data items. For instance, p denotes the program p treated as a data item (the text or abstract syntax tree of p), whereas $[p]$ denotes the function that p implements.

$$[p]\ i_1\ i_2 \ = \ out$$
$$[mix]\ p\ i_1 \ = \ p' \quad \text{such that } [p']\ i_2 \ = \ out$$

An interpreter can be thought of as a program that takes two inputs, namely a source program and the input to the source program. Partial evaluation of the interpreter (represented by the *first Futamura projection* (1) below) results in a target program that maps the input of *source* to the output of *source*. Therefore, partial evaluation of the interpreter has the effect of a compiler; it "compiles" (or translates) code from one language to another.

$$[int]\ source\ inp \ = \ out$$
$$[mix]\ int\ source \ = \ target \quad \text{such that } [target]\ inp \ = \ out \tag{1}$$

If the partial evaluator (*mix* above) can be applied to itself (*i.e.*, if it is self-applicable), it can be specialized to the given interpreter to produce a mapping from *source* to *target*, which is a compiler (the *second Futamura projection* (2) below.) Finally, the process can be carried a step further by applying the partial evaluator to itself twice; the result is a mapping from interpreters to compilers, which is a compiler generator (the *third Futamura projection* (3) below.)

$$[\![mix]\!]\ mix\ int \;=\; compiler \quad \text{such that } [\![compiler]\!]\ source \;=\; target \tag{2}$$

$$[\![mix]\!]\ mix\ mix \;=\; compilerGen \quad \text{such that } [\![compilerGen]\!]\ int \;=\; compiler \tag{3}$$

2.1.2 Partial evaluation as an optimization

The chief motivation for partial evaluation is as a program optimization: Partially evaluated versions of programs often run faster than the original programs. In general, partial evaluation allows a programmer to write one highly parameterized program for solving several similar problems, yet avoid the potential inefficiency associated with this style by automatically specializing the program with respect to different settings of the parameters to obtain efficient versions of the program for individual applications. Therefore, partial evaluation works well on programs that interpret static data to determine how dynamic data must be manipulated. Large speedups have been reported from partially evaluating circuit simulators [BW90], neural networks [Jac90], computations using networks of processors [RP89], pattern matchers [CD89, Bon90], and other programs that interpret part of their input. However, partial evaluation is not a general program transformation that can change a program's computational method. As a result, partial evaluation usually produces no more than linear speedups [JGS93, pp. 131].

2.1.3 Semantic foundations and correctness

The first formalization of the partial evaluation process was provided by Ershov in [Ers82]. He treated a program P as the source of a set of elementary computation steps C that could be partitioned into two disjoint components C' and C'' by a partition function μ. C' would represent the permissible computations based on only known inputs, and C'' would refer to the computations for which code would be produced (the residual program). In these terms, the binding-time analysis operation amounts to determining the appropriate partition μ, while the specialization operation involves executing the computations in C' and producing code for those in C''.

While this formalization accurately expresses the operations involved in partial evaluation, it does not provide any method to connect the markings produced by BTA algorithms to the underlying program semantics. Jones provided the first such characterization of a BTA algorithm based on the "data division" or S/D markings produced by it. In his terminology, a BTA can create either "uniform" or "pointwise" divisions. A uniform division is a mapping from the set of program variables to $\{S, D\}$; it identifies every variable in the program as either static or dynamic throughout the program. A pointwise division, on the other hand, is a mapping from $\mathit{points} \times \mathit{vars}$ to $\{S, D\}$; it identifies every program variable as either static or dynamic at a particular program point (*i.e.*, before the code at the program point is executed). For imperative languages with updates to variables, pointwise divisions are more useful because they can capture more static behaviour in a program: The same variable can take on static and dynamic values in different regions of code.

Jones characterizes the correctness of a BTA in terms of the "congruence" of the

$$
\begin{array}{ll}
 & x\ n\ a \\
\textbf{float}\ \mathit{power}\ (\ \textbf{float}\ x,\ \textbf{int}\ n\ )\ \{ & [SDS] \\
\quad \textbf{float}\ a = 1.0; & [SDS] \\
\quad \textbf{while}\ (\ n > 0\ )\ \{ & [SDS] \\
\quad\quad n = n - 1; & [SDS] \\
\quad\quad a = a * x; & [SDS] \\
\quad \} & \\
\quad \textbf{return}\ a; & [SDS] \\
\} &
\end{array}
$$

Figure 5: Incorrect results from a congruent BTA.
The power function from Chapter 1 is shown above, along with the results of a congruent BTA that uses flow dependences to identify dynamic behaviour. The BTA is supplied an initial division in which base x is static and exponent n is dynamic. Program variable a is identified as static, even though the data division satisfies the congruence condition.

data division produced by it. He has provided both extensional and intensional definitions for congruence on a small imperative language. By his intensional definition, a data division is congruent if any program point marked S has all of its flow predecessors marked S as well. As we pointed out in the introduction, the drawback with this model is that a BTA that satisfies the congruence condition may identify as static certain variables whose values are built up unboundedly under dynamic control. Thus, in Figure 5, the data division shown for the power function given known base x and unknown exponent n results from a congruent BTA. However, a specializer that uses this division to produce a residual power function will not terminate.

Partial evaluation has historically been applied to restricted academic languages, most often variants of the functional paradigm. The typical users of these tools have been expert programmers, who are familiar with the design of partial evaluators. Therefore, the lack of a termination guarantee has been considered an acceptable shortcoming of congruence-based BTA. However, partial evaluators are now being developed for imperative languages, in particular heavily used languages, for which the goal is to allow an arbitrary user to improve the performance of his or her program by using a partial evaluator as a "black box" that optimizes code in much the same way that optimizing compilers have been used for years. It is therefore important to develop a model for BTA that provides a termination guarantee for partial evaluation. In this thesis, we have developed a new semantic foundation for binding-time analysis that tackles this problem.

2.2 Dependence Graphs

In this section we present an overview of control-flow graphs and their semantics. We then explain the concepts of flow dependences and control dependences. We describe the program dependence graph, which is constructed using flow and control dependences, and we review different approaches to providing these graphs with a semantics that is consistent with the standard operational semantics of programs. Although control-flow graphs can be used to represent imperative programs with a variety of features including arbitrary control flow, we consider a restricted language of programs that includes the following constructs: Assignments, conditionals (**if**), loops (**while**), input (**read**), and output (**write**). The language provides only scalar integer variables.

2.2.1 The control-flow graph

The control-flow graph (CFG) [ASU86] is an intermediate representation for imperative programs that is useful for dataflow analysis and program optimization. It is a directed, rooted graph that has three kinds of vertices: Vertices represent either (i) assignment, input, or output statements, which have a single successor in the CFG; (ii) predicate vertices, which have one *true*-successor and one *false*-successor; or (iii) special **Entry** and **Exit** vertices. The **Entry** vertex is the root of the CFG; it has the **Exit** vertex as its *false* successor, and the first statement in the program as its *true* successor. The **Exit** vertex has no successors. Every vertex is reachable from the **Entry** vertex. Similarly, the **Exit** vertex is reachable from every vertex. The edges in the CFG are labeled with either *true* or *false* (in the case of edges arising from predicate vertices) or have no label (edges that arise from assignment/input/output vertices). The CFG for a program can be constructed in time linear in the size of the program using a syntax-directed translation scheme [Bal93]. As an example, the power function from Chapter 1 and its CFG are shown in Figure 6.

CFGs have a standard denotational semantics in which every program statement is modeled as a state-to-state transformer [Sch86]. The state represents a snapshot of the sequential execution of the program; it consists of a program point, which represents the current point of execution in the program, and a mapping from program variables to values, representing the values currently held by program variables. Assignment vertices update the state, while predicate vertices use the state to make branch decisions. In the operational semantics, execution starts at the **Entry** vertex, which is a predicate vertex that always evaluates to *true*. Execution terminates normally if the

(a)

$x = 2.0;$
read$(n);$
$a = 1.0;$
while $(\ n > 0 \ ) \ \{$
    $n = n - 1;$
    $a = a * x;$
$\}$
write$(a);$

(b)

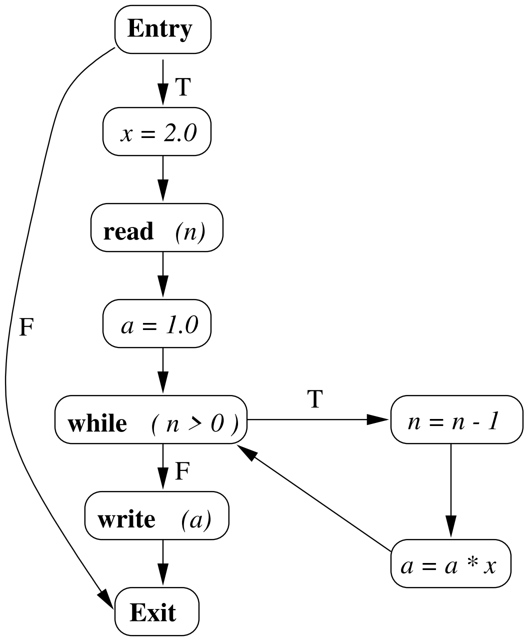


Figure 6: The power program and its control-flow graph.
Figure (a) above above depicts the power function, modified to conform to the language described in Section 2.2. The CFG for this program is shown in figure (b) above.

Exit statement is reached; execution may fail to terminate normally if an infinite loop is encountered, or if an exception occurs.

2.2.2 Flow dependences and control dependences

The control-flow graph represents the flow of control through a program. However, the concept that is often more useful in program analysis and optimization is the flow of data through a program. For this purpose, several forms of data dependence have been defined in the literature. We consider one form of data dependence termed flow dependence, defined below. We borrow all of the definitions of program dependence in this section from [Bal93].

A flow dependence from vertex w to vertex v indicates that a value computed at w may be used at v under some path through the control-flow graph. Reasoning about whether a value computed at w will in fact be used at v is an undecidable problem; a flow dependence from w to v is a conservative approximation which says that under some conditions a value computed at w may be used at v. Flow dependences can be computed in time quadratic in the size of the program (assuming that the used variables sets at CFG vertices are stored in bit vectors with constant-time membership tests) using standard techniques for computing reaching definitions.

Definition 1 Let v and w be vertices in a CFG G. There is a *flow dependence* from vertex w to vertex v iff vertex w assigns to variable x, vertex v uses x, and there is a path in G from w to v that does not include assignments to x (excluding w and v). $\square$

Some flow dependences represent the flow of data across iterations of a loop. Such dependences, which can be traced through the back edge of a loop in the CFG, are referred to as "loop-carried" flow dependences, as opposed to "loop-independent" flow dependences.

Flow dependences directly represent the flow of values through a program. However, it is well known that the values of a variable may be transmitted from program point p_1 to program point p_2 even though there is no path of flow dependence edges from p_1 to p_2 [DD77]. (An example of this will be given shortly.) In such instances, the flow of values is captured indirectly via "control dependences." Intuitively, a vertex v is (directly) *control dependent* on vertex w if the computation at w determines how many times vertex v is executed during the execution of the program. Alternatively, if the predicate vertex w executes and evaluates to a certain value (either *true* or *false*), vertex v is guaranteed to execute if the program terminates normally. If w does not evaluate to this value, v may not execute. Formally, control dependence is defined as below.

Definition 2 Let v and w be vertices in a CFG G. Vertex v *postdominates* vertex w iff $v \neq w$ and v is on every path from w to the **Exit** vertex. Vertex v postdominates the L-branch of predicate vertex w (where L is either *true* or *false*) iff v is the L-successor of w or if v postdominates the L-successor of w. $\square$

If a vertex v postdominates the L-branch of predicate vertex w, then v is guaranteed to execute whenever w evaluates to L, provided the program terminates normally. In addition, if v does not postdominate the predicate itself, whether v executes or not is dependent on w. Hence, v is control dependent on w.

Definition 3 Let v and w be vertices in a CFG G. There is a *(direct) control dependence* (labeled with L) from vertex w to vertex v iff v postdominates the L-branch of

read(x);
$y := 0$;
while $(x > 0)$ {
  $x := x - 1$
  $y := y + 1$
}

Figure 7: An example of a transfer loop.
The program above has the effect of assigning the value of positive integer x to y.

w, and v does not postdominate w. □

Control dependences can represent the transfer of values through a program even though there may be no flow dependences in the program that represent this transfer of values. For instance, in the program shown in Figure 7, the loop in the program has the effect of assigning the value of x to y, even though there are no flow dependences from the assingment of x to the assignment of y. Accounting for this phenomenon in BTA algorithms is the primary result of this thesis.

2.2.3 The program dependence graph

The program dependence graph (PDG) [FOW87] of a program P is a directed graph $G(P) = (V,E)$, where V is a set of vertices that is identical to the set of vertices in the CFG of P, except for the **Exit** vertex, and E is a set of directed dependence edges connecting pairs of vertices. In addition to flow and control dependences, the PDG also contains def-order edges that maintain ordering relationships between multiple definitions of the same variable [HPR88b]. An example of the use of def-order edges is shown in Figure 8. Because the number of flow dependences in a program is at most

(a) **if** P **then** $x := 0$
if Q **then** $x := 1$
$y = x$

(b) **if** Q **then** $x := 1$
if P **then** $x := 0$
$y = x$

Figure 8: An example of the use of def-order edges.
The programs shown in (a) and (b) above have the same set of flow and control dependences, even though they have different semantics. This is because the flow and control dependences do not capture the relative ordering of the two assignments to x. Therefore, a def-order edge is added from the assignment $x := 0$ to the assignment $x := 1$ in the program in (a) above, indicating their relative order. In the program in (b) above, the def-order edge is reversed.

quadratic in the size of the program, while the number of control dependences is linear in the size of the program, the PDG for a program can be constructed from its CFG in time quadratic in the size of the program. The PDG for the power program is shown in Figure 9.

Program dependence graphs have been used as an intermediate program representation in various contexts such as program understanding, maintenance [GL91], debugging [LW86], testing [Bin92, BH93], differencing [Hor90], specialization [RT96], reuse [NEK94], merging [HPR88a], and vectorization and parallelization [KKL+81]. They have the advantage that they make both flow and control dependences explicit in their structure, leading to efficient algorithms for tracing the relationships between program components. In addition, dependence graphs have the advantage that they do not impose a fixed sequential ordering on the statements in a program, making them useful representations for applications such as parallelization. Thus, the same PDG may represent multiple programs with different orderings of the same statements, as long as the dependence relationships in the different programs are identical. This is

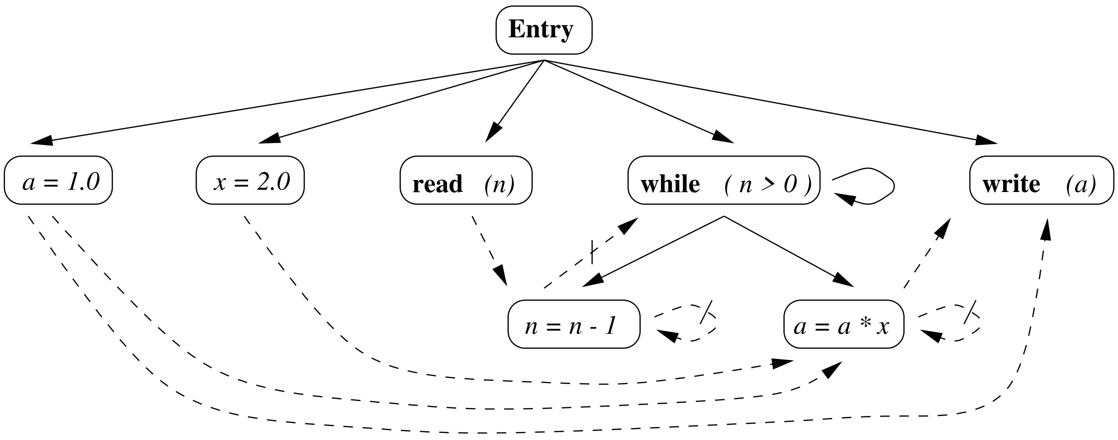


Figure 9: The program dependence graph of the power program.
In the graph shown above, solid arrows represent control dependences, while dashed arrows represent flow dependences. Note the control dependence from the loop predicate to itself; this arises because the loop predicate postdominates its own true successor, but it does not postdominate itself. Loop-carried flow dependence edges are shown with / marks.

highlighted by the programs shown in Figure 10.

The lack of sequential order in PDGs raises two questions, both of which are relevant to the work presented in this thesis. The first issue is whether PDGs have enough information to remain faithful to the semantics of the programs they represent; this is not readily apparent, because the same PDG may represent multiple programs, as in Figure 10. Horwitz *et al.* have shown that if the PDGs of two programs are isomorphic, given the same initial state either both programs diverge or both programs produce the same final state [HPR88a]. Therefore, the PDG is a suitable program representation for reasoning about the behaviour of a program.

Unlike CFGs, PDGs do not have an agreed-upon operational semantics that can be used to reason about the behaviour of the programs they represent. However, one of the

read(x);	**read**(x);
$y = x * x$;	$z = x + x$;
$z = x + x$;	$y = x * x$;

Figure 10: An example of order-independence in PDGs.
Both programs above have the same PDG representation, even though their statements are ordered differently. This is because each program above has the same semantics, modulo normal termination.

goals of this thesis is to develop the first semantic foundation for "correct" binding-time analysis algorithms. Therefore, we use an extension of the PDG known as the program representation graph (PRG). The PRG is an augmented version of the PDG that has a pure data-flow semantics, which we describe in detail in Chapter 3. An alternative approach would be to use the graph rewriting semantics for PDGs defined by Selke in [Sel89], in which computation steps are represented as graph transformations. However, the PRG semantics provide a more natural basis for characterizing static and dynamic behaviour.

The other issue that arises from the lack of sequential order in PDGs is whether it is possible to recover a sequential program from a PDG, if the original program from which the PDG was derived is not available. This problem is referred to as the "reconstitution" problem; Ramalingam [Ram89] has shown that the problem is in general NP-complete. In Chapter 7 of this thesis we present an approach to specialization that involves transforming the dependence graph rather than the control-flow graph. Therefore, in order to produce a residual program, we must carry out reconstitution on the transformed PDG. As explained in Chapter 7, we are able to avoid solving an NP-complete problem by keeping around some extra information through the steps of the transformation process.

2.3 Termination analysis

As we pointed out in the introduction, some authors have worked on an alternative approach to ensuring the termination of partial evaluation, in the context of functional programs. The goal is to conservatively identify as static only those variables that are BSV; a variable is BSV if the set of values it takes on over all possible dynamic inputs is bounded. These authors have designed termination analyses that conservatively identify all loops in the program that are limited to bounded iterations over all inputs. Intuitively, variables that take values from other BSV variables and whose values are built up only in such bounded loops must be BSV. Termination analysis is used to conservatively identify such variables and mark them as static. All other variables are identified as dynamic.

For functional programs without looping constructs and infinite data structures, non-terminating behaviour must result from infinite recursion. Holst has shown that in programs that manipulate S-expression data (*i.e.*, values built up using *cons* operations), it is possible to identify functions that are limited to finite recursion [Hol91]. He identifies parameters that are "in-situ decreasing": An in-situ decreasing parameter of a function f strictly decreases in size on every (recursive) chain of calls from f to f. A function that contains an in-situ decreasing parameter can only call itself a finite number of times before this parameter takes on the value *null* and recursion terminates.

Glenstrup and Jones have devised a second algorithm that identifies in-situ decreasing parameters [GJ96]. They define a structure, called the parameter dependency graph (we refer to this graph as the PG, to avoid confusing it with the PDG described earlier in this chapter), whose edges denote data dependences between function parameters. Edges are labeled to indicate their size-changing effects, as in Figure 11. In this framework, a "size-decreasing path" is a path free of $\uparrow$ edges but containing at least one $\downarrow$ edge. An in-situ decreasing parameter is one for which every path in the PG from the parameter to itself is size decreasing. Such parameters can be identified by solving a simple reachability problem on the PG: A parameter is in-situ decreasing if its vertex in the PG is reachable from itself only via paths that are size-decreasing. The presence of in-situ decreasing parameters is used to classify some static variables as BSV. All variables not marked as BSV (including some previously marked "static" by a congruent BTA) are reclassified as dynamic, and the modified annotations are passed to the specialization phase.

Example 1 The append function is shown in Figure 11. In every recursive call from *append* to itself, parameter l_1 strictly decreases in size. Hence, l_1 is in-situ-decreasing. If *append* is specialized with static l_1 and dynamic l_2, the function can go through at most a bounded number of iterations regardless of the values of the dynamic parameter l_2. Therefore, parameter l_2 is also BSV. Also shown in Figure 11 is the parameter dependency graph for *append.* The only cyclic paths from the node for l_1 to itself are strictly size-decreasing. Hence, l_1 is in-situ-decreasing. $\square$

Termination analyses such as the analysis reviewed above safely handle the phenomenon of static values built up under dynamic control, which is one of the thrusts of this thesis. However, they have the drawback that since they ignore control dependence, they cannot distinguish between static values built up under dynamic control

append(l_1, l_2) :
    case l_1 **of**
        nil : l_2
        cons(a, b) : *cons*(a, *append*(b, l_2))

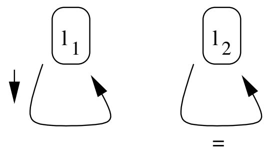


Figure 11: In-situ-decreasing parameters in the append function.
The append function is shown above, along with its parameter dependency graph.

and static-infinite computation. As explained in the introduction, this leads to conservative results for static loops. In addition, such analyses handle only functional languages that manipulate only S-expression data, and that do not have many of the complex features of common imperative languages such as C.

2.3.1 Context-free language reachability

In Chapter 6, we use a generalized form of graph reachability termed "context-free language reachability" (CFL-Reachability) to improve upon the results of the termination analysis described above. CFL-Reachability is defined formally as follows:

Definition 4 (Context-Free-Language Reachability; CFL-Reachability) Let L be a context-free language over alphabet Σ, and let G be a graph whose edges are labeled with members of Σ. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels on the edges on the path. A path in G is an *L-path* if its word is a member of L. The *all-pairs L-path problem* is to determine all pairs of vertices $v_1, v_2 \in V(G)$ such that there exists an *L-path* in G from v_1 to v_2. The *source-target L-path problem* is to determine whether there exists an *L-path* in G from a given source v_1 to a given target v_2. $\square$

Ordinary reachability (transitive closure) is a degenerate case of CFL-reachability: Let all edges of a graph be labeled with the letter e; transitive closure is the all-pairs e^*-path problem. More general instances of CFL-reachability are useful for focusing on certain paths of interest. By choosing an appropriate language L, we are able to enforce certain types of restrictions on when two vertices are considered to be "connected" (beyond just "connected by a sequence of edges", as one has with ordinary reachability).

CFL-reachability problems can be solved using a dynamic-programming algorithm. (The algorithm can be thought of as a generalization of the CYK algorithm for context-free recognition [Kas65, You67].) There is a general result that all CFL-reachability problems can be solved in time cubic in the number of vertices in the graph [Yan90]. Thus, CFL-Reachability affords higher precision than simple reachability, but at the cost of a cubic-time algorithm.

In this chapter, we have reviewed the concepts that are useful in developing termination guarantees for partial evaluation. In the following chapters, we develop analyses that use these concepts to provide such termination guarantees.

Chapter 3

Safe BTA for PRG programs

In this chapter, we describe three binding-time analysis algorithms for an imperative language with restricted features. In Chapter 1 we argued that in order for a BTA algorithm to guarantee termination of partial evaluation in the presence of dynamic control, the algorithm must account for the effect of control dependences on the values taken by program variables. In this chapter, we describe BTA algorithms that use dependence graphs to account for control dependences. First, we describe an intermediate form known as the program representation graph (PRG), a variant of the program dependence graph that makes both flow and control dependences explicit in its structure, and give a semantics for PRGs. Second, we use the semantics of the PRG to characterize several forms of static behaviour and static-infinite computation, and we use these definitions to characterize the safety of BTA algorithms. Third, we use the structure of the PRG to define three BTA algorithms that are simple reachability operations on the PRG. Fourth, we use an abstract interpretation of the PRG semantics to show that each of these algorithms conservatively identifies the program variables that exhibit a given form of static behaviour. Finally, we show how the results of these BTA algorithms can be converted into the markings produced by standard BTA algorithms, which identify program variables, as opposed to program points, as static or dynamic.

The three BTA algorithms described in this chapter identify progressively larger sets of static variables in a program. The first BTA follows all flow and control dependences in a program, while the other two BTA algorithms follow control dependences selectively, ignoring the effect of dynamic control where appropriate.

As we mentioned in the introduction, our goal is to define BTA algorithms that provide a restricted termination guarantee. In particular, we are interested in providing a termination guarantee for partial evaluation in the absence of static-infinite computation. All of the BTA algorithms defined in this chapter have this property.

A complete description of PRGs in available in [RR89]. Other material presented in this chapter may also be found in [DR95].

3.1 The PRG: A representation that formalizes dependences

In this section we describe the program representation graph, an intermediate form in which control dependences are represented explicitly. We describe the syntactic structure of PRGs in Section 3.1.1, and their value-sequence semantics in Section 3.1.2.

3.1.1 Syntactic Structure of PRGs

The PRG is an extension of the program dependence graph that represents programs from a standard imperative language without procedures, in which programs contain the following statements: Assignments, conditionals (**if**), loops (**while**), input (**read**), and output (**write**). The language provides only scalar variables of integral types.

The PRG of a program P is a directed graph $G(P) = (V,E)$, where V is a set of vertices and E is a set of directed edges connecting pairs of vertices. $V(G)$ includes a unique **Entry** vertex representing the start point of the program, and vertices that represent every assignment statement and predicate in the program. $E(G)$ consists of flow and control dependence edges as described in Chapter 2. $V(G)$ and $E(G)$ are augmented by the addition of "ϕ vertices," which are used for two purposes: First, in cases where multiple definitions of a variable reach the same use, gate vertices are introduced in order to "mediate" between the different definition points. These gate vertices are similar to the gate nodes in the SSA form of an imperative program [AWZ88, RWZ88]. Second, ϕ vertices are introduced in certain places to make it easier to define a dataflow-like semantics for PRGs. (The goal is to have a semantics in which vertices are associated with value sequences, and the value sequence produced at every vertex in $V(G)$ can be computed as a function of the value sequences produced at the predecessors of the vertex.)

The complete list of ϕ vertices present in the PRG is as follows:

- ϕ_{if} vertices: For variables defined within an if statement that are used before being defined after the if statement. These vertices are gate nodes that mediate between definitions of the same variable in different branches of an if statement.

- ϕ_{enter} vertices: For variables defined within a loop that are used before being defined within or after the loop. These vertices are gate nodes that mediate between outer and inner definitions of the same variable.

- ϕ_{copy} vertices: For variables used within a loop that are not defined within it. These vertices produce multiple copies of the same value assigned to a variable outside the loop, one copy for every iteration of the loop.

- ϕ_T vertices: For variables used before being defined within the true branch of an if statement. These vertices filter out values that are not used if the predicate evaluates to false.

- ϕ_F vertices: For variables used before being defined within the false branch of an if statement. These vertices filter out values that are not used if the predicate evaluates to true.

- ϕ_{exit} vertices: For variables defined within a loop that are used before being defined after the loop. These vertices filter out values that are not used outside the loop.

- ϕ_{while} vertices: For variables used within a loop that are defined within the loop. These vertices filter out values produced by the last iterations of the loop.

Example 2 The PRG for the power function from Chapter 1, modified in order to conform to the language represented by PRGs, is shown in Figure 12. □

3.1.2 Value-sequence semantics of PRGs

In the formal semantics of the PRG, every vertex is associated with a sequence of values, which represents the values computed at the corresponding program point during program execution. At assignment, input, and ϕ vertices, the value sequences correspond to values taken on at the corresponding program point by the variables defined at the statements, while at predicate vertices the values in the associated sequence are the *true/false* values computed for the expressions at the predicate. Every vertex in the PRG computes its own value sequence using the value sequences of its predecessors,

(a)

$x = 2.0;$
read$(n);$
$a = 1.0;$
while $(\ n > 0\ )\ \{$
    $n = n - 1;$
    $a = a * x;$
$\}$
write$(a);$

(b)

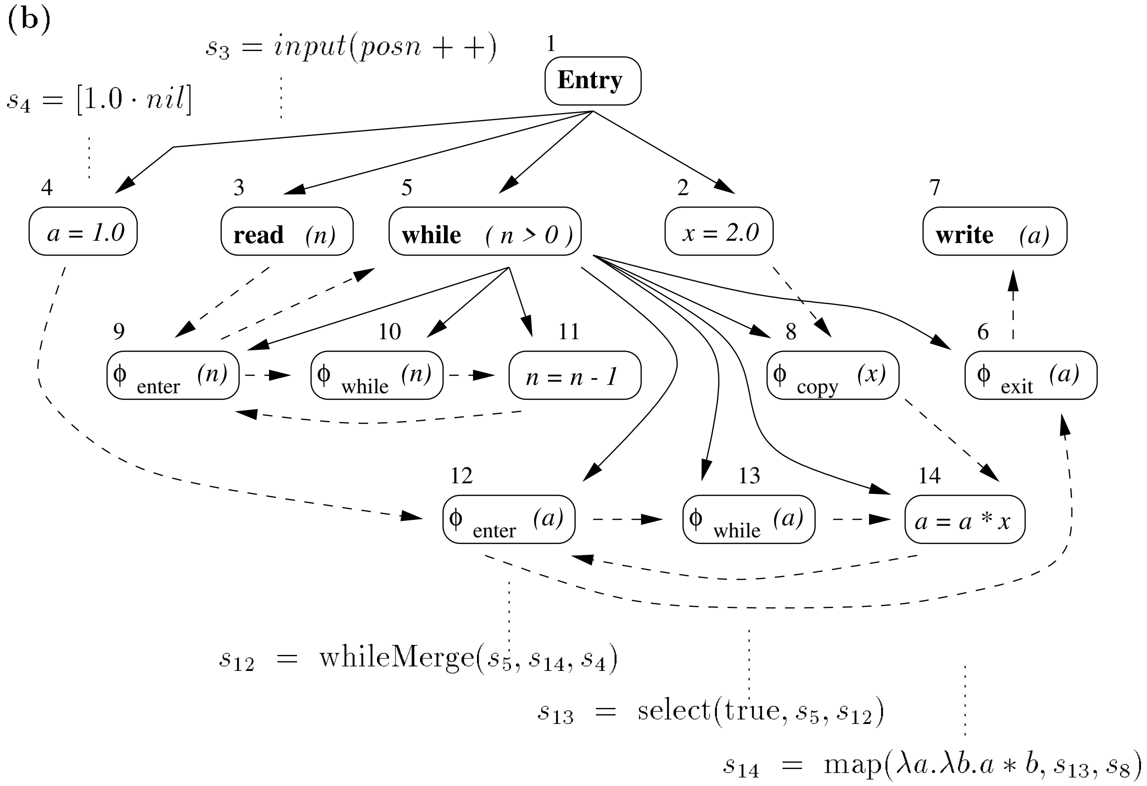


Figure 12: The power program and its program representation graph.
In the PRG in (b) above, solid lines indicate control dependences, while dashed lines indicate flow dependences. ϕ_{enter} vertices are control dependent on both their associated loop predicates and the control parents of their loop predicates. However, the latter control dependences have been omitted from the PRG above as they play no role in the value-sequence semantics. The PRG semantics uses a different set of dependences from the actual flow and control dependences in the PRG in two other ways: Control dependences to non-ϕ vertices are ignored, and loop predicates are related to associated ϕ_{exit} vertices.

and the expression at the vertex itself. Thus, in the PRG semantics, every vertex is modeled as a function that maps a set of input value sequences (the output sequences of its predecessors) to an output value sequence. Since the predecessors of a vertex in the PRG are (almost always) its flow and control dependence predecessors, the PRG semantics summarized above makes the role of flow and control dependences explicit. Complete details of the semantics of PRGs can be found in [RR89]; in this section, we summarize the relevant concepts.

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

$$
\begin{aligned}
&\mathit{Val} \;=\; \mathit{Booleans} \;+\; \mathit{Integers} \;+\; \mathit{Reals} \\
&\mathit{Sequence} \;=\; (\;\{\mathit{nil}\;,\;\mathit{err}\} \;+\; (\;\mathit{Val} \;\times\; \mathit{Sequence}\;)\;)_\perp \\
&\mathit{Stream} \;=\; (\;\mathit{Val} \;+\; (\;\mathit{Val} \;\times\; \mathit{Stream}\;)\;) \\
&\mathit{VertexFunc} \;=\; \mathit{Stream} \;\rightarrow\; \mathit{Vertex} \;\rightarrow\; \mathit{Sequence}
\end{aligned}
$$

Val is a standard domain of values related by the discrete partial order. *Sequence* is the domain of value sequences described in [Sch86, pp. 252-266], members of which are partially ordered as follows:

$$
\begin{array}{lll}
\text{(i)} & \perp \;\sqsubseteq\; s & \forall\; s \in \mathit{Sequence} \\
\text{(ii)} & s \;\sqsubseteq\; s & \forall\; s \in \mathit{Sequence} \\
\text{(iii)} & v \cdot s_1 \;\sqsubseteq\; v \cdot s_2 \text{ iff } s_1 \sqsubseteq s_2 & \forall\; s_1, s_2 \in \mathit{Sequence},\;\; v \in \mathit{Val}
\end{array}
$$

Sequences terminated by *err* indicate computational errors (such as division by zero). The domain of value sequences is a *pointed complete partial ordering*, such that the limit of every chain of elements in *Sequence* is also a member of *Sequence*, and the meet

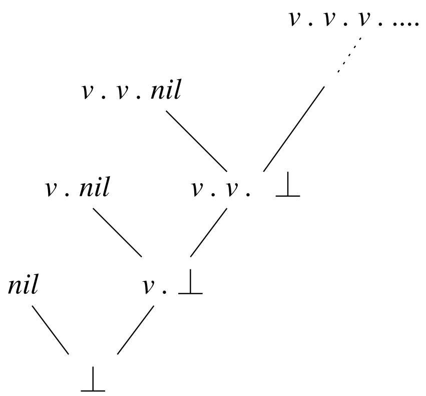


Figure 13: The pointed complete partial ordering of value sequences.
The partial ordering of value sequences is shown above. Sequences are incomparable if they have a position where the value at the position (including *nil*) in the two sequences differs. Therefore, if sequences s_1 and s_2 are incomparable, then s_1' and s_2' are incomparable, where $s_1 \sqsubseteq s_1'$ and $s_2 \sqsubseteq s_2'$.

operator $\sqcap$ is defined for every pair of elements. This domain is depicted graphically in Figure 13.

