Generation of Program Analysis Tools

Frank Tip

Generation of Program Analysis Tools

ILLC Dissertation Series 1995-5

institute *for* logic, language *and* computation

Generation of Program Analysis Tools

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr P.W.M. de Meijer
ten overstaan van een door het college van dekanen
ingestelde commissie in het openbaar te verdedigen
in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op vrijdag 17 maart 1995 te 15.00 uur

door

Frank Tip

geboren te Alkmaar.

Promotor:	prof. dr P. Klint
Co-promotor:	dr J.H. Field
Faculteit:	Wiskunde en Informatica

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Tip, Frank

Generation of program analysis tools / Frank Tip. -
Amsterdam : Institute for Logic, Language and
Computation. - Ill. - (ILLC dissertation series ; 1995-5)
Proefschrift Universiteit van Amsterdam.
ISBN 90-74795-22-6
NUGI 855
Trefw.: programmeertalen / computerprogramma's ; analyse.

Cover design by E.T. Zandstra.
Printed by CopyPrint 2000, Enschede, The Netherlands.

The research presented in this thesis was supported in part by the European Union under ESPRIT project #5399 (Compiler Generation for Parallel Machines—COMPARE).

Contents

Acknowledgments

Most of the research presented in this dissertation was carried out over the last four years in Paul Klint's "Programming Environments" group at CWI. However, part of this work took place at Mark Wegman's "Software Environments" group during a four-month interlude at the IBM T.J. Watson Research Center.

Working for Paul Klint has been a pleasant experience from the very beginning. Paul has always allowed me a lot of freedom in pursuing my own research interests, but was never short of good advice in times of confusion. I am deeply grateful to him for his continuous support, and for being my promotor.

My time at IBM as a research intern was an extremely useful and pleasant learning experience. I am especially indebted to my co-promotor, John Field, for taking care of (too) many "administratrivia" that made my stay at IBM possible. I am deeply grateful to John for actively participating in the realization of this thesis, but his support as a friend who pointed out the peculiarities of the American Way of Life is equally much appreciated.

I would like to thank the co-authors of the chapters of this thesis for their part of the work: Arie van Deursen (Chapter 2), John Field (Chapters 4 and 5), Paul Klint (Chapter 2), and G. Ramalingam (Chapter 5).

The members of the reading committee—dr Lex Augusteijn, prof. dr Jan Bergstra, prof. dr Tom Reps, and prof. dr Henk Sips—provided many useful comments. I wish to thank them for reviewing this thesis.

In addition to those mentioned above, the following persons made a significant contribution to this work by commenting on parts of it: Yves Bertot (Chapter 2), T.B. Dinesh (Chapter 6), Susan Horwitz (Chapter 3), J.-W. Klop (Chapter 2), and G. Ramalingam (Chapter 4). Jan Heering deserves a special acknowledgment for his constructive comments on various parts of this dissertation, but especially for always having an open mind (and an open door) to discuss murky research issues.

My colleagues were an essential part of the pleasant environments that I have worked in. At CWI and the University of Amsterdam: Huub Bakker, Mark van den Brand, Mieke Bruné, Jacob Brunekreef, Arie van Deursen, Casper Dik, T.B. Dinesh, Job Ganzevoort, Jan Heering, Paul Hendriks, Jasper Kamperman, Paul Klint, Steven Klusener, Wilco Koorn, Elena Marchiori, Emma van der Meulen, Jan Rekers, Jan Rutten, Susan Üsküdarlı, Frank Teusink, Eelco Visser, Paul Vriend, and Pum Walters. At the IBM T.J. Watson Research Center: John Field, Barry Hayes, Roger Hoover, Chris Laffra, G. Ramalingam, Jim Rhyne, Kim Solow, Mark Wegman, and Kenny Zadeck. These persons have influenced my work in many ways.

A special word of thanks goes to Frans Vitalis who really "started" this work by exposing me to the mysterious world of computers during visits to my father's office.

Finally I wish to thank my family and friends for their friendship and support in stressful times.

Frank Tip

Amsterdam, The Netherlands
December 1994

Chapter 1

Overview

1.1 Motivation

In recent years, there has been a growing interest in tools that facilitate the understanding, analysis, and debugging of programs. A typical program is extended and/or modified a number of times in the course of its lifetime due to, for example, changes in the underlying platform, or its user-interface. Such transformations are often carried out in an *ad hoc* manner, distorting the "structure" that was originally present in the program. Taken together with the fact that the programmers that worked on previous versions of the program may be unavailable for comment, this causes software maintenance to become an increasingly difficult and tedious task as time progresses. As there is an ever-growing installed base of software that needs to undergo this type of maintenance, the development of techniques that automate the process of understanding, analyzing, and restructuring of software—often referred to as *reverse engineering*—is becoming an increasingly important research topic [21, 117].

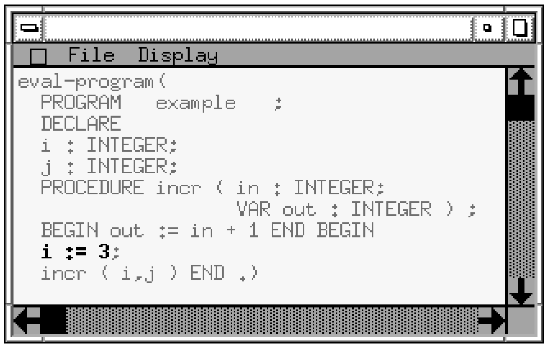
This thesis is concerned with tools and techniques that support the analysis of programs. Instead of directly implementing program analysis tools, our aim is to *generate* such tools from formal, algebraic specifications. Our approach has the pleasant property that it is largely language-independent. Moreover, it will be shown that—to a very large extent—the information needed to construct program analysis tools is already implicitly present in algebraic specifications.

Two particular types of program analysis tools are discussed extensively in the following chapters—tools for *source-level debugging* and tools for *program slicing*.

1.2 Source-level debugging

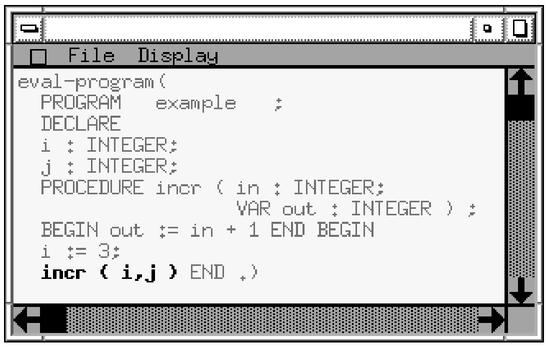
The basic feature of any source-level[1] debugger is to perform or simulate the execution of a program in a step-wise fashion, preferably in some visual way. Figure 1.1 shows a number of snapshots of such a single-stepping tool. In Figure 1.1 **(a)** the first statement of the program is executed; this fact is indicated by highlighting the corresponding statement in the program

[1]The phrase "source-level" indicates that communication between the debugging tool and the person debugging the program is done in terms of source-code.



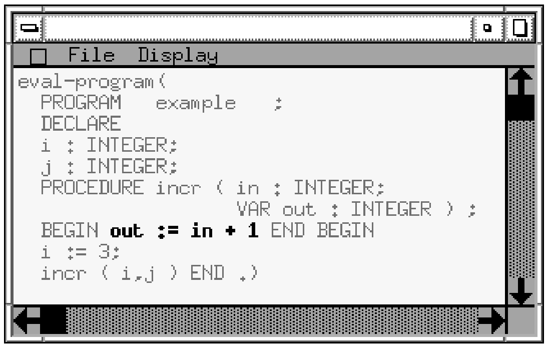
```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(a)



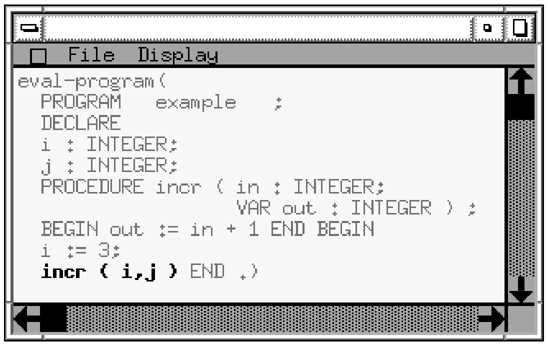
```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(b)



```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(c)



```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(d)

Figure 1.1: Snapshots of a source-level debugging tool.

text. In Figure 1.1 **(b)**, the execution of the next statement is visualized, in this case a call to procedure `incr`. Figure 1.1 **(c)** depicts the execution of the statement `out := in + 1` that constitutes the body of procedure `incr`. Finally, in Figure 1.1 **(d)** execution returns from the procedure.

Another common feature of a source-level debugger is *state inspection*, i.e., allowing the user to query the current values of variables or expressions whenever execution is suspended. For example, a user might ask for the value of variable `in` when the execution reaches the point shown in Figure 1.1 **(c)**. In reaction to this query, the debugging tool will determine the value `3`.

An extremely useful feature that can also be found in most debugging tools is a *breakpoint*. In general, a breakpoint may consist of any constraint on the program state that is supplied by the user. The basic idea is that the user asks to continue execution of the program until that constraint is met. *Control breakpoints* consist of locations in the program (typically statements). When execution reaches such a designated location, the debugging tool will return control to the user. Control breakpoints can be quite useful to verify whether or not a certain statement in a program is executed or not. *Data breakpoints* consist of constraints on values of expressions. For example, a user may ask the debugging tool to continue execution

```
read(n);
i := 1;
sum := 0;
product := 1;
while i <= n do
begin
  sum := sum + i;
  product := product * i;
  i := i + 1
end;
write(sum);
write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin



end;

write(product)
```

(c)

Figure 1.2: **(a)** Example program. **(b)** Static slice of the program with respect to the final value of variable `product`. **(c)** Dynamic slice of the program with respect to the final value of variable `product` for input `n = 0`.

of a program until the values of two designated variables are equal.

1.3 Program slicing

Program slicing [147, 136] is a technique for isolating computational threads in programs. Informally stated, a program slice contains the parts of a program that affect the values computed at some designated point of interest (typically a statement). An alternate view on the notion of a program slice is that of an executable *projection* of a program that replicates part of its behavior. Traditionally, a distinction between *static* and *dynamic* program slicing techniques is made in the literature. The former notion involves statically available information only, i.e., no assumptions regarding the inputs of a program are made. The latter notion, dynamic slicing, assumes a specific execution of the program, i.e., a specific test-case.

The notion of a program slice is best explained by way of an example. Figure 1.2 **(a)** shows a simple program that reads a natural number `n`, and computes the sum and product of the first `n` numbers. Figure 1.2 **(b)** shows a (static) slice of this program with respect to the value of variable `product` that is computed at statement `write(product)`. This slice consists of all statements in the program that are needed to compute the final value of `product` *in any execution*. Observe that neither of the assignments to variable `sum` is present in the slice, because these statements do not have an effect on the computation of `product`'s value for any value of `n`. Figure 1.2 **(c)** shows a dynamic slice of the program in Figure 1.2 **(a)** with respect to the final value of product for the specific test case `n = 0`. Note that the entire body of the **while** loop is omitted from the slice. This is the case because the loop body is not executed if `n` has the value `0`—therefore these statements cannot have an effect on any value computed by the program, and in particular they cannot have an effect on the final value of variable `product`. Also observe that the dynamic slice shown in Figure 1.2 **(c)** is only valid for input `n = 0`; this is evident from the fact that the slice will not terminate for any other value of `n`. Chapters 3, 5, and 6 present several approaches for computing program slices such as as the ones shown in Figure 1.2.

The value of program slicing for program understanding (an important aspect of reverse engineering) should be self-evident: it allows a programmer doing software maintenance to focus his attention on the statements that are involved in a certain computational thread, and to ignore potentially large sections of code that are irrelevant at the point of interest. In a similar way, slicing can be used to examine the effect of modifications to a program, by determining the parts of a program that may be affected by a change.