Stream is the domain of input streams from which read statements obtain values for the variables they define. It 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 14):

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

where $\mathbf{E_G}$ is the conditional expression of the form given in Figure 14 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.

$$
\begin{array}{l}
\mathbf{E_G} \;\doteq\; \lambda i.\lambda v.\lambda f. \\
\quad \begin{array}{lcll}
\mathbf{type}(v) & = & \mathbf{Entry} & \to \;\; \text{true} \cdot nil \\
\mathbf{type}(v) & = & \mathbf{read} & \to \;\; \text{input}(\; i \;) \\
\mathbf{type}(v) & \in & \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\} & \to \\
& & \multicolumn{2}{l}{\begin{cases} \text{replace}(\mathit{controlLabel}(v), \mathit{funcOf}(v), f\ i\ \mathit{parent}(v)) & \text{if } \#\mathit{dataPreds}(v) = 0 \\ \mathit{map}\ \mathit{funcOf}(v)\ (f\ i\ \mathit{dataPred}_1(v), \ldots f\ i\ \mathit{dataPred}_n(v)) & \text{otherwise} \end{cases}} \\
\mathbf{type}(v) & = & \phi_{\mathbf{enter}} & \to \\
& & \multicolumn{2}{l}{\text{whileMerge}(f\ i\ \mathit{whileNode}(v), f\ i\ \mathit{innerDef}(v), f\ i\ \mathit{outerDef}(v))} \\
\mathbf{type}(v) & = & \phi_{\mathbf{exit}} & \to \;\; \text{select}(\text{false}, f\ i\ \mathit{whileNode}(v), f\ i\ \mathit{dataPred}(v)) \\
\mathbf{type}(v) & = & \phi_{\mathbf{while}} & \to \;\; \text{select}(\text{true}, f\ i\ \mathit{whileNode}(v), f\ i\ \mathit{dataPred}(v)) \\
\mathbf{type}(v) & = & \phi_{\mathbf{T}} & \to \;\; \text{select}(\text{true}, f\ i\ \mathit{parent}(v), f\ i\ \mathit{dataPred}(v)) \\
\mathbf{type}(v) & = & \phi_{\mathbf{F}} & \to \;\; \text{select}(\text{false}, f\ i\ \mathit{parent}(v), f\ i\ \mathit{dataPred}(v)) \\
\mathbf{type}(v) & = & \phi_{\mathbf{if}} & \to \;\; \text{merge}(f\ i\ \mathit{ifNode}(v), f\ i\ \mathit{trueDef}(v), f\ i\ \mathit{falseDef}(v))
\end{array}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{replace}: & \text{replace}(x, y, \bot) = \bot & \text{replace}(x, y, nil) = nil \\
& \multicolumn{2}{l}{\text{replace}(x, y, z \cdot tail) = \mathbf{if}\ (x = z)\ \mathbf{then}\ y \cdot \text{replace}(x, y, tail)} \\
& & \mathbf{else}\ \text{replace}(x, y, tail) \\
\mathit{whileMerge}: & \text{whileMerge}(s_1, s_2, \bot) = \bot & \\
& \text{whileMerge}(s_1, s_2, nil) = nil & \\
& \multicolumn{2}{l}{\text{whileMerge}(s_1, s_2, x \cdot tail) = x \cdot \text{merge}(s_1, s_2, tail)} \\
\mathit{merge}: & \text{merge}(\bot, s_1, s_2) = \bot & \text{merge}(nil, s_1, s_2) = nil \\
& \text{merge}(true \cdot tail_1, \bot, s_2) = \bot & \\
& \text{merge}(true \cdot tail_1, nil, s) = nil & \\
& \text{merge}(false \cdot tail_1, s_1, \bot) = \bot & \\
& \text{merge}(false \cdot tail_1, s, nil) = nil & \\
& \multicolumn{2}{l}{\text{merge}(true \cdot tail_1, x \cdot tail_2, s) = x \cdot \text{merge}(tail_1, tail_2, s)} \\
& \multicolumn{2}{l}{\text{merge}(false \cdot tail_1, s, x \cdot tail_2) = x \cdot \text{merge}(tail_1, s, tail_2)} \\
\mathit{select}: & \text{select}(\; x, \bot, z \;) = \bot & \text{select}(x, y, \bot) = \bot \\
& \text{select}(x, nil, nil) = nil & \\
& \multicolumn{2}{l}{\text{select}(x, y \cdot tail_1, z \cdot tail_2) = \mathbf{if}\ (x = y)\ \mathbf{then}\ z \cdot \text{select}(x, tail_1, tail_2)} \\
& & \qquad \mathbf{else}\ \text{select}(x, tail_1, tail_2)
\end{array}
$$

Figure 14: The semantic equations associated with PRG vertices.
Some vertex types are omitted for brevity (see [RR89] for a complete definition of $\mathbf{E_G}$).

Definition 5 The meaning function $\mathbf{M}$ over the domain of PRGs is:

$$\begin{aligned}
\mathbf{M} &: \mathit{PRG} \rightarrow \mathit{VertexFunc} \\
\mathbf{M}\,[\![G]\!] &= \mathbf{fix}\ \mathbf{F} \text{ where} \qquad \mathbf{F} : \mathit{VertexFunc} \rightarrow \mathit{VertexFunc} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{F} = \lambda f.\lambda i.\lambda v.\ \mathbf{E_G}(i, v, f)
\end{aligned}$$

The meaning function for PRGs associates every vertex with a value sequence that is either an infinite sequence or a sequence terminated by *nil* or *err*.

Example 3 Figure 12 shows the power program and the semantic equations at selected vertices in its PRG. In particular:

- At vertex 3, 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 4, the function **replace** uses the sequence from the control predecessor (vertex 1) to produce the sequence containing the single value 1.0:

  $$f\ i\ v_4 = \text{replace(true, 1.0, } f\ i\ v_1\ \text{)}$$

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

- At vertex 12, the function **whileMerge** produces a value sequence s_{12} for variable a by merging the sequences for a from vertex 4 and vertex 14 (sequences s_4 and s_{14}, respectively). It uses the Boolean value sequence from its control-dependence predecessor (s_5) to determine how the two sequences for a should be merged:

$$f\ i\ v_{12}\ =\ \text{whileMerge}(\ f\ i\ v_5,\ f\ i\ v_{14},\ f\ i\ v_4\ )$$

- At vertex 13, the function **select** filters out values from the value sequence at the ϕ_{enter} vertex (vertex 12) that correspond to instances when the loop predicate evaluates to *false*:

$$f\ i\ v_{13}\ =\ \text{select}(\ \text{true},\ f\ i\ v_5, f\ i\ v_{12}\ )$$

- The functions at non-ϕ vertices (for instance, vertices 5, 11, and 14) are **map** functions. Thus $\mathbf{M}\ [\![G]\!]$ associates vertex 14 with output sequences as follows:

$$\mathbf{M}\ [\![G]\!]\ [1 \cdot nil]\ v_{14}\ =\ [2.0 \cdot nil] \qquad \mathbf{M}\ [\![G]\!]\ [2 \cdot nil]\ v_{14}\ =\ [2.0 \cdot 4.0 \cdot nil]$$

It should be pointed out that the PRG semantics is non-standard in one respect: It is *more defined* than the standard semantics in the case of inputs on which the program does not terminate. (Roughly, the reason is that value sequences transmitted along dependence edges can bypass non-terminating loops.) For inputs on which the program does not terminate, 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. For inputs on which the program terminates normally, it has been shown that the two sequences are identical [RR89].

3.2 Semantic characterizations of static and finite behaviour

In this section we use the PRG semantics to define three increasingly general forms of static behaviour that account for dynamic control. We then define the orthogonal concept of bounded static variation.

As noted in the introduction, the basis for this thesis is the observation that the usual notion of a "congruent division" is an unsatisfactory correctness criterion for BTA algorithms. For instance, a BTA algorithm that identifies variable a in the power program from Figure 12 as static would be congruent. As we pointed out in Chapter 1, the problem is that the entire sequence (or set) of values taken by n cannot be determined without the dynamic input. This leads directly to the definition of "strongly static" behaviour in terms of the value-sequence semantics of PRGs: A vertex is strongly static if the sequence produced at the vertex is independent of the input.

Definition 6 Vertex v in PRG G is *strongly (semantically) static* iff the sequences in $\{\mathbf{M}\ [\![G]\!]\ i\ v \mid i \in \mathit{Stream}\}$ form a chain in Sequence. □

The definition above says that a vertex is strongly static provided its behaviour (its value-sequence) is unaffected by changes in the run-time input. Note that we require the sequences at a strongly static vertex to form a chain, as opposed to requiring the same sequence over all inputs. This is because PRG vertices may produce $\perp$-terminated (incomplete) sequences in the presence of infinite loops. Intuitively, if the sequences produced at v on two different inputs i_1 and i_2 are incomparable, then the value of the input determines the values produced at v. For instance, for vertex v_{14} in the program

$i = -1;$
while $(i < 0)$ {
  $v : i = i - 1;$
}

Figure 15: An example of "infinite" strongly static behaviour.
In the program above, vertex v satisfies the strongly static property, even though the sequence produced at v contains infinitely many different values. Vertex v is in fact static-infinite.

from Figure 12, $\mathbf{M}\ [\![G]\!]\ [1 \cdot nil]\ v_{14}$ and $\mathbf{M}\ [\![G]\!]\ [2 \cdot nil]\ v_{14}$ are incomparable. Therefore, v_{14} is not strongly static.

The strongly static property defined above has the following advantage: If a vertex represents a static computation under dynamic control, it is not strongly static. Hence, if a BTA algorithm can be designed so that it conservatively identifies vertices that satisfy the strongly static property (*i.e*, if a vertex v is marked "S", then v is strongly static), then we will be able to eliminate one of the reasons why a partial evaluator may not terminate, namely because of static computations under dynamic control.

Note that the strongly static property, by itself, does not guarantee that a vertex will produce only finitely many different values. For instance, vertex v in the program from Figure 15 is strongly static. As discussed in the introduction, this may cause a partial evaluator to fail to terminate. Vertices such as vertex v in Figure 15 are precisely the "static-infinite" vertices.

The strongly static property has the following drawback: Some vertices that could

$$
\begin{array}{l}
\textbf{read}(\mathit{dyn});\\
\textbf{while}\ (\mathit{dyn} \neq 0)\ \{\\
\quad \mathit{count} = 0;\\
\quad \textbf{while}\ (\mathit{count} < 3)\ \{\\
\quad\quad v : \mathit{count} = \mathit{count} + 1;\\
\quad \}\\
\quad \mathit{dyn} = \mathit{dyn} - 1;\\
\}
\end{array}
$$

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

Figure 16: An example of weakly static behaviour.

be treated as static without compromising the requirement of termination under dynamic control will also fail the property. For instance, consider the behaviour of program point v in the program from Figure 16.

In this example program, the computation represented by the inner counting loop can be replaced by a single assignment, using a partial evaluator that treats variable *count* as static. Although the program may appear contrived, this kind of nested code is found frequently in specializable code. Unfortunately, vertex v fails the strongly static property, because its value-sequence depends on the dynamic input. The reason for this is as follows: The counting loop is nested within a dynamic predicate. Since control dependences reflect the nesting structure of a program, the control dependence from the dynamic outer loop predicate to the inner loop manifests itself in the value sequences of the vertices in the inner loop.

However, the key observation is that for every dynamic input, the value-sequence at v is formed from zero or more repetitions of the same base sequence (in this case,

$[1 \cdot 2 \cdot 3]$). Therefore, vertex v can be treated as static. 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 the example program, the control dependence from the outer loop predicate to the constant assignment feeding the inner loop represents the effect of run-time data on how many times v executes; under our generalized notion of staticness, this dependence is irrelevant. We term this generalized form of staticness "weakly static" behaviour.

Definition 7 Vertex v in PRG G is *weakly (semantically) static* iff at least one of the following holds:

(a) $\exists s \in Val^{*}$ s.t. $\forall i \in Stream$, $\mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{s^{n} \cdot nil | n \in \mathcal{N}\} \cup \{s^{\infty}\})$

or

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

where **DC** is the downwards-closure operator on *Sequence*. □

We call sets of the form $\{nil\} \cup \{s^{n} \cdot nil | n \in \mathcal{N}\} \cup \{s^{\infty}\}$ from property (a) above *rational repetitions*. As in the case of strongly static behaviour, we generalize rational repetitions to their downwards closures, in order to account for $\perp$-terminated sequences (we refer to subsets of these closures as *approximate 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 of the strongly static vertices.

Example 4 Vertex v from the program in Figure 16 is weakly static, because the sequences produced at v over all possible inputs are rational repetitions. In contrast, vertex v_{14} in the power program from Figure 12 does not satisfy the weakly static

```
read(x1);
if (x1 ≠ 0) {
   x2 := 0;
} else {
   x2 := 10;
};
v : x3 := x2;
```

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

Figure 17: An example of statically varying behaviour.

property in Definition 7 above, because there is no common base sequence from which the sequences in $\{[2.0 \cdot nil],\ [2.0 \cdot 4.0 \cdot nil],\ [2.0 \cdot 4.0 \cdot 8.0 \cdot nil] \ldots\}$ are formed. $\square$

Note that both Definition 6 and Definition 7 permit vertices that produce infinitely many different values to be classified as static. In the next section we show that vertices which satisfy either of these definitions and produce infinitely many different values are precisely the vertices that exhibit "static-infinite" behaviour. Both of these definitions characterize staticness purely in terms of dependence on dynamic input and not in terms of any finiteness condition. We define a third, more general, form of static behaviour that does involve boundedness conditions, namely "statically varying behaviour." The motivation for this notion of staticness comes from, for example, the behaviour at program point v in the program from Figure 17.

Under both Definition 6 and Definition 7, vertex v in the program from Figure 17 is not static. This is because the dynamic input determines the actual values in the value-sequence at v. In particular, property (a) from Definition 7 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 $\{0, 10\}$. Therefore, the values taken by x_3 at v can be enumerated during partial evaluation without resulting in non-termination. Hence we would like to treat such variables as static. We capture this behaviour by generalizing weak staticness to statically varying behaviour.

Definition 8 Vertex v in PRG G is *statically (semantically) varying* iff at least one of the following holds:

(a) $\exists B \subset Val, |B|$ finite, s. t. $\forall i \in Stream$,

$$\mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{v_1 \cdot \ldots \cdot v_k \cdot nil | v_1, \ldots, v_k \in B\} \cup B^\omega)$$

or

(b) v is weakly static.

where **DC** is the downwards-closure operator on *Sequence*. □

We refer to sets of the form $\{nil\} \cup \{v_1 \cdot \ldots \cdot v_k \cdot nil | v_1, \ldots, v_k \in B\} \cup B^\omega$ from the properties above as *static 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 statically varying behaviour generalizes weakly static behaviour, in the sense that every weakly static vertex is also statically varying.

Definitions 6-8, 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. Therefore, a BTA algorithm that conservatively approximates any of the three definitions above may still result in partial evaluation that does not terminate under certain conditions, in particular in the presence of static-infinite computation. In Chapter 1 we pointed out that a partial evaluator is guaranteed to terminate under all conditions if it treats only variables that are bounded static varying (BSV) as static. A variable is BSV if the set of distinct values taken on by it over all possible dynamic inputs is finite [JGS93, pp. 300].

Definition 9 Vertex v in PRG G is (*semantically*) *bounded static varying* iff:

$\exists B \subset Val, |B|$ finite, s. t. $\forall i \in Stream$,

$$\mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{v_1 \cdot \ldots \cdot v_k \cdot nil | v_1, \ldots, v_k \in B\} \cup B^{\omega})$$

where **DC** is the downwards-closure operator on *Sequence*.

Vertex v is (*semantically*) *non-BSV* iff it is not semantically BSV. □

Definition 9 above differs from Definition 8 by dropping property (b), thereby ensuring that only a finite set of different values is produced.

3.3 Notions of safety for binding-time analyses

In the previous section we presented semantic definitions for static and BSV behaviour in terms of the PRG semantics; we now show how these definitions can be used to establish a framework for "safe" binding-time analysis.

3.3.1 Specializable Vertices and Static-Infinite Computation

The key concept is as follows: If the BTA algorithm is conservative with respect to any of our definitions of staticness, partial evaluation is guaranteed to terminate in the absence of static-infinite computation. In addition, if the BTA is conservative with respect to our definition of BSV behaviour, partial evaluation is guaranteed to terminate under all conditions. We argue that the former guarantee is more appropriate for partial evaluation than the latter.

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

$$Static(G) = \{v \in V(G) | v \text{ is semantically static}\}$$
$$BSV(G) = \{v \in V(G) | v \text{ is semantically BSV}\}$$
$$Specializable(G) = Static(G) \cap BSV(G)$$

Vertices that belong to $Static(G)$ have the property that the set of values in their value-sequences is independent of the dynamic input. (Note that we are really defining three classes of specializable vertices, according to whether $Static(G)$ refers to strongly static, weakly static, or statically varying vertices.) Some of these vertices are also BSV; a specializer can perform the computation at these vertices, termed *specializable vertices*, without running the risk of falling into a non-terminating computation.

We can now define the semantic notion of static-infinite computation as follows: Any vertex in *Static(G)* takes on values that are independent of dynamic input; in particular, the values in the value-sequence at the vertex are not built up under dynamic control. If such a vertex is not BSV, it must be static-infinite.

Definition 10 PRG G is *static-infinite* iff $Static(G) - BSV(G) \neq \emptyset$. □

In contrast with the definition of Jones et al. in [JGS93, pp. 118], which is an informal description of static-infinite computation as an "infinite static loop," that is, "a loop not involving any dynamic tests," Definition 10 is a semantic definition that can be used to characterize the safety properties of a binding-time analysis algorithm.

3.3.2 BTA characterizations

With a formal notion of static-infinite computation in hand, we can now define the notions of safety and conditional safety for binding-time analyses.

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:

$$S(G) = \{v \in V(G) | bta\ G\ v\ =\ S\}$$
$$D(G) = V(G) - S(G)$$

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 11 Binding-time analysis *bta* is *safe* on a set $Gset$ of PRGs iff $\forall\ G \in Gset$, $S(G) \subseteq \mathit{Specializable}(G)$. □

A safe bta results in two-phase specialization that is guaranteed to terminate for all programs, including those that contain static-infinite computations. We are interested in BTAs that provide a weaker guarantee of termination, as follows:

Definition 12 Binding-time analysis *bta* is *conditionally safe* on a set $Gset$ of PRGs iff $\forall\ G \in Gset$, $S(G) \subseteq \mathit{Static}(G)$. □

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":

Corollary 13 For a set of PRGs *Gset* that contains no static-infinite PRGs, *bta* is conditionally safe on *Gset* $\Leftrightarrow$ *bta* is safe on *Gset*. $\square$

Thus, a conditionally safe BTA guarantees termination in the absence of static-infinite computation.

3.3.3 Termination versus computational completeness

As pointed out by Jones in [Jon95], specialization involves a basic tradeoff between "totality" and "computational completeness." A specializer can be computationally complete in that it executes every static computation in its subject programs, but then on a program that contains static-infinite computations, the specializer must diverge without producing a residual program, thereby violating the totality condition. On the other hand, a specializer can be total (*i.e.*, provide a termination guarantee) by attempting to execute static computations in a conservative manner, but it will lack computational completeness as a result.

In the context of the definitions provided in the previous section, both safe and conditionally safe BTAs favour totality over computational completeness, when compared with congruence-based BTAs. However, our experiments show that conditionally safe BTAs can provide totality via the termination guarantee without unduly compromising the ability of the partial evaluator to identify and execute away static code.

3.4 BTA algorithms for single-procedure programs

The definitions of static behaviour in Section 3.2 may seem somewhat arbitrary and unintuitive. However, they have the advantage that even though they are semantic properties, they are based on the PRG semantics, in which flow and control dependences are explicit. Therefore, it is possible to design BTA algorithms that are simple reachability operations on the PRG (or, that use flow and control dependence information) such that these algorithms can be proven conditionally safe.

In this section we define three such BTA algorithms as abstract interpretations of the PRG semantics, each of which conservatively identifies a form of static behaviour from the definitions in Section 3.2. The first algorithm follows control dependences blindly, and identifies only strongly static vertices as S; the second follows control dependences selectively, and thus identifies some weakly static vertices as S as well. The third BTA identifies some statically varying vertices as S by ignoring control dependences to vertices that have multiple static data dependence predecessors. We use the framework developed in the previous sections to prove the conditional safety of these analyses. We also show how the results of these BTA algorithms can be converted into the markings produced by standard BTA algorithms, which identify program variables as static or dynamic. Finally, we critique these algorithms and point out their benefits and drawbacks when compared with other BTA algorithms. All three algorithms can be viewed operationally as variants of operations for program slicing [Wei84], and can therefore be performed in time linear in the size of the PRG.

3.4.1 The Strong-Staticness BTA

The goal of the Strong-Staticness BTA is to identify a subset of all the strongly-static vertices in the PRG of a program. The idea is to follow all flow and control dependence edges from the set of **read** vertices in the PRG, marking with D all vertices that are encountered along the way. This operation is identical to a forward program slice [HRB90] from the set of **read** vertices in the PRG. Vertices that are not in this forward slice are marked with S.

Vertices that are not in the forward slice of any **read** vertex (i.e. vertices marked S) are guaranteed to have no **read** vertex in their backward slices. Intuitively, this means that changes in the behaviour of any **read** vertex (in particular, the input values) will not affect these vertices. In terms of the PRG semantics, this means that the value sequences produced at the S vertices are independent of the input stream, which is consistent with the characterization of strongly static behaviour in Definition 6. Thus, the algorithm identifies a subset of the strongly-static vertices in a PRG as static.

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 3.1.2. This interpretation is defined by the following recursive equation (see Figure 18) which resembles the PRG equation from Section 3.1.2:

$$\begin{aligned} \mathit{VertexAbs} &= \mathit{Vertex} \rightarrow \{S, D\} \text{ with } S \sqsubset D \\ &f_a : \mathit{VertexAbs};\ f_a = \lambda v.\mathbf{E^a_G}(v, f_a) \end{aligned}$$

All the abs_* functions in $\mathbf{E^a_G}$ are continuous, and propagate the value D if any of their inputs has the value D. The abstract semantics is defined as the least $f_a \in \mathit{VertexAbs}$ that satisfies the equation above:

$$\mathbf{M_a} : \mathit{PRG} \rightarrow \mathit{VertexAbs}$$

$$
\begin{array}{l}
\mathbf{E^a_G} \ \doteq \ \lambda v.\lambda f_a. \\
\quad \mathbf{type}(v) = \mathbf{Entry} \ \rightarrow \ S \\
\quad \mathbf{type}(v) = \mathbf{read} \ \rightarrow \ D \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\} \ \rightarrow \\
\qquad \begin{cases} \text{abs\_replace}(\ f_a \ \mathit{parent}(v)\ ) & \\ & \text{if } \#\mathit{dataPreds}(v) = 0 \\ \mathit{abs\_map}(\ f_a \ \mathit{dataPred_1}(v), \ldots f_a \ \mathit{dataPred_n}(v)) & \\ & \text{otherwise} \end{cases} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{enter}} \ \rightarrow \\
\qquad \text{abs\_whileMerge}(\ f_a \ \mathit{whileNode}(v),\ f_a \ \mathit{innerDef}(v),\ f_a \ \mathit{outerDef}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{exit}} \ \rightarrow \ \text{abs\_select}(\ f_a \ \mathit{whileNode}(v),\ f_a \ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{while}} \ \rightarrow \ \text{abs\_select}(\ f_a \ \mathit{whileNode}(v),\ f_a \ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{T}} \ \rightarrow \ \text{abs\_select}(\ f_a \ \mathit{parent}(v),\ f_a \ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{F}} \ \rightarrow \ \text{abs\_select}(\ f_a \ \mathit{parent}(v),\ f_a \ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{if}} \ \rightarrow \ \text{abs\_merge}(\ f_a \ \mathit{ifNode}(v),\ f_a \ \mathit{trueDef}(v),\ f_a \ \mathit{falseDef}(v)\ )
\end{array}
$$

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

$\text{abs\_replace}(\ a\ ) = a$
$\text{abs\_select}(\ a,\ b\ ) = a \sqcup b$
$\text{abs\_merge}(\ a,\ b,\ c\ ) = a \sqcup b \sqcup c$
$\text{abs\_whileMerge}(\ a,\ b,\ c\ ) = a \sqcup b \sqcup c$
$\text{abs\_map}(\ a_1,\ \ldots,\ a_n\ ) = a_1 \sqcup \ldots \sqcup a_n$

Figure 18: The abstract equations representing the Strong-Staticness BTA.

$$\mathbf{M_a}[\![G]\!] = \mathbf{fix}\ \mathbf{F_a} \text{ where } \qquad \begin{aligned} &\mathbf{F_a} : \mathit{VertexAbs} \rightarrow \mathit{VertexAbs} \\ &\mathbf{F_a} = \lambda f_a.\lambda v.\mathbf{E^a_G}(v, f_a) \end{aligned}$$

$\mathbf{F_a}$ 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 19. *abs* takes an element of type *VertexFunc* from the concrete domain, determines whether that function 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 14 *abs* is continuous on *VertexFunc*.

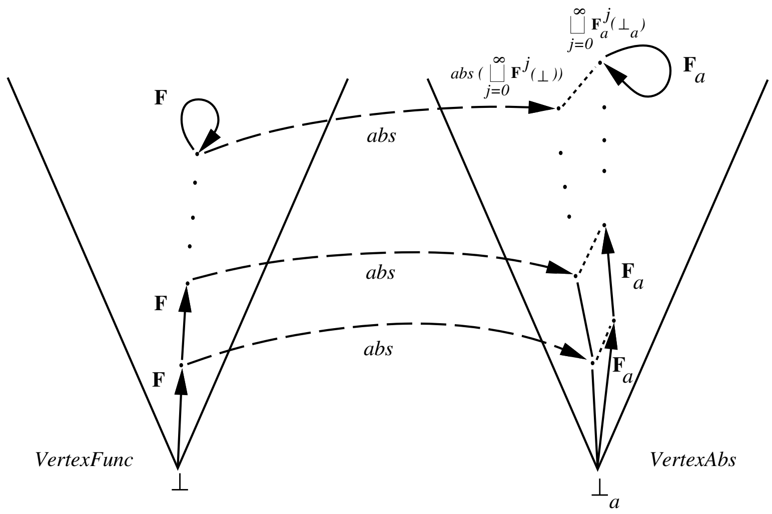
Proof. We prove the lemma in two parts:

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

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

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

else $abs(c)\ v = D$. From Definition 6, $\exists\ i_1,\ i_2 \in \mathit{Stream}$ 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 is a property of



$$abs \ : \ VertexFunc \ \rightarrow \ VertexAbs$$

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

Figure 19: *abs*, the abstraction function used to compare the results of $\mathbf{F}$ and $\mathbf{F_a}$.

Sequence that $c'\ i_1\ v$ and $c'\ i_2\ v$ are incomparable. Hence, $abs(c')\ v\ =\ D$.

(b) for any chain $C = c_1, \ldots c_j, \ldots$ in *VertexFunc*, $abs(\bigsqcup_{j=1}^{\infty} c_j)\ v = \bigsqcup_{j=1}^{\infty} abs(c_j)\ v$:

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

Hence, $\bigsqcup_{j=1}^{\infty} abs(c_j)\ v = S$ **else** $abs(\bigsqcup_{j=1}^{\infty} c_j)\ v = D$. Then $\exists\ i_1,\ i_2\ \in\ Stream$ s.t. $(\bigsqcup_{j=1}^{\infty} c_j)\ i_1\ v$ and $(\bigsqcup_{j=1}^{\infty} c_j)\ i_2\ v$ are incomparable. Let $k\ \in\ \mathcal{N}$ be the first position at which these sequences have different non-$\perp$ values. Then:

(i) $\exists\ m_1\ \in\ \mathcal{N}$ s.t. $|c_j\ i_1\ v|\ \geq\ k$ for all $j\ \geq\ m_1$

(i) $\exists\ m_2\ \in\ \mathcal{N}$ 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 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 $\bigsqcup_{j=1}^{\infty} abs(c_j)\ v\ =\ D$. $\square$

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

Lemma 15 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:

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

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

Our next task is to show that, at every step, the vertex function produced by $\mathbf{F}$ abstracts to a lower value than that produced by $\mathbf{F_a}$ at the corresponding step (see Figure 19).

Lemma 16 $abs(\mathbf{F}^j \perp) \;\sqsubseteq\; \mathbf{F_a}^j \perp_a \;\forall\, j \in \mathcal{N}$

Proof. We prove the lemma by induction on j:

Base case ($j = 0$): $abs(\perp)\ v \;=\; S \;\sqsubseteq\; \perp_a v$

Induction step: Assume $abs(\mathbf{F}^j \perp) \;\sqsubseteq\; \mathbf{F}^j_{\mathbf{a}} \perp_a$

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

else $abs(\mathbf{F}^{j+1} \perp)\ v \;=\; D$. From Lemma 15, either:

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

(ii) $\exists\, u \in preds(v)$ s.t. $abs(\mathbf{F}^j \perp)\ u \;=\; D$. Hence by assumption,

$\mathbf{F}^j_{\mathbf{a}} \perp_a u \;=\; D$. Then $\mathbf{F}^{j+1}_{\mathbf{a}} \perp_a v \;=\; D$ by definition of $\mathbf{F_a}$. □

Phrased differently, Lemma 16 says that at every step, if the value produced by $\mathbf{F_a}$ at a vertex is S then $\mathbf{F}$ 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 $\mathbf{F}$, demonstrates that the Strong-Staticness BTA is conditionally safe for all PRGs:

Theorem 1 For every vertex v in PRG G, $\mathbf{M_a}\,[G]\ v \;=\; S \;\Rightarrow\; v \in Static(G)$.

Proof. From Lemma 16, for all j, $abs(\mathbf{F}^j \perp)\ v \sqsubseteq \mathbf{F}^j_{\mathbf{a}} \perp_a v$.
Hence, $\bigsqcup_{j=0}^{\infty} abs(\mathbf{F}^j \perp)\ v \sqsubseteq \bigsqcup_{j=0}^{\infty} \mathbf{F}^j_{\mathbf{a}} \perp_a v$. Because abs is continuous (Lemma 14), it follows that: $abs(\bigsqcup_{j=0}^{\infty} \mathbf{F}^j \perp)\ v \sqsubseteq \bigsqcup_{j=0}^{\infty} \mathbf{F}^j_{\mathbf{a}} \perp_a v$. Or, $abs(\mathbf{M}\,[\![G]\!])\ v \sqsubseteq \mathbf{M_a}\,[\![G]\!]\ v$. In particular, if $\mathbf{M_a}[\![G]\!]v = S$, then $abs(\mathbf{M}\,[\![G]\!])\ v \;=\; S$. Hence $\{\mathbf{M}\,[\![G]\!]\ i\ v \mid i \in Stream\}$ is a chain in *Sequence*. 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 flow and control dependences, produces a conditionally safe BTA. 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. In the worst-case, the number of edges in the PRG of a program may be quadratic in the size of the program. Hence, the Strong-Staticness BTA has worst-case complexity quadratic in the size of the program (plus the time to construct the PRG).

3.4.2 The Weak-Staticness BTA

As discussed in Section 3.2, the strongly-static property is rather restrictive because it does not characterize static code nested within dynamic predicates as strongly static. In operational terms, the Strong-Staticness BTA is restrictive because it always considers control dependence edges to transmit dynamic behaviour. Therefore, we define the Weak-Staticness BTA, an analysis that identifies a subset of all the weakly-static vertices in the PRG of a program. The Weak-Staticness BTA is similar to the Strong-Staticness BTA, differing only in how constant vertices are treated.

Consider constant assignments. The sequence produced at a constant assignment vertex is given by (Figure 14):

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

where $\mathit{funcOf}(v)$ is the constant expression and $\mathit{parent}(v)$ is the control predecessor. The Strong-Staticness BTA would mark v with a D if $\mathit{parent}(v)$ has a D value, since $f\ i\ \mathit{parent}(v)$ determines the length of $f\ i\ v$ (and thus determines the sequence at v). However, regardless of the behaviour at $\mathit{parent}(v)$, the sequence at v will be a rational repetition in which $\mathit{funcOf}(v)$ is repeated a variable number of times over varying inputs. Hence, in the Weak-Staticness BTA an S value is produced at constant

```
read(dyn);
while (dyn ≠ 0) {
   count = 3;
   dyn = dyn − 1;
}
```

Figure 20: An example of specialization using the Weak-Staticness BTA. The program shown above is the residual program obtained by specializing the program from Figure 16, using the markings produced by the Weak-Staticness BTA.

assignments, in all cases. Formally, the function *abs_replace* in the abstract semantics is re-defined as follows:

$$\text{abs\_replace}(\ a\ ) = S$$

Example 5 In the program from Figure 16, the constant assignment $count = 0$ 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 residual program shown in Figure 20. The initialization of *count* 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 the Weak-Staticness BTA is conditionally safe mimics the one for the Strong-Staticness BTA, with two modifications:

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

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

(b) Lemma 15 is modified to account for weakly static behaviour, as follows:

Lemma 17 For any PRG vertex v that is not a **read** vertex, $\{\mathbf{F}^{j+1} \perp i\ v \mid i \in \mathit{Stream}\}$ is an approximate rational repetition if:

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

(b) *funcOf(v)* is a constant value. □

The functions at PRG vertices are all structured so that when predecessor sequences u_1, u_2, ..., u_k at vertex v are all rational repetitions, the output sequence at v is a rational repetition, with a base repeating sequence that is at most as long as the least common multiple of the lengths of the base repeating sequences in u_1, u_2, ..., u_k. This is the basis for the semantic justification of conditional safety for this analysis.

3.4.3 The Static-Variation BTA

The weakly-static property has the drawback that a vertex that chooses between two static data-predecessors based on a dynamic conditional is treated as dynamic. This is undesirable in situations where static computations nested beneath different branches of a dynamic predicate are used in later computations, as in the program from Figure 17 in Section 3.2. Therefore, we define the Static-Variation BTA, an analysis that identifies a subset of all the statically-varying vertices in the PRG of a program. The Static-Variation BTA is similar to the Weak-Staticness BTA, differing only at ϕ_{if} merge vertices.

Consider ϕ_{if} vertices. The sequence produced at a ϕ_{if} vertex is given by (Figure 14):

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

where *ifNode*(v) is the corresponding predicate and *trueDef*(v) (*falseDef*(v)) is the data predecessor within the *true* (*false*) branch of the conditional statement. Consider a case in which both data predecessors of v are marked S: the Weak-Staticness BTA would mark v with a D value if *ifNode*(v) has a D value, since $f\ i$ *ifNode*(v) determines the actual values in $f\ i\ v$. In the Static-Variation BTA, an S value is produced in this case, the idea being that if the data predecessors produce bounded values, the ϕ_{if} vertex produces bounded values as well, as it produces only values that are produced at either of its data predecessors.

Example 6 In program **P** in Figure 21, the assignment $x_3 = x_2$ is marked S by the Static-Variation BTA. As a result, the predicate following it is marked S, and specialization produces the program $\mathbf{P}'$ in Figure 21. □

The Static-Variation BTA is plausible because of the following observation: The functions at PRG vertices other than ϕ_{enter} vertices all have the property that when predecessor sequences u_1, u_2, ..., u_k at vertex v are all static variations, the output sequence at v is a static variation whose base set has at most as many values as the product of the number of values in the base sets of u_1, u_2, ..., u_k.

This BTA algorithm has the advantage that it captures a general form of static behaviour. Like the Strong-Staticness BTA and Weak-Staticness BTA, however, it has the drawback that it is applicable only to a small subset language of imperative programs. In particular, it is unclear how the PRG semantics or these algorithms can be extended to handle programs with arbitrary control-flow and aliased variables.

P :
```
read(x1);
if (x1 ≠ 0) {
   x2 = 0;
} else {
   x2 = 10;
};
x3 = x2;
if x3 < 10 {
   x4 = 0;
} else {
   read(x4);
}
```

P′ :
```
read(x1);
if (x1 ≠ 0) {
   x4 = 0;
} else {
   read(x4);
}
```

Figure 21: Specialization using the Static-Variation BTA.
Program $\mathbf{P}'$ above is obtained by specializing the program $\mathbf{P}$ above, using the markings produced by the Static-Variation BTA.

3.4.4 Standardizing the Results of PRG-Based BTA

Readers who are familiar with BTA algorithms in the literature that operate on control-flow representations may have observed two points of incompatibility between our algorithms and those in the literature. First, our algorithms obtain an initial division of static and dynamic parameters using constant assignments and input statements, respectively. In contrast, standard BTA algorithms process a goal function whose parameters are divided into static and dynamic parameters. Second, our BTAs mark PRG vertices as static or dynamic, whereas standard BTA algorithms mark program variables as static or dynamic.

The first difference is a matter of style. We have chosen to use the imperative language represented by PRGs without modification. However, we could modify the language to name a single function that takes a finite number of parameters, without changing any of the treatment in the previous sections. This is the approach that we have taken in our implementation of these algorithms.

BTA algorithms for imperative programs can be placed in three categories, based on the manner in which they mark program variables: Some BTAs produce "uniform divisions," in which every variable is marked as either static or dynamic throughout the program; other BTAs are "flow-sensitive," in that they mark every variable as static or dynamic at every program point; finally, "polyvariant" BTAs assign multiple markings to the same variable at a given program point.

Transforming PRG vertex markings to a uniform division of program variables is trivial: Every variable that is defined at any PRG vertex that is marked dynamic is marked dynamic. All other variables are marked static. In order to obtain static vs. dynamic markings for every program variable at every program point, we use the markings at PRG vertices as the input to a simple forward any-path Gen-Kill data flow problem [Hec77] on the CFG, as shown in Figure 22. There is a one-to-one correspondence between the nodes in the CFG of a program and the non-ϕ-vertices in the PRG of a program. Thus, the basic concept behind our solution is as follows: At the entry vertex, all variables are marked static. At every other CFG vertex, the marking for a program variable at the vertex is obtained by joining the output markings for the variable at all predecessors. If the vertex is an assignment vertex, the output marking for the variable that is assigned at the vertex is obtained by joining the marking for the corresponding PRG vertex with the marking for the variable at the vertex. For all other variables, the output marking is identical to the marking at the vertex. For all non-assignment vertices, the output marking for every variable is identical to the marking for the variable at the vertex.

$$
\begin{aligned}
In(v)[var] &= \bigsqcup_{w \in preds(v)} Out(v)[var] \\
Gen(v)[var] &= \textbf{if } \textit{typeof}(v) = \textit{assign} \textbf{ and } \textit{lhs}(v) = \textit{var} \textbf{ and } v \in D(G) \textbf{ then } D \\
&\quad\ \textbf{else } S \\
Out(v)[var] &= In(v)[var] \sqcup Gen(v)[var]
\end{aligned}
$$

Figure 22: Conversion of PRG markings to CFG markings.

The equations above represent a forward any-path data flow problem on a CFG. $\sqcup$ is the join operation on the two element domain $\{S, D\}$ with $S \sqsubset D$. $D(G)$ is the set of PRG vertices marked as dynamic by one of the BTA algorithms described in this section. The set of equations above can be solved using a structured approach in which the equations are solved for progressively larger blocks of code. This can be achieved in time $\mathcal{O}(n * v)$, where n is the number of nodes in the CFG, and v is the number of program variables.

3.5 Limitations and related work

The BTA algorithms described above have the advantage that they can be shown to be conditionally safe, in terms of the PRG semantics. In Chapter 4 we extend the treatment in this chapter to programs with procedures and procedure calls. However, it is unclear how the PRG semantics or these algorithms can be extended to handle programs with arbitrary control-flow and aliased variables. In Chapter 5 we provide an alternative solution to this problem, by first defining the concept of "loop dependence," and then designing a BTA algorithm that uses flow dependences and loop dependences to trace dynamic behaviour. This algorithm handles programs with arbitrary control-flow as well as aliased variables. However, it is not based on a formal semantics similar to the PRG semantics; therefore, unlike in the case of the BTA algorithms discussed in this chapter, it has the drawback that its correctness will be argued only informally.

One novelty of our approach to binding-time analysis is our use of control dependence to determine dynamic behaviour. Control dependences were introduced by Denning and Denning to formalize the notion of information flow in programs, in the context of computer-security issues [DD77]. Since then, they have played a fundamental role in vectorizing and parallelizing compilers (for instance, see [FOW87]). The role of control dependences in partial evaluation was first noted by Ershov in [Ers82]. He used the term "logical dependence" to refer to transitive control dependence. Jones hinted at the possibility of using control dependences during binding-time analysis in a remark about "indirect dependences" caused by predicates of conditional statements [Jon88], but this direction was not pursued. Neither author described how the conservative results of analyses that always follow control dependences could be avoided. In [JGS93], Jones *et al.* informally presented 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. These notions are clearly related to control dependence, but were developed not to obtain correct BTAs but to identify when partial evaluation might produce good results.

Our approach is based on dependence graphs because they make the role of control dependences explicit. One such graph is the program dependence graph (PDG) defined by Ferrante *et al.* in [FOW87]. The PDG has been extended in several different directions: Horwitz *et al.* defined the system dependence graph (SDG), an extension of the PDG to handle programs with procedures [HRB90]. Alpern *et al.* defined the static single-assignment (SSA) form [AWZ88] for programs, which includes ϕ or gate statements for merging data from multiple predecessors, while Yang *et al.* defined the program representation graph (PRG) as an extension of the PDG with ϕ nodes similar to the gate nodes in the SSA form [YHR92].

We are able to use dependence graphs to reason about program behaviour (in this case, static and dynamic behaviour) because they are faithful to the semantics of the program. For instance, Horwitz *et al.* have shown that if the PDGs of two programs are isomorphic, given the same initial state either both programs diverge or both produce the same final state [HPR88a]. Such a property makes it reasonable to develop a semantics for PDGs themselves. Selke has defined a graph rewriting semantics for PDGs [Sel89] that represents computation steps as graph transformations. Cartwright *et al.* decomposed the meaning function for a program into a function that transforms programs into "code trees" that resemble PDGs, and an interpreter for code trees that provides an operational semantics for code trees [CF89]. These two semantics for PDGs are unsuitable for binding-time analysis because they do not provide a basis for capturing static and dynamic behaviour. In particular, the behaviour of a particular vertex or the values produced by it cannot be captured directly from the semantics. As a result, we use Ramalingam's semantics for extended PRGs [RR89].

Program representations such as the PRG and SSA form are similar to the continuation passing style (CPS) [App92], in that information is stored about the future use of variables in the program. More formally, Kelsey has shown in [Kel95] that it is always possible to convert programs written in SSA form to CPS. This might suggest the use of CPS for BTA rather than the PRG used in our approach. Further, Consel and Danvy have shown that conversion of a subject program to CPS style improves the residual program produced for it by a partial evaluator [CD91]. The arguments against using CPS are two-fold: (a) As pointed out by Kelsey in [Kel95], converting an imperative program to CPS style requires the same flow analysis as that required to produce the SSA form (or the PRG), and (b) Given a choice between two essentially similar representations, the PRG is preferable as its semantics provides a straightforward way to define a semantic foundation for partial evaluation.

The first attempt at a semantic foundation for the partial evaluation process was provided by Ershov in [Ers82]; he defined it as a partitioning of elementary computation steps C into static computations C' and dynamic computations C''. Jones provided the first definition of what it means for a BTA algorithm to produce "correct" markings [Jon88] (congruence). As mentioned earlier, the notion of congruence is an unsatisfactory correctness criteria for BTA algorithms; we have shown that the PRG semantics can be used to define suitable criteria. Wand has presented a correctness criterion for BTA-based partial evaluation of terms in the pure λ-calculus in [Wan93], but it is not clear if that can be applied to specializers for imperative programs.

Other authors have also tackled the termination problem, and have defined termination analyses that can be combined with congruence-based BTA to produce analyses that provide a termination guarantee for all programs [Hol91, GJ96, AH96]. The chief differences between their work and ours are:

- Other termination analyses are applicable only to restricted languages, and can analyze only simple kinds of termination criteria. In Chapter 6 we describe a new termination analysis that improves upon previous analyses by using an optimistic approach and more complex termination criteria.

- More importantly, other termination analyses are based on conservatively identifying BSV variables, and do not consider control dependences explicitly. As a result, such analyses cannot distinguish between unbounded behaviour arising

> from dynamic control and unbounded behaviour arising from static-infinite computation. As mentioned in the introduction, this results in every static loop being treated conservatively, limiting the ability of the BTA to identify static variables. In contrast, the approach advocated in this chapter can be explained as follows: We selectively use control dependences to distinguish potential unbounded behaviour arising from dynamic control from unbounded behaviour resulting from static-infinite computation. We then use termination analyses to conservatively treat as static some of the variables identified as unbounded due to dynamic control.

In this chapter, we have presented conditionally safe BTA algorithms for single-procedure programs. In the next chapter, we extend this work to multi-procedure programs.

Chapter 4

Extending safe BTA to programs with procedures

In the previous chapter, we used the structure and semantics of PRGs to define conditionally safe BTA algorithms. The class of programs that can be represented by PRGs is restricted, since the PRG representation does not handle multi-procedure programs. In this chapter, we extend our BTA algorithms to programs with procedure definitions and calls.

In the case of single-procedure programs, we are able to use the PRG representation in a straightforward manner, to develop a semantic foundation for BTA, and to design BTA algorithms on the PRG. However, there is no equivalent representation for multi-procedure programs. Therefore, we define the system representation graph (SRG), which extends the dataflow semantics and structure of the PRG to programs with procedures.

As we mentioned earlier, the PRG is an extension of the program dependence graph, which represents single procedure programs. Horwitz et al have extended the PDG to programs with procedures by defining the system dependence graph (SDG) [HRB90]. The SDG contains a dependence graph similar to the PDG for every procedure in the program, and inter-procedural dependence edges that link the procedure dependence graphs in order to account for the procedure calls in the program. In this context, the SRG can be viewed as a semantic extension of the SDG, just as the PRG is a semantic extension of the PDG.

In this chapter, we first outline the structure of the system dependence graph, and its relationship with the program dependence graph. We then define the system representation graph, and present its semantics. We use this semantics to define the strongly static, weakly static and statically varying properties for vertices, as we did for PRGs in the previous chapter. We then define BTA algorithms as simple reachability operations on the system representation graph to conservatively identify vertices that satisfy these semantic properties. Like their single-procedure counterparts, these algorithms provide a termination guarantee in the absence of static-infinite computation.

Procedure calls add several complications to the task of developing a dependence-based representation for programs whose semantics is faithful to the standard operational semantics of a program. As a result, we restrict the class of multi-procedure programs represented by SRGs to programs that have "reducible" call graphs [ASU86]. A reducible call graph is one in which every call site on a procedure either produces calls that are "entry" calls to the procedure, or produces calls that are recursive calls to the procedure. Our experiments show that this class of programs is fairly general. For instance, 15 of the 17 programs we examined from the Spec95 benchmark suite have reducible call graphs. In Section 4.5, we discuss how we can extend our work to handle arbitrary multi-procedure programs.

4.1 The system dependence graph

The system dependence graph (SDG) is an extension of the program dependence graph, designed by Horwitz et al to extend the operations of program slicing and integration to programs with procedure definitions and calls [HRB90].

Programs represented by SDGs have the same features as the single procedure programs represented by PDGs, with the following extensions: There is a single main procedure and several auxiliary procedures, none of which may call the main procedure. A procedure may have any number of parameters, each of which is passed by value-result. Global variables are allowed; they are accommodated by adding them as extra parameters to all procedures that either use or define them.

An SDG includes a program dependence graph for the main procedure, and several procedure dependence graphs, one for every auxiliary procedure in the program. A procedure dependence graph is similar to a PDG, except that it includes additional vertices to handle the value-result parameters of the procedure. A procedure dependence graph has a *formal-in* vertex for each parameter of the procedure, and a *formal-out* vertex for every parameter that is re-defined within the procedure. For every call on a procedure, there is a *call* vertex, which serves as a control predicate for the call, and *actual-in* and *actual-out* vertices that match the formal-in and formal-out vertices in the procedure dependence graph of the called procedure. Inter-procedural flow dependence edges are added from actual-in vertices to corresponding formal-in vertices, and from formal-out vertices to corresponding actual-out vertices. In addition, an inter-procedural control dependence edge is added from the call vertex at a call site to the procedure entry vertex of the procedure dependence graph of the called procedure.

Finally, the SDG is augmented with *summary* edges that enable precise slicing operations; we omit them from this discussion. The SDG for the power program, modified to use recursive procedure calls, is shown in Figure 23.

The inter-procedural flow dependence edges between the formal and actual vertices in the SDG implement a value-result passing mechanism. For a call from procedure P to procedure Q, P copies the values of the actual parameters into temporaries before making the call. Q copies the actual values from the temporaries into its local parameter variables before executing the code in its body, and then copies the values from its parameters to temporaries for the return values of the parameters. P then copies the result values from the temporaries to the variables that are the actual parameters, after which the call is complete.

Example 7 The SDG for the power program is shown in Figure 23. Although there is no loop predicate in the program, the looping behaviour of procedure *prodSum* is controlled by the dynamic predicate involving the dynamic parameter *ctr*. Thus, parameter *prod*, which would be treated as static by a congruent BTA, is in fact in a dynamic loop. This loop is represented in the SDG by the flow dependence cycle from the formal-in vertex for *prod* to itself; the loop is controlled by a dynamic predicate. We wish to extend the SDG in such a way that the semantics of the extended graph indicates that the formal-in vertex for *prod* does not satisfy the analogues of the various static properties defined in Chapter 3. □

(a)

```
main () {
  float x = 2.0;
  read(n);
  float a = 1.0;
  call prodSum(n, a, x);
  write(a);
}
```

```
prodSum (ctr, prod, amt) {
  if ( ctr > 0 ) {
    ctr = ctr − 1;
    prod = prod ∗ amt;
    call prodSum(ctr, prod, amt);
  }
}
```

(b)

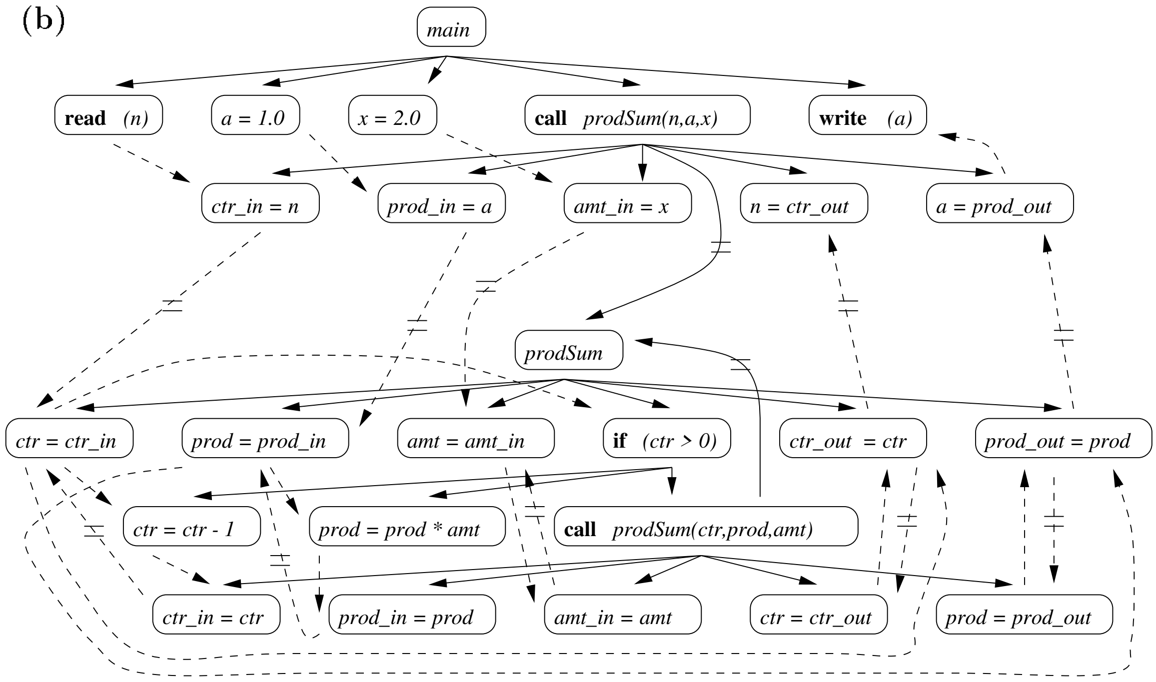


Figure 23: The system dependence graph of the power program.
The power program, written using recursive procedures, is shown in (a) above. Its SDG, with summary edges omitted, is shown in (b) above. Inter-procedural flow and control dependences are tagged with "=". For every parameter *par*, *par_in* is the copy-in temporary, and *par_out* is the copy-out temporary.

4.2 The system representation graph

The system dependence graph described in the previous section has the same drawback as the PDG, in that it does not have an agreed upon semantics that we can use to develop a semantic foundation for safe BTA. As a result, we define an extension of the SDG, which we term the system representation graph (SRG). The SRG is a semantic extension of the SDG; it is also an inter-procedural extension of the PRG. In this section, we define the structure and semantics of the SRG.

Before describing the SRG itself, we present our motivation for its design. Since the PRG has a value-sequence semantics that makes the definition of static properties straightforward, we would like the SRG to closely resemble the PRG. In particular, we would like to ensure that every procedure graph in the SRG is almost identical to the PRG representation of the procedure. In addition, we would like to develop a semantics for SRGs that is a generalization of the PRG semantics, so that we can extend the definitions of static behaviour from the previous chapter to the generalized semantics.

We assume that the call graph of the program to be represented is reducible. The concept of reducible graphs is generally used in the context of control-flow graphs. Reducibility can be expressed formally in many forms; for instance, a control-flow graph is reducible iff every loop in the graph has a single entry point. In the context of call graphs, a call graph is reducible if every call site on a procedure produces either only entry calls to the procedure, or only recursive calls. Therefore, we can claim that every call site on a procedure is either an "entry" call site or a recursive call site. An entry call site on a procedure P is one through which every call on P is an entry call, or a call in which there is no previous invocation of P on the call stack. In contrast, a recursive call site produces calls on P in which there are previous invocations of P on the call stack. This property allows us to generalize the loops in PRG programs to recursive procedures in the SRG. In addition, a program with a reducible call graph can be transformed into a program in which every procedure has exactly one entry or outer call site, every recursive call site on a procedure P is contained in the body of P itself (*i.e.*, there are no transitive recursive calls in the program), and non-recursive procedures have been inlined so that every remaining procedure has a recursive call site. We assume that this transformation has been carried out.[1] This allows us to selectively place appropriate ϕ nodes at the recursive call sites in the program, to account for the looping behaviour introduced by recursive procedure calls. Finally, we assume that every procedure has at most one recursive call site within its body. This assumption simplifies our presentation. All of our simplifying assumptions are discussed in further detail in Section 4.5 of this chapter.

4.2.1 Syntactic structure of SRGs

Structurally, the SRG contains a procedure representation graph for every procedure in the program. Procedure representation graphs extend procedure dependence graphs in exactly the same manner that PRGs extend PDGs. The distinction between PRGs and procedure representation graphs is that the latter include call and parameter vertices. Because of the presence of the PRG ϕ vertices in procedure representation graphs, SRGs do not require any actual-in or actual-out vertices. In addition, formal-in vertices

[1]The first transformation can be achieved by producing a new copy of the procedure for every outer call site. The remaining properties can be obtained by inlining function calls until recursive calls are encountered.

are replaced by ϕ_{value} vertices, which perform the copy-in actions of the value-result parameter passing mechanism. Similarly, formal-out vertices are replaced by $\phi_{resultI}$ and $\phi_{resultO}$ vertices, which pass on return values for parameters to inner and outer call sites, respectively. The SRG contains two kinds of call vertices, **callO** and **callI** vertices, for entry calls and recursive calls, respectively. Finally, ϕ_{call} vertices are added around recursive call sites. The role of these vertices is illustrated by the program in Figure 24, and is summarized below:

- ϕ_{value} vertices mediate between the definitions of an input parameter at the outer and inner call sites, in much the same way that ϕ_{enter} vertices mediate between the outer and inner definitions of a variable defined within a loop.

- $\phi_{resultO}$ vertices filter the return values for an output parameter, passing on the appropriate values to the outer call site. They play a role similar to that of ϕ_{exit} vertices.

- $\phi_{resultI}$ vertices filter the return values for an output parameter, passing on the appropriate values to the inner call site. They play a role similar to that of ϕ_{while} vertices.

- **callO** vertices represent entry call sites.

- **callI** vertices represent recursive call sites.

- ϕ_{call} vertices are placed at every nesting level of a procedure between the procedure entry vertex and the recursive call site. Their role is to mediate between inner and outer control predicates in such a way that a **callI** vertex can determine when a particular sequence of recursive calls has been completed. The label on the control dependence edge in the SRG from the outer predicate to the ϕ_{call} vertex is identical to the label on the control dependence edge in the SDG from the outer predicate to the inner predicate.

Construction of the SRG can be thought of as follows: The SDG for the program is constructed (without summary edges), PRG ϕ nodes are added to each procedure dependence graph, actual-in and actual-out vertices are eliminated, formal-in and formal-out vertices are replaced by ϕ_{value}, $\phi_{resultO}$ and $\phi_{resultI}$ vertices, and ϕ_{call} vertices are added around recursive call sites.

4.2.2 Semantics of SRGs

To meet our goal of generalizing PRGs to SRGs both structurally and semantically, we have devised a semantics for SRGs in which the behaviour of every vertex is represented by a value sequence. The value sequence contains, in order, the values computed at the vertex during program execution. Every value is instrumented with a tag, which is used by some vertex types to distinguish between entry calls and recursive calls. For instance, the procedure-entry vertex of every procedure produces values that are tagged with counts that make it possible for recursive call vertices to determine when recursion terminates. This mechanism is explained in detail later in this section. The SRG semantics is a generalization of the PRG semantics. In particular, all the vertices in the PRG of a single-procedure program can be considered to have output sequences in which values are tagged with the empty tag.

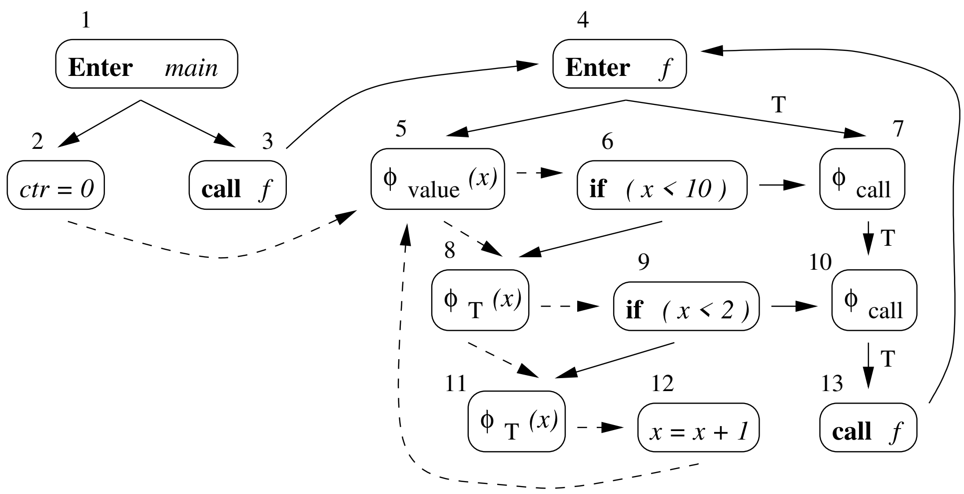
In the SRG semantics, every vertex type that is present in the PRG has exactly the

(a)

```
main () {
  int ctr = 0;
  call f(ctr);
}

f (x) {
  if ( x < 10 )
    if ( x < 2 ) {
      x = x + 1;
      call f(x);
    }
}
```

(b)



(c)

$v_1 \rightarrow [(T, \epsilon)]$ $\quad$ $v_2 \rightarrow [(0, \epsilon)]$

$v_3 \rightarrow [(O, \epsilon)]$ $\quad$ $v_4 \rightarrow [(O, \epsilon) \cdot (I, \epsilon) \cdot (I, \epsilon)]$

$v_5 \rightarrow [(0, \epsilon) \cdot (1, \epsilon) \cdot (2, \epsilon)]$ $\quad$ $v_6 \rightarrow [(T, \epsilon) \cdot (T, \epsilon) \cdot (T, \epsilon)]$

$v_7 \rightarrow [(T, 1) \cdot (T, 1) \cdot (T, 1)]$ $\quad$ $v_8 \rightarrow [(0, \epsilon) \cdot (1, \epsilon) \cdot (2, \epsilon)]$

$v_9 \rightarrow [(T, \epsilon) \cdot (T, \epsilon) \cdot (F, \epsilon)]$ $\quad$ $v_{10} \rightarrow [(T, 1) \cdot (T, 1) \cdot (F, 1)]$

$v_{11} \rightarrow [(1, \epsilon) \cdot (2, \epsilon)]$ $\quad$ $v_{12} \rightarrow [(0, \epsilon) \cdot (1, \epsilon)]$

$v_{13} \rightarrow [(I, 1) \cdot (I, 1) \cdot (F, 1)]$

Figure 24: The system representation graph for a recursive procedure program. A simple loop program, written using recursive procedures, is shown in (a) above. Its SRG is shown in (b) above. Solid arrows represent control dependences, and dashed arrows represent flow dependences. The value sequences produced at the SRG vertices are shown in (c) above.

same behaviour as in the PRG semantics, with the exception that the semantic equations are modified to handle tagged values. The semantic equations at SRG vertices are shown in Figure 25.

The new vertices introduced in the SRG behave as follows: Procedure-entry vertices combine the sequences from their call predecessors, producing O values for entry or outer calls and I values for recursive or inner calls. Values are tagged with a counter, which is used by ϕ_{call} and **callI** vertices to determine when a given sequence of recursive calls produced by an entry call terminates. These tags are ignored by vertices other than ϕ_{call} and recursive call vertices. ϕ_{value} vertices behave like the ϕ_{enter} vertices in the PRG, using the O/I values of their procedure-entry predecessors to mediate between the values from their inter-procedural data predecessors, just as ϕ_{enter} vertices use the T/F values from their loop predicate predecessors. $\phi_{resultI}$ and $\phi_{resultO}$ vertices use the values of their procedure-entry predecessors to filter the values produced by their data predecessors, retaining those values that must be returned to entry and recursive call sites, respectively. The behaviour at these return-value vertices is complicated by the fact that recursive calls are stacked. Therefore, for a given sequence $[o, i_1, \ldots, i_n]$ of recursive calls to a procedure P, control reaches the flow dependence predecessor of a return-value statement of invocation i_j of P before control reaches the same flow-dependence predecessor in invocation i_{j-1} of P. As a result, the values produced by the flow dependence predecessor of a return-value vertex are in the reverse order of the values produced by the procedure-entry predecessor, in terms of the actual calls or invocations to which they correspond. More precisely, for the sequence of calls $[o, i_1, \ldots, i_n]$, if the sequence at the flow predecessor is given by $[v_0, \ldots, v_n]$, the value that is returned to the entry call site is v_n, not v_0. The equations at return-value

$$
\begin{aligned}
&\mathbf{E_G} \;\doteq\; \lambda i.\lambda v.\lambda f.\\
&\quad \mathbf{type}(v) = \phi_{\mathbf{T}} \to \text{select}'((\text{true}, \epsilon), f\ i\ \mathit{ctrlPred}(v), f\ i\ \mathit{dataPred}(v))\\
&\quad \mathbf{type}(v) = \mathbf{callO} \to \text{replace}'(\mathit{controlLabel}(v), (O, \epsilon), f\ i\ \mathit{parent}(v))\\
&\quad \mathbf{type}(v) = \phi_{\mathbf{value}} \to \text{parMerge}(f\ i\ \mathit{ctrlPred}(v), f\ i\ \mathit{outerPred}(v), f\ i\ \mathit{innerPred}(v))\\
&\quad \mathbf{type}(v) = \phi_{\mathbf{resultO}} \to \text{outerSel}(f\ i\ \mathit{ctrlPred}(v), f\ i\ \mathit{dataPred}(v))\\
&\quad \mathbf{type}(v) = \phi_{\mathbf{resultI}} \to \text{innerSel}(f\ i\ \mathit{ctrlPred}(v), f\ i\ \mathit{dataPred}(v))\\
&\quad \mathbf{type}(v) = \mathbf{procEntry} \to \text{callMerge}(f\ i\ \mathit{outerPred}(v), f\ i\ \mathit{innerPred}(v))\\
&\quad \mathbf{type}(v) = \mathbf{callI} \to \text{tagReplace}(\mathit{controlLabel}(v), f\ i\ \mathit{parent}(v))\\
&\quad \mathbf{type}(v) = \phi_{\mathbf{call}} \to\\
&\qquad \text{callFill}(\mathit{outerCtrlLabel}(v), f\ i\ \mathit{outerPred}(v), f\ i\ \mathit{innerPred}(v))
\end{aligned}
$$

where *parMerge*, *outerSel*, *innerSel*, *tagReplace*, *callFill*, and *callMerge* are defined as follows:

$$
\begin{aligned}
&\mathit{parMerge}\text{:} && \text{parMerge}((O,t_1)\cdot \mathit{tail}_1,\ (x,t_2)\cdot \mathit{tail}_2,\ s) = (x,\epsilon)\cdot \text{parMerge}(\mathit{tail}_1,\ \mathit{tail}_2,\ s)\\
& && \text{parMerge}((I,t_1)\cdot \mathit{tail}_1,\ s,\ (x,t_2)\cdot \mathit{tail}_2) = (x,\epsilon)\cdot \text{parMerge}(\mathit{tail}_1,\ s,\ \mathit{tail}_2)\\
&\mathit{outerSel}\text{:} && \text{outerSel}((p,t_1)\cdot \mathit{nil},\ (d,t_2)\cdot \mathit{nil}) = (d,\epsilon)\\
& && \text{outerSel}(p\cdot (p',t_1)\cdot \mathit{tail}_1,\ (d,t_2)\cdot \mathit{tail}_2) = \mathbf{if}\ (p' = O)\ \mathbf{then}\\
& && \quad (d,\epsilon)\cdot \text{outerSel}((p',t_1)\cdot \mathit{tail}_1,\ \mathit{tail}_2)\ \mathbf{else}\ \text{outerSel}((p',t_1)\cdot \mathit{tail}_1,\ \mathit{tail}_2)\\
&\mathit{innerSel}\text{:} && \text{innerSel}((p,t_1)\cdot \mathit{nil},\ (d,t_2)\cdot \mathit{nil}) = \mathit{nil}\\
& && \text{innerSel}(p\cdot (p',t_1)\cdot \mathit{tail}_1,\ (d,t_2)\cdot \mathit{tail}_2) = \mathbf{if}\ (p' = O)\ \mathbf{then}\\
& && \quad \text{innerSel}((p',t_1)\cdot \mathit{tail}_1,\ \mathit{tail}_2)\ \mathbf{else}\ (d,\epsilon)\cdot \text{innerSel}((p',t_1)\cdot \mathit{tail}_1,\ \mathit{tail}_2)\\
&\mathit{tagReplace}\text{:} && \text{tagReplace}(l,\ (x,t)\cdot \mathit{nil}) = (F,t)\\
& && \text{tagReplace}(l,\ (x,t_1)\cdot (x',t_2)\cdot \mathit{tail}) = \mathbf{if}\ (x = l)\ \mathbf{then}\\
& && \quad (I,t_1)\cdot \text{tagReplace}(l,\ (x',t_2)\cdot \mathit{tail})\ \mathbf{else\ if}\ (t_1 < t_2)\ \mathbf{then}\\
& && \quad (F,t_1)\cdot \text{tagReplace}(l,\ (x',t_2)\cdot \mathit{tail})\ \mathbf{else}\ \text{tagReplace}(l,\ (x',t_2)\cdot \mathit{tail})\\
&\mathit{callFill}\text{:} && \text{callFill}(l,\ (x,t)\cdot \mathit{tail},\ \mathit{nil}) = (\delta,t)\cdot \text{callFill}(l,\ \mathit{tail},\ \mathit{nil})\\
& && \text{callFill}(l,\ (x,t_1)\cdot \mathit{tail}_1,\ (y,t_2)\cdot \mathit{tail}_2) = \mathbf{if}\ (x = l)\ \mathbf{then}\\
& && \quad (y,t_1)\cdot \text{callFill}(l,\ \mathit{tail}_1,\ \mathit{tail}_2)\ \mathbf{else}\ (\delta,t_1)\cdot \text{callFill}(l,\ \mathit{tail}_1,\ (y,t_2)\cdot \mathit{tail}_2)\\
&\mathit{callMerge}\text{:} && \text{callMerge}(\mathit{nil},\ s) = \mathit{nil}\\
& && \text{callMerge}((v,t)\cdot \mathit{tail},\ s) = (v,1)\cdot \text{callAux}(\mathit{tail},\ s)\\
& && \text{callAux}(s,\ (x,t)\cdot \mathit{tail}) = \mathbf{if}\ (x = I)\ \mathbf{then}\ (I,t)\cdot \text{callAux}(s,\ \mathit{tail})\\
& && \quad \mathbf{else\ if}\ (s = \mathit{nil})\ \mathbf{then}\ \mathit{nil}\ \mathbf{else}\ (\mathit{car}(s), t+1)\cdot \text{callAux}(\mathit{cdr}(s),\ \mathit{tail})
\end{aligned}
$$

Figure 25: The semantic equations associated with SRG vertices.

In the SRG equation above, the functions at all vertex types that are present in the PRG are unchanged, except that they are extended to produce tagged values, as shown for ϕ_T vertices. Some vertex types are omitted for brevity. ϵ is the empty tag, while δ is a "don't-care" value. **procEntry** vertices produce O/I values rather than T/F values. For the comparison test in function *callFill*, we assume that $O = T$ and $I = T$.

vertices are designed to accommodate this mismatch in the orderings of the predecessor sequences. Entry call vertices use the sequences of their control predecessors to produce O values representing procedure calls. The behaviour at recursive call vertices and ϕ_{call} vertices is less intuitive, and is therefore explained in detail below.

The implicit looping behaviour introduced by recursive calls in SRG programs can be thought of as a generalization of the explicit looping behaviour produced by while loops in PRG programs. Consider the behaviour of a loop predicate vertex in the PRG semantics: On each invocation of the loop, the vertex produces a series of zero or more *true* values, followed by a single *false* value. The *false* values in loop predicate sequences act as separators between invocations of the loop. They are used by ϕ vertices to merge or filter predecessor sequences appropriately. Therefore, in order to generalize explicit loops to recursive calls, we require procedure-entry vertices to produce similar separator values.

A recursive procedure can be thought of as a generalized loop, in which the predicates that control the execution of the recursive call site play the role of the loop predicate. Structurally, their role in the loop construct is made explicit by the addition of ϕ_{call} vertices in the superstructure of the procedure, from the procedure-entry vertex to the recursive call site.

In the semantics, these predicates play the following role: A recursive call site produces a single *false* value at the end of every sequence of recursive calls; this value is passed on to the procedure-entry vertex, which uses the value to produce appropriate O/I values. A sequence of recursive calls terminates when one of the predicates above the recursive call site in the control dependence tree of the procedure takes on a value that does not match the label on the control dependence edge from that predicate to the ϕ_{call} vertex at the next level between it and the recursive call site. The role of ϕ_{call} vertices in the semantics is to merge values from predicates at adjacent levels so that the recursive call site can determine when such a value, which indicates the end of a recursive sequence of calls, is produced at some level in the control dependence tree.

It is possible that in the same sequence of recursive calls, the predicates above the recursive call site may produce more than one value that suggests the termination of a recursive sequence of calls (for instance, if there are loop predicates between the procedure-entry vertex and the recursive call site). To account for this situation, we tag values with counters, which enable recursive call-site vertices to determine the last such value in every sequence of recursive calls. An example of the use of tags is given in Figure 24.

Example 8 The SRG for the power program from Figure 23 is shown in Figure 26. The sequences produced at selected vertices of the graph, given input values of 2 and 3 for variable n, are shown in Figure 27, and are explained below:

- At ϕ_{value} vertex v_7, the sequences of the outer data predecessor v_2 and the inner data predecessor v_{20} are merged, using the sequence from the procedure-entry predecessor v_6. This is done by function *parMerge*, which is similar to function *merge* from the PRG semantics.

- At ϕ_{call} vertex v_{16}, the values from the inner predicate v_9 are passed through, because control reaches the inner predicate on every invocation of the outer predicate v_6, the procedure-entry vertex. Each value is tagged using the tag from the outer predicate.

- At call vertex v_{22}, the function *tagReplace* produces an I for each value in its predecessor sequence (s_{16}) that indicates that control reaches the call vertex. It produces one *false* value at the end of each recursion sequence, mimicing the behaviour of a loop predicate.

- At procedure-entry vertex v_6, the sequences from its call predecessors (v_4 and v_{22}) are merged. The iteration tag is incremented for every new entry call value from its outer predecessor.

- At $\phi_{resultO}$ vertex v_{11}, the function *outerSel* passes on values corresponding to return values from entry calls. *outerSel* accounts for the reverse order of the data predecessor sequence (s_{19}) in relation to its procedure-entry predecessor sequence (s_6). In the power program, the same value for parameter *prod* is returned to all the calls on *prodSum*, because the procedure is tail-recursive. □

Since the SRG semantics defined in this section is a generalization of the PRG semantics, it shares the non-standard property of the PRG semantics. In particular, the SRG semantics is more defined than the standard semantics in the case of inputs for which the program does not terminate normally. As is the case with the PRG semantics, the SRG semantics is consistent with the standard operational semantics on programs that terminate normally.