1.4 Algebraic specifications

As was mentioned previously, our approach will be to generate program analysis tools from formal specifications. More precisely, we will use *algebraic specifications*[2] [23] of a language's semantics as a basis for tool generation. Two important properties of algebraic specifications that underlie our approach are:

- Algebraic specifications may be *executed* by way of *term rewriting* [95] or *term graph rewriting* [17]. This permits us to model the execution of a program abstractly, as a sequence of terms that arise in a rewriting process.
- In Chapters 2 and 4 we will show that algebraic specifications implicitly define *origin* and *dynamic dependence* relations on the terms that arise in any rewriting process according to that specification. These relations are the cornerstones for the generation of various *language-specific* program analysis tools that will be discussed in Chapters 5 and 6.

An algebraic specification consists of a set of (conditional) equations. For specifications of the semantics of imperative programming languages, these equations typically define the “meanings” of statements in terms of transformations of an “environment” or “store”, which is a representation of the values computed by the program.

For example, Figure 1.3 shows two conditional equations (taken from an algebraic specification of an interpreter for a small imperative language that will be presented in Chapter 6). Together, these equations define how the execution of an **if**–**then**–**else**-statement can be expressed in terms of the execution of the statements in the **then**-branch or the **else**-branch of the **if**, depending on the result of the evaluation of its control predicate. The auxiliary function `exec` used in the two equations specifies how environments are transformed by the execution of a list of statements. Figure 1.4 schematically depicts how an application of equation **[L16]** has the effect of transforming an **if**-term; dashed lines in

[2]In this thesis, we will take a rather operational perspective on algebraic specifications by considering these as an equational high-level programming language.

[L16] $\texttt{exec}(\textbf{if}\ \textit{Exp}\ \textbf{then}\ \textit{StatSeq}\ \textbf{else}\ \textit{StatSeq}'\ \textbf{end};\textit{StatSeq}'', \textit{Env}) = \texttt{exec}(\textit{StatSeq}'', \texttt{exec}(\textit{StatSeq}, \textit{Env}))$
 when $\texttt{eval}(\textit{Exp}, \textit{Env}) \neq \texttt{0}$

[L17] $\texttt{exec}(\textbf{if}\ \textit{Exp}\ \textbf{then}\ \textit{StatSeq}\ \textbf{else}\ \textit{StatSeq}'\ \textbf{end};\textit{StatSeq}'', \textit{Env}) = \texttt{exec}(\textit{StatSeq}'', \texttt{exec}(\textit{StatSeq}', \textit{Env}))$
 when $\texttt{eval}(\textit{Exp}, \textit{Env}) = \texttt{0}$

Figure 1.3: Algebraic specification of the execution of an **if**-statement.

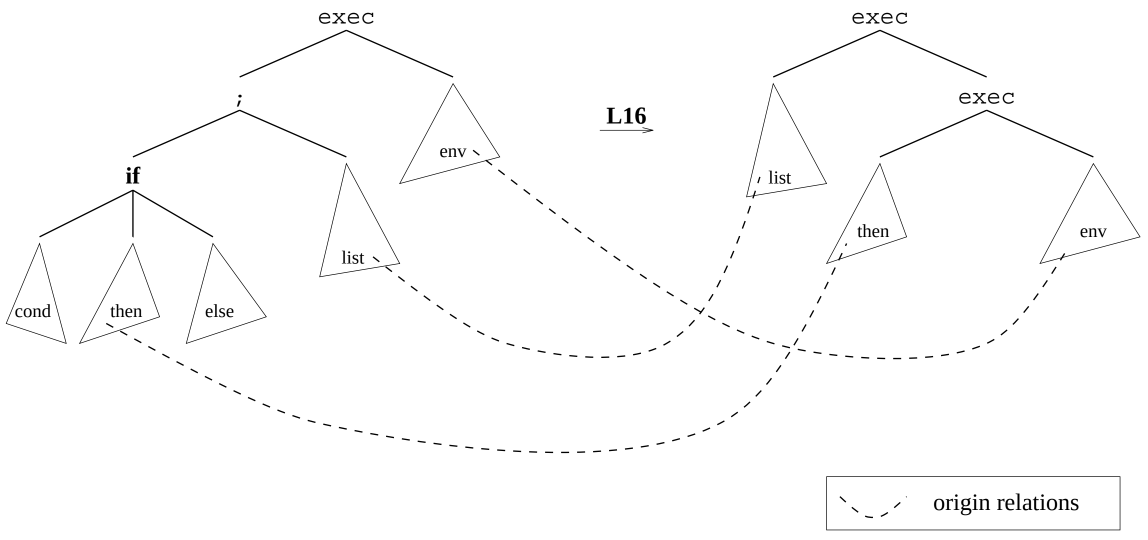


Figure 1.4: Schematic view of origin relations induced by an application of equation **[L16]**.

the figure indicate the origin relations[3] between subterms of the **if**-term, and the term it is rewritten to. Tracing back origin relations in the sequence of terms for the execution of some program is a mechanism for formalizing the notion of a "current locus of execution". This enables us to generate a source-level debugging tool from an algebraic specification of an interpreter.

Figure 1.5 shows two simple axioms for integer arithmetic[4]. Equation **[A1]** states that multiplying the constant `0` with any integer number yields the value `0`, and **[A2]** states that multiplication is an associative operation. Figure 1.6 depicts how, according to these two rules, a term `intmul(intmul(0, 1),2)` may be rewritten to the constant `0`. In this figure, dotted lines indicate dynamic dependence relations. Intuitively, dynamic relations indicate which symbols are necessary for producing certain other symbols. By tracing back dynamic dependence relations from the final term, one may determine which function symbols in the initial term were necessary for creating it. Observe that in the example reduction of Figure 1.6, neither of the constants `1` and `2` in the initial term was necessary for creating the final term `0`.

[3]The reader should be aware that this depiction is a slight simplification—a formal definition of the origin function follows in Chapter 2.

[4]This example is an excerpt of a similar example that occurs in Chapter 6.

```
[A1]  intmul(0,X)                =  0
[A2]  intmul(intmul(X, Y),Z)     =  intmul(X,intmul(Y, Z))
```

Figure 1.5: Some equations for integer arithmetic.

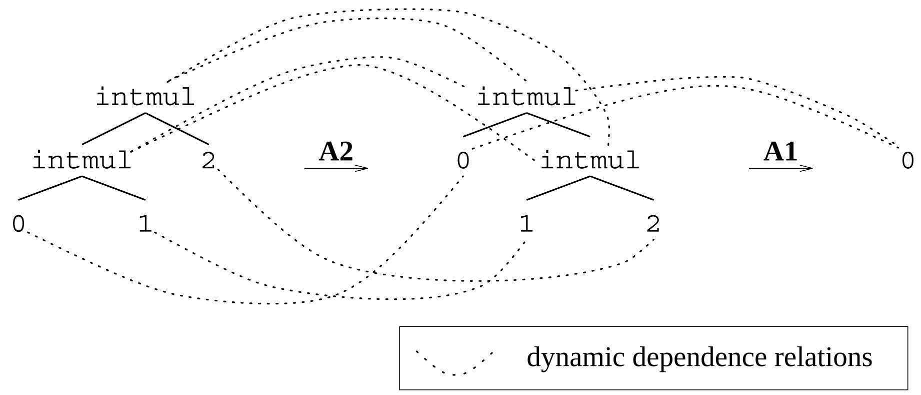


Figure 1.6: Schematic view of dynamic dependence relations induced by applications of **[A1]** and **[A2]**.

In Chapters 5 and 6, it will be shown that applying the dynamic dependence relation to specifications of the semantics of programming languages produces various types of program slices.

In Chapter 6, we present a framework for constructing advanced source-level debuggers that incorporates the features discussed above, and various others.

1.5 Organization of this thesis

The subsequent chapters of this dissertation (except Chapter 7) were originally written as a collection of separate articles on related topics. Although these chapters have since undergone substantial modifications in various places, they can still be read as self-contained papers. Nonetheless, there are some dependences between the material covered in the different chapters. In each of these cases, a small amount of overlap (in the form of reiteration of definitions and examples) was deliberately left in place, for the sake of making the work more accessible.

The foundations of the work in this thesis consist of two relations, origin tracking and dynamic dependence tracking, between an original term and a term it rewrites to:

- *Origin tracking* establishes relations between "equal" terms. In Chapter 2, a formal definition of the origin relation for arbitrary conditional term rewriting systems is presented.

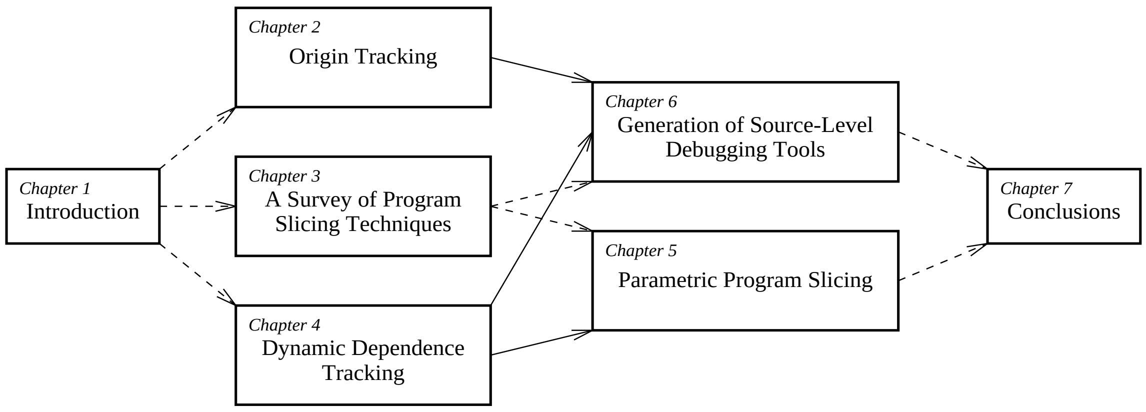

Chapter 1
Introduction
Chapter 2
Origin Tracking
Chapter 3
A Survey of Program Slicing Techniques
Chapter 4
Dynamic Dependence Tracking
Chapter 6
Generation of Source-Level Debugging Tools
Chapter 5
Parametric Program Slicing
Chapter 7
Conclusions

Figure 1.7: Dependences between the chapters in this thesis.

- *Dynamic dependence tracking*, defined in Chapter 4, determines the symbols of the initial term that are necessary for producing symbols of the rewritten term. For the casual reader, the formal definition of dynamic dependence tracking in Chapter 4 could be skipped on an initial reading, as Chapters 5 and 6 contain an informal presentation of dynamic dependence tracking that may be more accessible.

These relations will be used for the generation of program analysis tools, in particular for *program slicing* tools. To put our work in context, related work—in the form of an extensive survey of the current literature on program slicing and its applications—is presented in Chapter 3.

Then, two settings are explored in which these relations are exploited for the generation of tools:

- In Chapter 5, a *translational setting* is described, in which rewrite rules are used to translate terms to an intermediate representation called PIM [55]. Then, other rewrite rules serve to simplify and execute the resulting PIM term. By using different subsets of PIM's simplification and execution rules in combination with the dynamic dependence relation of Chapter 4, various types of program slices are obtained.
- Chapter 6 describes an *interpretive setting*, where program terms are directly manipulated by a set of rewrite rules. In this setting, the origin relation of Chapter 2 is used for the definition of a number of source-level debugging features, and the dynamic dependence relation of Chapter 4 permits the support of dynamic program slicing features.

Finally, in Chapter 7, conclusions and directions for future work are reported. Figure 1.7 depicts the main interdependences between the chapters that follow.

1.6 Origins of the chapters

The subsequent chapters of this thesis are derived from a collection of articles that have previously appeared elsewhere.

Chapter 2, 'Origin Tracking' is a slightly modified version of a paper[5] that appeared in a special issue on 'Automatic Programming' of the *Journal of Symbolic Computation* [47]. This paper was co-authored by Arie van Deursen and Paul Klint. Chapter 3, 'A Survey of Program Slicing Techniques' is an updated version of CWI technical report CS-R9438 [136], and has also been submitted for journal publication. Chapter 4, 'Dynamic Dependence Tracking' is an extended version of a paper[6] entitled 'Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing' that was presented at the *Sixth International Symposium on Programming Language Implementation and Logic Programming* held in Madrid, Spain from September 14–16, 1994 [58]. This paper was written jointly with John Field. Chapter 5, 'Parametric Program Slicing' is an extended version of a paper [57] presented at the *Twenty-Second ACM Symposium on Principles of Programming Languages*, in San Francisco, California from January 23–25, 1995. This paper was written jointly with John Field and G. Ramalingam. Chapter 6, 'Generation of Source-Level Debugging Tools' appeared as CWI technical report CS-R9453, entitled 'Generic Techniques for Source-Level Debugging and Dynamic Program Slicing' [135], and will be presented at the *Sixth International Joint Conference on the Theory and Practice of Software Development*, to be held in Aarhus, Denmark, May 22–26, 1995. Chapter 6 is also loosely based on a paper entitled 'Animators for Generated Programming Environments' that was presented at the *First International Workshop on Automated and Algorithmic Debugging* held in Linköping, Sweden from May 3–5, 1993 [134].

Some of the papers mentioned above have also appeared as deliverables of the COMPARE project. For an overview of this project, the reader is referred to [9, 91].

[5]Academic Press is acknowledged for their permission to reprint parts of this paper.
[6]Springer-Verlag is acknowledged for their permission to reprint parts of this paper.

Chapter 2

Origin Tracking

(joint work with Arie van Deursen and Paul Klint)

Summary

We are interested in generating interactive programming environments from formal language specifications and use term rewriting to execute these specifications. Functions defined in a specification operate on the abstract syntax tree of programs, and the initial term for the rewriting process will consist of an application of some function (e.g., a type-checker, evaluator or translator) to the syntax tree of a program. During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms. We want to formalize these recurrences and use them, for example, for associating positional information with messages in error reports, visualizing program execution, and constructing language-specific debuggers. *Origins* are relations between subterms of intermediate terms and subterms of the initial term. *Origin tracking* is a method for incrementally computing origins during rewriting. We give a formal definition of origins, and present a method for implementing origin tracking.

This chapter is mainly concerned with technical foundations; Chapter 6 will discuss in detail how origin tracking can be used for the generation of source-level debugging tools.

2.1 Introduction

We are interested in generating interactive development tools from formal language definitions. Thus far, this has resulted in the design of an algebraic specification formalism, called ASF+SDF [23, 68] supporting modularization, user-definable syntax, associative lists, and conditional equations, and in the implementation of the ASF+SDF Meta-environment [69, 93].

Given a specification for a programming (or other) language, the Meta-environment generates an interactive environment for the language in question. More precisely, the Meta-environment is a tool generator that takes a specification in ASF+SDF and derives a lexical analyzer, a parser, a syntax-directed editor and a rewrite engine from it. The Meta-environment provides fully interactive support for writing, checking, and testing specifications—all tools are generated in an incremental fashion and, when the input specification is changed, they are updated incrementally rather than being regenerated from scratch. A central objective in this research is to maximize the direct use that is made of the formal specification of a language when generating development tools for it.

We use Term Rewriting Systems (TRSs) [95] to execute our specifications. A typical function (such as an evaluator, type checker, or translator) is a specification that operates on the abstract syntax tree of a program (which is part of the initial term). During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms. We want to formalize these recurrences and use them, for example, for:

- associating positional information with messages in error reports;
- visualizing program execution;
- constructing language-specific debuggers.

Our approach to formalize recurrences of subterms consists of two stages. First, we define relations for elementary reduction steps $t_i \to t_{i+1}$; these relations are described in Section 2.1.3. Then, we extend these relations to compound reduction sequences $t_0 \to t_1 \to \ldots \to t_n$. In particular, we are interested in relations between subterms of an intermediate term t_i, and subterms of the initial term t_0. We will call this the *origin relation*. Intuitively, it formalizes from which parts of the initial term a particular subterm originates. The process of incrementally computing origins we will call *origin tracking*.

2.1.1 Applications of origin tracking

In TRSs describing programming languages terms such as

```
program(decls(decl(n,natural)), stats(assign(n,34)))
```

are used to represent abstract syntax trees of programs. A typical type-check function takes a program and computes a list of error messages. An example of the initial and final term when type checking a simple program is shown in Figure 2.1.

The program uses an undeclared variable `n1`, and the result of the type-checker is a term representing this fact, i.e., a term with `undeclared-var` as function symbol and the name `n1` of the undeclared variable as argument. The dashed line represents an origin: it relates the occurrence of `n1` in the result to the `n1` in the initial term. One can use this to highlight the exact position of the error in the source program. Figure 2.2 shows an application of this technique.

Similarly, program evaluators can be defined. Consider for example a rule that evaluates a list of statements by evaluating the first statement followed by the remaining statements:

[L1] `ev-list(cons(`*Stat*`,`*S-list*`),`*Env*`)` $\to$ `ev-list(`*S-list*`,ev-stat(`*Stat*`,`*Env*`))`

The variables (*Stat*, *S-list*, and *Env*) are used to pass information from the left to the right-hand side. The origins of these variable occurrences in the right-hand side are shown by dashed lines in Figure 2.3.

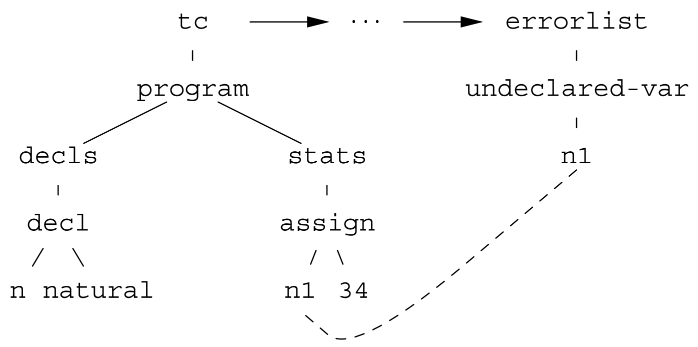
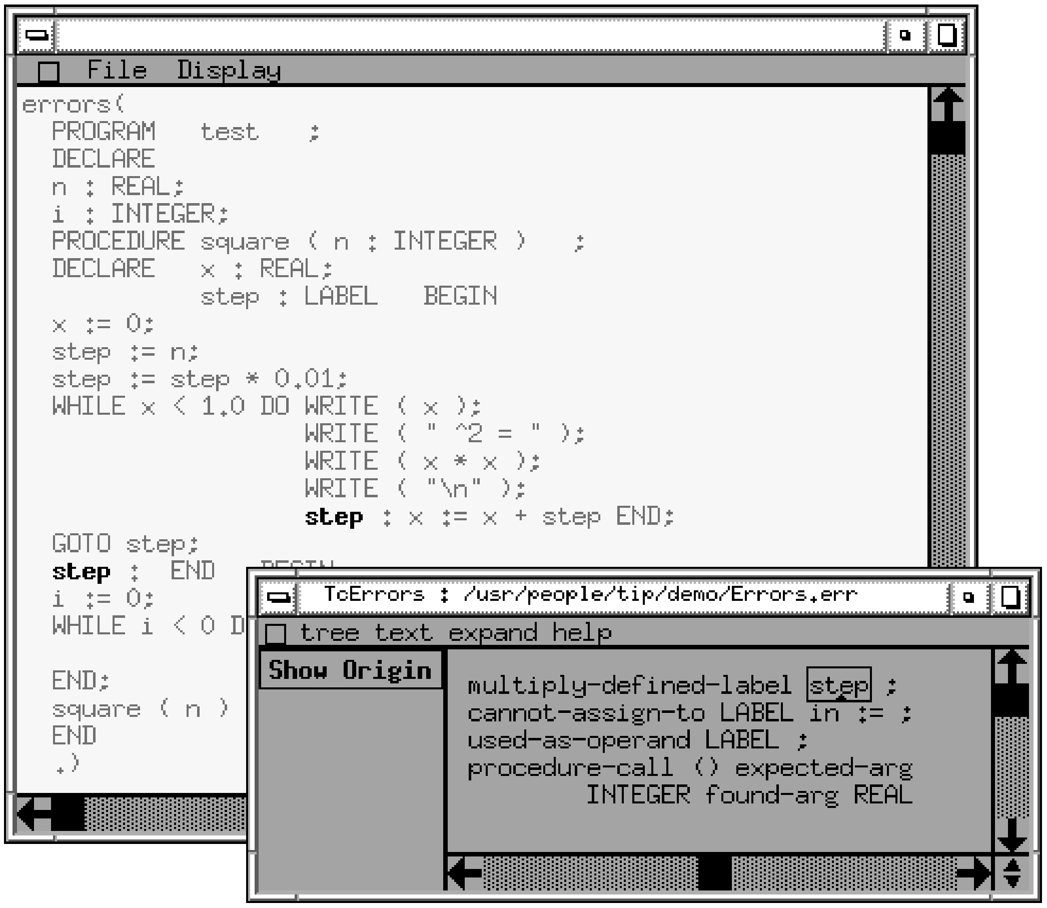


Figure 2.1: Type-checking a simple program.



```
errors(
  PROGRAM   test   ;
  DECLARE
  n : REAL;
  i : INTEGER;
  PROCEDURE square ( n : INTEGER )   ;
  DECLARE   x : REAL;
            step : LABEL   BEGIN
  x := 0;
  step := n;
  step := step * 0.01;
  WHILE x < 1.0 DO WRITE ( x );
                   WRITE ( " ^2 = " );
                   WRITE ( x * x );
                   WRITE ( "\n" );
                   step : x := x + step END;
  GOTO step;
  step :  END
  i := 0;
  WHILE i < 0 D

  END;
  square ( n )
  END
  .)
```

```
multiply-defined-label step ;
cannot-assign-to LABEL in := ;
used-as-operand LABEL ;
procedure-call () expected-arg
        INTEGER found-arg REAL
```

Figure 2.2: Highlighting occurrences of errors.

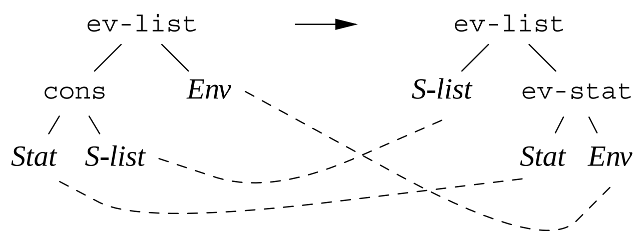


Figure 2.3: One step in the evaluation of a simple program. Dashed lines indicate how the rule application induces origin relations.

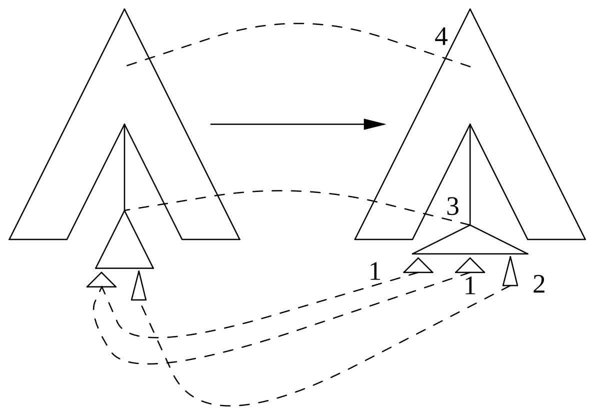
Visualization of program execution is a natural application of origin tracking. The basic idea is that, during execution, the statement currently being executed is highlighted in the source text. In the sequel, it will be shown how this can be accomplished by matching redexes against the pattern `ev-stat(`*Stat*`,` *Env*`)`; whenever such a match occurs, the origin of the first argument of `ev-stat` indicates the statement that is currently being executed. Dinesh and Tip [49, 134] have shown how, by employing multiple patterns, program execution can be animated in a very fine-grained manner: the execution of any language construct (e.g., expressions, declarations) can be traced. This is particularly useful for applications such as source-level debugging and tutoring.