4.3 Semantics of static behaviour for SRG vertices

In this section we use the SRG semantics to generalize the three increasingly general forms of static behaviour and the concept of bounded static variation from Chapter 3

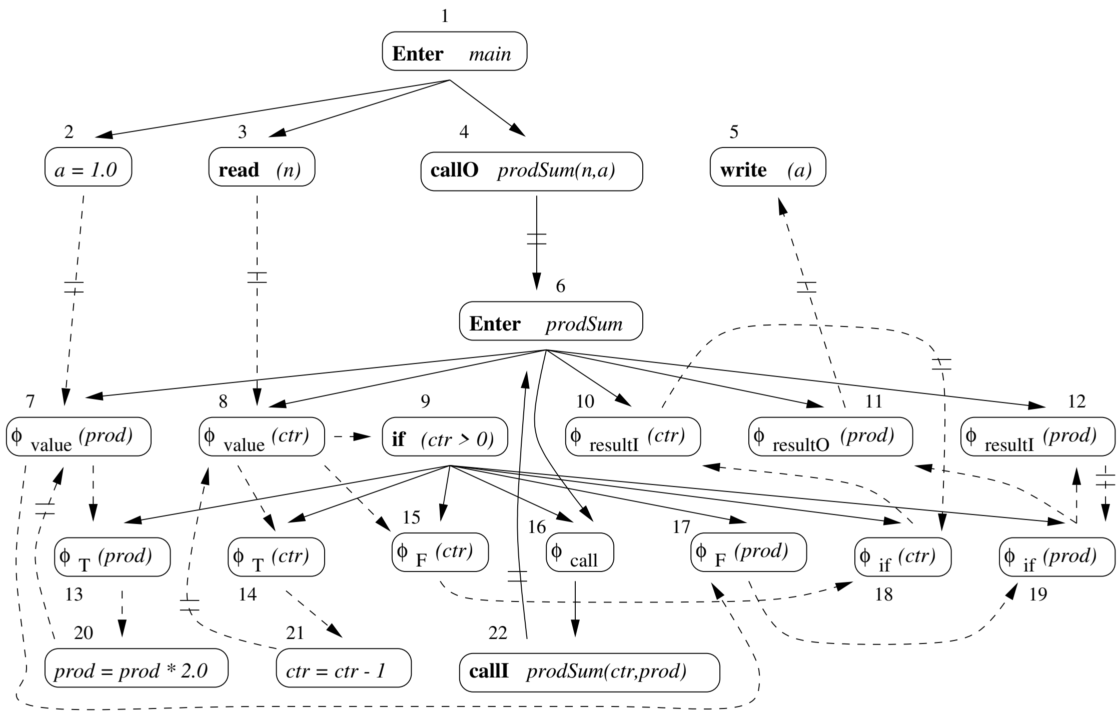


Figure 26: The system representation graph of the power program.

The SRG of the power program, written using recursive procedures, is shown above. Inter-procedural flow and control dependences are tagged with "=". Labels on control dependence edges are omitted; all the control dependence edges in the SRG for the power program above are labeled with *true*. The third parameter to procedure *prodSum* has been omitted in order to make the SRG more readable. Control dependence edges that play no role in the semantic equations have been omitted.

$\mathbf{M}\ [\![G]\!]\ [2] \rightarrow$	$\mathbf{M}\ [\![G]\!]\ [3] \rightarrow$
$v_2 \rightarrow [1.0]$	$v_2 \rightarrow [1.0]$
$v_3 \rightarrow [2]$	$v_3 \rightarrow [3]$
$v_4 \rightarrow [O]$	$v_4 \rightarrow [O]$
$v_5 \rightarrow [4.0]$	$v_5 \rightarrow [8.0]$
$v_6 \rightarrow [(O,1)\cdot(I,1)\cdot(I,1)]$	$v_6 \rightarrow [(O,1)\cdot(I,1)\cdot(I,1)\cdot(I,1)]$
$v_7 \rightarrow [1.0\cdot 2.0\cdot 4.0]$	$v_7 \rightarrow [1.0\cdot 2.0\cdot 4.0\cdot 8.0]$
$v_8 \rightarrow [2\cdot 1\cdot 0]$	$v_8 \rightarrow [3\cdot 2\cdot 1\cdot 0]$
$v_9 \rightarrow [T\cdot T\cdot F]$	$v_9 \rightarrow [T\cdot T\cdot T\cdot F]$
$v_{10} \rightarrow [0\cdot 0]$	$v_{10} \rightarrow [0\cdot 0\cdot 0]$
$v_{11} \rightarrow [4.0]$	$v_{11} \rightarrow [8.0]$
$v_{12} \rightarrow [4.0\cdot 4.0]$	$v_{12} \rightarrow [8.0\cdot 8.0\cdot 8.0]$
$v_{13} \rightarrow [1.0\cdot 2.0]$	$v_{13} \rightarrow [1.0\cdot 2.0\cdot 4.0]$
$v_{14} \rightarrow [2\cdot 1]$	$v_{14} \rightarrow [3\cdot 2\cdot 1]$
$v_{15} \rightarrow [0]$	$v_{15} \rightarrow [0]$
$v_{16} \rightarrow [(T,1)\cdot(T,1)\cdot(\delta,1)]$	$v_{16} \rightarrow [(T,1)\cdot(T,1)\cdot(T,1)\cdot(\delta,1)]$
$v_{17} \rightarrow [4.0]$	$v_{17} \rightarrow [8.0]$
$v_{18} \rightarrow [0\cdot 0\cdot 0]$	$v_{18} \rightarrow [0\cdot 0\cdot 0\cdot 0]$
$v_{19} \rightarrow [4.0\cdot 4.0\cdot 4.0]$	$v_{19} \rightarrow [8.0\cdot 8.0\cdot 8.0\cdot 8.0]$
$v_{20} \rightarrow [2.0\cdot 4.0]$	$v_{20} \rightarrow [2.0\cdot 4.0\cdot 8.0]$
$v_{21} \rightarrow [1\cdot 0]$	$v_{21} \rightarrow [2\cdot 1\cdot 0]$
$v_{22} \rightarrow [(I,1)\cdot(I,1)\cdot(F,1)]$	$v_{22} \rightarrow [(I,1)\cdot(I,1)\cdot(I,1)\cdot(F,1)]$

Figure 27: The sequences at SRG vertices for the power program.

The sequences of tagged values produced at selected vertices of the SRG for the power program, given two different input values for variable n, are shown above. In these sequences, values that have empty tags are shown without tags, while values with tags are shown as pairs (v,t), where v is the value and t is its associated tag. The sequences at SRG vertices are similar to the sequences at PRG vertices, except that procedure-entry, ϕ_{call} and call vertices produce tagged values, and procedure-entry and call vertices produce values from $\{O, I, \mathit{false}\}$ as opposed to $\{\mathit{true}, \mathit{false}\}$.

```
main () {                     outer (ctr) {                 inner (index) {
    read(dyn);                    if ( ctr ≠ 0 ) {              if ( index < 3 ) {
    call outer(dyn);                  stat = 0;                     v : index = index + 1;
}                                     call inner(stat);             call inner(index);
                                      ctr = ctr − 1;            }
                                      call outer(ctr);      }
                                  }
                              }
```

$$\begin{aligned}
\mathbf{M}\ [\![G]\!]\ [0]\ v &= [] \\
\mathbf{M}\ [\![G]\!]\ [1]\ v &= [1 \cdot 2 \cdot 3] \\
\mathbf{M}\ [\![G]\!]\ [2]\ v &= [1 \cdot 2 \cdot 3 \cdot 1 \cdot 2 \cdot 3]
\end{aligned}$$

Figure 28: An example of weakly static behaviour in SRG programs.
The program above is a modified version of the program in Figure 16 from Chapter 3.

to programs with procedures. Because the value-sequence semantics of SRGs is closely related to the PRG semantics, the definitions of these properties are unchanged for SRG vertices. The only modification is that the meaning function **M** for SRGs is implicitly modified to produce value sequences in which the tags on the values are omitted.

We first define strongly static behaviour using the SRG semantics.

Definition 18 Vertex v in SRG G is *strongly (semantically) static* iff the sequences in $\{\mathbf{M}\ [\![G]\!]\ i\ v \mid i \in \mathit{Stream}\}$ form a chain in Sequence. □

As shown by the behaviour of program point v in the program from Figure 28, the strongly static property may be too restrictive. The notion of rational repetitions from the PRG semantics carries over directly to the SRG semantics. The distinction is that in SRG programs, the repetitions may be generated by recursive calls.

Definition 19 Vertex v in SRG G is *weakly (semantically) static* iff at least one of the following holds:

(a) $\exists s \in Val^* \text{ s.t. } \forall i \in Stream,\ \mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{s^n \cdot nil | n \in \mathcal{N}\} \cup \{s^\infty\})$

or

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

where **DC** is the downwards-closure operator on *Sequence*. □

The SRG semantics can be used to define statically varying and BSV behaviour, in exactly the same fashion as for PRG programs.

Definition 20 Vertex v in SRG G is *statically (semantically) varying* iff at least one of the following holds:

(a) $\exists B \subset Val, |B|$ finite, s. t. $\forall i \in Stream$,

$$\mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{v_1 \cdot \ldots \cdot v_k \cdot nil | v_1, \ldots, v_k \in B\} \cup B^\omega)$$

or

(b) v is weakly static.

where **DC** is the downwards-closure operator on *Sequence*. □

Definition 21 Vertex v in SRG G is *(semantically) bounded static varying* iff:

$\exists B \subset Val, |B|$ finite, s. t. $\forall i \in Stream$,

$$\mathbf{M}\ [\![G]\!]\ i\ v \in \mathbf{DC}(\{nil\} \cup \{v_1 \cdot \ldots \cdot v_k \cdot nil | v_1, \ldots, v_k \in B\} \cup B^\omega)$$

where **DC** is the downwards-closure operator on *Sequence*.

Vertex v is *(semantically) non-BSV* iff it is not semantically BSV. □

The tags on the values in the value sequences produced at SRG vertices are ignored in the definitions above. This is because these tags do not provide any information relevant to the static/dynamic nature of an SRG vertex. Rather, they are used as instrumentation at procedure-entry, ϕ_{call} and recursive call vertices, to ensure that a *false* value is produced at the call vertex at the end of every sequence of recursive calls to the procedure.

A semantic foundation for safe and conditionally safe BTA on SRG programs follows directly from the characterization of safe and conditionally safe BTA for single-procedure programs in the previous chapter. This is achieved by replacing the definitions of static and BSV behaviour for PRG programs in the treatment in Section 3.3 with the definitions above.

4.4 BTA algorithms for SRG programs

As in the case of PRG programs, we are able to define BTA algorithms that use the flow and control dependences that are explicit in the structure of SRGs to conservatively identify static vertices. In this section, we define three BTA algorithms as abstract interpretations of the SRG semantics. Operationally, these algorithms are simple reachability operations on the SRG, and can be viewed as variants of operations for inter-procedural program slicing [HRB90]. Therefore, each of these algorithm has a running time that is linear in the number of edges in the SRG. Our BTA algorithms are "context-insensitive," in the sense that they do not account for calling context precisely. In order to account for calling context, it is necessary to match the call and return edges in flow dependence paths accurately, which increases the complexity of the slicing algorithm. We discuss this issue in greater detail later in the chapter.

The algorithms described in this section are direct extensions of our BTA algorithms from the previous chapter. As a result, the first algorithm follows control dependences blindly, and identifies only strongly static vertices as S; the second follows control dependences selectively, and thus identifies some weakly static vertices as S as well. The third BTA identifies some statically varying vertices as S by ignoring control dependences to vertices that have multiple static data dependence predecessors.

4.4.1 The Strong-Staticness BTA for SRGs

The goal of this BTA is to identify a subset of all the strongly-static vertices in the SRG of a program. The idea is to follow all flow and control dependence edges from the set of **read** vertices in the SRG, marking with D all vertices that are encountered along the way. This operation is identical to a forward inter-procedural program slice [HRB90] from the set of **read** vertices in the SRG. Vertices that are not in this forward slice are marked with S.

We present the Strong-Staticness BTA for SRGs as the fixed point of an abstract interpretation that is consistent with the SRG semantics. This interpretation is defined by the recursive equation in Figure 29. Because ϕ_{call} vertices are placed at every level in the nesting structure of a recursive procedure, the Strong-Staticness BTA is able to correctly account for the effect of a dynamic predicate on the recursion of a procedure even when the predicate is many nesting levels higher than the call site in the control dependence tree of the procedure.

Our correctness argument for this BTA is identical to the argument we used for the

$$
\begin{array}{l}
\mathbf{E^a_G} \doteq \lambda v.\lambda f_a. \\
\quad \mathbf{type}(v) = \mathbf{Entry} \to S \\
\quad \mathbf{type}(v) = \mathbf{read} \to D \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\} \to \\
\qquad\qquad \begin{cases} \text{abs\_replace}(\ f_a\ \mathit{parent}(v)\ ) & \text{if } \#\mathit{dataPreds}(v) = 0 \\ \text{abs\_map}(\ f_a\ \mathit{dataPred}_1(v), \ldots f_a\ \mathit{dataPred}_n(v)) & \text{otherwise} \end{cases} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{enter}} \to \\
\qquad\qquad \text{abs\_whileMerge}(\ f_a\ \mathit{whileNode}(v),\ f_a\ \mathit{innerDef}(v),\ f_a\ \mathit{outerDef}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{exit}} \to \text{abs\_select}(\ f_a\ \mathit{whileNode}(v),\ f_a\ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{while}} \to \text{abs\_select}(\ f_a\ \mathit{whileNode}(v),\ f_a\ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{T}} \to \text{abs\_select}(\ f_a\ \mathit{parent}(v),\ f_a\ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{F}} \to \text{abs\_select}(\ f_a\ \mathit{parent}(v),\ f_a\ \mathit{dataPred}(v)\ ) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{if}} \to \text{abs\_merge}(\ f_a\ \mathit{ifNode}(v),\ f_a\ \mathit{trueDef}(v),\ f_a\ \mathit{falseDef}(v)\ ) \\
\quad \mathbf{type}(v) = \mathbf{callO} \to \text{abs\_replace}(f\ \mathit{parent}(v)) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{value}} \to \text{abs\_cMerge}(f\ \mathit{ctrlPred}(v), f\ \mathit{outerPred}(v), f\ \mathit{innerPred}(v)) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{resultO}} \to \text{abs\_select}(f\ \mathit{ctrlPred}(v), f\ \mathit{dataPred}(v)) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{resultI}} \to \text{abs\_select}(f\ \mathit{ctrlPred}(v), f\ \mathit{dataPred}(v)) \\
\quad \mathbf{type}(v) = \mathbf{procEntry} \to \text{abs\_call}(f\ \mathit{outerPred}(v), f\ \mathit{innerPred}(v)) \\
\quad \mathbf{type}(v) = \mathbf{callI} \to \text{abs\_tagReplace}(f\ \mathit{parent}(v)) \\
\quad \mathbf{type}(v) = \phi_{\mathbf{call}} \to \text{abs\_call}(f\ \mathit{outerPred}(v), f\ \mathit{innerPred}(v))
\end{array}
$$

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

$$
\begin{array}{l}
\text{abs\_replace}(\ a\ ) = a \\
\text{abs\_select}(\ a,\ b\ ) = a \sqcup b \\
\text{abs\_merge}(\ a,\ b,\ c\ ) = a \sqcup b \sqcup c \\
\text{abs\_whileMerge}(\ a,\ b,\ c\ ) = a \sqcup b \sqcup c \\
\text{abs\_map}(\ a_1,\ \ldots,\ a_n\ ) = a_1 \sqcup \ldots \sqcup a_n \\
\text{abs\_call}(\ a,\ b\ ) = a \sqcup b \\
\text{abs\_tagReplace}(\ a\ ) = a \\
\text{abs\_cMerge}(\ a,\ b,\ c\ ) = a \sqcup b \sqcup c
\end{array}
$$

Figure 29: The abstract equations for the Strong-Staticness BTA on SRG programs.

```
main () {                S      outer (ctr) {               D      inner (index) {               S
    read(dyn);           D          if ( ctr ≠ 0 ) {        D          if ( index < 3 ) {        S
    call outer(dyn);     S              stat = 0;           S              index = index + 1;    S
}                                       call inner(stat);   S              call inner(index);    S
                                        ctr = ctr − 1;      D          }
                                        call outer(ctr);    D      }
                                    }
                                }
```

Figure 30: Application of the Weak-Staticness BTA.

Strong-Staticness BTA in the previous chapter, with the exception that Lemma 15 is adapted to SRG vertices.

4.4.2 The Weak-Staticness BTA for SRGs

As in the case of PRG programs, we define the Weak-Staticness BTA, an analysis that identifies a subset of all the weakly-static vertices in the PRG of a program. The Weak-Staticness BTA is similar to the Strong-Staticness BTA, differing only at constant assignment vertices and at entry call vertices. Entry call vertices are treated as constant assignments; they produce sequences of O values, which are always rational repetitions. In the abstract semantics, the function *abs_replace* is re-defined as follows:

$$\text{abs\_replace}(\ a\ ) = S$$

Example 9 The program from Figure 28 is shown in Figure 30, along with the markings produced by Weak-Staticness BTA. The SRG is omitted for brevity. □

The proof that the Weak-Staticness BTA is conditionally safe mimics the one for the Weak-Staticness BTA on PRGs, with Lemma 17 modified to SRG vertices:

Lemma 22 For any SRG vertex v that is not a **read** vertex, $\{\mathbf{F}^{j+1} \perp i\ v \mid i \in \mathit{Stream}\}$ is an approximate rational repetition if:

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

(b) *funcOf(v)* is a constant value, or v is a **callO** vertex. □

The functions at SRG vertices are all structured so that when predecessor sequences u_1, u_2, ..., u_k at vertex v are all rational repetitions, the output sequence at v is a rational repetition, with a base repeating sequence that is at most as long as the least common multiple of the lengths of the base repeating sequences in u_1, u_2, ..., u_k. This is the basis for the semantic justification of conditional safety for this analysis.

4.4.3 The Static-Variation BTA for SRGs

We extend the Weak-Staticness BTA for SRG programs to the Static-Variation BTA for SRG programs in exactly the same way as we did for PRG programs. The Weak-Staticness BTA is similar to the Strong-Staticness BTA, differing only at ϕ_{if} vertices. Therefore, the function *abs_merge* in the abstract semantics is re-defined as follows:

$$\text{abs\_merge}(\ a,\ b,\ c\ ) = b \sqcup c$$

4.5 Limitations and related work

In our treatment of multi-procedure programs in this chapter, we have imposed certain restrictions on the programs that are represented by system representation graphs. In this section, we begin by discussing these restrictions. We then discuss related work.

The first restriction we have imposed on an SRG program is that the call graph of the program must be reducible. The concept of reducible graphs is generally used in the context of control-flow graphs. Reducibility can be expressed formally in many forms; for instance, a control-flow graph is reducible iff every loop in the graph has a single entry point. In the context of call graphs, a call graph is reducible if every call site on a procedure produces either only entry calls to the procedure, or only recursive calls. We impose this restriction because it allows us to introduce the ϕ_{call} vertex superstructure in a selective manner, in particular around recursive call sites. There are three different ways in which we can justify this restriction: First, our experiments show that this constraint does not restrict the set of representable programs unduly. For instance, 15 of the 17 programs we examined from the Spec95 benchmark suite have reducible call graphs.[2] Second, given a program with an irreducible call graph, it is always possible to automatically transform the program into one that has a reducible call graph, by using an algorithm similar to the node-splitting algorithm described in, for instance, [ASU86, pp. 664-668]. We omit the details here. The algorithm is exponential in the worst case; however, it is unlikely that a program will have more than a few irreducible components. Finally, we can extend the SRG itself to account for irreducible control flow, in particular by placing ϕ_{call} vertices conservatively. However, this approach is not conceptually appealing.

Given that the subject programs have reducible call graphs, we have also assumed that every procedure has exactly one entry call site, and that there are no transitive

[2]The exceptions are *perl* and *li*. *perl* has an irreducible component consisting of roughly 5% of the total code base. Transforming *perl* to produce a reducible call graph should therefore not blow up the size of the code significantly. In contrast, the lisp interpreter *li* has a significant irreducible component.

recursive calls. These assumptions do not restrict the input language. In fact, every program with a reducible call graph can be converted to this form through a simple transformation. This transformation is acceptable because, in any case, the goal of the analysis for which we have defined the SRG is to enable partial evaluation to be carried out, which itself is a transformation of the program. The advantage of this assumption is that we do not need to associate SRG vertices with sets of sequences, each tagged with a context tag indicating the call site from which procedure calls relevant to the given sequence are made. Alternatively, given a procedure that has multiple entry call sites, we would require extra machinery in the SRG to impose the appropriate ordering on the calls generated from these call sites. We simplify the problem by assuming a single entry call site.

Finally, we insist that every procedure must have only one recursive call site. Again, we impose this restriction in order to simplify the problem of ordering the recursive calls correctly. If we removed this restriction, the BTA algorithms defined in the previous section would remain unchanged. It might appear that the single recursive call site restriction has the effect of restricting the recursive nature of the program to linear chains of calls, as opposed to general recursion, which may produce a tree of recursive calls. However, this is not the case. For instance, we allow recursive call sites within loops, which may produce a tree of recursive calls rather than a linear chain of calls. We have imposed the restriction of single recursive call sites only because we are interested in associating SRG vertices with ordered value sequences. The assumption allows us to simplify the machinery for producing values across inter-procedural edges in the correct order. In order to handle procedures with multiple recursive call sites, we can extend the tagging mechanism of the SRG semantics, while retaining the structure of the SRG.

All of the BTA algorithms described in the previous section are "context-insensitive," meaning that they do not account for calling context precisely. In this respect, our algorithms are similar to that of *cmix*, which is not context-sensitive. Horwitz et al. have shown that this problem can be handled efficiently, in the context of program slicing, by introducing "summary" edges in the SDG [HRB90]. We omit these edges from the SRG, because it is not clear how we would follow summary edges in a BTA algorithm such as Weak-Staticness BTA that follows dependence edges selectively. However, we can use summary edges to implement a context-sensitive version of Strong-Staticness BTA, since that algorithm follows all the dependence edges in the SRG. Reps et al. have presented an alternative approach to this problem, which involves using context-free language reachability to "match" parameter-in and parameter-out edges appropriately [RHS95]. It is possible that we could phrase our BTA algorithms declaratively as context-free language reachability problems, and use the scheme in [RHS95, Rep97] to account for the calling context on a procedure more precisely. We have not investigated this possibility.

In this chapter, we have extended our work on safe BTA for PRG programs in one direction, by expanding the input language in such a way that the language retains a dependence graph representation whose data-flow semantics can be used to argue the correctness of BTA algorithms. In the next chapter, we use a different approach. We develop a conditionally safe BTA algorithm for a richer language, with complex features such as arbitrary control flow and pointer variables.

Chapter 5

Loop dependences and safe BTA for C programs

In the preceding chapters, we have described BTA algorithms that provide a termination guarantee for partial evaluation, in the absence of static-infinite computation. These BTA algorithms have the advantage that they are based on dependence graph representations whose dataflow semantics can be used to establish their conditional safety. In addition, the algorithms have the advantage that they are simple reachability operations on dependence graphs. However, they are applicable only to a limited class of imperative programs, in particular programs without arbitrary control flow and pointer variables.

In this chapter, we describe a different approach towards our goal of safe BTA for imperative programs. We tackle the problem of developing safe BTA algorithms for C programs. It remains unclear how the dependence graphs from the previous chapters can be extended to handle arbitrary control flow and pointer variables. Therefore, we design a new kind of dependence graph, which we term the "loop dependence graph" (LDG). This graph contains "loop dependences," in addition to the standard data dependences and control dependences that are present in the system dependence graph representation of a program. Loop dependences have the advantage that they can be constructed for programs with arbitrary control flow. In addition, they make the role of dynamic control explicit, without following control dependences arising from the nesting structure of the program. Thus, loop dependences can be thought of as selective control dependences, appropriate for identifying situations in which a seemingly static variable may take on unbounded values through a loop in the program.

The intuition behind our development of loop dependences is as follows: If a program variable is built up under dynamic control, the program must have two properties: There must be a loop in the control structure of the program (through jump statements, explicit loops, recursive functions, or a combination of these) that is controlled in part by a dynamic predicate, and the variable must be updated using its previous values within this loop. The latter condition is represented by the presence of a flow dependence cycle in the dependence graph of the program. We introduce loop dependences from predicates to flow dependence cycles: Roughly, a loop dependence from a predicate p to a flow cycle c indicates that the behaviour of p may determine whether the cycle c is traversed during computation. In other words, the predicate may control how many times the loop in the control structure of the program that gives rise to the flow dependence cycle is iterated. Therefore, a loop dependence from a dynamic predicate to a flow cycle can be used to conservatively represent the phenomenon we are interested in tracking, namely static computation under dynamic control.

Unlike the PRG and SRG representations defined earlier, the loop dependence graph does not have a dataflow semantics that makes the role of the dependences in the program explicit in the semantics. As a result, we cannot make an argument for the conditional safety of the BTA algorithm described in this chapter using the LDG semantics. Instead, we rely on a structural argument, based on our characterization of static-infinite computation, and the definition of loop dependence.

The BTA algorithm defined in this chapter has the advantage that it can handle programs with arbitrary control structures. Since we are interested in developing a safe BTA for C programs, we must also extend our analysis to handle pointer variables. Our approach to this problem is as follows: Pointer variables affect the flow dependences in a program, not the control dependences or loop dependences. Therefore, we follow the approach used by *cmix* [And94], which is a prototype partial evaluator for C programs. *cmix* employs a congruence-based BTA algorithm, which ignores dynamic control, and therefore provides no termination guarantee. However, it uses a pointer analysis algorithm [And93] that conservatively identifies aliasing relationships. We employ the same algorithm, and use its results to conservatively add flow dependence edges to the LDG. Thus, we are able to account for the presence of pointer variables in a manner that is orthogonal to our use of loop dependences.

We have implemented our loop dependence algorithm for C programs, and have tested it on several programs that have been used to measure the performance of *cmix*. Our results show that by using loop dependences, we are able to provide a termination guarantee for partial evaluation in the presence of dynamic control, without compromising the ability of the analysis to identify static behaviour. In particular, when our loop dependence BTA is combined with a simple precision analysis, described later in this chapter, it is able to identify as static every variable in the test suite of programs that is also identified as static by the traditional congruence-based BTA of *cmix*. These results suggest that control dependences may be used to provide a safety guarantee for partial evaluation, without resulting in unduly conservative BTA algorithms. In fact, as we show later in this chapter, our loop dependence BTA algorithm is able to process programs that have been transformed using "The Trick," by using simple user annotations.

5.1 Loop dependence

As we have shown in the previous chapters, a partial evaluator may fail to terminate because a variable that is built up from static values under dynamic control is treated as static. The goal of all of our BTA algorithms is to conservatively identify the variables in a program that exhibit this behaviour. In this section, we define a new kind of dependence termed "loop dependence," which enables us to identify such variables in the presence of arbitrary control flow.

To motivate loop dependences, we first recap the intuition behind our approach for PRG and SRG programs: A program variable that is re-defined within a loop in the program is potentially built up under dynamic control, if the loop predicate is dynamic, and if the variable is built up from its own previous values in the loop. In the case of PRG programs, every loop is explicitly represented by a while loop. In our BTA algorithms, we approximate the above condition by treating all ϕ_{enter} nodes controlled by dynamic predicates as dynamic. In the case of SRG programs, we follow the same approach, but we identify loops that result from recursive procedure calls as well.

In the case of programs with arbitrary control flow, therefore, we are interested in identifying the loops in the control structure of the program. These loops could be contained in a single procedure, or they could span several procedures in the program. For each such loop, we conservatively identify the predicates that control the exit conditions on the loop, and the variables whose values are built up within the loop. We introduce loop dependences between the predicates controlling the exit conditions on the loop and the variables whose values are built up in the loop, represented by flow dependence cycles. Intuitively, a loop dependence can be regarded as a control dependence from a predicate to a flow dependence cycle, as opposed to a control dependence to a program statement.

More formally, we consider the language of programs represented by PRGs, augmented with procedure definitions and calls, and **goto**, **return**, **break** and **continue** statements, with the usual meaning. Parameter passing is by value-result, and only scalar variables are permitted. In a later section of this chapter, we extend the language to include pointer variables.

We first characterize a flow dependence cycle. The definition of flow dependences from Chapter 2 says that there is a flow dependence from a vertex u that defines a variable x to a vertex v that uses x iff there is a path in the control flow graph of the program from u to v such that x is not defined at any vertex along the path except u and v. There may be multiple paths from u to v in the control flow of the program along which this property holds, all of which may be represented by a single flow dependence edge from u to v. A flow dependence cycle consists of a sequence of flow dependence edges, some of which may be inter-procedural parameter edges.

Definition 23 A flow dependence cycle c is a sequence of edges $[e_1 \ldots e_i \ldots e_n]$ such that each edge e_i is a flow dependence edge from vertex v_{i-1} in the extended CFG G of program P to vertex v_i in G, and $v_0 = v_n$. □

In the definition above, the extended CFG of a program is composed of the CFGs of all the procedures in the program, augmented with call and return edges representing procedure calls.

Consider a vertex v such that v is in a flow dependence cycle. By the definition above, there is at least one path p in the extended CFG such that v depends on itself along p. The flow dependence cycle conservatively indicates that each time control proceeds from v back to v along p during the execution of the program, the variable assigned at v takes a new value, which may be derived from its value on the previous occasion that control reached v. In such an instance, the values taken on by the variable may be built up on each occasion that control passes around the loop represented by p during program execution.

We are interested in identifying the predicates that determine whether the flow dependence cycle c from v to itself is realised during program execution. Consider a flow dependence edge e that is part of the cycle. If a predicate u controls the execution of e, then u may also control the execution of c. This is the basis for our definition of loop dependence. Predicate u controls flow edge e if the behaviour of u (*i.e.* the value computed at u) determines whether execution flows in such a way that the dependence represented by e is realised. This is formalized in Definition 24 below.

Definition 24 Let c be a flow dependence cycle of edges $[e_1 \ldots e_i \ldots e_n]$, and let u be a predicate vertex in the extended CFG G of program P. There is a *loop dependence* from u to c iff there is an edge e_i in c from vertex v_{i-1} that defines variable x to vertex v_i that uses x for which both of the following properties hold:

(a) there is a path p from v_{i-1} to v_i via u in G such that x is not re-defined at any vertex in p except v_{i-1} and v_i, and

(b) there is an edge e from u in G such that if every edge from u other than e is removed from G, there is no flow dependence from v_{i-1} to v_i. □

In the definition above, property (a) says that there is at least one branch direction from u along which flow cycle c is present, while property (b) says that there is at least one other branch direction from u along which c cannot be realised. In other words, the execution of c depends on the behaviour of u. Hence, there is a loop dependence from u to c.

Example 10 The loop dependences in the power program, modified to use jump statements, are shown in Figure 31. The loop dependences from the if predicate that controls the exit condition of the loop in the program are closely related to the control dependences from the while loop predicate in the power program from Figure 1 in Chapter 1. The remaining control dependences from Figure 1 are not replaced with loop dependences. This is because they do not control any flow dependence cycles in the program. □

5.2 Constructing loop dependences

In the previous section, we outlined a new kind of dependence called loop dependence, which is useful in designing BTA algorithms for programs with arbitrary control flow. In this section, we describe an algorithm that constructs loop dependences by solving several dataflow problems on the extended CFG. We assume that flow dependences are constructed in the usual way, resulting in a system dependence graph representation of the program, as described in the previous chapter.

(a)

```
float power ( float x, int n ) {
  float a = 1.0;
  L : if ( n > 0 ) {
    n = n − 1;
    a = a ∗ x;
    goto L;
  }
  return a;
}
```

(b)

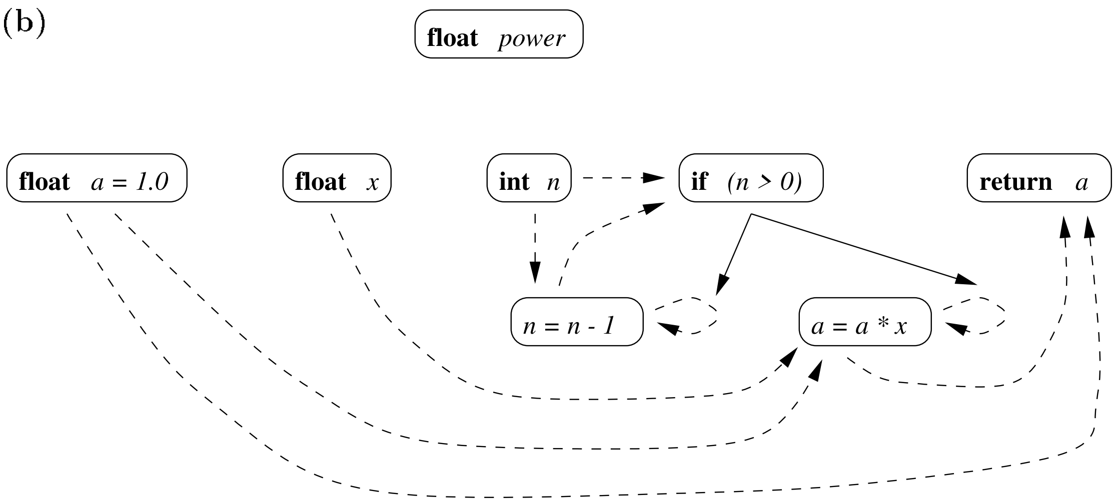


Figure 31: The loop dependences in the power program.
The power function, written with goto statements, is shown in figure (a) above. The flow dependences and loop dependences for *power* are shown in figure (b) above. Flow dependence edges are shown as dashed lines, while loop dependence edges are shown as solid lines.

Loop dependences as defined in the previous section connect a program's predicates to flow dependence cycles. Unfortunately, the number of simple flow dependence cycles in a program is exponential in the size of the program text, in the worst case. Therefore, we identify an approximation of the precise loop dependence property from Definition 24. In particular, we build loop dependences from predicates to strongly connected components in the flow dependence graph. In other words, if a predicate controls the behaviour of a flow dependence edge, we introduce loop dependences to every strongly connected component that contains the given flow dependence edge. Because we limit the precision of loop dependences in this manner, we can introduce a loop dependence edge from a predicate to the target vertex of the flow dependence edge that it controls, rather than the flow dependence edge or flow dependence cycle.

Our algorithm for constructing loop dependences is shown as Algorithm 1 below. The algorithm has three steps:

- In the first step, we identify strongly connected components in the flow dependence sub-graph of the SDG, which includes intra-procedural and inter-procedural flow dependence edges, using a standard algorithm from [CLR90, pp. 488-493]. This operation is linear in the size of the flow dependence graph, which is at most quadratic in the size of the program.

- In the second step, we identify predicates that control intra-procedural flow dependence edges. We do this by solving two dataflow problems on every CFG. First, for every vertex that defines a variable, we identify the predicates that are reachable from the definition without the variable being re-defined. This information can be extracted directly from the reaching definitions sets at CFG vertices. The worst-case complexity of this step is therefore quadratic in the size of the given procedure. Next, for every predicate and every variable, we identify the uses of the variable that are reachable along at least one outgoing CFG edge from the predicate without the variable being re-defined, and not reachable along at least one outgoing CFG edge from the predicate without the variable being re-defined. If the procedure has only if predicates or while predicates, which have exactly two outgoing CFG edges, this operation requires solving two reachability problems on the CFG for every predicate. Its worst case complexity is therefore quadratic in the size of the given procedure. If the procedure includes switch predicates with multiple branch possibilities, the worst case complexity of this operation is cubic in the size of the procedure, because it requires solving a linear number of reachability problems on the CFG for every predicate. We store the predicate information at definitions and uses in bit-vectors to obtain constant-time access for the final step of the algorithm. If the source of a flow dependence edge includes a predicate p in its set of predicates that are reachable from the previous step, and if the target of the edge includes p in its set of predicates from which it may/may not be reached, then the predicate (conservatively) controls the flow dependence edge, as per Definition 24.

- In the final step, we introduce a loop dependence edge from a predicate to the target vertex of a flow dependence edge it controls, as per the above criteria, if the flow dependence edge connects two vertices that are in the same strongly connected component of the flow dependence sub-graph of the SDG. This operation involves relating flow dependence edges with predicates, and therefore has a worst-case complexity that is cubic in the size of the program. Hence, Algorithm 1 below has a worst-case running time that is cubic in the size of the program.

Algorithm 1 Construct Loop Dependences.

Given a program p with procedures P_i, corresponding CFGs G_i, and SDG G:

1. Identify strongly connected components in the flow dependence sub-graph of G.
2. For each G_i:
   (a) $\forall u, v \in V(G_i), \forall var \in Vars(G_i)$: set $forw(u)[v] = false$, $back(u)[var, v] = false$.
   (b) Compute reaching definitions on G_i.
   (c) $\forall u, v \in V(G_i)$, if v is a predicate vertex and $u \in RDefsIn(v)$ then

      set $forw(u)[v] = true$.

   (d) For each $p \in V(G_i)$ s.t. p is a predicate vertex, for each $var \in Vars(G_i)$:
      (i) Let E be the set of outgoing edges from p in G_i. Remove E from G_i.
      (ii) Set $lhs(p) = var$. $\forall v \in V(G_i)$, set $count(v) = 0$.
      (iii) For each edge e in E:

         Add e to G_i. Compute $ReachingUses(p,var)$.

         $\forall v \in ReachingUses(p, var)$, increment $count(v)$.

         Remove e from G_i.

      (iv) Add all edges in E to G_i. Set $lhs(p) = \epsilon$.
      (v) $\forall v \in V(G_i)$, if $0 < count(v) < outdegree(p)$ then

         set $back(v)[var, p] = true$.

   (e) For each predicate vertex p in G_i, for each flow dependence in G from u in G_i to v in G_i s.t. $scc\#(u) = scc\#(v)$, if $forw(u)[p]$ and $back(v)[lhs(u), p]$ then

      add a loop dependence from edge p to v. □

5.3 The loop dependence graph

The loop dependence graph (LDG) of a program is constructed by augmenting the flow-dependence graph of the program with loop dependences. Therefore, it includes the following kinds of dependence edges: Intra-procedural flow dependences, including loop-carried flow dependences and loop-independent flow dependences, inter-procedural flow dependences linking actual-in vertices with formal-in vertices and formal-out vertices with actual-out vertices, and loop dependences. The LDG does not contain summary edges.

The number of edges in the loop dependence graph is quadratic in the size of the program, in the worst case. However, the cost of building the LDG, using Algorithm 1, is cubic in the size of the program in the worst case. Therefore, the LDG is more expensive to build than a standard dependence graph such as the SDG (with no summary edges). However, it affords greater precision for binding-time analysis in the presence of arbitrary control flow. The LDG for the power program from Figure 31 is shown in Figure 32.

5.4 The Loop-Dependence BTA algorithm

The loop dependence graph defined in the previous section provides a basis for the development of a BTA algorithm that identifies static computations under dynamic control in programs with arbitrary control flow. As is the case with our BTA algorithms for PRG and SRG programs, the Loop-Dependence BTA is a simple reachability operation on the LDG, and can be thought of as a variant of the operation of program slicing on dependence graphs.

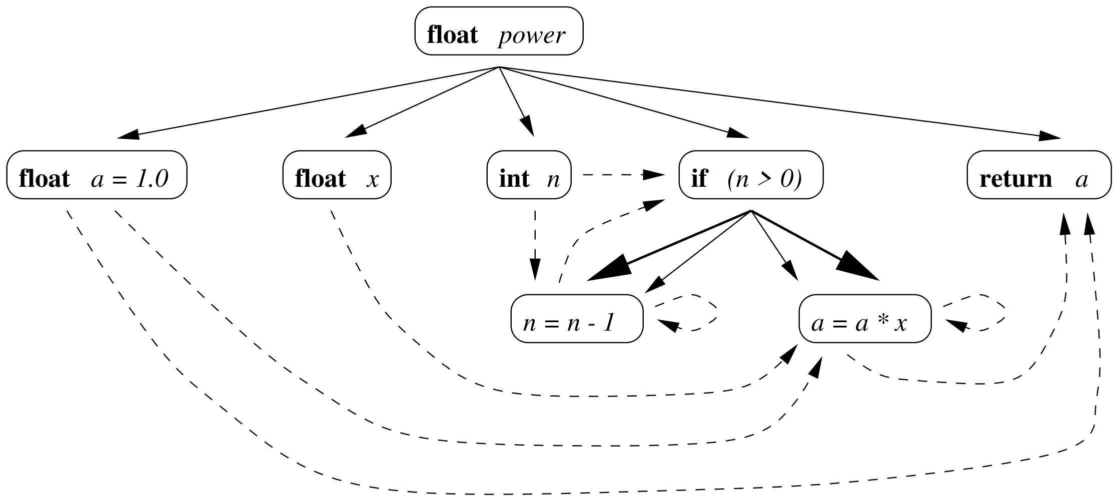


Figure 32: The loop dependence graph of the power program.
In the LDG above, flow dependence edges are shown as dashed lines, control dependence edges are shown as solid lines, and loop dependence edges are shown as thick lines.

Operationally, the algorithm proceeds as follows: Initially, all the vertices in the LDG are marked as static, except for read vertices, which are marked as dynamic, and call and procedure entry vertices, which are not marked. The algorithm follows all flow dependence and loop dependence edges forward from the set of read vertices, marking as dynamic all vertices that are encountered along the way. As is the case with our BTA algorithms for SRG programs described in the previous chapter, the Loop-Dependence BTA is context-insensitive, in the sense that it does not account for calling context precisely. This is because it does not match parameter-in edges with parameter-out edges in flow dependence paths. An example of the application of this BTA is shown in Figure 33.

We have defined the Loop-Dependence BTA operationally, as a reachability operation, rather than as an abstract interpretation of a concrete semantics. This is because

(a)

```
main () {
  read (dyn);
  call outer(dyn);
}

outer (ctr) {
  if ( ctr > 0 ) {
    if ( ctr > 10 ) {
      x = 0;
      L : if ( x < 3 ) {
        x = x + 1;
        goto L;
      }
      call outer(ctr − 1);
    }
  }
}
```

(b)

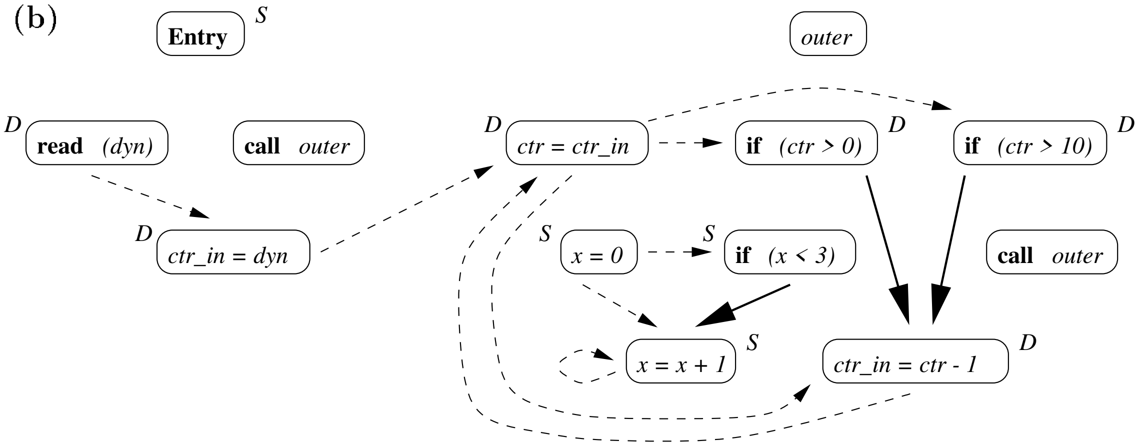


Figure 33: An application of the Loop-Dependence BTA.
The program in (a) above is a modified version of the nested loops program from Figure 16 in Chapter 3. The LDG of the program is shown in (b) above. Control dependences have been omitted to avoid clutter. The results of the Loop-Dependence BTA are shown as markings on every LDG vertex. The inner counting loop is marked static, as expected. Note that the call and procedure entry vertices have not been marked, as they cannot be targets of flow dependence or loop dependence edges. Call vertices are emitted depending on whether the given call is unrolled, as determined by the markings on the parameters of the called procedure. The results of the Loop-Dependence BTA can be converted into markings for program variables using the algorithm in Section 3.4.4 of Chapter 3.

the LDG does not have a dataflow semantics that relates the behaviour of a vertex to that of its dependence predecessors. In particular, it is not clear how ϕ vertices can be added to the SDG to account for the role of jump statements in the program. As a result, we argue the correctness of the Loop-Dependence BTA using the structure of the LDG.

We claim that the Loop-Dependence BTA is conditionally safe. By the definition of Conditionally safe BTA in Definition 12 from Chapter 3, a BTA is conditionally safe if every vertex identified as static by the BTA that fails the BSV property is static-infinite. Accordingly, the correctness of the Loop-Dependence BTA is established by the theorem below.

Theorem 2 If vertex v in the LDG G of program p is marked static by the Loop-Dependence BTA, and v does not satisfy the BSV property, then v is static-infinite.

Proof Sketch. The proof is by induction on the strongly connected components of the sub-graph of the LDG obtained by eliminating control dependence edges and def-order dependence edges. Because the flow and loop dependence sub-graph may not be connected, there may be more than one scc that has no incoming edges from other sccs in the graph. The markings produced by the Loop-Dependence BTA are such that for every scc, either every vertex in the scc is marked static, or every vertex in the scc is marked dynamic. We consider only sccs in which every vertex is marked static by the Loop-Dependence BTA.

Base Case: Scc s has no incoming flow or loop dependence edges from other sccs. Consider a vertex v in s that defines variable x, such that v is not BSV. Because s has no incoming flow edges from other sccs, it must be that x is initialized within s.

Therefore, since v is not BSV, the value of x must be built up in a flow cycle c contained in s. If c is not controlled by any predicates, v must be static-infinite. If c is controlled by one or more predicates p, then consider each such predicate p. By the definition of loop dependence, there must be a loop dependence edge from p to a vertex in c. Therefore, p must be in s. As a result, the variables used in the expression at p are initialized and built up in s. Hence, p must be static. Hence, v must be static-infinite.
Induction Step: Assume that for all sccs s' such that there is a flow dependence or loop dependence from a vertex u in s' to a vertex v in s, the vertices in s' are marked correctly by the Loop-Dependence BTA. We show that it must be the case that the vertices in s are also marked correctly by the Loop-Dependence BTA. Suppose s has an incoming edge from vertex u that is not in s. By assumption, u is marked correctly. Therefore, if u is not BSV and not static-infinite, it must be marked dynamic. By the definition of the Loop-Dependence BTA, every vertex in s must then be marked dynamic. We are left with the case where all incoming edges from outside the scc are from vertices that are static. Then, the proof is identical to that for the base case. $\square$

5.5 Precision analysis

The Loop-Dependence BTA algorithm described in the previous section has the advantage that it uses loop dependences to account for the role of control dependences in a selective manner. In particular, loop dependences represent control dependences that feed values built up in the loops in a program. Therefore, the Loop-Dependence BTA is able to guarantee termination in the presence of dynamic control while differing from the standard congruence-based BTA algorithms only to the extent that loop dependences representing dynamic control are also followed by the analysis.

However, loop dependences are constructed using flow dependences, which have the following drawback: Flow dependences are built using only the defined and used variable sets at CFG vertices. The actual expressions at the vertices are ignored. As a result, a flow dependence cycle may be present from a vertex to itself even though the variable defined at the vertex does not take on new values on every iteration of the program loop represented by the flow dependence cycle. Consider a flow dependence cycle that is controlled by a dynamic predicate. All of the vertices in the cycle will be marked as dynamic by the Loop-Dependence BTA, under the argument that the value of a variable may be built up unboundedly through the loop. However, this may not always be the case. For instance, consider the following assignment statement within a dynamic loop in the structure of a program, where x is an array whose elements are shifted in a loop:

$$x[i] = x[i-1];$$

A dependence analysis that does not look inside the index expressions would add a flow dependence from the assignment to itself, which would result in a loop dependence from the predicate controlling the loop to the assignment. However, the assignment above does not generate new values for x on every iteration of the loop. We term flow cycles such as the one above "grounded flow dependence cycles".

Grounded cycles can be identified by adding a simple criterion to the loop dependence construction algorithm that conservatively determines whether a flow dependence cycle can produce new values on every iteration. We omit the details here.

It is also possible that a flow dependence cycle may generate new values for a

int *bsearch* (**int** x, **intArr** a) {	
int $low = 0, high = 99, mid$;	
while $(low \leq high)$ {	D
$mid = (low + high)/2$;	D
if $(a[mid] < x)$	D
$low = mid + 1$;	D
else if $(a[mid] > x)$	D
$high = mid - 1$;	D
else return mid;	D
}	
return -1;	
}	

Figure 34: The Loop-Dependence BTA applied to the binary search program. The binary search program is shown above. The markings from the Loop-Dependence BTA, given an input division in which parameters x and a are both dynamic, are shown for certain vertices. *intArr* is assumed to be an integer array type.

variable on every iteration, but it may do so in such a manner that the number of iterations of the loop is bounded. For instance, consider one version of the binary search program, shown in Figure 34. Variables *mid*, *low*, and *high* are involved in flow dependence cycles that are controlled by all three predicates in the program, of which the predicates involving parameters a and x are clearly dynamic. As a result, all of the vertices in the loop are marked as dynamic by the Loop-Dependence BTA. This marking appears reasonable, because the predicates involving a and x do in fact determine the iteration count of the program loop that is associated with the flow dependence cycles for *mid*, *low*, and *high*.

However, examination of the expressions in the program loop shows that the number of iterations of the loop is in fact bounded by the initial values of *low* and *high*, because the loop monotonically approaches its exit condition on every iteration. In particular, on every path through the loop, either *low* takes on a new value that is greater than its previous value, or *high* takes on a new value that is smaller than its previous value. We term such loops "grounded loops". Loop dependences from dynamic predicates can be ignored if the flow cycles that are their targets are in grounded loops. Grounded loops can be identified by solving an any-path dataflow problem: If there is any path of control flow through the loop such that the variables in the loop predicate expression do not take on new values given which the loop approaches its exit condition, the loop is conservatively identified as a loop that is not grounded. Given a particular path of control flow through the loop, the condition above can be tested by symbolically evaluating the expressions at vertices along the path, and then testing these expressions to determine whether they approach the exit condition for all possible values of the variables in the expressions.

The analyses described above can be used to improve the precision of the Loop-Dependence BTA as follows: First, the LDG is constructed. Next, groundedness analysis is performed to identify grounded loops and grounded flow cycles. Loop dependence edges to or from grounded vertices are eliminated from the LDG. Finally, the Loop-Dependence BTA is performed on the modified LDG.

5.6 Pointer variables

The Loop-Dependence BTA has the advantage that it can handle programs with arbitrary control structures. Since we are interested in developing a safe BTA for C programs, we must also extend our analysis to handle pointer variables. In this section, we summarize our approach to integrating pointer variables into the Loop-Dependence

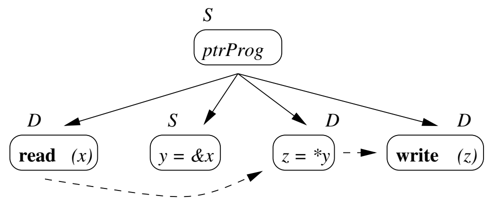
BTA.

Conceptually, pointer variables have the effect of creating aliasing relationships among program variables. Alternatively, every program variable "points-to" zero or more other program variables. A variable x points-to a variable y if the value of x may represent the address of y. We borrow the pointer analysis algorithm of Lars Andersen from [And93], and use it to build "points-to sets" for all the variables in the program. Note that Andersen's pointer analysis is "flow-insensitive," in the sense that a variable has a single points-to set throughout the program. This leads to a loss of precision in the case of pointer variables that point to distinct variables in different areas of the program.

We use the points-to sets for program variables to augment the sets of "defined" and "used" variables at LDG vertices. This transformation has the effect that vertices that appear to assign to a single variable may in fact define multiple variables. Our dataflow algorithms for building flow dependences and loop dependences are extended to handle definition sets with multiple variables in a straightforward manner. Therefore, we are able to incorporate pointer variables in a manner that is orthogonal to our use of loop dependences for providing a termination guarantee. In other words, our BTA algorithm does not depend on the algorithm used to produce defined and used sets at program points. Therefore, any pointer analysis algorithm can be used in place of the algorithm from [And93]. An example of our approach is shown in Figure 35.

In the case of multi-procedure programs, aliasing relationships may span procedure boundaries. This is especially true in C programs, which frequently use pointer-valued parameters to simulate call-by-reference parameters. We approach this problem as follows: Consider a procedure P that transitively calls a procedure Q, such that a

	Used	Defined
void *ptrProg*() {		
int $x, *y, z$;		
read (x);	$\{\}$	$\{x\}$
$y = \&x$;	$\{\}$	$\{y\}$
$z = *y$;	$\{x\}$	$\{z\}$
write (z);	$\{z\}$	$\{\}$
}		



Points-to sets: $x : \{\}$, $y : \{x\}$, $z : \{\}$

Figure 35: Incorporating pointer variables into the Loop-Dependence BTA.
In the program above, pointer variable y points-to variable x. The resulting defined and used sets are shown above, along with the LDG of the program and the results of the Loop-Dependence BTA on the LDG. Notice that variable y is static even though the object it points to is dynamic. This is because the value of y is statically known to be equal to the address of x. During specialization, occurrences of y can be replaced with the static expression "$\&x$".

pointer variable that is dereferenced in Q has a local variable from P in its points-to set. We add an extra parameter to the procedures between P and Q in the call graph, including Q, to represent the local variable of P that is referenced in Q. Parameter-in vertices are added, with appropriate defined and used sets, at appropriate call sites. If the variable is updated through a pointer, parameter-out vertices are added as well. An example of our solution is shown in Figure 36.

Finally, the static or dynamic behaviour of a pointer variable can be thought of on two levels: The pointer itself may be static or dynamic, while the objects it points to may be static or dynamic. Here again, we copy the approach used by Andersen in [And93]. Therefore, our approach to incorporating pointer variables is identical to that of *cmix* [And94], with the exception that we use extra parameters to handle flow dependences induced by aliasing relationships across procedures, whereas *cmix* uses a type-inference engine to deduce these dependences. In addition, the results of pointer

```
void outer() {
  int x;
  read (x);
  call incr(&x);
  write (x);
}

void incr(int *y) {
  *y = *y + 1;
}
```

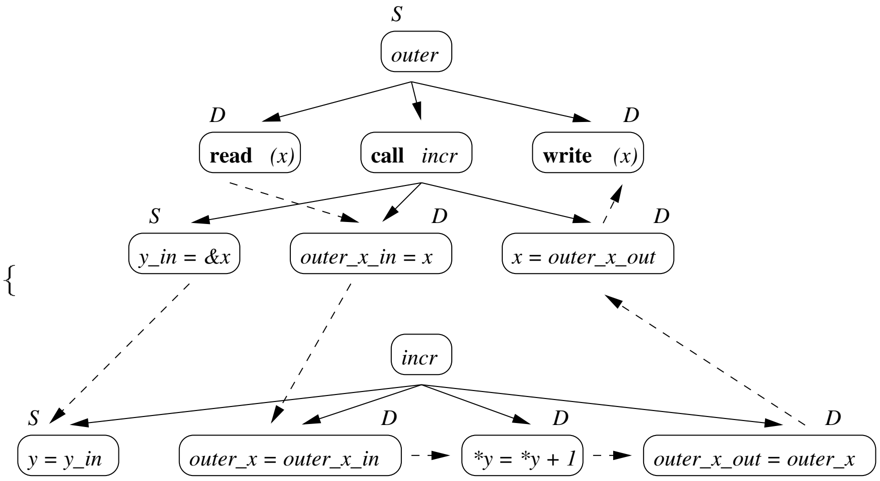


Figure 36: Inter-procedural aliasing through pointer variables.
In the program above, parameter y of procedure *incr* points-to local variable x of procedure *outer*. In the LDG for the program, also shown above, parameter *outer_x* is added to account for the update to x within *incr*.

analysis affect the construction of loop dependences.

5.7 Implementation and experimental results

We have implemented the Loop-Dependence BTA for a subset of C that includes almost all of the features of Ansi C, with the following exceptions: We do not handle separate compilation, setjmp and longjmp function calls, pointer arithmetic, and dynamic memory allocation via calls to *malloc*. We disallow separate compilation because the implementation of the pointer-analysis algorithm we used does not support separate compilation. We do not handle calls to *malloc* because it is not clear how pointers to dynamically allocated storage should be treated in the specialization phase. However, our BTA algorithm is able to process calls to *malloc*. In these respects, we impose the same restrictions that are imposed by *cmix*, the partial evaluator for C programs developed by Lars Andersen at DIKU [And95a]. Finally, we assume that all pointer arithmetic arises from array accesses, because it makes the task of building defined and used sets from points-to sets in the presence of array pointers significantly simpler.

The goal of the work presented in this chapter is a BTA algorithm that provides a termination guarantee without compromising the ability of the analysis to identify static behaviour. Therefore, we have implemented only the first phase of partial evaluation for C programs, *i.e* the BTA phase. In evaluating our results, we use the BTA markings produced by the BTA phase of *cmix*, which uses a standard congruence-based BTA algorithm, as our reference point. Because we have followed the pointer analysis algorithm and programming model of *cmix*, we claim that differences in the results of *cmix* and our implementation can be attributed solely to our use of loop dependences in tracing dynamic behaviour, modulo imprecisions in our implementation.

We have developed our implementation, which we refer to as *Spec* (safe partial evaluation for C), on top of a version of the Wisconsin Program-Slicing Tool developed at the University of Wisconsin [oWM97]. This tool provides an infrastructure for building dependence graph representations of C programs. The system has a front-end for Ansi C programs, and a back-end that generates a system dependence graph, given a set of input C programs. The front-end includes an implementation of Andersen's pointer analysis algorithm. However, the results of this analysis are not transmitted to the back-end of the system.

To this framework, we have added the following components: An algorithm to augment the defined and used sets at CFG vertices using the results of pointer analysis, an algorithm to add parameters to procedures in order to account for inter-procedural

Test Program	Specialized w.r.t.	Terminates?		Identifies All Static Variables?		
		cmix	*Spec*	*cmix*	*Spec-*	*Spec+*
power	base	*No*	*Yes*	*Yes*	*Yes*	*Yes*
bsearch	array bounds	*Yes*	*Yes*	*Yes*	*No*	*Yes*
mix^1	program	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
mix^2	program	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
ray	scene	*Yes*	*Yes*	*Yes*	*No*	*Yes*

Figure 37: Experimental comparison of *cmix* and *Spec*.
In the table above, *Spec-* refers to our implementation of the Loop-Dependence BTA without precision analysis, while *Spec+* includes our precision analysis algorithm. *power* is the power function, *bsearch* refers to several versions of the binary search function, mix^1 and mix^2 are different versions of a specializer for an imperative language, and *ray* is an implementation of ray tracing (approx. 2000 lines of C code) by Peter Andersen at DIKU [And95b].

aliasing information, an algorithm for constructing loop dependences, and a component that identifies grounded flow dependence cycles and grounded loops using a linear constraint solving technique.

We evaluate the performance of the Loop-Dependence BTA as follows: We run the input program through *cmix*, obtaining a division of the program variables into static and dynamic variables. We then run the same program through *Spec*, and we compare the data divisions obtained. We do not report the run-time performance of the specialized programs in each case, using the argument that if two BTA algorithms produce the same division of program variables, they must produce the same residual program, given the same specialization algorithm. Our set of test programs represents the programs for which results of *cmix* have been reported. Our experimental results are presented in Figure 37.

The table from Figure 37 shows that *cmix* does not terminate on the power function, when specialized to a known base value, as reported earlier. This is because the congruence-based BTA of *cmix* ignores dynamic control. Exactly the same problem arises in, for instance, the NORMA interpreter from [JSS85]. We have not included these programs here because we have not obtained the source code for them. The goal of our experiments is to show that the Loop-Dependence BTA can handle the common occurrences of static behaviour. As shown in the table of results, our BTA algorithm identifies all the static variables identified by *cmix* in the specializer mix^1. The more interesting case is mix^2, which is a version of mix^1 that uses The Trick to obtain non-trivial specialization of a specializer, given a known input program. For this program, the Loop-Dependence BTA is able to identify static variables as desired. The binary-search function requires the use of precision analysis, as discussed earlier. Finally, the ray-tracing program *ray* contains one grounded loop and one use of The Trick. These are identified as grounded by the precision analysis. All other parts of the program are identified consistently with *cmix*. The ability of the BTA to process loops produced by The Trick is discussed in detail in the next section.

5.8 Handling The Trick

It has been argued in the literature that dynamic control should be ignored in designing BTA algorithms, because programs that have been transformed using The Trick cannot be processed suitably. In this section, we clarify this statement, and then argue that this is not the case, in particular with the BTA algorithm that has been described in this chapter.

The motivation for The Trick comes from the self-application of partial evaluators, as described by the Futamura projection equations in Chapter 2. The second Futamura projection indicates that a compiler can be generated by applying a partial evaluator to itself, where the static argument to the partial evaluator is an interpreter, and the dynamic argument contains the inputs to the interpreter. The idea is to use the knowledge of the text of the interpreter to simplify the code of the partial evaluator.

Consider a partial evaluator *mix* of the form described by Jones et al. in [JGS93, pp. 85-87]. The specialization component of *mix* has a pending loop, which selects a program point from the pending list and performs specialization actions at that program point. The pending list in turn may be updated by the specialization actions at that program point. For each program point, the specializer examines the statements in the code at the program point and processes the code accordingly. Consider the fragment of code within the pending loop of *mix* below, taken from [JGS93, pp. 91-93]:

$bb = lookup(pp, prog);$

In the code above, *pp* is current program point being processed, *bb* is the basic block at *pp*, and *prog* is a linked list of the basic blocks in the interpreter. *pp* is dynamic, because it is taken from the pending list, which depends on the values for the dynamic parameters to the interpreter. Therefore, *bb*, which is obtained through the lookup on *prog* using *pp*, is also dynamic. This causes the rest of the code within the body of the pending loop to be treated as dynamic, leading to trivial specialization.

However, *pp* always refers to one of the statically known set of program points in *prog*. In other words, *pp* is BSV. This information can be exploited by replacing the code above with the loop shown below:

$pp' = pp_0;$

while $(pp' \neq pp)$ {

$pp' = next(pp', prog);$

}

$bb = lookup(pp', prog);$

In the loop above, pp_0 is the initial program point in the program *prog*, while *next* returns the program point immediately after pp' in the list *prog*.

Given a BTA algorithm that follows only flow dependences, the variable pp' in the loop above is treated as static, even though it is built up under dynamic control. Because pp is BSV, specialization can proceed safely even with pp' marked as static, since the values enumerated for pp' are limited to the program points in *prog*. During specialization, the remaining code in the body of the pending loop is specialized for every value taken by bb, resulting in non-trivial specialization.

In contrast, a BTA algorithm that uses control dependences – even selectively – such as the Loop-Dependence BTA, would identify pp' as dynamic, because it is built up under dynamic control. The result would be trivial specialization.

The transformation described above, in which the lookup using a dynamic pp is replaced by a search loop, is referred to as The Trick. This transformation is done by hand, by a user whose examination of the code has uncovered the fact that the dynamic control variable (pp in the example above) is in fact BSV. The example above has been used to argue against the use of control dependence in BTA algorithms, as follows: In order to properly exploit BSV behaviour, many programs are transformed using The Trick. If a BTA algorithm that accounts for dynamic control is applied to these programs, variables that have been enumerated using The Trick (for example, pp'

in the example above) are treated as dynamic. This eliminates the advantage provided by using The Trick.

Let us re-phrase the argument above: In cases where The Trick is applied, it is convenient to ignore dynamic control, because this leads to better results. Therefore, the model that should be used for BTA is one that does not use control dependence, even though this leads to incorrect results in all the other cases where The Trick is not employed and dynamic control does lead to non-BSV behaviour.

Our claim is that this model is not suitable for partial evaluation. We approach the problem as follows: The key point is that The Trick is a manual transformation, which is not automated. In fact, it relies on the user observing that a dynamic variable in the program is BSV. Consider the lookup loop in the example above. In the terminology of the Loop-Dependence BTA, the loop above is a grounded loop. In other words, there is a loop dependence from a dynamic predicate to a flow dependence cycle, where the flow dependence cycle is in a loop whose iteration count is bounded.

Therefore, we modify The Trick by requiring the user to add a simple annotation to the loop header of the transformed lookup loop, indicating that the loop is grounded. This annotation is used by the precision analysis of the Loop-Dependence BTA to eliminate the loop dependences from the loop predicate to the flow dependence cycles controlled by it. As a result, variables enumerated in the search loops produced by The Trick are treated as static, as desired.

In summary, we have designed a BTA algorithm that guarantees termination of partial evaluation in the presence of dynamic control, without compromising the ability of the analysis to identify static behaviour. As we have argued and demonstrated above, our BTA algorithm is able to suitably process programs that have been transformed using The Trick, by using the notion of grounded loops. Note that the notion of grounded loops can be applied to the Static-Variation BTA algorithms for PRG and SRG programs as well. Therefore, these algorithms are also able to suitably process code produced by The Trick, using the user annotation scheme described above.

The Loop-Dependence BTA has the advantage that it handles a large class of imperative programs. As is the case with all of our BTA algorithms, it provides only a partial termination guarantee. In particular, it guarantees termination in the absence of static-infinite computation. As we argued in the introduction, we believe that this is the most appropriate model for partial evaluation. This is especially true in the case of real imperative programs, which have complex features that make termination analysis of all static loops in a program unduly conservative.

In the next chapter, we explore an alternative approach to the termination problem. We devise a termination analysis algorithm that can be used to provide a termination guarantee for partial evaluation on all programs that belong to a restricted functional language.

Chapter 6

Termination analysis for functional programs

In previous chapters of this thesis, we have tackled the termination problem in partial evaluation by designing BTA algorithms that conservatively identify static behaviour in the presence of dynamic control. In this chapter, we present work on another approach to solving the termination problem. In particular, we present a termination-analysis algorithm that guarantees termination of partial evaluation under all conditions. Our work improves upon previous termination analyses in two respects: Our algorithm is optimistic, whereas previous algorithms are pessimistic, and we use a richer language of shape-modification operators. Both of these differences allow our analysis to obtain greater precision.

In contrast with our BTA algorithms described in previous chapters, termination analyses are limited in the language features and data types they can handle. This is because such analyses must look inside the expressions at program points and reason about the possible shapes of data values. Therefore, our termination analysis operates only on functional programs with list data types, as is the case with previous work in this area. Since the functional programs are essentially a subset of the procedural imperative programs, we can compare the approach described in this chapter with our other BTA algorithms in the following way: The termination analysis of this chapter has the advantage that it guarantees that the partial evaluator will terminate on a program even if the program contains static-infinite computations. However, it sacrifices precision, because every static loop is treated conservatively. Further, only a restricted class of programs can be handled.

In the context of functional programs without explicit looping constructs and infinite data structures, non-terminating partial evaluation results from infinite recursion (or, infinite unfolding). Holst has shown that in programs that manipulate S-expressions or list data it is possible to identify functions that are limited to finite recursion [Hol91]. He identifies parameters that are "in-situ decreasing": An in-situ decreasing parameter of a function f strictly decreases in size on every (recursive) chain of calls from f to f. A function that contains an in-situ decreasing parameter can only call itself a finite number of times before this parameter takes on the value *null*. Hence, such a function can go through only a finite number of levels of recursion. In addition, if the initial or entry values for the parameter are bounded, the function can go through only a bounded number of recursive calls. Thus, even a parameter whose values increase in size on successive recursive calls can be treated as BSV as long as it belongs to such a function (*i.e.*, one that has some *other* parameter that is in-situ-decreasing), and its initial values are bounded.

Glenstrup and Jones have defined a second algorithm that identifies in-situ decreasing parameters [GJ96]. They define a structure, called the parameter dependency graph (we refer to this as the PG to distinguish it from the PDG described in Chapter 2), whose edges denote flow dependences between function parameters. Edges are labeled to indicate their size-changing effects, as in Example 11 below. In this framework, a "size-decreasing path" is a path free of $\uparrow$ edges but containing at least one $\downarrow$ edge. An in-situ decreasing parameter is one for which every path in the PG from the parameter to itself is size decreasing. Such parameters can be identified by solving a simple reachability problem on the PG: A parameter is in-situ decreasing if its vertex in the PG is reachable from itself only via paths that are size-decreasing, and there is at least one path from the vertex to itself. A standard congruent BTA is applied to identify static and dynamic parameters, following which termination analysis is used to re-classify non-BSV static parameters as dynamic.

In this chapter, we characterize BSV behaviour in terms of the in-situ-decreasing and in-situ-increasing properties for parameters of functions. We then use these properties as the basis for a termination-analysis algorithm that generalizes the algorithm of Glenstrup and Jones.

The first difference between our algorithm for identifying BSV parameters and Glenstrup and Jones's algorithm is that our algorithm uses a more precise technique for identifying in-situ-decreasing parameters. Their approach requires that size-decreasing paths must be free of $\uparrow$ edges. In general, size-decreasing paths may include $\uparrow$ edges that are "matched" by $\downarrow$ edges, as is the case for a path in Example 11 below.

Example 11 In the program shown in Figure 38 (a), *eval* is an infix expression evaluator that takes a list of operators (*ops*), a list of values (*vals*), the current value of the expression (*tot*), and an error token (*err*) that it returns if an invalid operator is encountered. Function *checkValid* sets the list of remaining values to *null* if an invalid operator is detected, while *accum* updates the expression value at each step.

On each successive call to *eval*, the operations *cdr*, *cons*, and *cdr* are applied to parameter *vals*. The net effect is that *cdr(vals)* (or *null*) is passed to the next call on *eval*. The graph shown in Figure 38 (b) is a snippet of the PG for the program as defined by Glenstrup and Jones, while the graph in Figure 38 (c) is a snippet of the dependence graph used in our approach. Each graph represents the same cycle from vertex *vals* to itself that is present in the two dependence graphs.

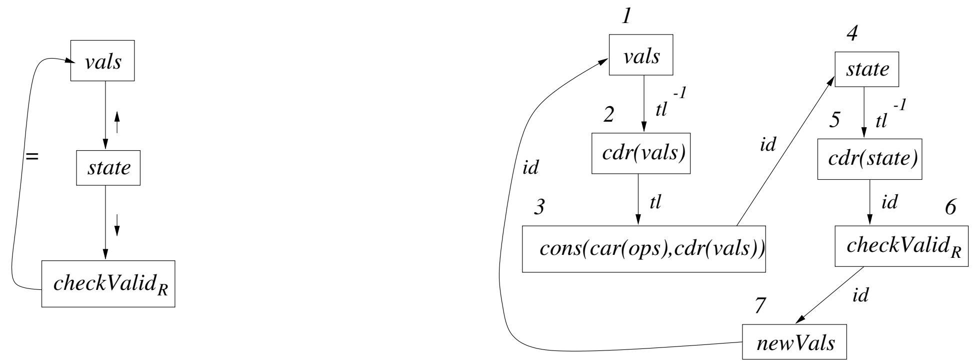
The path from *vals* to itself in (b) contains an $\uparrow$ edge, and is therefore not a size-decreasing path under Glenstrup and Jones's approach. For the corresponding path in (c), the label tl on the edge from v_2 to v_3 indicates that the expression *cons(car(ops),cdr(vals))* has the expression *cdr(vals)* as its tail, while the label tl^{-1} from v_4 to v_5 indicates that the expression *cdr(state)* is obtained by extracting the tail of *state*. This allows our technique to determine that the path is size-decreasing: Because the *cons* operation on *cdr(vals)* (shown by the edge from v_2 to v_3) is "balanced" by the *cdr* operation on *state* (shown by the edge from v_4 to v_5), the net effect is that just a single *cdr* is applied to *vals* (as indicated by the edge from v_1 to v_2). $\square$

To handle such cases, we use CFL-Reachability [Yan90, Rep95], a generalized form of graph reachability. A CFL-Reachability problem is one in which a path is considered to connect two vertices in a graph only if the concatenation of the labels on the edges of the path is a word in a certain context-free language. Thus the path from parameter *vals* to itself in (b) from Figure 38 above has the concatenated label $tl^{-1}.tl.id.tl^{-1}.id.id.id$, which is in the language *decr_path* defined later in the paper (*decr_path* is a context-free language that defines the notion of size-decreasing paths).