In a similar way, various notions of breakpoints can be defined. Source-level debuggers often have a completely fixed notion of a breakpoint, based on line-numbers, procedure calls and machine addresses. By contrast, the origin relation enables one to define breakpoints in a much more uniform and generic way. For instance, a *positional* breakpoint can be created by having the user select a certain point in the source text. The path from the root to that point is recorded and the breakpoint becomes effective when—in this example—the origin of the first argument of `ev-stat` equals that path. *Position-independent* breakpoints can be defined by using patterns describing statements of a certain form (e.g., an assignment with `x` as left-hand side). The breakpoint becomes effective when the argument of `ev-stat` matches that pattern; its origin shows the position in the original program. The definition of these, and other debugging concepts will be further explored in Chapter 6.

2.1.2 Points of departure

Before sketching the origin relation (in Section 2.1.3) we briefly state our points of departure:

- No assumptions should be made regarding the choice of a particular reduction strategy.
- No assumptions should be made concerning confluence or termination; origins can be established for arbitrary reductions in any TRS.
- The origin relation should be obtained by a static analysis of the rewrite rules.



1. **common variables**
2. **common subterms**
3. **redex-contractum**
4. **contexts**

Figure 2.4: Single-step origin relations.

- Relations should be established between any intermediate term and the initial term. This implies that relations can be established even if there is no normal form.
- Origins should satisfy the property that if t has an origin t', then t' can be rewritten to t in zero or more steps.
- The origin relation should be *transitive*.
- An efficient implementation should exist.

These requirements do not lead, however, to a unique solution. We will therefore only present one of the possible definitions of origins, although we can easily imagine alternative ones.

2.1.3 Origin relations

The definition of the origin relation is based on the transitive and reflexive closure of a number of *single-step origin relations* for elementary reductions, which will now be studied in some detail. In the description that follows it is assumed that a rewrite rule $r : t_1 \rightarrow t_2$ is applied in context C with substitution σ, giving rise to the elementary reduction $\mathrm{C}[t_1^\sigma] \rightarrow_r \mathrm{C}[t_2^\sigma]$. Figure 2.4 depicts the four types of single-step origin relations that occur:

common variables. If a variable X appears in both sides, t_1 and t_2, of rule r, then relations are established between each function symbol in the instantiation X^σ of X in $\mathrm{C}[t_1^\sigma]$ and the corresponding function symbol in each instantiated occurrence of X in $\mathrm{C}[t_2^\sigma]$.

Figure 2.5 illustrates how the variable *X* induces relations between corresponding function symbols for a specific application of the rule `f(`*X*`)` $\rightarrow$ `g(`*X*`)`.

The common variables relation becomes a bit more complicated for left-nonlinear rules, i.e., rules where some variable *X* occurs more than once in the left-hand side, e.g., `plus(`*X*`,`*X*`)` $\rightarrow$ `mul(2,` *X*`)`. In this case, all occurrences of *X* in the left-hand side give

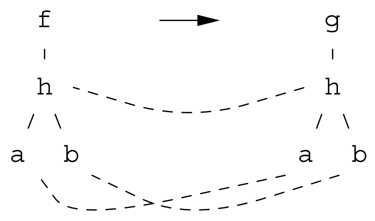


Figure 2.5: Relations according to a variable occurrence in both sides of a rule.

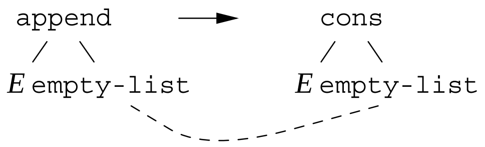


Figure 2.6: Relations according to common subterms rule.

rise to origin relations. In other words, nonlinearity in the left-hand side may cause non-uniqueness of origins.

common subterms. If a term s is a subterm of both t_1 and of t_2, then these occurrences of s give rise to common subterms relations between their instantions. Consider, for example, the following rules that define how an element is appended to the end of a list:

[A1] `append(`E`,empty-list)` $\rightarrow$ `cons(`E`,empty-list)`
[A2] `append(`E_1`,cons(`E_2`,`L`))` $\rightarrow$ `cons(`E_2`, append(`E_1`,`L`))`

Using the common variables relation, several useful origin relations will be constructed. However, no such relation is present for the constant `empty-list` that occurs in either side of **[A1]**. This relation is established by the common subterms rule, and is depicted in Figure 2.6. A more elaborate example involving the common subterms relation is the conditional rule **[W1]** for evaluating `while`-statements:

[W1] `ev-stat(while(`*Exp*`,`*S-list*`),`*Env*`)` $\rightarrow$
`ev-stat(while(`*Exp*`,`*S-list*`),ev-list(`*S-list*`,`*Env*`))`
when `ev-exp(`*Exp*`, `*Env*`) = true`

When evaluation of *Exp* yields true, the same while-statement is evaluated in a modified environment that is obtained by evaluating the body of the while-statement (*S-list*) in the initial environment (*Env*). The common subterms relation links these `while`-symbols.

redex-contractum. The top symbol of the redex t_1^σ and the top symbol of its contractum t_2^σ are related, as shown in Figure 2.7 for rule **[A2]**. An essential application of this relation can be seen in

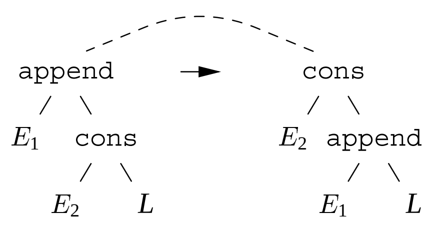


Figure 2.7: Relations according to redex-contractum rule.

[R1] `real-const(`*Char-list*`)` $\rightarrow$ `real-type`

where a real constant containing a list of characters is rewritten to its type denotation `real-type`. Observe that the redex-contractum relation may cause non-uniqueness of origins.

contexts. Relations are established between each function symbol in the context C of the left-hand side and its counterpart in the context C of the right-hand side.

It is obvious how in a chain of elementary reductions, the transitive closure of the single-step origin relations can be used to determine the origins of any subterm in the reduction.

In an alternate, more implementation-oriented view, subterms are annotated with their origins (as sets of paths in the original term). For each reduction, the origins of the redex are propagated to the contractum in accordance with the single-step origin relations.

Origin tracking for conditional TRSs (CTRSs) is an extension of the origin function for unconditional TRSs, but is slightly more complicated. The main complications arise from the fact that we want to be able to determine origins of terms that appear in the (sub)reductions that are necessary for the evaluation of conditions.

If evaluation of a condition involves reduction of a term t, the origins of the redex are passed to t, according to the common variables rule and the common subterms rule. These origins are subsequently propagated to the normal form of t, according to the usual origin relations. If a condition introduces variables, then these are matched against normal forms that have already origins associated with them. Such variables may be re-used in other conditions, and in the right-hand side of the rewrite rule.

2.2 Formal definition

In this section, we present a formal definition of origin tracking. A basic knowledge of term rewriting systems (TRSs), and conditional term rewriting systems (CTRSs) is assumed. For a detailed discussion of these, the reader is referred to [95].

The remainder of this section is organized as follows. First, we introduce basic concepts and rewriting histories for unconditional TRSs. Subsequently, the origin function for unconditional TRSs is defined, and illustrated by way of an example. After discussing basic concepts and rewriting histories for CTRSs, we consider the origin function for CTRSs.

We have used the formal definition of the origin relation to obtain an executable specification of origin tracking. The examples that will be given in this section have been verified automatically using that specification.

2.2.1 Basic concepts for unconditional TRSs

A notion that will frequently recur is that of a *path* (occurrence), consisting of a (possibly empty) sequence of natural numbers between brackets. Paths are used to indicate subterms of a term by interpreting numbers as argument positions of function symbols. For instance, (2 1) indicates subterm `b` of term `f(a, g(b, c))`. This is indicated by the '$/$' operator: `f(a, g(b, c))`$/$(2 1) = `b`. The associative operator '$\cdot$' concatenates paths, e.g., (2) $\cdot$ (1) = (2 1). The operators '$\prec$', '$\preceq$', and '$|$' define the prefix ordering on paths. The fact that p is a prefix of q is denoted $p \prec q$; '$\preceq$' is the reflexive closure of '$\prec$'. Two paths p and q are disjoint (denoted by $p \mid q$) if neither one is a prefix of the other.

The set of all valid paths in a term t is $O(t)$. The set of variables occurring in t is denoted $Vars(t)$. We use $t' \subset t$ to express that t' appears as a subterm of t; the reflexive closure of '$\subset$' is '$\subseteq$'. The negations of '$\subset$' and '$\subseteq$' are '$\not\subset$' and '$\not\subseteq$', respectively. Finally, *Lhs*(r) and *Rhs*(r) indicate the left-hand side and the right-hand side of a rewrite rule r.

2.2.2 A formalized notion of a rewriting history

A basic assumption in the subsequent definitions is that the complete history of the rewriting process is available. This is by no means essential to our definitions, but has the following advantages:

- The origin function for CTRSs can be defined in a declarative, non-recursive manner. We encountered ill-behaved forms of recursion in the definition itself when we experimented with more operational definition methods, due to the convoluted structure of rewriting histories for CTRSs.
- Uniformity of the origin functions for unconditional TRSs and for CTRSs. The latter can be defined as an extension of the former.

In the case of unconditional TRSs, the *rewriting history* $\mathcal{H}$ is a single *reduction sequence* $\mathcal{S}$. This sequence consists of a list of *sequence elements* $\mathcal{S}_i$ that contain all information involving the i^{th} rewrite-step. Here, i ranges from 1 to $|\mathcal{S}|$ where $|\mathcal{S}|$ is the length of sequence $\mathcal{S}$.

Each sequence element is a 5-tuple $(n,\ t,\ r,\ p,\ \sigma)$ where n is the name of the sequence element (consisting of a sequence name and a number), t denotes the i^{th} term of sequence $\mathcal{S}$, r the i^{th} rewrite rule applied, p the path to the redex in t, and σ the substitution used in the application of r. Access functions $n(s)$, $t(s)$, $r(s)$, $p(s)$, and $\sigma(s)$ are used to obtain the components of s. The last element of a sequence is irregular, because the term associated with this element is in normal form: the rule, path and substitution associated with $\mathcal{S}_{|\mathcal{S}|}$ consist of the special value *undefined*.

Below, s, s', and s'' denote sequence elements. Moreover, it will be useful to have a notion $\mathcal{H}^-$ denoting the history $\mathcal{H}$ from which the last sequence element, $\mathcal{S}_{|\mathcal{S}|}$, is excluded. For our convenience, we introduce $Lhs(s)$ and $Rhs(s)$ to denote the left-hand side and right-hand side of $r(s)$. Finally, $Succ(\mathcal{H},\ s)$ denotes the successor of s, for s in $\mathcal{H}^-$, and $Start(\mathcal{H})$ determines the first element of the reduction sequence in $\mathcal{H}$.

2.2.3 The origin function for unconditional TRSs

2.2.3.1 Auxiliary notions

The auxiliary function *Com* (Definition 2.1 below) is frequently used in the definitions of the origin functions below, to compute positions of common variables and common subterms. The arguments of *Com* are a substitution σ and two terms t and t'. The result computed by *Com* is a set containing pairs $(p,\ p')$ such that either a variable X or a common subterm t'' occurs both at path p in t and at path p' in t'.

Definition 2.1 (*Com*)

$$\begin{aligned} \mathit{Com}(\sigma,\ t,\ t') \ \triangleq\ & \{\ (p \cdot q,\ p' \cdot q) \mid t/p \in \mathit{Vars}(t),\ t/p = t'/p',\ q \in O(\sigma(t/p))\ \} \cup \\ & \{\ (p,\ p') \mid t/p \notin \mathit{Vars}(t),\ t/p = t'/p'\ \} \end{aligned}$$

For one-step reductions, the basic origin relation *LR* (short for Left-hand side to Right-hand side) relates common subterms of a redex and its contractum that appear as a result of the presence of a common variable or a common subterm in the applied rewrite rule.

Definition 2.2 (*LR*) *For s in $\mathcal{H}^-$: $\mathit{LR}(s) \triangleq \mathit{Com}(\sigma(s),\ \mathit{Lhs}(s),\ \mathit{Rhs}(s))$*

2.2.3.2 Definition of *ORG*

The origin function *ORG* for unconditional TRSs is defined using *LR*. Relations are represented by *relate* clauses: a clause $relate(\mathcal{H},\ s',\ p',\ s,\ p)$ indicates a relation between the subterm at path p' in $t(s')$ and the subterm at path p in $t(s)$ in history $\mathcal{H}$. In **(u1)**, all relations between symbols in the redex in $t(s)$ and its contractum in $t(Succ(\mathcal{H},\ s))$ are defined, excluding the top symbols of the redex and the contractum. The fact that all symbols in the context of the redex remain unchanged is expressed in **(u2)**. In addition, the top symbols of the redex and the contractum are related by **(u2)**.

For s in $\mathcal{H}$ and a path p in $t(s)$, the set of related subterms (according to the transitive and reflexive closure of *relate*) in the initial term, $t(Start(\mathcal{H}))$, is denoted $\mathit{ORG}(\mathcal{H},\ s,\ p)$.

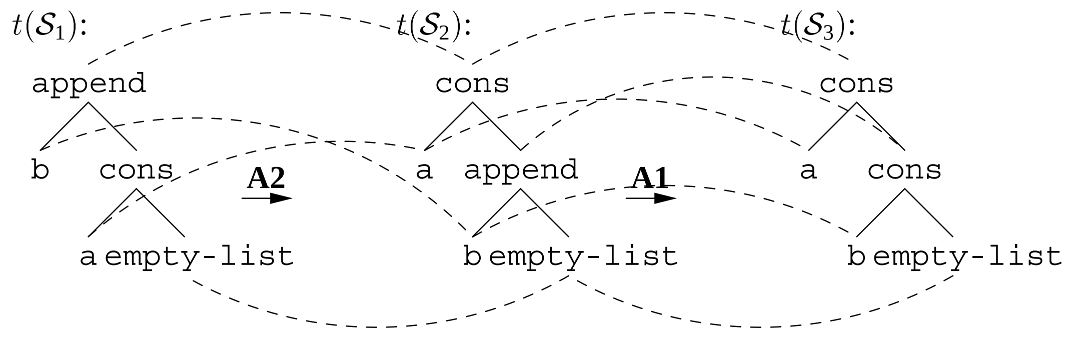


Figure 2.8: History $\mathcal{H}_{app}$ for `append(b, cons(a, empty-list))`. Dashed lines indicate origin relations.

Definition 2.3 (*ORG*) *For s in $\mathcal{H}^-$ and s' in $\mathcal{H}$:*

(**u1**) $\forall(q,\ q') \in \mathit{LR}(s)$: $\mathit{relate}(\mathcal{H},\ s,\ p(s)\cdot q,\ \mathit{Succ}(\mathcal{H},\ s),\ p(s)\cdot q')$
(**u2**) $\forall p : (p \preceq p(s)) \vee (p \mid p(s))$: $\mathit{relate}(\mathcal{H},\ s,\ p,\ \mathit{Succ}(\mathcal{H},\ s),\ p)$

$$\mathit{ORG}(\mathcal{H},\ s,\ p)\ \triangleq \begin{cases} \{\, p \,\} & \textit{when } s = \mathit{Start}(\mathcal{H}) \\ \\ \{\, p'' \mid\ p'' \in \mathit{ORG}(\mathcal{H},\ s',\ p'), & \textit{when } s \neq \mathit{Start}(\mathcal{H}) \\ \qquad\quad \mathit{relate}(\mathcal{H},\ s',\ p',\ s,\ p)\,\} & \end{cases}$$

In principle, the availability of all *relate* clauses allows us to determine relationships between subterms of two arbitrary intermediate terms that occur during the rewriting process. However, we will focus on relations involving the initial term.

2.2.3.3 Example

As an example, we consider the TRS consisting of the two rewrite rules **[A1]** and **[A2]** of section 2.1.3. Figure 2.8 shows a history $\mathcal{H}_{app}$, consisting of a sequence $\mathcal{S}$, as obtained by rewriting the term `append(b, cons(a, empty))`.

Below, we argue how the origin relations shown in Figure 2.8 are derived from Definition 2.3. For the first sequence element, $\mathcal{S}_1$, we have $p(\mathcal{S}_1) = ()$, $r(\mathcal{S}_1)$ = **[A2]**, and $\sigma(\mathcal{S}_1) = \{\ E_1 \mapsto$ `b`, $E_2 \mapsto$ `a`, $L \mapsto$ `empty-list` $\}$. As all variable bindings are constants here, we have: $O({E_1}^{\sigma(\mathcal{S}_1)})$ = $O({E_2}^{\sigma(\mathcal{S}_1)})$ = $O(L^{\sigma(\mathcal{S}_1)})$ = $\{\ ()\ \}$. From this, we obtain:

$$\mathit{LR}(\mathcal{S}_1) = \mathit{Com}(\sigma(\mathcal{S}_1),\ \mathit{Lhs}(\mathcal{S}_1),\ \mathit{Rhs}(\mathcal{S}_1)) = \{\ ((1),\ (2\,1)),\ ((2\,1),\ (1)),\ ((2\,2),\ (2\,2))\ \}$$

In a similar way, we compute:

$$\mathit{LR}(\mathcal{S}_2) = \mathit{Com}(\sigma(\mathcal{S}_2),\ \mathit{Lhs}(\mathcal{S}_2),\ \mathit{Rhs}(\mathcal{S}_2)) = \{\ ((1),\ (1)),\ ((2),\ (2))\ \}$$

From Definition 2.3 we now derive the following *relate* relationships. (Note that the last three relationships are generated according to **(u1)** of Definition 2.3.)

$$\begin{array}{ll}
relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (1),\ \mathcal{S}_2,\ (2\ 1)) & relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1),\ \mathcal{S}_2,\ (1)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 2),\ \mathcal{S}_2,\ (2\ 2)) & relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (),\ \mathcal{S}_2,\ ()) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2\ 1),\ \mathcal{S}_3,\ (2\ 1)) & relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2\ 2),\ \mathcal{S}_3,\ (2\ 2)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (),\ \mathcal{S}_3,\ ()) & relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1),\ \mathcal{S}_3,\ (1)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2),\ \mathcal{S}_3,\ (2)) &
\end{array}$$

As an example, we compute the subterms related to the constant `a` at path (1) in $t(\mathcal{S}_3)$:

$$\begin{aligned}
\mathit{ORG}(\mathcal{H}_{app},\ \mathcal{S}_3,\ (1)) &= \{\, p'' \mid p'' \in \mathit{ORG}(\mathcal{H}_{app},\ s',\ p'),\ relate(\mathcal{H}_{app},\ s',\ p',\ \mathcal{S}_3, (1)) \,\} \\
&= \{\, p'' \mid p'' \in \mathit{ORG}(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1)) \,\} \\
&= \mathit{ORG}(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1)) \\
&= \{\, p'' \mid p'' \in \mathit{ORG}(\mathcal{H}_{app},\ s',\ p'),\ relate(\mathcal{H}_{app},\ s',\ p',\ \mathcal{S}_2, (1)) \,\} \\
&= \{\, p'' \mid p'' \in \mathit{ORG}(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1)) \,\} \\
&= \mathit{ORG}(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1)) \\
&= \{\ (2\ 1)\ \}
\end{aligned}$$

Hence, the constant `a` at path (1) in $t(\mathcal{S}_3)$ is related to the constant `a` at path (2 1) in the initial term.

We conclude this example with a few brief remarks. First, some symbols in $t(\mathcal{S}_3)$ are not related to any symbol of $t(\mathcal{S}_1)$. For instance, symbol `cons` at path (2) in $t(\mathcal{S}_3)$ is only related to symbol `append` in $t(\mathcal{S}_2)$; this symbol, in turn, is not related to any symbol in $t(\mathcal{S}_1)$. Second, we have chosen a trivial example where no origins occur that contain more than one path. Such a situation may arise when a rewrite rule is not left-linear, or when the right-hand side of a rewrite rule consists of a common variable or a common subterm.

2.2.4 Basic concepts for CTRSs

A conditional rewrite-rule takes the form:

$$\mathit{lhs} \rightarrow \mathit{rhs} \quad \textbf{when} \quad l_1 = r_1,\ \cdots,\ l_n = r_n$$

We assume that CTRSs are executed as join systems [95]: both sides of a condition are instantiated and normalized. A condition succeeds if the resulting normal forms are syntactically equal. It is assumed that the conditions of a rule are evaluated in left-to-right order. As an extension, we allow *one* side of a condition to introduce variables[1]; we will refer to such variables as *new* variables (as opposed to *old* variables that are bound during the matching of the left-hand side, or during the evaluation of a previous condition). To avoid complications in our definitions, we impose the non-essential restriction that no condition side may contain old as well as new variables. New variables may occur in subsequent conditions as well as in the right-hand side. Variable-introducing condition sides are *not*

[1]An example CTRS with variable-introducing conditions will be discussed in Section 2.2.6.3 below.

normalized, but matched against the normal form of the non-variable-introducing side (for details, see [140]). Given the above discussion, conditional term rewriting can be regarded as the following cyclic 3-phase process:

1. Find a match between a subterm t and the left-hand side of a rule r.
2. Evaluate the conditions of r: instantiate and normalize non-variable-introducing condition sides.
3. If all conditions of r succeed: replace t by the instantiated right-hand side of r.

In will be convenient to introduce some auxiliary notions that formalize the introduction of variables in conditions. Let $|r|$ be the number of conditions of r. For $1 \leq j \leq |r|$, the left-hand side and the right-hand side of the j^{th} condition of r are denoted $Side(r,\ j,\ \mathit{left})$ and $Side(r,\ j,\ \mathit{right})$, respectively. Moreover, let $\overline{\mathit{left}} = \mathit{right}$ and $\overline{\mathit{right}} = \mathit{left}$. The function *VarIntro* (Definition 2.4) indicates where new variables occur; tuples $(h,\ side)$ are computed, indicating that $Side(r,\ h,\ side)$ is variable-introducing.

Definition 2.4 (*VarIntro*)

$$\begin{aligned} \mathit{VarIntro}(r) \triangleq\ & \{\, (h,\ \mathit{side}) \mid X \not\subseteq \mathit{Lhs}(r),\ X \subseteq \mathit{Side}(r,\ h,\ \mathit{side}), \\ & \quad \forall j\ (j < h)\ \forall \mathit{side}' : X \not\subseteq \mathit{Side}(r,\ j,\ \mathit{side}')\, \} \end{aligned}$$

For convenience, we also define a function *NonVarIntro* (Definition 2.5) that computes tuples $(h,\ side)$ for all non-variable-introducing condition sides.

Definition 2.5 (*NonVarIntro*)

$$\begin{aligned} \mathit{NonVarIntro}(r) \triangleq\ & \{\, (h,\ \mathit{side}) \mid 1 \leq h \leq |r|,\ \mathit{side} \in \{\, \mathit{left},\ \mathit{right}\,\}, \\ & \quad (h,\ \mathit{side}) \notin \mathit{VarIntro}(r)\, \} \end{aligned}$$

2.2.5 Rewriting histories for CTRSs

In phase 2 of the 3-phase process sketched in Section 2.2.4 above, each normalization of an instantiated condition side is a situation similar to the normalization of the original term, involving the same 3-phase process. Thus, we can model the rewriting of a term as a tree of reduction sequences. The *initial reduction sequence* named $\mathcal{S}^{init}$ starts with the initial term and contains sequence elements $\mathcal{S}^{init}_i$ that describe successive transformations of the initial term. In addition, $\mathcal{H}$ now contains a sequence for every condition side that is normalized in the course of the rewriting process. Two sequences appear for non-variable-introducing conditions, but for variable-introducing conditions only one sequence occurs in $\mathcal{H}$ (for the non-variable-introducing side).

Formally, we define the history as a flat representation of this tree of reduction sequences. A history now consists of two parts:

- A set of uniquely named reduction sequences. Besides the initial sequence, $\mathcal{S}^{init}$, there is a sequence $\mathcal{S}^k$ (with k an integer) for every condition side that is normalized in the course of the rewriting process.
  As before, a sequence consists of one or more sequence elements, and each sequence

element is a 5-tuple $(n,\ t,\ r,\ p,\ \sigma)$, denoting the name, term, rule, path, and substitution involved. As in the unconditional case, access functions are provided to obtain the components of s. A name of a sequence element is composed of a sequence name and a number, permitting us to find out to what sequence an element belongs.