Andersen and Holst have also addressed the problem of identifying in-situ-decreasing

```
eval(ops,vals,tot,err)                      checkValid(state)            accum(op,val,totVal)
  if ops = null and vals = null               if invalid(car(state))       if op = '+
    tot                                         null                         totVal + val
  else if vals = null                         else                         else if op = '-
    err                                         cdr(state)                   totVal - val
  else
    eval(cdr(ops),newVals,newTot,err)
    where newVals := checkValid(cons(car(ops),cdr(vals)))
          newTot := accum(car(ops),car(vals),tot)
```

(a)



(b) *(c)*

Figure 38: Parameter dependence graphs for an infix expression evaluator. An infix expression evaluator is shown in figure (a) above. A snippet of the PG for this program is shown in figure (b) above. The corresponding snippet from the dependence graph used in our work is shown in (c) above.

parameters more precisely, in the context of higher-order languages, using a different approach than the one we use. A comparison of our work with their results is given in a later section of this chapter.

The second difference between our algorithm for identifying BSV parameters and Glenstrup and Jones's algorithm is that whereas their algorithm is pessimistic, our algorithm is optimistic. In particular, Glenstrup and Jones treat every parameter as non-BSV until it can be argued to be BSV. Their algorithm proceeds by alternating two BSV-identification steps, as follows: In the first step, a parameter is treated as BSV if all of its acyclic predecessors in the PG are BSV and if no cycles that involve the parameter are size increasing. In the second step, a parameter is identified as BSV if all of its acyclic predecessors are BSV, if one of its sibling parameters (*i.e.*, a parameter associated with the same function) is in-situ-decreasing and has only BSV predecessors, and if all of its sibling predecessors are BSV. This process may result in some BSV parameters never being identified as BSV, as is shown in Example 12 below.

Example 12 In the program shown in Figure 39 (a), *game* is a function that plays a game involving two players, whose holdings are represented by *plyr1* and *plyr2*. *moves* is a list of moves that are executed during the game. Two possible moves are swapping the players' holdings, and moving a holding from one player to another. When the moves are exhausted, the players' holdings are returned as a pair.

The parameter dependency graph for the game program is shown in Figure 39 (b). Given an input division under which initial values for all three parameters are BSV, it is the case that all three parameters are BSV. This is because even though parameters *plyr1* and *plyr2* may increase in size on successive iterations, the number

(a) *game(plyr1,plyr2,moves)*
    if *moves = null*
      cons(plyr1,plyr2)
    else if *car(moves) = 'swap*
      game(plyr2,plyr1,cdr(moves))
    else if *car(moves) = 'capture*
      game(cons(car(plyr2),plyr1),cdr(plyr2),cdr(moves))

(b)

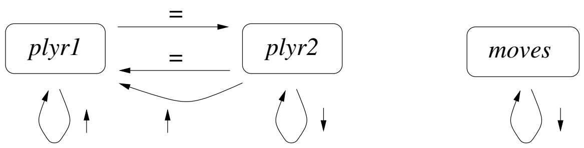


Figure 39: An example of pessimistic BSV identification.
A game playing function is shown in figure (a) above. The PG for this program is shown in figure (b) above.

of iterations is bounded by parameter *moves*. However, the algorithm of Glenstrup and Jones will identify parameters *plyr1* and *plyr2* as non-BSV, because initially, each parameter depends on the other, which is a non-BSV sibling. Under our optimistic approach, in which every parameter is initially BSV, both *plyr1* and *plyr2* are identified as BSV, as desired. □

In contrast, our algorithm is optimistic, and proceeds as follows: Every parameter is initially identified as BSV. A pre-processing phase is applied to identify in-situ-decreasing parameters. All parameters that are in-situ-decreasing are initially classified as "anchors". Parameter markings are then updated at every step, until no further parameters can be updated, as follows: Any parameter that has a possibly size-increasing path to itself and that has no sibling anchors is re-classified as dynamic. Such an update may be propagated in two ways: Any successor of a dynamic parameter is re-classified as dynamic, and any anchor that is re-classified as dynamic is no longer an anchor. This change may result in previously anchored parameters to be re-classified as dynamic, and so on. For the program shown in Example 12 above, this optimistic algorithm will identify *plyr1* and *plyr2* as BSV.

Thus, we are able to improve upon the precision of Glenstrup and Jones's algorithm in two orthogonal directions: (i) By using a richer language of paths, and (ii) by using an optimistic approach.

The contributions of the work presented in this chapter can be summarized as follows:

- We give a formal characterization of *in-situ-decreasing*, *in-situ-increasing*, *entry-BSV*, and *anchoring* parameters.
    - A parameter is *in-situ-decreasing* iff it strictly decreases in size on every recursive call from its associated function to itself.
    - A parameter is *in-situ-increasing* iff it strictly increases in size on some recursive call from its associated function to itself.
    - A parameter is *entry-BSV* iff the set of values taken by the parameter on "entry" or non-recursive calls to its associated function is finite.
    - A parameter is *anchoring* iff the parameter is in-situ-decreasing and entry-BSV.
- We provide an algorithm that identifies a subset of all the in-situ-decreasing parameters and a superset of all the in-situ-increasing parameters in a program.

  - The algorithm uses CFL-Reachability to identify a broader class of strictly size-decreasing paths than the algorithm of Glenstrup and Jones.

- We provide an algorithm that identifies a subset of all the BSV parameters in a program.

  - Our algorithm has a different structure than that of Glenstrup and Jones. Their algorithm treats all vertices as non-BSV until proven otherwise, and uses two BSV identification phases that must be iterated in order to identify as many BSV vertices as possible. On the other hand, our algorithm is optimistic, in the sense that it treats every vertex as BSV until there is evidence (conservatively) that the vertex may not be BSV. By identifying the roots of non-BSV behaviour using the in-situ-increasing property, we are able to propagate enough non-BSV information through the program to produce correct results.
  - Whereas Glenstrup and Jones use a single constant vertex for all the constants in the program, we use a distinct constant vertex for every function that defines a constant. This allows us to distinguish between constants that represent "entry" values for a parameter and those that represent recursive values, leading to greater precision.

It should be pointed out that our algorithm for identifying BSV parameters, like other termination-analysis algorithms, does not use control dependences explicitly. However, it does account for non-BSV behaviour arising from dynamic control, because it implicitly accounts for control dependences by considering both branch possibilities at all predicates. Because the analysis ignores control dependences, however, it cannot distinguish between non-BSV behaviour arising from dynamic control and non-BSV behaviour arising from static-infinite computation, leading to conservative treatment of all static loops.

This chapter is organized as follows: In Section 6.1, we present an overview of the subject functional language and its semantics. In Section 6.2, we use this semantics to define the BSV, in-situ-decreasing, in-situ-increasing, entry-BSV, and anchoring properties. In Section 6.3, we present the augmented parameter dependence graph (APG), an extended form of the parameter dependency graph. In Section 6.4, we define several context-free path languages on the APG. In Section 6.5, we present an algorithm that uses these languages to identify in-situ-decreasing and in-situ-increasing parameters in a program. In Section 6.6, we present an algorithm that uses in-situ-decreasing and in-situ-increasing markings to identify a subset of all the BSV parameters in a program. In Section 6.7, we summarize related work.

6.1 A simple functional language and its semantics

In this section we present a simple, first-order call-by-value functional language, `F`, and the semantics of programs in `F`. The language and its description are taken directly from Glenstrup and Jones's work in [GJ96]; we use the same language so that our results can be compared with previous work, while we reproduce the language description here for completeness. The language `F` is defined by the grammar below:

p	: *Program*	$::=$	$f1(x_1, \ldots, x_{m_1}) \;=\; e_1 \;\;\ldots\;\; fn(z_1, \ldots, z_{m_n}) \;=\; e_n$
e	: *Expression*	$::=$	$se \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid e_1 \textbf{ where } x \;=\; e_2$

$$
\begin{array}{llll}
 & & ::= & fi(se_1, \ldots, se_{m_i}) \\
se & : \textit{SimpleExpression} & ::= & be \mid basefcn(be_1, \ldots, be_k) \\
be & : \textit{BasicExpression} & ::= & x \mid constant \\
x,\, z & : \textit{Variable} & &
\end{array}
$$

Base functions consist of *cons*, *car* and *cdr*, all of which have the usual meanings. Constants may be list constants or integers. $f1$ (or *main*) is the main function, and is not called anywhere in the program. We follow Glenstrup and Jones in treating the *tail recursive* subset of `F` for ease of presentation.

6.1.1 Transition sequences

A *state* is a pair $(f, \vec{v})$ with f defined in p, where $\vec{v}$ is a shorthand for $(v_1, \ldots, v_n)$, $n = arity(f)$, and $\vec{v}_i$ is the ith component of a tuple. Given a value set V, *call-free evaluation* of expression e is defined as:

$$
[\![e]\!]_0 \vec{v} = \begin{cases} w, & \text{if } e\text{'s value on } \vec{v} \text{ is computed without making any function calls.} \\ (g, \vec{w}), & \text{if } e\text{'s value on } \vec{v} \text{ is } g\text{'s value on } \vec{w}. \end{cases}
$$

A single step *state transition*, written $(f, \vec{v}) \rightarrow (g, \vec{w})$, occurs if p contains $f\vec{x} = e$, and $[\![e]\!]_0 \vec{v} = (g, \vec{w})$. Total evaluation of an expression is then defined as:

$$
[\![e]\!] \vec{v} = \begin{cases} v, & \text{if } e'\text{s value on } \vec{v} \text{ is computed without making any function calls.} \\ [\![body(g)]\!] \vec{w}, & \text{if } [\![e]\!]_0 \vec{v} = (g, \vec{w}) \end{cases}
$$

A multi-step *transition sequence* $\tau = [(f^1, \vec{v}^1), \ldots, (f^k, \vec{v}^k)]$ is obtained by composing several state transitions $(f^1, \vec{v}^1) \rightarrow (f^2, \vec{v}^2)$, $(f^2, \vec{v}^2) \rightarrow (f^3, \vec{v}^3)$, etc. Every recursive call from a function f to itself is associated with a transition sequence of the form

$[(f, \vec{v}^1), \ldots, (f, \vec{v}^k)]$ where $\vec{v}^1$ is the vector of parameter values with which the caller f is called, and $\vec{v}^k$ is the vector of parameter values with which the callee f is called.

6.1.2 Call paths

We can approximate the transition sequences from the concrete semantics with the abstract *call path* constructs defined by Glenstrup and Jones in [GJ96]. A call path is an abstraction of a transition sequence where every vector of values is replaced by a vector of syntactic expressions. More precisely, a *call path* of length $k-1$ from a function f^1 to a function f^k is a sequence $\pi = [(f^1, \vec{x}), (f^2, \vec{e}^2), \ldots, (f^k, \vec{e}^k)]$, where p (assumed to be tail recursive) contains definitions $f^{i-1}\vec{y} = \ldots f^i\vec{e}^i \ldots$ for $2 \leq i \leq k$. $\vec{e}^2$ is a vector of arguments for f^2 obtained by unfolding the call from f^1 to f^2 without doing any computation, and is expressed in terms of $\vec{x}$. Similarly, $\vec{e}^k$ is obtained by unfolding the calls from f^1 to f^2 … f^{k-1} to f^k without doing any computation, and is expressed as a function of $\vec{x}$. Argument $\vec{e}^k_j$ is said to "depend" on argument $\vec{x}_i$ iff the "simplified" expression for $\vec{e}^k_j$ in terms of $\vec{x}$ contains the symbol $\vec{x}_i$. An expression is simplified by applying the following simplification rules to the expression: $car(x, ?) = x$, and $cdr(?, x) = x$.

We define the size operators $\ll$ and $\lesssim$ on expressions as follows: Given two expressions e_1 and e_2, $e_1 \ll e_2$ iff $\forall \vec{v}$ s.t. $[\![e_1]\!]\vec{v}$ and $[\![e_2]\!]\vec{v}$ are well defined : $[\![e_1]\!]\vec{v} < [\![e_2]\!]\vec{v}$, and $e_1 \lesssim e_2$ iff $\forall \vec{v}$ s.t. $[\![e_1]\!]\vec{v}$ and $[\![e_2]\!]\vec{v}$ are well defined : $[\![e_1]\!]\vec{v} \leq [\![e_2]\!]\vec{v}$, where $<$ is the "proper-substructure-of" relation on S-expressions, and $\leq$ is the substructure relation on S-expressions. In particular, we will often be interested in whether the relations $e \ll x$ and $e \lesssim x$ hold between an expression e and (the degenerate) expression

x.

Example 13 In the program from Example 11, one call path that represents a possible call from *eval* to itself is the path $[(eval, [o, v, t, e]), (eval, [cdr(o), cdr(v), t + car(v), e])]$. Parameter *tot* ($t$ for short) depends on parameters *tot* (t) and *vals* (v) because the expression $t + car(v)$ contains the symbols t and v. □

In general, every real computation or transition sequence is represented by some call path, although the converse does not hold. Hence, we can use the abstract semantics of call paths to define the properties of interest in termination analysis, with the guarantee that if a parameter satisfies a property defined in terms of the abstract semantics, the property holds over all real computations.

6.2 Semantics of BSV behaviour

In this section, we are concerned with semantics and, in particular, semantic properties of parameters to functions. We first define the BSV and "entry-BSV" properties in terms of the transition sequence semantics. We then define two related properties – "in-situ-decreasing", and "in-situ-increasing". In later sections, we will develop static-analysis algorithms that (safely) identify subsets (or supersets) of the parameters that possess these two properties. These algorithms are subroutines of our algorithm for identifying (a subset of the) parameters with the BSV property. We define the "anchoring" property and show how this property can be used to establish the BSV behaviour of in-situ-increasing parameters.

A parameter is BSV if the set of all the different values taken by the parameter, given fixed static inputs to *main*, is finite. We borrow the definition used by Glenstrup and Jones in [GJ96] as Definition 25 below.

Definition 25 A parameter f_j of function f in p is *bounded-static-varying (BSV)* iff for every static input $\vec{v}_s^1 \in \vec{V}_s$, $\{\ \vec{v}_j^k \mid \exists\ \tau = [(f^1, [\vec{v}_s^1, \vec{v}_d^1]), \ldots, (f^k, \vec{v}^k)]$ s.t. $\vec{v}_d^1 \in \vec{V}_d$, $f^1 = main,\ f^k = f\ \}$ is a finite set. □

In general, every function other than *main* may be called either through a recursive call, or through an entry call. (We made the same distinction in our discussion of SRG programs in Chapter 4.) An entry call is one in which there is no previous invocation of the function on the stack. In terms of transition sequences, an entry call is represented by a sequence from *main* to the function with no other instances of the function in the sequence. For a parameter of a recursive function, its values on entry calls are termed "entry values". We say a parameter is "entry-BSV" if this set of values is finite, given fixed static inputs to *main*.

Definition 26 A parameter f_j of function f in p is *entry bounded-static-varying (entry-BSV)* iff for every static input $\vec{v}_s^1 \in \vec{V}_s$, $\{\ \vec{v}_j^k \mid \exists\ \tau = [(f^1, [\vec{v}_s^1, \vec{v}_d^1]), \ldots, (f^k, \vec{v}^k)]$ s.t. $\vec{v}_d^1 \in \vec{V}_d$, $f^1 = main$, $f^k = f$ and $\forall i \in (2, k-1)\ f^i \neq f\ \}$ is a finite set. □

A parameter f_j that is identified as static by a congruent BTA may exhibit non-BSV behaviour if it depends on a parameter g_i such that parameter g_i takes larger values on successive recursive calls to its associated function g, and function g goes through an unbounded number of recursive calls. However, if function g has another parameter g_k such that g_k is in-situ-decreasing and the entry values of g_k are bounded, then parameter g_i may be BSV even though it takes larger values on successive recursive calls.

We formalize this idea by first characterizing in-situ-decreasing behaviour in terms of the abstract call paths defined in the previous section. A parameter of a recursive function is in-situ-decreasing if its values strictly decrease in size on every recursive call to the function from itself, as in Definition 27 below.

Definition 27 A parameter f_j of function f in p is *in-situ-decreasing* if for every call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ where $f^1 = f^k = f$, $\vec{e}^k_j \ll \vec{x}_j$. □

Since every real computation is represented by a call path, it follows that any parameter that satisfies the property above must take successively smaller values on all recursive calls to its associated function. The property above is stricter than required, because every call path may not correspond to any real computation. However, it is useful because call paths are closely related to the edges in the dependence graph that we use as the basis for our termination-analysis algorithm. Thus, the set of parameters that satisfy the property above is in general a subset of the set of semantically in-situ-decreasing parameters.

A parameter f_j that has the in-situ-decreasing property as defined above limits its associated function f to finite recursion. In addition, if the set of entry values for the parameter is finite, the parameter places a bound on the number of levels of recursion the function can go through. Thus, for every static input, there is a number n such that recursions of f always "bottom-out" after at most n levels. We refer to such a parameter as "anchoring":

Definition 28 A parameter f_j of function f in p is *anchoring* iff f_j is in-situ-decreasing and f_j is entry-BSV. □

From Definitions 25 and 26 above, it follows that a parameter that is BSV must also be entry-BSV. Conversely, a parameter that is entry-BSV and in-situ-decreasing must also be BSV. This leads to the lemma below:

Lemma 29 A parameter f_j of function f in p is anchoring iff f_j is in-situ-decreasing and f_j is BSV. □

As discussed earlier in this section, the "seeds" of non-BSV behaviour in static parameters (as identified by a congruent BTA) are parameters that build up their values on successive recursive calls to their associated functions. We say that a parameter is "in-situ-increasing" (ISI) if it takes larger values on recursive calls to its associated function, and if the larger values on successive recursive calls are built (at least partly) from previous values of the same parameter. This concept is formalized in Definition 30 below.

Definition 30 A parameter f_j of function f in p is *in-situ-increasing* if there is a call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ where $f^1 = f^k = f$, such that $\vec{e}^k_j \leq\not\leq \vec{x}_j$ and $\vec{e}^k_j$ depends on $\vec{x}_j$. □

Once again, the characterization of in-situ-increasing behaviour in terms of call paths is more conservative than a characterization of the property in terms of transition sequences, since a call path may not correspond to any transition sequence, while every transition sequence is conservatively represented by some call path. Thus, the set of in-situ-increasing parameters according to Definition 30 is in general a superset of the set of semantically in-situ-increasing parameters.

Although no parameter can be both in-situ-decreasing and in-situ-increasing, a parameter may satisfy neither property. The relationship between the two properties can be expressed as follows: If a parameter that is identified as static by a congruent BTA is not BSV, and if the parameter is entry-BSV and does not depend on any non-BSV sibling parameters or recursive predecessors, then it must be that the parameter has no siblings that are anchoring. In other words, even if a parameter is in-situ-increasing, it will satisfy the BSV property as long as it is entry-BSV, it depends only on other BSV parameters, and if it is "anchored" by one or more siblings that are anchoring. This idea is formalized in Lemma 31 below.

Lemma 31 If parameter f_j of function f in p is not BSV and f_j does not depend on any dynamic parameters, then one of the following must hold:

(a) f_j is not entry-BSV, or

(b) f_j depends on a sibling parameter f_i that is not BSV, or

(c) f_j depends on a parameter g_i that is not BSV, or

(d) f_j is in-situ-increasing, and f_j has no sibling f_i such that f_i is anchoring. □

The lemma above expresses the intuition behind both the algorithm of Glenstrup and Jones, which is pessimistic as it initially treats all parameters as non-BSV, and our algorithm, which is optimistic as it treats every parameter as BSV and then re-classifies some parameters as non-BSV. The lemma is a re-statement of Theorem 8 from [GJ96].

6.3 The augmented parameter dependence graph

In this section we present the augmented parameter dependence graph (APG), an extension of Glenstrup and Jones' parameter dependency graph, in which flow dependences between parameters are captured through edges in the graph. The APG differs from the parameter dependency graph defined by Glenstrup and Jones in four ways:

- An APG includes nodes for every intermediate expression in the program,
- An APG may have multiple constant vertices,
- Flow edges in the APG are labeled differently, and
- The call graph is embedded in the APG.

Control dependences are not represented explicitly. This is because, as we argued earlier in this chapter, termination-analysis algorithms account for the effect of dynamic control implicitly, by considering all possible branch directions at predicates.

Formally, the *augmented parameter dependence graph* for a 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 *header vertex* and *return-value vertex* for each function in p, one *constant vertex* c_f for every function f in p whose body includes one or more constants (we say that f "encloses" c_f), a vertex for every intermediate expression in the body of any function in p, a vertex for each variable in p, and a vertex for every parameter of a function in p. $E(G)$ includes flow edges and control edges (these control edges are distinct from the control dependence edges in program dependence graphs). Control edges in $E(G)$ are a superset of the call edges in the call graph of p: A control edge exists from f to g iff the expression body of f contains a call on g, or if g is a constant vertex that is enclosed by f. Every control edge has the label *control*. Flow edges in $E(G)$ represent standard flow dependence relationships between vertices. (A flow edge exists from u to v if the expression at v is defined in terms of the expression at u.) Every flow edge in G has a label from the set $\{id, hd, tl, hd^{-1}, tl^{-1}\}$, where the label on the edge is determined as follows: Edge e from vertex u to vertex v has the label l where:

$l = id$	if v is a parameter vertex or if v is a return-value vertex
$l = hd$	if $v = cons(u, w)$
$l = tl$	if $v = cons(w, u)$
$l = hd^{-1}$	if $v = car(u)$
$l = tl^{-1}$	if $v = cdr(u)$

Essentially, it is the flow-edge labels that distinguish APGs from parameter dependency graphs.

Example 14 The APG for the program from Example 11 is shown in Figure 40. Consider the path in the graph from vertex *vals* to itself with concatenated label $tl^{-1}.tl.id.tl^{-1}.id.id.id$. This path arises from a recursive call on *eval* from *eval* in which the value taken by *vals* at the callee is the tail of the value of *vals* at the caller. □

The number of vertices in the APG G of a program p is bounded by $\mathcal{O}(F + P + Var + Ops)$, where F is the number of functions in p, P is the number of parameters in p, Var is the number of *where* variables in p, and Ops is the number of cons/car/cdr operations in p. The number of edges in G is bounded by $\mathcal{O}(F^2 + (P + Var)^2 + Ops)$.

6.4 Paths in the APG

In Section 6.2, we gave a semantic characterization of in-situ-decreasing parameters and in-situ-increasing parameters. Our purpose in this section is to characterize a subset of

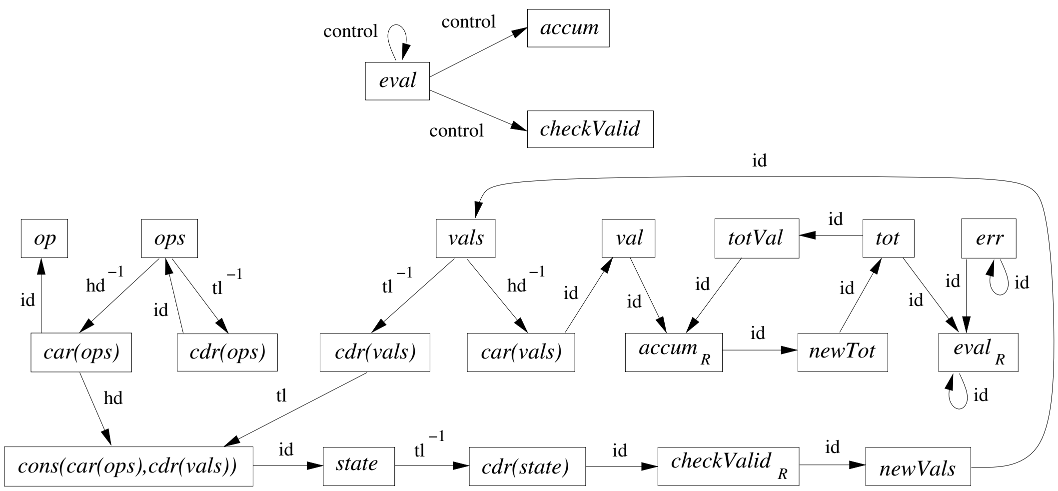


Figure 40: The APG for an infix expression evaluator.
The APG for the infix expression evaluator from Example 11 is shown above. Edges in the APG have labels from the set $\{id, hd, tl, hd^{-1}, tl^{-1}\}$, as described earlier.

the in-situ-decreasing parameters and a superset of the in-situ-increasing parameters in a program, in terms of properties of the augmented parameter dependence graph. We are able to do this because the paths in the APG of a program are directly related to the dependence relationships in the calls paths of the program.

However, not every path in the APG represents a dependence between expressions. For instance, the following example illustrates that an arbitrary path of flow edges from a vertex u to a vertex v in the APG may represent a "false" dependence because the values at u and v have no common substructure (later in this section, we show how "false" dependences can be identified):

Example 15 In the APG for the program shown in Example 11, the path from vertex *ops* to vertex *vals* with label $hd^{-1}.hd.id.tl^{-1}.id.id.id$ suggests that the value of *vals* depends on the value of *ops*. However, the path represents a "false" dependence, because the tl^{-1} (or *cdr*) operation on parameter *state* extracts its tail, whereas the *hd* (or *cons*) operation on $car(ops)$ places $car(ops)$ in the the head of *state*. Similarly, the path from *ops* to *vals* with label $tl^{-1}.id.hd^{-1}.hd.id.tl^{-1}.id.id.id$ represents a "false" dependence. □

We use the presence or absence of certain kinds of paths in the APG to determine whether given parameters in the program satisfy the in-situ-decreasing and in-situ-increasing properties. We do this by solving several path problems in which a path is considered to connect two vertices if and only if the concatenation of the labels on the edges of the path is a word in a certain context-free language.

Definition 32 (Context-Free-Language Reachability; CFL-Reachability) Let L be a context-free language over alphabet Σ, and let G be a graph whose edges are labeled with members of Σ. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is a member of L. The *all-pairs L-path problem* is to determine all pairs of vertices $v_1, v_2 \in V(G)$ such that there exists an *L-path* in G from v_1 to v_2. The *source-target L-path problem* is to determine whether there exists an *L-path* in G from a given source v_1 to a given target v_2. □

Ordinary reachability (transitive closure) is a degenerate case of CFL-Reachability: Let all edges of a graph be labeled with the letter e; transitive closure is the all-pairs e^*-path problem. More general instances of CFL-Reachability are useful for focusing on certain paths of interest. By choosing an appropriate language L, we are able to enforce certain types of restrictions on when two vertices are considered to be "connected" (beyond just "connected by a sequence of edges", as one has with ordinary reachability).

CFL-Reachability problems can be solved using a dynamic-programming algorithm. (The algorithm can be thought of as a generalization of the CYK algorithm for context-free recognition [Kas65, You67].) There is a general result that all CFL-Reachability problems can be solved in time cubic in the number of vertices in the graph [Yan90, MR97]. (This holds even if the context-free language is specified with an ambiguous grammar.) Because the number of vertices in the APG for a program p is bounded by $\mathcal{O}(F + P + Var + Ops)$, the problem of identifying whether an L-path exists from every vertex in the APG to every other vertex (the all-pairs L-path problem) can be solved in time $\mathcal{O}((F + P + Var + Ops)^3)$.

Every flow edge from a vertex u in the APG to a vertex v in the APG has a label from the alphabet $\{id, hd, tl, hd^{-1}, tl^{-1}\}$, indicating the relationship between the expressions at u and v along a certain execution path. Similarly, the concatenated label on a path from vertex u to vertex v indicates the relationship between the structures of the expressions at u and v along a certain execution path. Therefore, we can define context-free languages such that all paths whose labels are in a given language L relate the values at their source and target vertices in some particular manner. In particular, we are interested in three kinds of relationships between vertices in the APG: Size-decreasing, possibly size-increasing, and equal. In addition, for some purposes we may want to exclude the empty path from consideration, and so sometimes we are interested in languages that do not derive ε, the empty path. We define the following context-free languages, which are described in detail below and summarized in Figure 41:

- The language L_1 represents paths in which each hd (resp. tl) is balanced by a hd^{-1} (resp. tl^{-1}); these paths correspond to values transmitted along execution paths in which each cons operation (which gives rise to a hd or tl label on an edge in the path) is eventually "taken apart" by a car (or hd^{-1}) or cdr (or tl^{-1}) operation:

$$
\begin{array}{lll}
L_1\!: & eq\_path^{\varepsilon} & \rightarrow \mathbf{hd}\;\; eq\_path^{\varepsilon}\;\; \mathbf{hd^{-1}}\;\; eq\_path^{\varepsilon} \\
 & eq\_path^{\varepsilon} & \rightarrow \mathbf{tl}\;\; eq\_path^{\varepsilon}\;\; \mathbf{tl^{-1}}\;\; eq\_path^{\varepsilon} \\
 & eq\_path^{\varepsilon} & \rightarrow \mathbf{id}\;\; eq\_path^{\varepsilon} \\
 & eq\_path^{\varepsilon} & \rightarrow \varepsilon
\end{array}
$$

- The language L_2 is similar to L_1, but excludes empty paths:

$$
\begin{array}{lll}
L_2\!: & eq\_path & \rightarrow \mathbf{hd}\;\; eq\_path^{\varepsilon}\;\; \mathbf{hd^{-1}}\;\; eq\_path^{\varepsilon} \\
 & eq\_path & \rightarrow \mathbf{tl}\;\; eq\_path^{\varepsilon}\;\; \mathbf{tl^{-1}}\;\; eq\_path^{\varepsilon} \\
 & eq\_path & \rightarrow \mathbf{id}\;\; eq\_path^{\varepsilon}
\end{array}
$$

- An L_3-path is a path that has one or more hd^{-1} (resp. tl^{-1}) labels that is not balanced by any hd (resp. tl) label. Such paths are "strictly decreasing" in the sense that they correspond to execution paths in which the expression at the target vertex of the path is a proper substructure of the expression at the source vertex:

$$
\begin{array}{lll}
L_3\!: & decr\_path & \rightarrow eq\_path^{\varepsilon}\;\; \mathbf{hd^{-1}}\;\; decr\_path \\
 & decr\_path & \rightarrow eq\_path^{\varepsilon}\;\; \mathbf{tl^{-1}}\;\; decr\_path \\
 & decr\_path & \rightarrow eq\_path^{\varepsilon}\;\; \mathbf{hd^{-1}}\;\; eq\_path^{\varepsilon} \\
 & decr\_path & \rightarrow eq\_path^{\varepsilon}\;\; \mathbf{tl^{-1}}\;\; eq\_path^{\varepsilon}
\end{array}
$$

- Similarly, L_4-paths have zero or more hd^{-1} (resp. tl^{-1}) labels that are not balanced by any hd (resp. tl) label. Such paths are "equal or decreasing":

$$
\begin{array}{lll}
L_4\text{:} & \mathit{eq\_or\_decr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{hd^{-1}} \ \ \mathit{eq\_or\_decr\_path}^{\varepsilon} \\
 & \mathit{eq\_or\_decr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{tl^{-1}} \ \ \mathit{eq\_or\_decr\_path}^{\varepsilon} \\
 & \mathit{eq\_or\_decr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon}
\end{array}
$$

- The languages L_5 and L_6 represent paths that have one or more hd (resp. tl) labels that are not balanced by any hd^{-1} (resp. tl^{-1}) labels. L_5 paths are "strictly increasing" and correspond to execution paths in which the expression at the source of the path is a proper substructure of the value at the target, while L_6 paths are "equal or increasing" (with empty paths allowed):

$$
\begin{array}{lll}
L_5\text{:} & \mathit{incr\_path} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{hd} \ \ \mathit{incr\_path} \\
 & \mathit{incr\_path} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{tl} \ \ \mathit{incr\_path} \\
 & \mathit{incr\_path} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{hd} \ \ \mathit{eq\_path}^{\varepsilon} \\
 & \mathit{incr\_path} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{tl} \ \ \mathit{eq\_path}^{\varepsilon}
\end{array}
$$

$$
\begin{array}{lll}
L_6\text{:} & \mathit{eq\_or\_incr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{hd} \ \ \mathit{eq\_or\_incr\_path}^{\varepsilon} \\
 & \mathit{eq\_or\_incr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon} \ \ \mathbf{tl} \ \ \mathit{eq\_or\_incr\_path}^{\varepsilon} \\
 & \mathit{eq\_or\_incr\_path}^{\varepsilon} & \rightarrow \mathit{eq\_path}^{\varepsilon}
\end{array}
$$

The path languages $L_1 - L_6$ defined above all represent certain kinds of flow dependence relationships between expressions, or the APG vertices that represent expressions. All of these languages are subsets of a language consisting of paths that

Language	Root Nonterminal	Informal Characterization	ε in Language?
L_1	eq_path^{ε}	Equal	Yes
L_2	eq_path	Equal	No
L_3	$decr\_path$	Strictly decreasing	No
L_4	$eq\_or\_decr\_path^{\varepsilon}$	Equal or decreasing	Yes
L_5	$incr\_path$	Strictly increasing	No
L_6	$eq\_or\_incr\_path^{\varepsilon}$	Equal or increasing	Yes
L_7	$dependence\_path^{\varepsilon}$	Shared substructure	Yes
L_8	$eq\_or\_possibly\_incr\_path$	$L_7 - \{\varepsilon\} - L_3$	No
L_9	$possibly\_incr\_path$	$L_7 - \{\varepsilon\} - L_2 - L_3$	No
L_{10}	$control\_path$	Function call	No

Figure 41: Context-free languages used to specify paths of interest in the APG.

do not represent "false" dependences. This language is defined as language L_7 below. Vertices u and v in an APG are connected by an L_7-path from u to v if and only if the expressions at u and v may share a common substructure:

$$L_7\colon\quad dependence\_path^{\varepsilon} \quad \rightarrow eq\_or\_decr\_path^{\varepsilon}\ \ eq\_or\_incr\_path^{\varepsilon}$$

The intuition behind the definition of L_7 is as follows: The first component of a dependence path from u to v represents operations that "dig into" the structure of the value at u to retrieve a substructure of the value. The second component of the dependence path builds up the value at v by combining this substructure of the value at u with other values.

Paths in the language L_7 provide a test for the notion of dependence between parameters in a call path. In other words, if a parameter f_j depends on parameter g_i along some call path, then there must be a path in L_7 from g_i to f_j. This is expressed in Lemma 33 below.

Lemma 33 Parameter f_j of function f in p depends on parameter g_i of function g in

p along some call path iff there is a path p from vertex g_i to vertex f_j in the APG of p such that the concatenated label of p is in $dependence\_path^{\varepsilon}$. □

We would like to use the path languages defined above to identify parameters that must have the in-situ-decreasing property. More precisely, if every (non-ε) path from a parameter vertex to itself that is in the language L_7 is also in the language L_3, and if successive values of the parameter on recursive calls are determined completely by the previous values of the same parameter, then the parameter must be in-situ-decreasing. The reason for excluding ε is that there is always an empty path from a parameter vertex to itself, but this does not represent a transmission path to a recursive invocation of the function.

The characterization of in-situ-decreasing behaviour above presents a subtle difficulty: CFL-Reachability can only be used to test whether *there exists* an L-path from a source to a target. Fortunately, there is a way to finesse this difficulty: We can identify the language L_8 of all paths in $L_7 - \{\varepsilon\} - L_3$, and we can use CFL-Reachability to test the existence of paths in L_8. If there are no paths in L_8 from a parameter vertex to itself, then every path in L_7 from the parameter to itself is also in L_3. The language of paths in L_8 is defined as follows:

$$
\begin{array}{lll}
L_8\text{:} & eq\_or\_possibly\_incr\_path & \rightarrow eq\_or\_decr\_path^{\varepsilon} \;\; incr\_path \\
 & eq\_or\_possibly\_incr\_path & \rightarrow eq\_path
\end{array}
$$

Paths in the language of L_8 are either equal (i.e., balanced) or contain at least one hd (resp. tl) label that is not balanced later in the path by a hd^{-1} (resp. tl^{-1}). Such paths are not strictly-size-decreasing because the expression at the target of the path may not be a proper substructure of the expression at the source of the path. It can be shown that $L_7 = L_3 \cup \{\varepsilon\} \cup L_8$, and furthermore, that L_3, $\{\varepsilon\}$, and L_8 partition L_7. In other words, every L_7-path is either an L_3-path or an L_8-path or the empty path.