- A mechanism indicating the connections between the various reduction sequences. This mechanism takes the form of a relation that determines a sequence name given a name of a sequence element s, a condition number j, and a condition side $side$, for all $(j,\ side) \in NonVarIntro(s)$. E.g., a tuple $(n(s),\ j,\ side,\ sn)$ indicates that a sequence named *sn* occurred as a result of the normalization of $Side(s,\ j,\ side)$.

Two functions *First* and *Last* are defined, both taking four arguments: the history $\mathcal{H}$, a sequence element s, a condition number j, and a condition side $side$. $First(\mathcal{H},\ s,\ j,\ side)$ retrieves the name of s, determines the name of the sequence associated with side $side$ of condition j of $r(s)$, looks up this sequence in $\mathcal{H}$, and returns the first element of this sequence. $Last(\mathcal{H},\ s,\ j,\ side)$ is similar: it determines the last element of the sequence associated with side $side$ of condition j of $r(s)$.

Furthermore, $\mathcal{H}^-$ now denotes the history $\mathcal{H}$ from which all last elements of sequences are excluded, $Succ(\mathcal{H},\ s)$ now denotes the successor of s in the same sequence, for s in $\mathcal{H}^-$, and $Start(\mathcal{H})$ determines the first element of the initial sequence in $\mathcal{H}$. Finally, we introduce the shorthands $Side(s,\ j,\ side)$, $VarIntro(s)$, and $NonVarIntro(s)$ for $Side(r(s),\ j,\ side)$, $VarIntro(r(s))$, and $NonVarIntro(r(s))$, respectively.

2.2.6 The origin function for CTRSs

2.2.6.1 Basic origin relations

The basic origin relation *LR* (Definition 2.2) defines relations between consecutive elements s and $Succ(\mathcal{H},\ s)$ of the same sequence. The basic origin relations *LC*, *CR*, and *CC* define relations between elements of different sequences. Each of these relations reflects the following principle: common subterms are only related when a common variable or a common subterm appears at corresponding places in the left-hand side, right-hand side and condition side of the rewrite rule involved.

Definition 2.6, *LC* (Left-hand side to Condition side), defines relations that result from common variables and common subterms of the left-hand side and a condition side of a rule. An *LC*-relation connects a sequence element s to the first element s' of a sequence for the normalization of a condition side of $r(s)$. The relation consists of triples $(q,\ q',\ s')$ indicating a relation between the subterm at path q in the redex and the subterm at path q' in $t(s')$.

We do not establish *LC*-relations for variable-introducing condition sides, because such relations are always *redundant*. To understand this, consider the fact that we disallow instantiated variables in variable-introducing condition sides. Thus, *LC* relations would always correspond to a common *subterm* t of the left-hand side and a variable-introducing condition side. Then, only if t also occurs in a subsequent condition side, or in the right-hand side of the rule can the relation be relevant for the remainder of the rewriting history. But if this is the case, this other occurrence of t will be involved in an *LC*-relation anyway.

Definition 2.6 (*LC*) *For s in $\mathcal{H}^-$:*

$$\begin{aligned} \mathit{LC}(\mathcal{H},\ s) \triangleq \{\ & (q,\ q',\ s') \mid (j,\ \mathit{side}) \in \mathit{NonVarIntro}(s), s' = \mathit{First}(\mathcal{H},\ s,\ j,\ \mathit{side}), \\ & (q,\ q') \in \mathit{Com}(\sigma(s), \mathit{Lhs}(s),\ \mathit{Side}(s,\ j,\ \mathit{side}))\ \} \end{aligned}$$

In Definition 2.7 and Definition 2.8 below, the final two basic origin relations, *CR* (Condition side to Right-hand side) and *CC* (Condition side to Condition side) are presented. These relations are concerned with common variables and common subterms in variable-introducing condition sides. In addition to a variable-introducing condition side, these relations involve the right-hand side, and a non-variable-introducing condition side, respectively. The following technical issues arise here:

- There are no *CR* and *CC* relations for non-variable-introducing conditions, because both condition sides are normalized in this case, and no obvious correspondence with the syntactical form of the rewrite rule remains.
- As mentioned earlier, no reduction sequence appears in $\mathcal{H}$ for a variable-introducing condition side. To deal with this issue, the variable-introducing side $Side(s,\ j,\ side)$ is used to indicate relations with the term $t(Last(\mathcal{H},\ s,\ j,\ \overline{side}))$ it is matched against.

CR-relations are triples $(q,\ q',\ s')$ indicating that the subterm at path q in $t(s')$ is related to the subterm at path q' in the contractum; *CC*-relations are quadruples $(q,\ q',\ s',\ s'')$ that express a relation between the subterm at path q in $t(s')$ and the subterm at path q' in $t(s'')$.

Definition 2.7 (*CR*) *For s in $\mathcal{H}^-$:*

$$\begin{aligned} \mathit{CR}(\mathcal{H},\ s) \triangleq \{ & (q,\ q',\ s') \mid (j,\ \mathit{side}) \in \mathit{VarIntro}(s),\ s' = \mathit{Last}(\mathcal{H},\ s,\ j,\ \overline{\mathit{side}}), \\ & (q,\ q') \in \mathit{Com}(\sigma(s),\ \mathit{Side}(s,\ j,\ \mathit{side}),\ \mathit{Rhs}(s))\ \} \end{aligned}$$

Definition 2.8 (*CC*) *For s in $\mathcal{H}^-$:*

$$\begin{aligned} \mathit{CC}(\mathcal{H},\ s) \triangleq \{ & (q,\ q',\ s',\ s'') \mid (j,\ \mathit{side}) \in \mathit{VarIntro}(s),\ (h,\ \mathit{side}') \in \mathit{NonVarIntro}(s), \\ & j < h,\ s' = \mathit{Last}(\mathcal{H},\ s,\ j,\ \overline{\mathit{side}}),\ s'' = \mathit{First}(\mathcal{H},\ s,\ h,\ \mathit{side}'), \\ & (q,\ q') \in \mathit{Com}(\sigma(s),\ \mathit{Side}(s,\ j,\ \mathit{side}),\ \mathit{Side}(s,\ h,\ \mathit{side}'))\ \} \end{aligned}$$

2.2.6.2 Definition of *CORG*

The origin function *CORG* for CTRSs (Definition 2.9) is basically an extension of *ORG*. Using the basic origin relations *LC*, *CR*, and *CC*, relations between elements of different reduction sequences are established in **(c1)**, **(c2)**, and **(c3)**. Again, the origin function computes a set of paths in the initial term according to the transitive and reflexive closure of *relate*. For any sequence element s in $\mathcal{H}$, and any path p in $t(s)$, *CORG* computes a set of paths to related subterms in $t(Start(\mathcal{H}))$.

Definition 2.9 (*CORG*) *For s in $\mathcal{H}^-$ and s' in $\mathcal{H}$:*

$$\begin{array}{lll}
(\mathbf{u1}) & \forall (q,\, q') \in \mathit{LR}(s): & \mathit{relate}(\mathcal{H},\, s,\, p(s)\cdot q,\, \mathit{Succ}(\mathcal{H},\, s),\, p(s)\cdot q') \\
(\mathbf{u2}) & \forall p : (p \preceq p(s)) \vee (p \mid p(s)): & \mathit{relate}(\mathcal{H},\, s,\, p,\, \mathit{Succ}(\mathcal{H},\, s),\, p) \\
(\mathbf{c1}) & \forall (q,\, q',\, s') \in \mathit{LC}(\mathcal{H},\, s): & \mathit{relate}(\mathcal{H},\, s,\, p(s)\cdot q,\, s',\, q') \\
(\mathbf{c2}) & \forall (q,\, q',\, s') \in \mathit{CR}(\mathcal{H},\, s): & \mathit{relate}(\mathcal{H},\, s',\, q,\, \mathit{Succ}(\mathcal{H},\, s),\, p(s)\cdot q') \\
(\mathbf{c3}) & \forall (q,\, q',\, s',\, s'') \in \mathit{CC}(\mathcal{H},\, s): & \mathit{relate}(\mathcal{H},\, s',\, q,\, s'',\, q')
\end{array}$$

$$\mathit{CORG}(\mathcal{H},\, s,\, p) \triangleq \begin{cases} \{\, p \,\} & \text{when } s' = \mathit{Start}(\mathcal{H}) \\[2ex] \{\, p'' \mid \; p'' \in \mathit{CORG}(\mathcal{H},\, s',\, p'), & \text{when } s' \neq \mathit{Start}(\mathcal{H}) \\ \qquad \mathit{relate}(\mathcal{H},\, s',\, p',\, s,\, p) \,\} & \end{cases}$$

2.2.6.3 Example

We extend the example of section 2.2.3.3 with the following conditional rewrite rules for a function `rev` to reverse lists.

[R1]	`rev(empty-list)`	$\rightarrow$	`empty-list`
[R2]	`rev(cons(`E`, `L_1`))`	$\rightarrow$	`append(`E`, `L_2`)` **when** L_2 `= rev(`L_1`)`

In rule **[R2]**, a variable L_2 is introduced in the left-hand side of the condition. Actually, the use of a new variable is not necessary in this case: we may alternatively write `append(`E`, rev(`L_1`))` for the right-hand side of **[R2]**. The new variable is used solely for the sake of illustration. Figure 2.9 shows the rewriting history $\mathcal{H}_{rev}$ for the term `rev(cons(b, empty-list))`. Note that besides the initial sequence, $\mathcal{S}^{init}$, only one sequence, $\mathcal{S}^1$, appears for the normalization of the condition of **[R2]**, because it is variable-introducing.

For sequence element $\mathcal{S}_1^{init}$ we have $p(\mathcal{S}_1^{init}) = ()$, $\sigma(\mathcal{S}_1^{init}) = \{\ E \mapsto$ `b`, $L_1 \mapsto$ `empty-list`, $L_2 \mapsto$ `empty-list` $\}$. It follows that $O(E^{\sigma(\mathcal{S}_1^{init})}) = O({L_1}^{\sigma(\mathcal{S}_1^{init})}) = O({L_2}^{\sigma(\mathcal{S}_1^{init})})$ = { () }. Moreover, $\mathit{VarIntro}(\mathcal{S}_1^{init}) = \{\ (1,\ \mathit{left})\ \}$. Consequently, we obtain:

$$\begin{array}{ll}
LR(\mathcal{S}_1^{init}) = \{\ ((1\ 1),\ (1))\ \}, & LC(\mathcal{H}_{rev},\ \mathcal{S}_1^{init}) = \{\ ((1\ 2),\ (1),\ \mathcal{S}_1^1)\ \} \\
CR(\mathcal{H}_{rev},\ \mathcal{S}_1^{init}) = \{\ ((),\ (1),\ \mathcal{S}_2^1)\ \}, & CC(\mathcal{H}_{rev},\ \mathcal{S}_1^{init}) = \emptyset
\end{array}$$

As a result, the following relationships are generated for $\mathcal{S}_1^{init}$:

$$\begin{array}{ll}
relate(\mathcal{H}_{rev},\ \mathcal{S}_1^{init},\ (),\ \mathcal{S}_2^{init},\ ()) & relate(\mathcal{H}_{rev},\ \mathcal{S}_1^{init},\ (1\ 1),\ \mathcal{S}_2^{init},\ (1)) \\
relate(\mathcal{H}_{rev},\ \mathcal{S}_1^{init},\ (1\ 2),\ \mathcal{S}_1^1,\ (1)) & relate(\mathcal{H}_{rev},\ \mathcal{S}_2^1,\ (),\ \mathcal{S}_2^{init},\ (2))
\end{array}$$