We would also like to use the path languages defined above to identify parameters that may have the in-situ-increasing property. For this purpose, we can once again partition the language L_7 so as to group together all paths that are possibly size increasing, as follows:

$$L_9\text{:}\quad \textit{possibly\_incr\_path} \quad \rightarrow \textit{eq\_or\_decr\_path}^{\varepsilon} \quad \textit{incr\_path}$$

$$L_8 = L_2 \cup L_9$$

$$L_7 = L_3 \cup \{\varepsilon\} \cup L_2 \cup L_9$$

Paths in L_9 are size increasing because they contain at least one hd (resp. tl) label that is not balanced later in the path by a hd^{-1} (resp. tl^{-1}), and are of interest because of the following property: If a parameter v is in-situ-increasing, then there exists a cyclic path from v to itself in the APG such that the path is in L_9.

The final path language of interest in the APG is the language of control paths. A control path from function f to function g indicates that a call on f may produce a call on g, and is a sequence of one or more edges labeled with *control*:

$$\begin{aligned} L_{10}\text{:}\quad & \textit{control\_path} \rightarrow \textbf{control} \quad \textit{control\_path} \\ & \textit{control\_path} \rightarrow \textbf{control} \end{aligned}$$

Constant vertices may also be targets of edges labeled with *control*. A path in L_{10} from a function f to a constant vertex c indicates that c is enclosed by a function that may be transitively called by f. (This represents the fact that a call on f may generate an instance of a constant represented by c.)

The languages of path labels defined in this section are all context-free languages. Every source-target L-path problem for each of these languages can therefore be solved in time cubic in the number of nodes in the APG.

6.5 Identifying ISD and ISI parameters

In this section, we define an algorithm that uses the presence or absence of certain paths in the APG to identify in-situ-decreasing and in-situ-increasing parameters in a program. This algorithm serves as a pre-processing phase for our algorithm for identifying BSV parameters, which is defined in Section 6.6.

In any recursive call from a function f to itself, represented by a call path from f to f, a parameter f_j of f can only receive values from previous values of f_j itself, from previous values of other sibling parameters f_i of f, from constant values, or from a combination of these. Therefore, a parameter must be in-situ-decreasing if it satisfies two basic conditions:

(i) it does not take values from sibling parameters or constants, and

(ii) it receives a value from itself in only a size-decreasing manner.

Condition (i) can be tested using APG paths as follows: From Lemma 33 in the previous section, we can conclude that the absence of any paths in $dependence\_path^{\varepsilon}$ from sibling vertices f_i to vertex f_j ensures that f_j does not depend on sibling parameters, while the absence of any paths in $dependence\_path^{\varepsilon}$ to vertex f_j from constant vertices enclosed by functions that are transitively called by f ensures that f_j does not take on constant values on recursive calls to f. Condition (ii) can be tested in two parts: The presence of a path in *decr_path* from vertex f_j to itself ensures that parameter f_j depends on itself, while the absence of any paths in *eq_or_possibly_incr_path* from vertex f_j to itself ensures that these dependences are strictly size-decreasing. The sufficiency of these tests for identifying a subset of the in-situ-decreasing parameters in a program is formalized in Lemma 34 below.

Lemma 34 Parameter f_j of recursive function f in p is in-situ-decreasing if all of the following properties hold on the APG of p:

(a) there does not exist a vertex f_i such that f_i is a sibling parameter of f_j and there is an L_7-path (*dependence_path*$^\varepsilon$) from f_i to f_j;

(b) there is no constant vertex c such that there is an L_7-path (*dependence_path*$^\varepsilon$) from c to f_j and there is an L_{10}-path (*control_path*) from f to c;

(c) there exists an L_3-path (*decr_path*) from f_j to f_j, and there does not exist an L_8-path (*eq_or_possibly_incr_path*) from f_j to f_j.

Properties (a) and (b) above ensure that successive values of f_j are derived only from previous values of f_j, while property (c) above ensures that successive values of f_j in a recursive call on f are always derived from previous values of f_j in a size-decreasing manner. $\square$

Similarly, we can use APG paths to identify a superset of all the in-situ-increasing parameters in a program. In particular, a parameter may be in-situ-increasing for the following reason: The parameter depends on itself in a size-increasing manner. This condition can be tested via the presence of paths in the language *possibly_incr_path* from the parameter vertex to itself.

Lemma 35 If parameter f_j of function f in p is in-situ-increasing, then:

(a) there is an L_9-path (*possibly_incr_path*) from f_j to f_j in the APG of p. □

The lemmas above lead directly to Algorithm 2 defined below, which conservatively identifies in-situ-decreasing and seed-in-situ-increasing parameters using CFL-Reachability.

Algorithm 2 Identify in-situ-decreasing and in-situ-increasing parameters.

Input: APG G

Output: APG G, in which forall p in *params(G)*,

$ISD(p) \;\Rightarrow\; p$ is in-situ-decreasing, and p is in-situ-increasing $\Rightarrow\; ISI(p)$.

Construct the relations *ISD* and *ISI* defined below.[1]

ISD(v)	**if**	$\neg$*sibl-dep(v)* **and** $\neg$*const-dep(v)* **and** *decr(v,v)* **and** $\neg$*eq_or_incr(v,v)*
ISI(v)	**if**	*incr(v,v)*

where

decr(m,n)	**if**	*decr_path(m,n)*
eq_or_incr(m,n)	**if**	*eq_or_possibly_incr_path(m,n)*
incr(m,n)	**if**	*possibly_incr_path(m,n)*
sibl-dep(m)	**if**	$\text{dependence\_path}^{\varepsilon}(n, m)$ **and** *function(n)* $=$ *function(m)*
const-dep(m)	**if**	$\exists\ c$ *s.t.* *control_path(function(m),c)* **and** *dependence_path*$^{\varepsilon}$*(c,m)*

[1]We use the notation *foo_path(m,n)* to mean that that n is reachable from m in G via a path in the language *foo_path*. Function *params* returns the set of parameters in an APG, while *function* returns the function associated with a parameter.

Every parameter in an APG may belong to one of *ISD* or *ISI*; the parameter is in *ISD* if it is identified as definitely in-situ-decreasing, while the parameter is in *ISI* if it is possibly in-situ-increasing. Thus *ISD* and *ISI* are mutually exclusive, but not complementary. □

Example 16 The result of applying Algorithm 2 to the program from Example 11 is as follows: All parameters of functions *checkValid* and *accum* are trivially included in *ISD*, because these functions are non-recursive. Parameter *err* is not in relation *ISD* because there is an *eq_or_possibly_incr_path* from *err* to itself (with label *id*.) Similarly, parameter *tot* is not in *ISD* because of the path with label *id.id.id.id* from *tot* to itself. Parameter *ops* (similarly, *vals*) is included in *ISD* because the only path in $dependence\_path^{\varepsilon}$ that has *ops* (similarly, *vals*) as its target originates at *ops* (similarly, *vals*) and is in *decr_path*. Finally, there are no paths in *possibly_incr_path* in the APG of the program. Hence, there are no parameters in relation *ISI*. □

Since CFL-Reachability problems can be solved in time cubic in the number of vertices in the graph [Yan90], Algorithm 2 has running time $\mathcal{O}((F + P + Var + Ops)^3)$, where F is the number of functions in p, P is the number of parameters in p, Var is the number of *where* variables in p, and Ops is the number of cons/car/cdr operations in p.

6.6 Identifying BSV parameters

The algorithm for identifying ISD and ISI parameters defined in the previous section serves as a pre-processing phase for our main algorithm for identifying BSV parameters, which is described in this section. This algorithm has a structure that is significantly different from that of Glenstrup and Jones' algorithm in [GJ96]. Their algorithm is pessimistic, in the sense that every static parameter is treated as non-BSV until proven otherwise through the steps of the algorithm. In contrast, our algorithm is optimistic: Every static parameter is treated as BSV until (conservatively) shown otherwise. As shown by Example 12 earlier in this chapter, this difference sometimes allows the optimistic approach to produce better results.

Our algorithm for identifying BSV parameters is shown as Algorithm 3 below. The algorithm is described declaratively, as a fixed-point procedure on a recursive equation involving several properties of function parameters. Operationally, the algorithm can be thought of as a reachability operation on the APG, with additional information concerning anchoring parameters. The intuition behind the procedure is as follows: A static parameter whose predecessors in the APG are all BSV and which is not involved in any size-increasing loops must be BSV. Further, a static parameter that is involved in size-increasing loops will be BSV if its predecessors are BSV, and it is anchored by one or more siblings that are anchoring. A sibling parameter is anchoring if it is in-situ-decreasing and BSV.

Algorithm 3 Identify bounded-static-varying parameters.

Obtain the least fixed-point of equation (1) below. Re-classify all static parameters for which predicate D is true as dynamic.

$$D(v) = \left(\bigvee_{w \in preds(v)} D(w) \right) \ \vee \ \left(ISI(v) \ \wedge \ \left(\bigwedge_{w \in siblings(v)} \neg Anchoring(w) \right) \right)$$

$$Anchoring(v) = ISD(v) \wedge \neg D(v) \qquad (1)$$

where initially, $D(v) = false$ for all static parameters, and

ISI and *ISD* are determined by Algorithm 2.

The equation above is solved over the domain $\{true, false\}$, where $false \sqsubset true$. The predicates D, ISI, and ISD are either *true* or *false* for every parameter. □

Equation (1) from the algorithm above involves negation on a predicate that is not a base predicate. However, the equation can be re-written as equation (2) below, in which negation is restricted to the base predicate *ISD*:

$$D(v) = \left(\bigvee_{w \in preds(v)} D(w) \right) \vee \left(ISI(v) \wedge \left(\bigwedge_{w \in siblings(v)} (\neg ISD(w) \vee D(w)) \right) \right) \quad (2)$$

This formulation makes it clear that the least-fixed point of Equation (2) (and hence of Equation (1)) is well-defined.

Algorithm 3 proceeds as follows. Initially, every static parameter is treated as BSV (D is *false* for the parameter vertex), and every parameter in *ISD* is treated as anchoring, since it is also treated as BSV. Parameter markings are then updated at every step, until no further parameters can be updated, as follows: Any parameter that is in *ISI* and that has no anchoring siblings is marked as non-BSV (D is set to *true* for the parameter vertex). Such an update may be propagated in two ways: Any successor of a parameter marked as non-BSV is re-classified as non-BSV, and any anchoring parameter that is re-classified as non-BSV is no longer anchoring. This change may cause previously anchored parameters to be re-classified as non-BSV, and so on.

The correctness of our algorithm for identifying BSV parameters is established by Theorem 3 below.

Theorem 3 If parameter f_j of function f in p is identified as static by Algorithm 3, then parameter f_j is BSV.

Proof Sketch. The proof is by induction on the "level" of a function. The level of a function is defined as a number assigned to a function by topologically sorting the call graph of the program, as follows: The function *main* has level 1. All functions in the same strongly connected component (scc) of the call graph have the same level. If a function in one scc can call a function in another scc, the functions in the scc of the caller have a lower level than those in the scc of the callee. We prove the theorem by induction; we assume that BSV parameters are marked correctly at all functions whose level number is less than the current level. We then show that the parameters of the functions at the current level must be marked correctly.

Base Case: Level 1, function *main*. The hypothesis is trivially true.

Induction Step: Assume that for all parameters f_j of functions f such that the level of f is less than i, if $D(f_j) = \mathit{false}$ then f_j is BSV. We must show that this is true for all parameters f_j of functions f at level i. We first show that anchoring parameters are marked correctly at level i:

We give an argument by contradiction. Suppose parameter f_j is identified as Anchoring by Algorithm 3, but is not in fact anchoring. Then by Lemma 29, f_j is either not BSV, or not in-situ-decreasing. But if f_j is marked as *ISD* (as it must be, for Algorithm 3 to have identified f_j as having the Anchoring property), it must be in-situ-decreasing, by Lemma 34. Hence f_j must not be BSV.

We now obtain a contradiction as follows: Since f_j is identified as in-situ-decreasing, it cannot depend on sibling parameters or on constants enclosed by functions in the same scc of the call graph as function f. Also, if f_j depends on a parameter g_i of function g in the same scc as f, g_i must also depend on f_j. This is because g_i must depend on some parameter of f or on a constant enclosed by f, since it can be called through f. Now if g_i depends on any other parameter f_k of f or on a constant enclosed by f, this dependence would be transitively passed on to f_j, but that is ruled out because f_j is identified as *ISD* by Algorithm 2. Hence, the predecessors of f_j consist of parameters in an scc of the APG consisting of parameters of functions g that are in the same scc of the call graph as f, or parameters of functions g whose level is lower than the level of f. Let us call the latter set P. The parameters in P must be marked as BSV, or f_j would not be marked as anchoring. By assumption, the parameters in P are marked correctly.

The final step is to show that the BSV-ness of the parameters in P permits us to conclude that *all* the elements of the scc containing f_j (and hence f_j) are also BSV. The predecessors g_i in the same scc of the APG as f_j must have the following property: Let S be the scc containing g_i and f_j; then all cycles in S (even those not containing f_j) must be in $eq\_or\_decr\_path^{\varepsilon}$. Otherwise, f_j would not have only strictly size decreasing paths to itself. This means that the values taken on by the parameters in S can only be substructures of the (bounded sets of) values taken on by the members of P. Hence, all the parameters g_i in the scc of f_j, including f_j itself, must be BSV, which contradicts our assumption. Hence, f_j must be anchoring (*i.e.*, $Anchoring(f_j) \Rightarrow anchoring(f_j)$). This argument is described pictorially in Figure 42.

Through a similar argument, using the property that anchors at level i must be marked correctly, we can show that all other parameters of functions at level i that are marked as BSV must be BSV. We omit the proof for brevity. $\square$

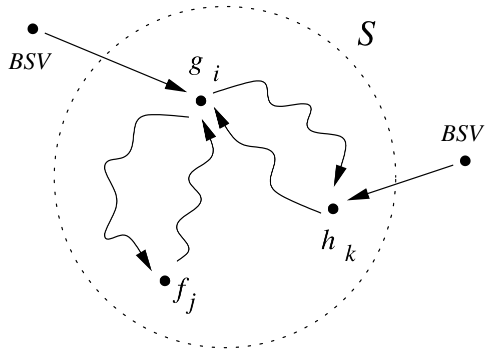


Figure 42: Correctness of Algorithm 3.
The figure above represents an scc S of the APG involving *ISD* parameter f_j, which is marked as Anchoring by Algorithm 3. All cycles in S are in $eq\_or\_decr\_path^{\varepsilon}$. In addition, all parameterss not in S that have flow dependence edges to parameters in S are BSV. Therefore, all parameters in S, including f_j, must be BSV.

The algorithm above is a reachability operation that is linear in the sum of the number of edges in the APG and the number of anchor links from sibling parameters, which is $\mathcal{O}(F^2 + (P + Var)^2 + Ops)$. If we use simple reachability to identify in-situ-decreasing and in-situ-increasing parameters, our algorithm has the same worst-case complexity as that of Glenstrup and Jones, with better results due to its optimistic approach. If we use CFL-Reachability in the pre-processing phase, the running time of the algorithm is controlled by the running time of the pre-processing phase, which is $\mathcal{O}((F + P + Var + Ops)^3)$. This option yields better results, as it identifies a broader class of in-situ-decreasing parameters, and a narrower class of in-situ-increasing parameters.

6.7 Related work

The work described in this chapter is also presented in [DR96]. The basis for this work is Holst's definition of the in-situ-decreasing property for function parameters in [Hol91]: An in-situ decreasing parameter of a function f strictly decreases in size on every (recursive) chain of calls from f to f.

Glenstrup and Jones define a second algorithm for identifying in-situ-decreasing parameters, which uses the markings $\uparrow$, $\downarrow$ and $=$ on edges in the parameter dependency graph [GJ96]. The algorithm described in this paper extends their work by using CFL-Reachability to identify a broader class of size-decreasing paths, and by using an optimistic algorithm.

Andersen and Holst have described an extension of Holst's analysis to a higher-order lambda calculus [AH96]. Although their primary emphasis was termination analysis for programs with higher-order functions, they observe that their technique for handling higher-order functions can be adapted to discover some size-decreasing paths containing $\uparrow$ edges.

Our approach was conceived independently of their work, but we became aware of it shortly after it was presented at SAS '96. Some of the differences between their work and our approach are:

- Their approach is based on tree grammars, whereas our approach is an extension of Glenstrup and Jones's approach, which is based on graph reachability. In order to identify a greater number of in-situ-decreasing parameters than Glenstrup and Jones, we extend the parameter dependency graph with new nodes and new edge markings and we use CFL-Reachability rather than a closed semi-ring graph algorithm [AHU74, GJ96].

- There is a general result that all "context-free language reachability problems" can be solved in time cubic in the number of vertices in the graph [Yan90]. This allows us to obtain a cubic-time bound for our algorithm. (Andersen and Holst did not report a bound on the running time of their algorithm, although we suspect that for the class of problems we are addressing, their methods would also run in cubic time.)
- We are able to provide a formal justification of our method for identifying in-situ-decreasing parameters.

CFL-Reachability has also been used for a number of other program-analysis problems: Reps, Sagiv, and Horwitz have used CFL-Reachability techniques to solve interprocedural dataflow-analysis problems [RSH94, RSH95] and to perform interprocedural slicing [RHSR94]. Reps has used CFL-Reachability to develop a shape-analysis algorithm [Rep95]. He used edge markings that are identical to the markings used on the edges of the APG defined in this paper.

Melski and Reps have shown that CFL-Reachability problems are convertible into a class of set-constraint problems (and vice versa) [MR97]. Because set-constraints are related to regular tree grammars, this result also has some bearing on the relationship between our work and that of Andersen and Holst. The exact relationship is somewhat fuzzy at this point because the tree grammars used in the Andersen and Holst paper do not make a direct application of the Melski-Reps result possible. (Even if it were applicable, the details of the general-case construction would obscure the relatively simple concepts expressed by the context-free grammars given in Section 6.4.)

Chapter 7

Specialization of dependence graphs

As we explained in the introduction, partial evaluation can be implemented as a two-step process, in which the first phase is an analysis phase where static and dynamic program variables are identified, while the second phase is a specialization phase in which static parts of the program code are executed away and dynamic code fragments are emitted as the specialized program. In previous chapters of this thesis, we have shown how dependence graphs can be used to enable the analysis phase of partial evaluation. The natural question that arises is whether dependence graphs provide a suitable basis for the specialization phase of partial evaluation as well.

In this chapter, we show that dependence graphs can be used as a basis for the specialization phase. In particular, we develop this phase of partial evaluation as an operation that transforms the dependence graph representation of the subject program, as opposed to its CFG representation. Dependence graphs have the advantage that they do not explicitly order independent statements within a program, allowing greater flexibility during execution. Under a control-flow-based model of execution, program statements are executed in a particular sequential order. In contrast, a dependence graph is executed as a data-flow graph, in which every vertex in the graph is a producer of output values that consumes input values produced by its dependence predecessors.

In this chapter, we describe our work on developing a specializer for dependence graphs. We show how specialization on PRGs can be modeled as a partial execution of the PRG semantics, in exactly the same way that specialization of CFGs can be modeled as partial execution of the CFG's state-transformation semantics. We describe a data-flow specialization algorithm for PRGs that transforms a PRG with static and dynamic markings into a residual PRG that represents the specialized program. We also outline a reconstitution algorithm that allows us to construct a sequential program from the residual PRG. These algorithms are applicable to imperative programs that have PRG representations.

We have implemented our specialization scheme, using the results of BTA algorithms that identify both strongly static and weakly static vertices. Our experiments reveal that dependence-graph-based specialization has the following drawbacks:

- Data-flow execution of the PRG is slower than sequential execution of the CFG. In our experiments, over a range of programs with looping constructs, PRG specialization is roughly 20% slower than CFG specialization. This is because the data-flow execution engine requires the transmission of every value through the PRG. Each value must be tagged at the source vertex, and this tag must be matched as the destination vertex, which leads to expensive tests. In addition, values must be queued at consumer vertices, in case values from other producers are not available. Finally, the PRG representation of a program may have a greater number of edges than its CFG representation. The running time of the specializer is related to the number of edges in the graph, in both CFG specialization and PRG specialization. However, the higher execution cost of PRG specialization does not affect the running time of the residual program itself.

- Reconstitution of sequential code from the residual PRG is an expensive operation. Our reconstitution algorithm has a worst-case running time that is equal to the cost of sorting the vertices in the residual PRG. In practice, this cost appears to be linear in the size of the residual PRG, because vertices are almost always emitted in sorted order. However, this is not guaranteed to be the case.

- At this time, only a limited class of programs can be specialized, in particular the programs represented by PRGs. It is not clear how we would extend the specialization algorithm to handle programs with arbitrary control flow.

Regardless, our work on specialization of dependence graphs serves as a useful exercise in examining the role of dependence graphs in partial evaluation.

7.1 Specialization of control flow graphs

In this section, we describe the traditional method used in the specialization phase of partial evaluation. In particular, we are interested in identifying the difference between the total execution of a CFG during computation and the partial execution of a CFG during specialization, so that we can develop a specializer for PRGs by similarly modifying the operational behaviour of PRGs.

At the semantic level, the specialization phase of partial evaluation can be thought of as a partial execution of the program given incomplete input values, as opposed to total execution given complete inputs. Under the conventional approach to partial evaluation, the specialization phase is carried out on the control flow graph. In the total or operational semantics of the CFG, every program point is a state-to-state transformer, where the program state contains a mapping from every program variable to a value. In the partial semantics of the CFG, every program point either transforms the "partial state", if it can be computed using values of static variables, or it has a side-effect of producing a residual program point. The partial state contains a mapping of every static program variable to a value.

Operationally, total execution of the CFG is carried out by an interpreter, which sequentially executes program points in the CFG, updating the state at every assignment vertex, and choosing a branch direction at every predicate vertex. In contrast, the specialization phase of partial evaluation can be thought of as a partial execution of the CFG. A specializer either executes or residuates program points in the CFG, as follows:

- If the code at the current program point pp can be evaluated using only static values, the partial state of the specializer is updated or a branch direction is chosen, and the specializer moves on to the next statement pp'.
- If dynamic values are required to evaluate the code at the current program point pp, code is generated in the residual program; the program point in the residual program is labeled (pp, $state$), where $state$ is the current partial state.

The actions of a specializer as described above can be thought of as a regrouping of the control flow graph. The original program has run-time states of the form $(pp, (stateS, stateD))$, where pp is the program point, $stateS$ is the part of the state mapping static variables to values, and $stateD$ is the part of the state mapping dynamic variables to values. In the specialized program, program points are of the form $(pp, stateS)$, and so run-time states are of the form $((pp, stateS), stateD)$. Thus,

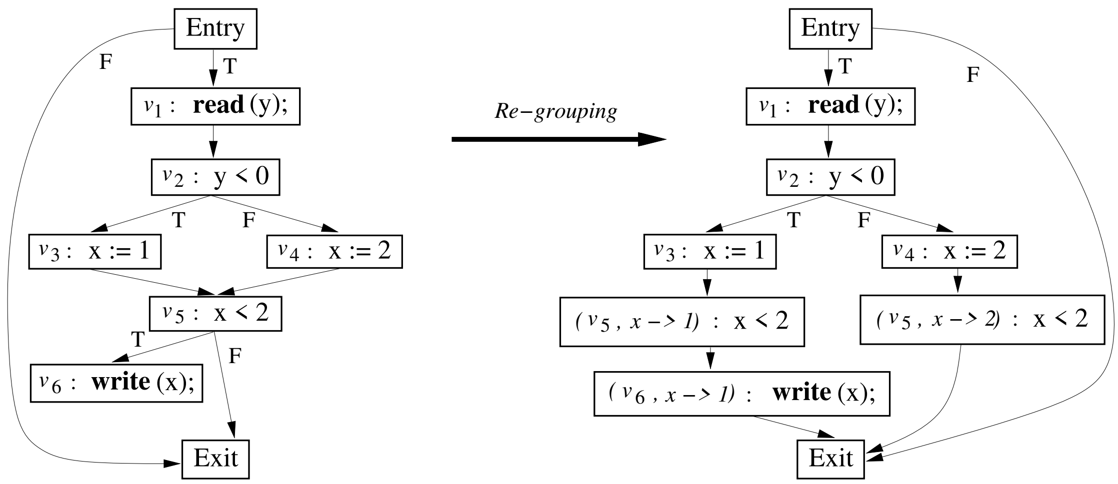


Figure 43: Specialization as regrouping on the CFG.

states of the specialized program can be identified with states of the original program: $((pp, stateS), stateD)$ corresponds to $(pp, (stateS, stateD))$.

An example of specialization via regrouping on the CFG is shown in Figure 43.

7.2 Specialization of dependence graphs

As we described in the previous section, partial execution of CFGs is similar to their total execution, with the distinction that some CFG vertices may produce code-generation actions rather than state-transformation actions. In this section, we show that the PRG semantics can be similarly extended to define a specialization operation on PRGs.

In the total semantics of PRGs described in Chapter 3, every vertex is associated with a value sequence, which represents its behaviour during computation, or total execution. Operationally, a PRG can be executed as a data-flow graph, in which every vertex is both a producer and a consumer of values. A vertex produces a new value when it has "enough" values from its predecessors to compute another value; this value is then passed on to its successors, which serve as consumers for the value, and which may, in turn, produce a new value when supplied with the given value from the producer.

Similarly, we define a partial semantics for PRGs that represents the behaviour of PRG vertices during specialization, or partial execution. In the partial semantics, every vertex either produces a value sequence, as in the total semantics, or code-generation actions, depending on the static/dynamic nature of the vertex and its dependence predecessors.

During specialization, the PRG is executed as a data-flow graph, in which the actions of a vertex are triggered when it receives values from its predecessors. Therefore, it is necessary even for dynamic vertices that perform code-generation actions to produce special wild-card values that are passed on to their successors. This distinction from CFG specialization arises because in PRG execution, the "execution" of individual nodes is controlled locally, according to received values from predecessors, whereas in CFG execution, there is a unique locus of control (and thus control flow is a global concept of the semantics).

The partial semantics for PRGs that represents PRG specialization is determined by the form of static behaviour that is identified by the BTA phase. For instance, when the BTA phase identifies only strongly static vertices, a constant assignment that is nested within a dynamic conditional is treated as dynamic, and must perform a code-generation action and produce suitable wild-card output values. In contrast, when the BTA phase identifies weakly static vertices, a constant assignment nested within dynamic conditionals is treated as static, and does not perform any code-generation actions. The partial semantics for PRG specialization, given a BTA phase that identifies strongly static vertices, is shown in Figure 44.

The equation shown in Figure 44 is a straightforward extension of the equation representing the total semantics of PRGs in Figure 14 from Chapter 3. PRG vertices that have only static predecessors have the same behaviour in both the total semantics and the partial semantics. Vertices with dynamic predecessors have code-generation actions in the partial semantics, and produce only wild-card values.

When weakly static vertices are identified by the BTA phase, some vertices that have dynamic predecessors may also be treated as static. Such vertices will not perform any code-generation actions. The semantics for specialization with weakly static behaviour is shown in Figure 45.

Specialization on dependence graphs, as represented by the partial semantics for PRGs, can also be thought of as a regrouping of the dependence graph. The original PRG has local run-time states $(v, (sPred[i], dPred[i]))$, where $sPred[i]$ represents the values in the sequences produced at the static dependence predecessors of vertex v, and $dPred[i]$ represents the values in the sequences produced at the dynamic dependence predecessors of v. In the specialized PRG, the values from static predecessors are absorbed into the vertices, resulting in run-time local states that are of the form $((v, sPred[i]), dPred[i])$. As in the case of CFG-based specialization, the states of the specialized PRG can be identified with states of the original PRG: $((v, sPred[i]), dPred[i])$ corresponds to $(v, (sPred[i], dPred[i]))$.

$$
\begin{array}{l}
\mathbf{E_G} \;\doteq\; \lambda v.\lambda f. \\
\quad \mathbf{type}(v) = \mathbf{Entry} \rightarrow \text{true} \cdot \mathit{nil},\ \{\text{emit } \mathbf{Entry}\} \\
\quad \mathbf{type}(v) = \mathbf{read},\ \mathit{ctrlPred(v)} \text{ is } S \rightarrow \\
\qquad \text{replace}(\mathit{controlLabel}(v), ?, f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \mathbf{read}\} \\
\quad \mathbf{type}(v) = \mathbf{read},\ \mathit{ctrlPred(v)} \text{ is } D \rightarrow \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \mathbf{read}\} \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ \#\mathit{dataPreds}(v) = 0,\ \mathit{ctrlPred(v)} \text{ is } S \rightarrow \\
\qquad \text{replace}(\mathit{controlLabel}(v), \mathit{funcOf}(v), f\ \mathit{ctrlPred}(v)),\ \{\} \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ \#\mathit{dataPreds}(v) = 0,\ \mathit{ctrlPred(v)} \text{ is } D \rightarrow \\
\qquad \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \mathbf{assign/if/while}\} \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ w \text{ is } S\ \forall w \in \mathit{dataPreds}(v) \rightarrow \\
\qquad \mathit{map}\ \mathit{funcOf}(v)(f\ \mathit{dataPred_1}(v) \ldots f\ \mathit{dataPred_n}(v)),\ \{\} \\
\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ \exists\ w \in \mathit{dataPreds}(v) \text{ s.t. } w \text{ is } D \rightarrow \\
\qquad \mathit{map}\ \mathit{funcOf}(v)(f\ \mathit{dataPred_1}(v) \ldots f\ \mathit{dataPred_n}(v)),\ \{\text{emit } \mathbf{assign/if/while}\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{T}},\ \mathit{ctrlPred(v)} \text{ is } S \rightarrow \text{select}(\text{true}, f\ \mathit{parent}(v), f\ \mathit{dataPred}(v)),\ \{\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{T}},\ \mathit{ctrlPred(v)} \text{ is } D \rightarrow \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \phi_{\mathbf{T}}\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{if}},\ \mathit{ctrlPred(v)} \text{ is } S \rightarrow \\
\qquad \text{merge}(f\ \mathit{ifNode}(v), f\ \mathit{trueDef}(v), f\ \mathit{falseDef}(v)),\ \{\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{if}},\ \mathit{ctrlPred(v)} \text{ is } D \rightarrow \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \phi_{\mathbf{if}}\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{enter}},\ \mathit{ctrlPred(v)} \text{ is } S \rightarrow \\
\qquad \text{whileMerge}(f\ \mathit{whileNode}(v), f\ \mathit{innerDef}(v), f\ \mathit{outerDef}(v)),\ \{\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{enter}},\ \mathit{ctrlPred(v)} \text{ is } D,\ \mathit{outerPred(v)} \text{ is } S \rightarrow \\
\qquad \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \phi_{\mathbf{enter}}, \text{ emit outer } \mathbf{assign}\} \\
\quad \mathbf{type}(v) = \phi_{\mathbf{enter}},\ \mathit{ctrlPred(v)} \text{ is } D,\ \mathit{outerPred(v)} \text{ is } D \rightarrow \\
\qquad \text{copySeq}(f\ \mathit{ctrlPred}(v)),\ \{\text{emit } \phi_{\mathbf{enter}}\}
\end{array}
$$

where *replace*, *whileMerge*, *select*, and *merge* are defined as in Figure 14 and *copySeq* is defined as follows:

$$
\begin{array}{lll}
\mathit{copySeq}: & \text{copySeq}(\ \bot\ ) = \bot & \text{copySeq}(\ \mathit{nil}\ ) = \mathit{nil} \\
 & \text{copySeq}(\ a \cdot \mathit{tail}\ ) = a \cdot \text{copySeq}(\ \mathit{tail}\ ) &
\end{array}
$$

Figure 44: PRG specialization equations for strongly static behaviour.
In the equation above, code-generation actions are shown in braces. ? is the wild-card value. Every vertex with an emit action emits a residual vertex for every value it computes in its output value sequence. A ϕ_{enter} vertex within a dynamic loop that has a static outer predecessor must residuate the outer predecessor as well, once for every value it receives from the predecessor. We have omitted the machinery for determining how emitted residual vertices are linked to their predecessors in the residual PRG. This is discussed in Section 7.3. Some vertex types have been omitted for brevity.

$\mathbf{E_G} \;\doteq\; \lambda v.\lambda f.$
$\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ \#\mathit{dataPreds}(v) = 0,\ \mathit{ctrlPred(v)}$ is $S \rightarrow$
$\qquad \text{replace}(\mathit{controlLabel}(v), \mathit{funcOf}(v), f\ \mathit{ctrlPred}(v)),\ \{\}$
$\quad \mathbf{type}(v) \in \{\mathbf{assign}, \mathbf{if}, \mathbf{while}\},\ \#\mathit{dataPreds}(v) = 0,\ \mathit{ctrlPred(v)}$ is $D \rightarrow$
$\qquad \text{blindReplace}(\mathit{funcOf}(v), f\ \mathit{ctrlPred}(v)),\ \{\}$

where *blindReplace* is defined as follows:

$$\begin{array}{ll} \mathit{blindReplace}: & \text{blindReplace}(\ v,\ \perp\ ) = \perp \qquad\qquad \text{blindReplace}(\ v,\ \mathit{nil}\ ) = \mathit{nil} \\ & \text{blindReplace}(\ v,\ z \cdot \mathit{tail}\ ) = v \cdot \text{blindReplace}(\ v,\ \mathit{tail}\ ) \end{array}$$

Figure 45: PRG specialization equations for weakly static behaviour.
In the equation above, only those vertex types are shown for which the partial semantics for weakly static specialization differs from the partial semantics for strongly static specialization. Vertices with constant expressions are not emitted even if their control predecessors are dynamic.

7.3 An algorithm for PRG specialization

The partial semantics defined in the previous section gives us a handle on a method for specializing dependence graphs. In this section, we describe an algorithm that specializes a PRG by mimicing its data-flow execution. The algorithm is an extension of our algorithm for total execution of PRGs, described below.

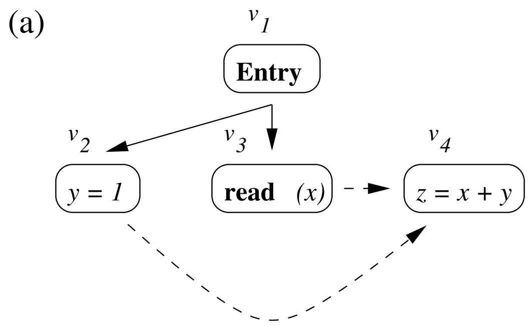
We use a round-robin iteration scheme, in which a vertex that is ready to produce a new value does so by placing this value, tagged with the vertex identifier, on a global worklist queue. At every step of the main execution engine, a value is taken off the worklist and forwarded to all of the PRG successors of the vertex that produced the value. Each such successor receives the value and updates its local state, determining whether is has "enough" pending values from all of its predecessors in its local state to produce a new value. If it does, it produces the value, consumes the corresponding predecessor values by updating its local state, and adds the new value to the global worklist. PRG execution is initiated by the production of a *true* value from the **Entry** vertex, and terminates when no values remain on the global worklist.

Example 17 An example PRG is shown in Figure 46 (a). The values propagated through the PRG during its data-flow execution are shown in Figure 46 (b). Vertex v_1 produces a single *true* value. It propagates this value, tagged with the identifier of the source vertex (v_1). Vertex v_2 produces the value 1 when it receives a *true* value from its control dependence predecessor (v_1). Vertex v_3 consumes a value from the input stream when it receives a *true* value from its control dependence predecessor (v_1). It propagates this value (10 in this example). Finally, vertex v_4 produces the value 11, when it has received values from its predecessors (v_2 and v_3). □

The PRG semantic equations guarantee that if the program terminates normally, every value that is on the worklist is eventually consumed by all of the PRG successors of the vertex that produced the value. The vertex-identifier tag on a value is necessary so that each successor can determine which of its predecessors produced the value, in order to correctly update its local state. In addition, a successor vertex that cannot consume a propagated value immediately on receiving it must store this value in a local queue for the corresponding predecessor. Queues are necessary because one predecessor of the vertex may propagate several values before its other predecessors are able to supply corresponding values required for the computation at the vertex.