In a similar way, the following *relate* relationships are computed for $\mathcal{S}_2^{init}$ and $\mathcal{S}_1^1$:

$$\begin{array}{ll}
relate(\mathcal{H}_{rev},\ \mathcal{S}_2^{init},\ (),\ \mathcal{S}_3^{init},\ ()) & relate(\mathcal{H}_{rev},\ \mathcal{S}_2^{init},\ (1),\ \mathcal{S}_3^{init},\ (1)) \\
relate(\mathcal{H}_{rev},\ \mathcal{S}_2^{init},\ (2),\ \mathcal{S}_3^{init},\ (2)) & relate(\mathcal{H}_{rev},\ \mathcal{S}_1^1,\ (),\ \mathcal{S}_2^1,\ ()) \\
relate(\mathcal{H}_{rev},\ \mathcal{S}_1^1,\ (1),\ \mathcal{S}_2^1,\ ()) &
\end{array}$$

Finally, we compute the subterms related to `empty-list` at path (2) in $t(\mathcal{S}_3^{init})$:

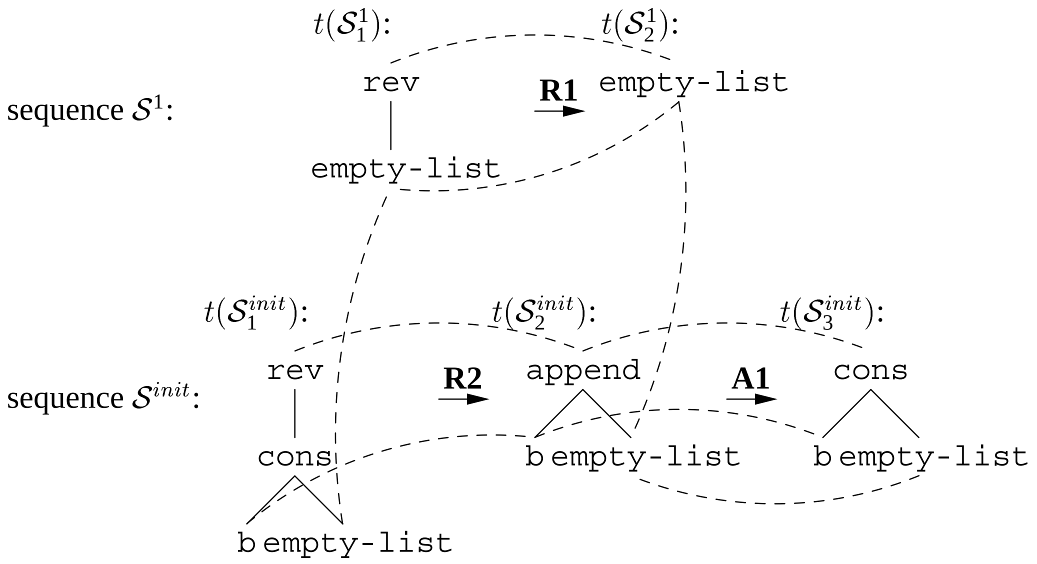

$t(\mathcal{S}_1^1)$:
$t(\mathcal{S}_2^1)$:
sequence $\mathcal{S}^1$:
rev
R1
empty-list
empty-list
$t(\mathcal{S}_1^{init})$:
$t(\mathcal{S}_2^{init})$:
$t(\mathcal{S}_3^{init})$:
sequence $\mathcal{S}^{init}$:
rev
R2
append
A1
cons
cons
b empty-list
b empty-list
b empty-list

Figure 2.9: History $\mathcal{H}_{rev}$ for `rev(cons(b, empty-list))`. Dashed lines indicate origin relations.

$$
\begin{aligned}
&CORG(\mathcal{H}_{rev},\ \mathcal{S}_3^{init},\ (2)) = \\
&\quad = \{\, p'' \mid p'' \in CORG(\mathcal{H}_{rev},\ s',\ p'),\ relate(\mathcal{H}_{rev},\ s',\ p',\ \mathcal{S}_3^{init},\ (2))\,\} \\
&\quad = \{\, p'' \mid p'' \in CORG(\mathcal{H}_{rev},\ \mathcal{S}_2^{init},\ (2))\,\} \\
&\quad = CORG(\mathcal{H}_{rev},\ \mathcal{S}_2^{init},\ (2)) \\
&\quad = \cdots = \{\ (1\ 2)\ \}
\end{aligned}
$$

Consequently, the constant `empty-list` in $t(\mathcal{S}_3^{init})$ is related to the constant `empty-list` in $t(\mathcal{S}_1^{init})$.

2.3 Properties

The origins defined by *CORG* have the following property: if the origin of some intermediate term t_{mid} contains a path to initial subterm t_{org}, then t_{org} can be rewritten to t_{mid} in zero or more reduction steps. This property gives a good intuition of the origin relations that are established in applications such as error handling or debugging.

To see why this property holds, we first consider one reduction step:

Lemma 2.10 *Let $\mathcal{H}$ be a rewriting history, s, s' arbitrary sequence elements in $\mathcal{H}$, and p, p' paths. For any relate$(\mathcal{H}, s, p, s', p')$ we have $t(s) \equiv t(s')$ or $t(s) \to t(s')$.*

Informally stated, directly *relate*d terms are either syntactically equal or one can be reduced to the other in exactly one step. This holds because the context, common variables, and common subterms relations all relate identical terms. Only the redex-contractum relation links non-identical terms, but these can be rewritten in one step. Since the origin relation *CORG* is defined as the transitive and reflexive closure of *relate*, we now have the desired property:

Theorem 2.11 *Let $\mathcal{H}$ be a history. For every term $t(s)$ occurring in some sequence element s in history $\mathcal{H}$, and for every path $p \in O(t(s))$, we have:*

$$q \in \mathit{CORG}(\mathcal{H}, s, p) \Rightarrow t(\mathit{Start}(\mathcal{H}))/q \to^* t(s)/p$$

One may be interested in the *number of paths* in an origin. To this end, we introduce:

Definition 2.12 *Let o be an origin, and let $|o|$ denote the number of paths in o. Then: o is* empty *iff $|o| = 0$,* non-empty *iff $|o| \geq 1$,* precise *iff $|o| \leq 1$, and* unitary *iff $|o| = 1$.*

For some applications, unitary origins are desirable. In animators for sequential program execution, one wants origins that refer to exactly one statement. On the other hand, when error-positioning is the application, it can be desirable to have non-unitary origins, as for instance in errors dealing with multiple declarations of the same variable (see, e.g., the label declaration in Figure 2.2).

The theorems below indicate how non-empty, precise and unitary origins can be detected through static analysis of the CTRS. In the sequel r denotes an arbitrary rule, j is a number of some condition in r, and *side* $\in \{\mathit{left}, \mathit{right}\}$ denotes an arbitrary side. In the sequel, a term that (possibly) contains variables will be referred to as an *open* term.

Theorem 2.13 (Non-empty origins) *Terms with top symbol f have non-empty origins if for all open terms u with top function symbol f:*

$$\begin{array}{ll} (1) & u \subseteq \mathit{Rhs}(r) \Rightarrow u \subseteq \mathit{Lhs}(r) \\ (2) & u \subseteq \mathit{Side}(r,\ j,\ \mathit{side}) \Rightarrow u \subseteq \mathit{Lhs}(r) \end{array}$$

This can be proven by induction over all *relate* clauses, after introducing an ordering on all sequence elements. Informally, all terms with top symbol f will have non-empty origins if no f is introduced that is not *relate*d to a “previous” f. Note that relations according to variables have no effect on origins being (non-)empty.

In order to characterize sufficient conditions for precise and unitary origins, we first need some definitions:

Definition 2.14 *Let r be a conditional rewrite rule and u an open term. Then r is an u-collapse rule if $\mathit{Rhs}(r) \equiv u$, and $u \subseteq \mathit{Lhs}(r)$.*

Definition 2.15 *For open terms t and u, t is linear in u if u occurs at most once in t.*

Definition 2.16 *The predicate LinearIntro(u, r) holds if u has at most one occurrence in either the left-hand side or any variable-introducing condition side. Formally, LinearIntro(u, r) $\Leftrightarrow$ there is (1) at most one $t \in \{$Lhs$(r)\} \cup \{$Side$(r, h, side) \mid (h, side) \in$ VarIntro$(r)\}$ such that $u \subseteq t$, and (2) this t, if it exists, is linear in u.*

Theorem 2.17 (Precise origins) *Terms with top symbol f have precise origins if the following holds for all open terms u having either f as top symbol or solely consisting of a variable:*

(1) The CTRS does not contain u-collapse rules
(2) $u \subseteq$ Rhs$(r) \Rightarrow$ LinearIntro(u, r)
(3) $u \subseteq$ Side$(r,\ j,\ side) \Rightarrow$ LinearIntro$(u,\ r)$

Again, this theorem can be proven by induction over all *relate*s. The crux is that no term with top function symbol f is introduced in a way that it is *relate*d to more than one "previous" term.

Theorem 2.18 (Unitary origins) *Since "non-empty" and "precise" implies "unitary", combining the premises of Theorems 2.13 and 2.17 yields sufficient conditions for unitary origins*

For many-sorted CTRSs, some special theorems hold. We assume CTRSs to be sort-preserving, i.e., the redex and the contractum belong to the same sort. Hence, *CORG* is sort-preserving. Thus, we have the following theorem (which in the implementation allows for an optimization—Section 2.4):

Theorem 2.19 *relate can be partitioned into subrelations for each sort.*

One may be interested whether all terms *of some particular sort* `S` have non-empty, precise, or unitary origins. This happens under circumstances very similar to those formulated for the single-sorted case (Theorems 2.13 to 2.18). For precise and unitary origins, however, it is not sufficient to consider only terms of sort `S`; one also needs to consider sorts `T` that can have subterms of sort `S` (since duplication of `T`-terms may imply duplication of `S`-terms). Hence, we define:

Definition 2.20 *For two sorts `S` and `T`, we write `S` $\sqsubseteq$ `T` if terms of sort `T` can contain subterms of sort `S`.*

Using this, we can formulate when terms of sort `S` have precise or unitary origins. This is similar to the single-sorted case (see Theorem 2.17), but in (1) u must be of sort `S`, and (2) and (3) must hold for all u of sort `T` such that `S` $\sqsubseteq$ `T`. Unitary origins of sort `S` are obtained by combining the premises for the non-empty origins and precise origins.

We refer to [46] for more elaborate discussions of the above results.

2.4 Implementation aspects

An efficient implementation of origin tracking in the ASF+SDF system has been completed. In this section, we briefly address the principal aspects of implementing origin tracking.

2.4.1 The basic algorithm

In our implementation, each symbol is *annotated* with its origin, during rewriting. Two issues had to be resolved:

- annotation of the initial term
- propagation of origins during rewriting

The first issue is a trivial matter because—by definition—the origin of the symbol at path p is $\{ p \}$. The second issue is addressed by copying origins from the redex to the contractum according to the basic origin relation *LR*. In a similar way, propagations occur for the basic origin relations *LC*, *CC*, and *CR*. Observe that no propagations are necessary for the origins in the context of the redex, as the origins of these symbols remain unaltered.

2.4.2 Optimizations

Several optimizations of the basic algorithm have been implemented:

- All positional information (i.e., the positions of common variables and common subterms) is computed in advance, and stored as annotations of rewrite rules.
- The rewriting engine of the ASF+SDF system explicitly constructs a list of variable bindings. Origin propagations that are the result of common variables can be implemented as propagations to these bindings. When a right-hand side or condition side is instantiated, all common variable propagations are handled as a result of the instantiation. The advantage of this approach is that the number of propagations decreases, because we always propagate to only one subterm for each variable.
- Origins are implemented as a set of pointers to function symbols of the initial term. The advantages are twofold: less space is needed to represent origins, and set union becomes a much cheaper operation.

2.4.3 Associative lists

In order to implement origin tracking in the ASF+SDF system, provisions had to be made for *associative lists* [69, 140]. Associative lists can be regarded as functions with a variable arity. Allowing list functions in CTRSs introduces two minor complications:

- A variable that matches a sublist causes relations between *arrays of* adjacent subterms. In the implementation, we distinguish between ordinary variables and list variables, and perform propagations accordingly.
- Argument positions below list functions depend on the actual bindings. Therefore, when computing the positions of common variables and common subterms, positions below lists are marked as *relative*. The corresponding *absolute* positions are determined during rewriting.