Apart from its structure itself, this algorithm differs from the standard execution of a CFG by an interpreter in the following ways:

- The algorithm has an overhead when compared with CFG execution, because at



(b) Value propagation:

v_1: $[T, v_1]$ to v_2 and v_3
v_2: $[1, v_2]$ to v_4
v_3: $[10, v_3]$ to v_4
v_4: $[11, v_4]$

Figure 46: An example of PRG execution by value-propagation.
An example PRG is shown in figure (a) above.The values propagated through the PRG during execution are shown in figure (b) above.

every vertex, the tags associated with received values must be matched against all of the predecessors of the vertex, possibly resulting in several tests at each step. Also, every value is pushed and popped off value queues at vertices that are successors of the vertex that produced the value.

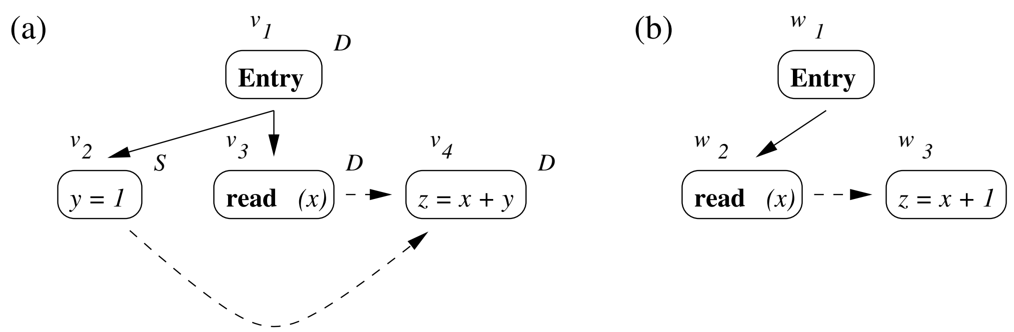
- The PRG has no global state. Therefore, every vertex maintains its own local state, which determines the state of the computation at the vertex at any given point during PRG execution.

The specialization algorithm is an extension of the total execution scheme above. The central "execution-by-value-propagation" engine is unchanged. Static vertices behave as before, placing new values on the worklist and consuming values from predecessors. Dynamic vertices emit residual PRG vertices when they are supplied with values from their predecessors, in addition to propagating wild-card values. In order to ensure that an emitted residual vertex is linked correctly with its dependence predecessors in the residual PRG, we use the following scheme: A vertex that produces a new residual vertex obtains a unique identifier for this vertex. The output value that is produced at the vertex (a wild-card value) is tagged with two identifiers: The identifier of the source vertex in the original PRG, and the identifier of the residual vertex in the residual PRG. The latter identifier represents the vertex that will serve as a dependence predecessor for the vertices emitted by the successors of the source vertex. Thus, when a dynamic vertex produces a residual vertex, its state has the information necessary to create the appropriate dependence edges in the residual PRG that have this residual vertex as their target. This scheme is sufficient because no residual vertex can be emitted before the residual vertices that serve as its predecessors in the residual PRG have been emitted. This property is guaranteed by the partial semantics in Figure 44 and Figure 45.

Example 18 An example PRG is shown in Figure 47 (a). The residual PRG produced by applying our specialization algorithm to this PRG is shown in Figure 47 (b). The values propagated through the PRG during specialization are shown in Figure 47 (c). Vertex v_1 produces a single *true* value, and performs a code-generation action by emitting a residual **Entry** vertex. It propagates the value *true*, tagged with the identifier of the original vertex that produced the value (v_1) and the identifier of the residual vertex (w_1). Vertex v_2 is static, and therefore has no code-generation action. It produces the value 1 when it receives a *true* value from its control dependence predecessor (v_1). The second tag on this value is a don't-care, because v_2 does not produce any residual vertex. Vertex v_3 is dynamic, and emits a residual **read**. It propagates a don't-care value. The second tag on this value indicates the residual **read** vertex (w_2) that is emitted. This residual obtains the identifier of its control dependence predecessor from the second tag on the value propagated by v_1. Finally, vertex v_4 emits a residual assignment (w_3), in which the static value from predecessor v_2 has been absorbed into the right-hand-side expression. The residual assignment obtains the identifier of its flow-dependence predecessor from the second tag on the value produced by v_3. $\square$

7.4 Reconstitution of sequential code

The specialization algorithm described in the previous section allows us to transform a PRG representing the subject program into a residual PRG, which in turn represents



(c) Value propagation:

v_1:	$[T, v_1, w_1]$ to v_2 and v_3	{emit **Entry**(id=1)}
v_2:	$[1, v_2, ?]$ to v_4	{}
v_3:	$[?, v_3, w_2]$ to v_4	{emit **read**(id=2,cPred=1)}
v_4:	$[?, v_4, w_3]$	{emit **assign**(id=3,dPreds=[2])}

Figure 47: An example of PRG specialization by value-propagation.
An example PRG is shown in figure (a) above. The residual PRG produced by applying our specialization algorithm to this PRG is shown in figure (b) above. The values propagated through the PRG during specialization are shown in figure (c) above.

(a) **read**(x);
$y = x * x$;
$z = x + x$;

(b) **read**(x);
$z = x + x$;
$y = x * x$;

(c)

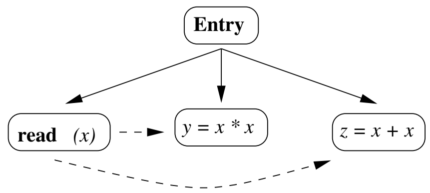
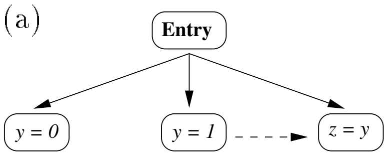


Figure 48: Distinct programs with identical PRG representations.
The programs shown in (a) and (b) above have the same PRG representation, shown in (c). Although the programs are different, they have identical semantic behaviour.

the specialized program. Because specialization is carried out by a data-flow execution of the PRG, it is not possible to modify the CFG of the program at every step of the specialization process. Hence, at the end of the specialization process, there is no CFG corresponding to the residual PRG. In order to obtain sequential code that can be fed to a conventional compiler or code generator, it is therefore necessary to "reconstitute" sequential code from the residual PRG.

In general, a given dependence graph or PRG may represent multiple sequential programs. This is because statements that are independent in terms of dependences are not explicitly ordered in the PRG. An example of two different sequential programs that have the same PRG is given in Figure 48. Because the same PRG may represent multiple programs, the residual PRG obtained by specialization may not correspond to a unique program. Therefore, the goal of reconstitution is to generate *a* sequential program that corresponds to the given PRG. Any program that is represented by the residual PRG would be satisfactory. This is because, as shown by Horwitz et al in [HPR88b], if two programs share the same PDG, they must have the same behaviour in the standard semantics. This result has been extended to PRGs as well, in [RR89].

(a)



(b) $y = 1;$
$y = 0;$
$z = y;$

(c)

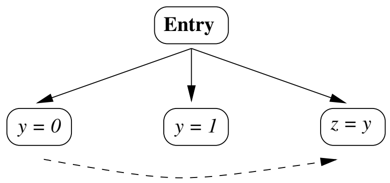


Figure 49: Incorrect reconstitution using topological sorting.
Reconstitution of sequential code from the PRG shown in (a) above may yield the program shown in (b) above, if a topological sort of the PRG is used to order the generated code. The program generated does not have the same behaviour as the program represented by the PRG in (a). It is represented by a different PRG, shown in (c) above, than the PRG in (a).

In other words, a PRG contains, in its dependence edges, enough information to ensure that a PRG is faithful to the semantics of any programs it represents.

However, the information contained in the dependence edges of a PDG or PRG is not in the correct form for making sequential code generation simple. In particular, the intuitive idea of code generation via a topological sort of the PRG may not yield a sequential program that corresponds to the PRG. For instance, consider the PRG in Figure 49 (a). Topological sort may yield the program in Figure 49 (c), which has a different semantics, and therefore a different PRG representation.

The problem arises because program dependence graphs do not contain explicit "output-dependences" and "anti-dependences", which can be used to order vertices that are not connected by flow and control dependences, but whose relative order affects the behaviour of the program. These data dependences were defined by Kuck et al in [KKL+81]: An output-dependence edge connects two definitions of the same variable, indicating that one definition must appear before the other in the program text. Similarly, an anti-dependence edge links a use of a program variable to a definition of the same variable. Such an anti dependence indicates that the use of the variable must occur before the definition in the program text. For instance, for the PRG shown in Figure 49 (a), there is an output-dependence from the assignment $y = 0$ to the assignment $y = 1$, indicating that the assignments must appear in that order in the program text. The program in Figure 49 (b) violates this requirement.

Horwitz et al. have described a method to reconstitute sequential code from PDGs for which program text is not available, in the context of program integration [HPR88a]. Their solution involves constructing dependence edges similar to output-dependence and anti-dependence edges from the PDG, and then emitting program text using a topological sort on the augmented dependence graph. However, the problem of constructing output-dependence and anti-dependence edges from an arbitrary PDG is NP-complete [Ram89]. In the case of reconstitution from the residual PRG, the problem is NP-complete in the size of the residual PRG, which is directly related to the amount of loop unrolling performed by the specialization phase.

Fortunately, we are able to side-step this problem in the following manner: The PRG that exists at the beginning of the specialization phase has a corresponding sequential representation, which is the subject program. We can build certain output-dependence and anti-dependence edges on the original PRG, and modify this information through the steps of specialization. At the end of the specialization process, we then have enough ordering information to produce sequential code using a simple matching scheme that has a worst-case running time limited by the cost of sorting the vertices in the residual PRG. We describe our reconstitution algorithm in detail in the following section.

7.4.1 An algorithm for reconstitution of residual code

The key observation behind our reconstitution algorithm is as follows: Every vertex in the original PRG is replicated zero or more times in the residual PRG. No vertices that do not correspond to vertices in the original PRG are ever produced. As a result, the output-dependences and anti-dependences in the residual PRG are directly related to those in the original PRG. In particular, for every pair of vertices in the original PRG that are related by an output-dependence or anti-dependence, there are zero or more pairs of vertices in the residual PRG that are related by the same dependence. The complication lies in determining, for a pair of vertices in the original PRG, which copies of these vertices in the residual PRG must be related to each other by the given dependence.

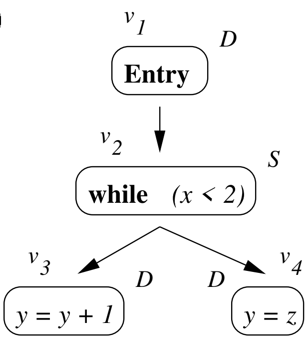
Our algorithm proceeds as follows: First, we build the output-dependences and anti-dependences in the original PRG, by solving straightforward data-flow problems on the CFG of the subject program. We classify these dependences as either loop-independent or loop-carried, as with flow dependences. Next, we perform the BTA phase, which marks every vertex static or dynamic. We are interested in ordering vertices in the residual PRG, not in the original PRG. Therefore, we consider only output-dependences and anti-dependences that connect dynamic vertices. For each such pair, we determine the highest (closest to **Entry**) ancestors in the control dependence tree such that the ancestors are dynamic, and will be siblings in the residual PRG. These ancestors must be ordered correctly in the residual PRG (we refer to these vertices as *source* and *target* for the remainder of this discussion). Pairs *source* and *target* can be identified through simple reachability operations on the PRG.

We first consider loop-independent or forward dependences. In such dependences, both *source* and *target* may in fact be nested within multiple static predicates that will not appear in the residual PRG. This complicates the problem of "matching up" the incarnations of the vertices in the residual PRG correctly. We solve this problem by using "witness" vertices. Recall that both *source* and *target* are dynamic. In addition, they share a common ancestor in the control dependence tree that is also dynamic, such that every predicate between this dynamic ancestor and either *source* or *target* in the control dependence tree of the original PRG is static. For both *source* and *target*, we identify ancestors in the control tree that are one step below the common dynamic ancestor in the control tree. These serve as the witnesses to the dependence. During specialization, we must keep track of iteration counts for these vertices, so as to ensure that an output-dependence or anti-dependence edge is introduced between every copy of *source* and every copy of *target* for which the iteration counts of the witness vertices are identical.

Example 19 In the program shown in Figure 50 (a), vertices u and v are related by an output-dependence. A snippet of the control-dependence tree for the program is shown in Figure 50 (b). The output dependence from u to v is relevant because copies of u in the residual PRG will be siblings of copies of v in the residual PRG. The vertices that are involved in our processing of this dependence are shown in Figure 50 (c). Vertices v_3 and v_4 are the source and target of the dependence, respectively. They have a common dynamic ancestor in v_1; copies of v_3 and v_4 have copies of v_1 as their common parent in the residual PRG. Vertices v_3 and v_4 have the same witness vertex, v_2, which is a static predicate. □

(a) $x = 0;$
read$(y);$
$z = y;$
while$(x < 2)$ $\{$
  $u:\ y = y + 1;$
  $v:\ y = z;$
  $x = x + 1;$
$\}$
write$(y);$

(b)



(c)

source:	v_3	source witness:	v_2
target:	v_4	target witness:	v_2
common ancestor:	v_1		

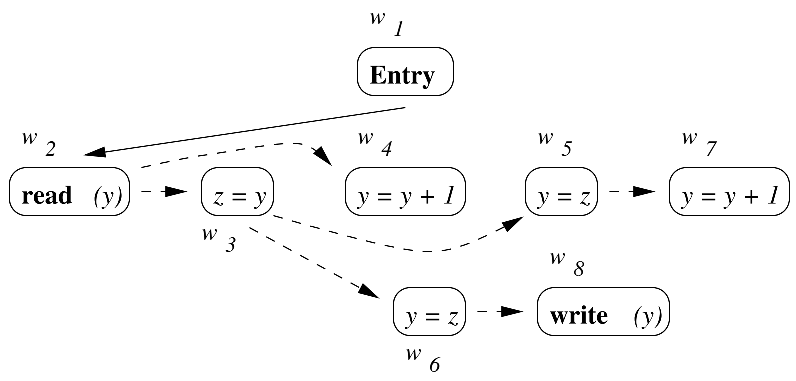
Figure 50: An example of witness vertices.

In the program shown in figure (a) above, vertices u and v are related by an output-dependence. A snippet of the control dependence tree for the program is shown in figure (b) above. The vertices that are involved in our processing of the output-dependence from u to v are shown in figure (c) above.

In the specialization phase, we maintain iteration counts for all vertices that are witnesses for any output-dependences or anti-dependences. Every time a witness vertex produces a new value, it increments its iteration count, and adds a tag to its output value that stores the current iteration count. When a residual vertex is produced by either the source or target vertex of a dependence, the vertex stores iteration counts for the witness along with the particular residual vertex. It also maintains a list of all the residual vertices generated by it. After the specialization phase is complete, and all the residual PRG vertices have been generated, dependences are added to the residual PRG as follows: The lists of residuals for *source* and *target* are sorted, and then walked in tandem. Ordering edges are added between vertices with identical witness counts.

Example 20 The PRG produced by specializing the program from Figure 50 (a) is shown in Figure 51 (a). Note that a topological sort on the PRG may produce code in which the assignment at w_6 appears before the assignment at w_7, which would violate the semantics of the residual PRG. The lists of residual vertices produced by vertices v_3 and v_4 from Figure 50 (b) are shown in Figure 51 (b). During specialization, we maintain an iteration count for v_2 from Figure 50 (b). Each residual copy of vertices v_3 and v_4 is tagged with an iteration count for v_2. These counts are used to "match up" incarnations of v_3 with incarnations of v_4. The resulting output-dependence edges are added to the residual PRG, resulting in the graph shown in Figure 51 (c). Output-dependence edges are shown as dotted arrows. The addition of these edges ensures that the assignment at w_7 appears before the assignment at w_6, as desired. Note that there are other output dependences in the program as well. We ignore those dependences in this discussion. $\square$

(a)



(b) residuals from v_3: $(w_4,1)$, $(w_7,2)$
residuals from v_4: $(w_5,1)$, $(w_6,2)$

(c)

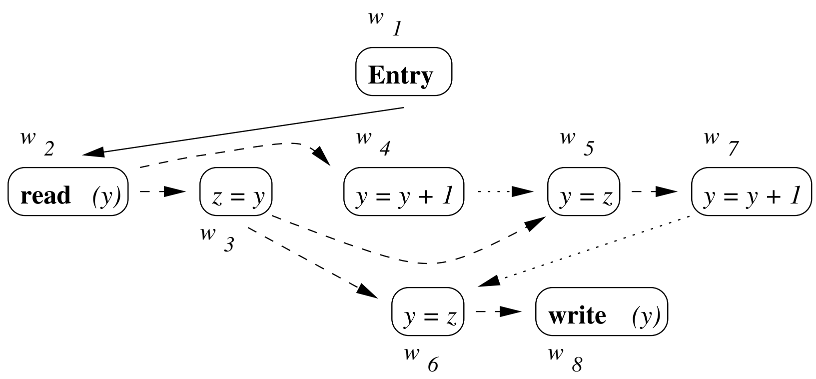


Figure 51: The use of witness vertices in reconstitution.
The PRG produced by specializing the program from Figure 50 (a) is shown in figure 51 (a) above. The lists of residual vertices produced by vertices v_3 and v_4 from Figure 50 (b) are shown in figure (b) above. Output-dependence edges are added between these residual vertices, resulting in the graph shown in figure (c) above.

Finally, we use a code generation phase, in which we generate sequential code using a simple topological sort on a graph that includes the flow and control dependence edges in the residual PRG, as well as the dependence edges introduced as described above. We have omitted many of the details of the procedure from this discussion, for brevity.

In the case of loop-carried output-dependences and anti-dependences, the solution is more involved. A loop-carried dependence is relevant only if the predicate of the loop that carries the dependence is static. In such a case, the loop that carries the dependence will be unrolled during specialization. Therefore, the vertices at the ends of the dependence must be ordered correctly across iterations of the loop. For this purpose, we maintain a second counter at loop vertices, namely the loop count. For a loop vertex, the iteration count represents how many times the loop has been reached, while the loop count represents how many times the loop has been unrolled after being reached on a given occasion. A vertex emitted by the source of a dependence must be placed before a vertex emitted by the target of the dependence if the loop count of the carrying loop corresponding to the source residual is exactly one less than the loop count corresponding to the target residual. In addition, any static loops that are ancestors of the carrying loop in the control dependence tree of the original PRG must have the same loop counts at both the source residual and the target residual. To test this condition, we use a similar matching scheme to the one we use for loop-independent dependences, modified to account for the different nature of the witness vertices.

The reconstitution algorithm sketched above introduces an overhead in the specialization process, even though it is not NP-complete. As the amount of loop unrolling (in terms of the number of iterations of the loop being unrolled in the subject program) increases, the cost of reconstitution increases. For each pair of vertices in the original PRG linked by a dependence edge, the cost of the matching process includes the cost of sorting each vertex's list of residuals by iteration count, and then scanning the lists, matching residuals with related counts. Therefore, the cost of the reconstitution algorithm is controlled by the cost of sorting the vertices in the residual PRG. In practice, the list of residuals corresponding to a given vertex appears to be in sorted order, resulting in a linear sorting cost. However, this cannot be guaranteed.

7.5 Experimental results

We have implemented our specializer using a version of the Wisconsin Program-Slicing Tool developed at the University of Wisconsin [oWM97]. The specializer includes two independent components: The first component parses the source program, constructs the PRG, builds output-dependence and anti-dependence edges, and performs the BTA phase. This component is implemented on top of the Wisconsin slicing tool, which contains modules to construct and slice PDGs. The second component of the implementation is the specializer and code generator.

In the first phase, we use the algorithm described by Cytron et al in [CFR+88] to introduce ϕ_{if} and ϕ_{enter} gate vertices into the CFG. For the remaining filtering ϕ vertices, we use a procedure described partially by Ramalingam in [RR89] to add the appropriate vertices to the PDG augmented with ϕ_{if} and ϕ_{enter} vertices. The BTA algorithms are implemented as slicing operations on the PRG, while output-dependences and anti-dependences are constructed by solving straightforward data-flow

Loop Iterations	CFG Specialization (running time in seconds)	PRG Specialization	PRG Specialization with Reconstitution
1000	0.54	0.63	1.18
5000	2.62	3.15	5.82
10000	5.32	6.25	11.75
20000	10.74	12.56	23.51
30000	16.27	19.00	35.42

Figure 52: Running times for CFG and PRG specialization.
The table above shows running times for CFG specialization (using *cmix*) and PRG specialization using our implementation. The table shows running times for our specializer both with and without reconstitution of sequential code enabled. The numbers in the table above were obtained on a Sparc10 workstation with 64MB RAM, running AFS.

problems on the CFG.

The specialization engine is implemented as a data-flow execution of the PRG, using an abstract vertex class and virtual methods: A vertex class is derived for every kind of PRG vertex, with a local state and a code-generation component. The local state determines how values supplied by predecessor vertices are used to produce new output values and code-generation actions at the given vertex. The code-generation component emits residual vertices, tagged appropriately for reconstitution.

The table in Figure 52 shows the results of one experiment using our implementation. We wish to compare the cost of PRG specialization with the cost of conventional CFG specialization. For purposes of comparison, we use the specialization phase of *cmix* [And94] as the reference CFG-based specializer. The subject program contains a simple static loop with dynamic code nested within it. We report results for our specializer both with and without the reconstitution algorithm enabled.

As shown by the execution times in Figure 52, all three forms of specialization scale linearly with the size of the residual program. PRG specialization without reconstitution is roughly 20% slower than CFG specialization. This is because:

- Every value is tagged with a source vertex identifier and the identifier of the residual vertex corresponding to the source vertex in the residual PRG.
- When a value is received by a target vertex, it must place the value in the appropriate predecessor queue. This can involve several test operations.
- Every value is pushed and popped off queues at successor vertices.

A graphical comparison of the execution times is shown in Figure 53. As shown in the figure, PRG specialization with reconstitution has a running time that is linear in the size of the residual program. This is because vertices are emitted in sorted order, making the running time of the sorting algorithm (we use an insertion sort algorithm) linear. However, this is not guaranteed. It appears reasonable to assume that vertices will almost always appear in sorted order, resulting in linear performance if we use an insertion sort algorithm. The overhead of reconstitution is significant, because iteration-count and loop-count information must be transmitted through the PRG.

7.6 Limitations and related work

We have not designed a PRG-specialization scheme for use with a BTA phase that identifies statically varying vertices rather than weakly static vertices. Extending our algorithm in this direction would involve modifying the reconstitution strategy to account for PRG vertices that may move across the control tree during specialization. When only weakly static vertices are identified, a PRG vertex may only move up the

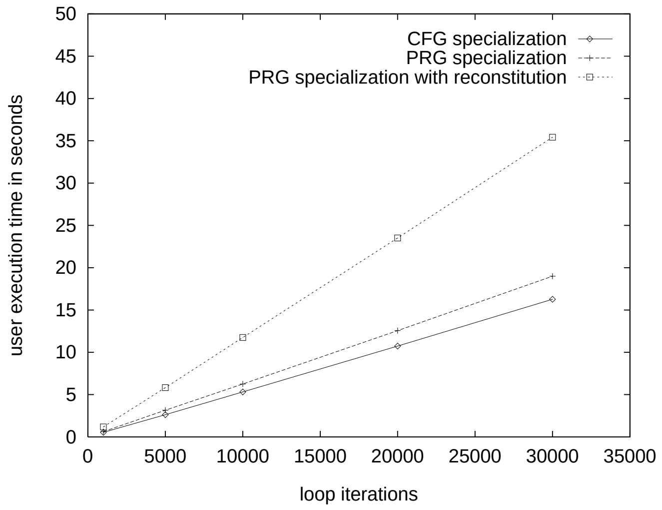


Figure 53: Comparison of execution costs for CFG and PRG specialization. The figure above compares the running time of CFG specialization with that of PRG specialization, using the data from Figure 52. All three methods scale linearly. PRG specialization with reconstitution enabled is roughly twice as slow as CFG specialization.

control tree, as the PRG is "flattened" via loop unrolling and evaluation of branch predicates.

The class of programs that can be specialized is limited to the set of programs that can be represented by PRGs. In particular, it is not clear how we would extend our specialization scheme to handle programs with arbitrary control flow. Extending the algorithm to programs with procedures appears straightforward, although we have not implemented such an extension as of this time.

Much work has been done in the past on executing data-flow graphs, and in designing data-flow machines that can execute data-flow graphs directly. For instance, see [Pap88], [NA89], and [GKW85]. All of these designs use a producer-consumer model, in which a consumer is ready to fire or produce a new output value when it has values, tagged appropriately to ensure that input values are matched correctly, available on all of its input arcs from its predecessors. The same concept has also been applied to other structures such as petri nets [Pet81]. Our PRG execution model is similar to this model. The difference is that we are interested in partial execution of the graph, rather than total execution. The PRG semantics allows us to express PRG specialization as a data-flow computation. As a result, we are able to use the same data-flow execution method to perform the specialization operation.

Chapter 8

Conclusions

This thesis has tackled a known problem in the area of partial evaluation, namely the lack of a termination guarantee. As we have shown and argued in this thesis, the lack of termination arises in two distinct ways: A program may have variables whose values are built up in loops controlled by dynamic data, or a program may have infinite loops that are controlled by only static data. Therefore, we are able to place BTA algorithms in three categories, based on the strength of the termination guarantee they provide for partial evaluation:

(a) Algorithms that provide no termination guarantee.

(b) Algorithms that provide a termination guarantee in the absence of static-infinite computations.

(c) Algorithms that provide a termination guarantee for all programs.

Traditional BTA algorithms that use only flow dependences fall in category (a). We have argued that these algorithms are unsuitable for certain applications. As we showed in Chapter 1, using the power function as an example, values can be transmitted from one program point in a program to another, even though there may be no path of flow dependences relating these program points.

The BTA algorithms presented in Chapters 3, 4, and 5 account for the transmission of values through transfer loops controlled by dynamic data. They use either control dependences or loop dependences, in addition to the usual flow dependences, in order to trace the flow of dynamic values through a program. Therefore, these algorithms can be placed in category (b). Our argument for developing BTA algorithms that provide this partial termination guarantee is as follows: Providing a termination guarantee for programs with static-infinite computations requires conservative analysis of all the static loops in every program, even though static-infinite computations are unlikely to occur in practice, and represent poor programming practice. In imperative programs with complex expressions (in particular, data types that are not downwards closed) and complex control constructs, this leads to conservative results. In Chapter 3, we used the PRG to define a conditionally-safe BTA for single-procedure programs. In Chapter 4, we extended the PRG representation by designing the SRG, and we used this representation to extend our BTA algorithms for single-procedure programs to those with multiple procedures. In Chapter 5, we tackled the problem of developing BTA algorithms in category (b) for programs with arbitrary control flow and pointer-valued variables. It is not clear how we would design a dependence graph representation with a dataflow semantics for such programs. Therefore, we devised a new form of program dependence termed "loop dependence". We used these dependences to develop a BTA algorithm whose correctness can be argued less formally, using the structure of the loop dependence graph. We presented experimental evidence using our implementation to show that the Loop-Dependence BTA is able to identify static behaviour where desired. For this purpose, we used a set of programs that has been used to demonstrate the capability of *cmix*. We claim that this is a reasonable experiment, because *cmix* uses a BTA algorithm that is in category (a).

The drawback of the Loop-Dependence BTA is that the algorithm for constructing loop dependences has a running-time that may be cubic in the size of the program. This clearly limits the applicability of the BTA algorithm to large programs. A conservative alternative is to use control dependences in place of loop dependences, which would eliminate the cost of constructing loop dependences. However, this will result in an unduly conservative BTA algorithm. Identifying a less expensive approximation to loop dependence that retains it selectivity in terms of ignoring dynamic predicates where possible remains an open area for future investigation. The applicability of the Loop-Dependence BTA is also limited by the treatment of heap-allocated storage. However, this problem is independent of our use of loop dependence.

Another area of partial evaluation that remains open to examination is the development of termination analysis algorithms for imperative programs, for use in applications where BTA algorithms in category (c) may be more suitable.

A further open question in the area of partial evaluation of imperative programs is the handling of arrays – for example, identifying partially static arrays. Here dependence graph structures like PRGs may also play some role, as there is a substantial literature, in the context of automatic program parallelization, on array subscript analysis for the purpose of identifying *in*dependent statements.

In the context of functional programs with limited data types, BTA algorithms in category (c) appear most suitable. Several such algorithms have already been devised and reported in the literature. In Chapter 6, we extended one of these algorithms in two ways: (i) By using CFL-Reachability in place of ordinary graph reachability, and (ii) by using an optimistic approach.

In this thesis, we have also used dependence graphs, in particular the PRG, as the basis for the specialization phase of partial evaluation. Our experiments show that there is a significant overhead in partially executing the PRG of a program, when compared with the cost of partially executing the CFG representation of the program. In practice, PRG specialization with reconstitution is roughly twice as slow as CFG specialization, while in the worst case, it may be asymptotically slower. It remains to be determined whether PRG specialization offers any advantages over CFG specialization that would make the additional overhead of the specialization operation worthwhile.

Finally, with partial evaluators increasingly being developed for real imperative languages such as C, there is a need for a standard set of test programs that can be used to evaluate the practical applicability of a particular partial evaluation scheme or implementation. Any such set of programs should include multimedia programs, which appear to hold promise as an application area for partial evaluation of imperative programs [KR96]. This remains as one of the many steps that must be taken before partial evaluation is adopted as a practical technique for improving program performance.

Bibliography

[AH96] P. H. Andersen and C. K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Proceedings of the Third International Static Analysis Symposium*, 1996.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[And92] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 54–61. New Haven, CT: Yale University, June 1992.

[And93] L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 47–58. New York: ACM, 1993.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[And95a] P. H. Andersen. *C-Mix User Manual (DRAFT)*. DIKU, Copenhagen, Denmark, 1995.

[And95b] P.H. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[App92] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[AWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, (San Diego, CA, January 13-15, 1988)*, pages 1–11, 1988.

[Bal93] T. J. Ball. *The Use of Control-Flow and Control Dependence in Software Tools*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1993. UW Computer Sciences Technical Report #1169.

[BFR90] A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, Germany, 1990.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 119–132, 1994.

[BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages, (Charleston, SC, January 10-13, 1993)*, pages 384–396, 1993.

[Bin92] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proc. of the IEEE Conf. on Softw. Maint.* Orlando, FL, Nov. 9-12, 1992, pages 41–50, 1992.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[Bul88] M.A. Bulyonkov. A theoretical approach to polyvariant mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 51–64. Amsterdam: North-Holland, 1988.

[BW90] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[CD89] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. ACM, Berlin: Springer-Verlag, 1991.

[CF89] R. Cartwright and M. Felleisen. The semantics of program dependence. *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation, (Portland, OR, June 21-23, 1989), ACM SIGPLAN Notices*, 24(7), July 1989.

[CFR+88] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages, (Austin, TX, January 11-13, 1989)*, pages 25–35, 1988.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction To Algorithms.* McGraw-Hill Book Company, New Tork, 1990.

[DD77] D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Commun. of the ACM*, 20(7):504–513, July 1977.

[DR95] M. Das and T. Reps. Semantic foundations of binding-time analysis for imperative programs. In *Conference Record of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, San Diego, CA, June 1995*, pages 100–110, 1995.

[DR96] M. Das and T. Reps. BTA termination using cfl-reachability. Technical Report 1329, Computer Sciences Department, University of Wisconsin-Madison, November 1996.

[Ers82] A.P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GJ89] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

[GJ91] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GJ96] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. *Andrei Ershov Second International Conference 'Perspectives of System Informatics', Lecture Notes in Computer Science, 1996*, 1996.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *CACM*, 28(1):34–52, January 1985.

[GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Softw. Eng.*, SE-17(8):751–761, August 1991.

[Har78] A. Haraldsson. A partial evaluator, and its use for compiling iterative statements in Lisp. In *Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 195–202. New York: ACM, 1978.

[Hec77] M. S. Hecht. *Flow analysis of computer programs*. North-Holland, New York, 1977.

[Hol91] C.K. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Berlin: Springer-Verlag, 1991.

[Hor90] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation, (White Plains, NY, June 20-22, 1990), ACM SIGPLAN Notices*, 25(6):234–245, June 1990.

[HPR88a] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, (San Diego, CA, January 13-15, 1988)*, pages 133–145, 1988.

[HPR88b] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, (San Diego, CA, January 13-15, 1988)*, pages 146–157, 1988.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[Jac90] H.F. Jacobsen. Speeding up the back-propagation algorithm by partial evaluation. Student Project 90-10-13, DIKU, University of Copenhagen, Denmark. (In Danish), October 1990.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[Jon88] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.

[Jon95] N. D. Jones. Mix: Ten years after. In *Conference Record of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, San Diego, CA, June 1995*, page ??, 1995.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[Kas65] J. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[Kel95] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Workshop on Intermediate Representations (IR '95) (Technical Report MSR-TR-95-01, Microsoft Research, Microsoft Corporation)*, pages 13–22, 1995.

[KKL+81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, (Williamsburg, VA, January 26-28, 1981)*, pages 207–218, 1981.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Princeton, NJ: D. van Nostrand, 1952.

[Kom81] H.J. Komorowski. *A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[KR96] T. Knoblock and E. Ruf. Data specialization. *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation, (Philadelphia, PA, May 21-24, 1996), ACM SIGPLAN Notices*, pages 215–225, 1996.

[LW86] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Proc. of the First Conf. on Empirical Studies of Programming,* June 1986, pages 133–145, 1986.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 94–105. New York: ACM, 1991.

[Mog92] T. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 116–121. New Haven, CT: Yale University, 1992.

[MR97] D. Melski and T. Reps. Interconvertibility of set constraints and context-free langauge reachability. In *PEPM '97: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 12-13, 1997*, pages 74–89. New York: ACM, 1997.

[MS92] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[NA89] R. S. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture, IEEE/ACM, Jerusalem, Israel, May 1989*, 1989.

[NEK94] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *CACM*, 37(5):50–57, May 1994.

[oWM97] University of Wisconsin-Madison. *Wisconsin Program-Slicing Tool 1.0 Reference Manual.* University of Wisconsin-Madison, Madison, WI, USA, 1997.

[Pap88] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Mulitprocessor.* PhD thesis, Massachussetts Institute of Technology Laboratory for Computer Science, 1988. MIT Laboratory for Computer Science Technical Report 432.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, New Jersey, 1981.

[Ram89] G. Ramalingam, 1989. Personal communication to Tom Reps.

[Rep95] T. Reps. Shape analysis as a generalized path problem. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 1–11. New York: ACM, 1995.

[Rep97] T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97: International Logic Programming Symposium, (Port Jefferson, NY, Oct. 12-16, 1997), J. Maluszynski (ed.)*, pages 5–19, Cambridge, MA, 1997. The M.I.T. Press.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages, (San Francisco, CA, January 23-25, 1995)*, pages 49–61, 1995.

[RHSR94] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, (New Orleans, LA, December 7-9, 1994), ACM SIGSOFT Software Engineering Notes 19(5)*, pages 11–20, December 1994.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation, (Portland, OR, June 21-23, 1989), ACM SIGPLAN Notices*, 24(7):69–80, July 1989.

[RR89] G. Ramalingam and T. Reps. Semantics of program representation graphs. Technical Report TR-900, Computer Sciences Department, University of Wisconsin, Madison, WI, December 1989.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94/14, DIKU, University of Copenhagen, Denmark, April 1994.

[RSH95] T. Reps, M. Sagiv, and S. Horwitz. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Syposium on Principles of Programming Languages, (San Francisco, CA, Jan. 23-25, 1995)*, pages 49–61, 1995.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. *Proceedings of the Dagstuhl Seminar on Partial Evaluation,* Schloss Dagstuhl, Wadern, Germany, Feb. 12-16, 1996*, Lecture Notes in Computer Science*, 1110:409–429, January 1996.

[RWZ88] B. K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, (San Diego, CA, January 13-15, 1988)*, 1988.

[Sch86] D. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., Boston, MA, 1986.

[Sel89] R. P. Selke. A rewriting semantics for program dependence graphs. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages, (Austin, TX, January 11-13, 1989)*, pages 12–24, 1989.

[Wan93] M. Wand. Specifying the correctness of binding-time analysis. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 137–143. ACM, New York: ACM, 1993.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Yan90] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Symposium on Principles of Database Systems, 1990*, pages 230–242, 1990.

[YHR92] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Software Engineering and Methodology*, 1(3):310–354, July 1992.

[You67] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.

Index