Consider the following example, where `l` is a list function, and *X** is a list variable that matches sublists of any length:

[L1] `f(l(`$X*$`, a))` $\rightarrow$ `g(l(`$X*$`, a))`

When we rewrite the redex `f(l(b, c, a))` according to **[L1]**, the contractum is `g(l(b, c, a))`. Variable $X*$ gives rise to both a relation between the constants `b` in the redex and the contractum, and a relation between the constants `c` in the redex and the contractum. Moreover, constant `a` appears at path (1 2) in the left-hand side of **[L1]**, but at path (1 3) in the redex.

2.4.4 Sharing of subterms

For reasons of efficiency, implementations of CTRSs allow sharing of subtrees, thus giving rise to DAGs (Directed Acyclic Graphs) instead of trees. The initial term is represented as a tree, and sharing is introduced by instantiating nonlinear right-hand sides and condition sides. For every variable, the list of bindings contains a *pointer* to one of the subterms it was matched against. Instantiating a right-hand side or condition side is done by copying these pointers (instead of copying the terms in the list of bindings). Sharing has the following repercussions for origin tracking:

- No propagations are needed for variables that occur exactly once in the left-hand side (and for new variables that occur exactly once in the introducing condition). This results in a radical reduction of the number of propagations.
- Variables that occur nonlinearly in the left-hand side of a rule (and new variables that occur nonlinearly in the introducing condition) present a problem. When sharing is allowed in this case, inconsistent origins with respect to the definition may arise because different origins may be associated with a shared function symbol when it is “accessed” via different paths. A solution to this problem consists of using a pointer to a *copy* of the term matched against such a variable in the list of bindings. This corresponds to disallowing sharing in a limited number of situations.

2.4.5 Restricting origin tracking to selected sorts

Often, one is only interested in the origins of subterms of a particular sort. A straightforward result of Property 2.19 is the following: to compute the origins of subterms of sort `S`, only propagations for common subterms of sort `S`, and for common variables of sorts `T` such that `S` $\sqsubseteq$ `T` are necessary.

2.4.6 Time and space overhead of origin tracking

Origins are represented by sets of pointers to symbols of the initial term, and associated with every symbol is exactly one such set. The size of these sets is bounded by the number of function symbols in the initial term because, in the worst case, a set contains a pointer to every symbol in the initial term. Thus, the space overhead of origin tracking is linear in the size of the initial term. In practice, only small sets arise, resulting in little space overhead. The use of efficient set representations would reduce this overhead even further.

We have measured the time overhead caused by origin tracking. In all measurements, the run-time overhead lies between 10% and 100%, excluding the costs of pre-computing positional information.

2.5 Related work

In TRS theory, the notion of *descendant* [95] (or *residual* [118, 78]) is used to study properties such as confluence or termination, and to find optimal orders for contracting redexes (see [112] for some recent results). For a reduction $t \rightarrow t'$ contracting a redex $s \subseteq t$, a different redex $s' \subseteq t$ may reappear in the resulting term t'. The occurrences of this s' in t' are called the descendants of s'.

Descendants are similar to origins, but more restricted. Only relations according to contexts and common variables are established (Bergstra and Klop [24] also use *quasi-descendants* linking the redex and contractum as well). Moreover, descendants are defined for a smaller class of TRSs; only orthogonal (left-linear and non-overlapping) TRSs without conditional equations are allowed.

Bertot [27, 26] studies residuals in TRSs and λ-calculus, and introduces *marking functions* to represent the residual relation. He provides a formal language to describe computations on these marking functions, and shows how the marking functions can be integrated in formalisms for the specification of programming language semantics (*viz.* term rewriting systems and collections of inference rules). Bertot works in the realm of left-linear, unconditional TRSs and only considers precise origins.

The ideas of Bertot concerning origins in inference rules have been used in the framework of TYPOL [41], a formalism to specify programming languages, based on natural semantics [85]. For compositional definitions of evaluators or type-checkers (in which the meaning of a language construct is expressed in terms of its substructures), the implementation of TYPOL keeps track of the construct currently processed (the *subject*). A pointer to the subject is available in tools derived from the specification, particularly debuggers or error handlers. In addition to automatic subject tracking, TYPOL has been equipped with special language constructs to manipulate origins explicitly. This contrasts with our approach, where origin tracking is invisible at the specification level.

Berry [25] aims at deriving animators from relational rules (similar to operational semantics). He defines a *focus* that is either equal to the *subject* (as in TYPOL) or to the *result* of the evaluation of some subexpression. The theory he develops uses the concept of an *inference tree*, a notion similar to our rewriting histories.

In the context of the PSG system [10], a generator for language-specific debuggers was described. Debuggers are generated from a specification of the denotational semantics of a language and some additional debugging functions. Bahlke et al. insist that programs are explicitly annotated with their position in the initial syntactic structure before running their semantic tool.

2.6 Concluding remarks

2.6.1 Achievements

Summarizing the results described in this chapter, we have:

- A definition of origins that does not depend on a particular rewrite strategy, nor on the confluence or strong-normalization of the underlying CTRS. It establishes only relations that can be derived from the syntactic structure of the rewrite rules.
- The property that whenever a term t_{mid} has a subterm t_{org} in the initial term as origin, this term t_{org} can be rewritten to t_{mid}.
- Sufficient criteria that a specification should satisfy to guarantee that an origin consisting of at least one, or exactly one path is associated with each subterm of a given sort.
- An efficient implementation method for origin tracking.
- A notion of sort-dependent "filtering" of origins, when only the origins of terms of certain sorts are needed.
- A prospect of applying origin tracking to the generation of interactive language-based environments from formal language definitions. In particular, generic techniques for debugging and error reporting have been discussed.

2.6.2 Limitations

The current method for origin tracking has limitations, most of which are related to the introduction of new function symbols. Some typical problem cases are:

- In the context of translating arithmetic expressions to a sequence of stack machine instructions, one may encounter an equation of the form

  `trans(plus(`E_1`,`E_2`))` $\rightarrow$ `seq(trans(`E_1`),seq(trans(`E_2`),add))`

  The `plus` of the expression language is translated to the `add` stack-instruction. It seems intuitive to relate both `seq` function symbols to the `plus` symbol at the left-hand side. However, the current origin mechanism do not establish this relation.
- In specifications of evaluators it frequently occurs that the evaluation of one construct is defined by reducing it to another construct, as in

  `eval(repeat(`*S*`,`*Exp*`),`*Env*`)` $\rightarrow$
  `eval(seq(`*S*`,while(not(`*Exp*`),`*S*`)),`*Env*`)`

  where the evaluation of the `repeat`-statement is defined in terms of the `while`-statement. In this example, `seq` is a constructor for statement sequences. Here again, the `while`-statement on the right-hand side does not obtain an origin although the `repeat`-statement on the left-hand side would be a good candidate for this.

These examples have the flavor of translating terms from one representation to another and they illustrate that *more* origin relations have to be established in these cases. In [45, 44], the compositional structure of *primitive recursive schemes* [114, 115] (a well-behaved subclass of algebraic specifications) is exploited to establish additional origin relations. An alternative approach would be to allow user-provided annotations in specifications that indicate more origin relations.

2.6.3 Applications

The main applications of origin tracking have already been sketched. These applications can be summarized as follows:

Animation of program execution: Origin tracking has been used successfully for the construction of tools that visualize program execution [49, 134].

Source-level debugging: Chapter 6 describes how origin tracking can be used to generate powerful source-level debugging tools from algebraic specification of interpreters.

Error reporting: Origin tracking has also been used in conjunction with algebraic specifications of type-checkers [49, 48, 44] in order to obtain positional information of type-checkers.

Experience has shown that the origin function that was described in this chapter is insufficiently powerful to be applicable to any type-checking specification. A solution to this problem, in the form of a specialized origin function for *primitive recursive schemes* is proposed in [45, 44].

Chapter 3

A Survey of Program Slicing Techniques

Summary

The subsequent Chapters 4, 5, and 6 revolve, in one way or another, around the concept of *program slicing*. To put this work in perspective, this chapter presents a comprehensive survey of program slicing and its applications.

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a location in the program in combination with a subset of the program's variables. The task of computing program slices is called *program slicing*. The original definition of a program slice was presented by Weiser in 1979. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is that between a *static* and a *dynamic* slice. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific test case.

Procedures, unstructured control flow, composite data types and pointers, and concurrency each require a specific solution. Static and dynamic slicing methods for each of these features are compared and classified in terms of their accuracy and efficiency. Moreover, the possibilities for combining solutions for different features are investigated. Recent work on the use of compiler-optimization and symbolic execution techniques for obtaining more accurate slices is discussed. The chapter is concluded with an overview of the applications of program slicing, which include debugging, program integration, dataflow testing, and software maintenance.

^{1} **slice** \\'slīs\\ n **1 :** a thin flat piece cut from something **2 :** a wedge-shaped blade (as for serving fish) **3 :** a flight of a ball (as in golf) that curves in the direction of the dominant hand of the player propelling it
^{2} **slice** *vb* **sliced**; **slic-ing 1 :** to cut a slice from; *also* to cut into slices **2 :** to hit (a ball) so that a slice results

The Merriam-Webster Dictionary

3.1 Overview

We present a survey of algorithms for program slicing that can be found in the present literature. A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a pair $\langle$program point, set of variables$\rangle$. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with respect to criterion* C. The task of computing program slices is called *program slicing*.

The original concept of a program slice was introduced by Weiser [144, 145, 147]. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program, and advocates the integration of program slicers in debugging environments. Various slightly different notions of program slices have since been proposed, as well as a number of methods to compute slices. The main reason for this diversity is the fact that different applications require different properties of slices. Weiser defined a program slice S as a *reduced, executable program* obtained from a program P by removing statements, such that S replicates part of the behavior of P. Another common definition of a slice is a *subset* of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute an executable program. An important distinction is that between a *static* and a *dynamic* slice. The former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case. Below, in Sections 3.1.1 and 3.1.2, these notions are introduced in some detail.

Features of programming languages such as procedures, unstructured control flow, composite data types and pointers, and concurrency each require a specific solution. Static and dynamic slicing methods for each of these features are classified and compared in terms of accuracy and efficiency. In addition, possibilities for integrating solutions for different language features are investigated. Throughout this survey, slicing algorithms are compared by applying them to similar examples.

3.1.1 Static slicing

Figure 3.1 **(a)** shows an example program that asks for a number `n`, and computes the sum and the product of the first `n` positive numbers. Figure 3.1 **(b)** shows a slice of this program with respect to criterion (10, `product`). As can be seen in the figure, all computations not relevant to the (final value of) variable `product` have been "sliced away".

In Weiser's approach, slices are computed by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependences. Only statically available information is used for computing slices; hence, this type of slice is referred to as a *static* slice. An alternative method for computing static slices was suggested by Ottenstein and Ottenstein [120], who restate the problem of static slicing in terms of a reachability problem in a *program dependence graph* (PDG) [101, 53]. A PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependences. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all PDG vertices from which the vertex under consideration can be reached. Various program slicing approaches discussed below utilize modified and extended versions of PDGs as their underlying program representation. Yet another approach

```
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i <= n do
       begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

Figure 3.1: **(a)** An example program. **(b)** A slice of the program w.r.t. criterion (10, `product`).

was proposed by Bergeretti and Carré [22], who define slices in terms of information-flow relations, which are derived from a program in a syntax-directed fashion.

The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward* traversal of the program's control flow graph (CFG) or PDG, starting at the slicing criterion. Therefore, these slices are referred to as *backward* (static) slices. Bergeretti and Carré [22] were the first to define the notion of a *forward* static slice, although Reps and Bricker [127] were the first to use this terminology. Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, a statement being "dependent" on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not. Backward and forward slices[1] are computed in a similar way; the latter requires tracing dependences in the forward direction.

3.1.2 Dynamic slicing

Although the exact terminology "dynamic program slicing" was first introduced by Korel and Laski [99], dynamic slicing may very well be regarded as a non-interactive variation of Balzer's notion of flowback analysis [15]. In flowback analysis, one is interested in how information flows through a program to obtain a particular value: the user interactively traverses a graph that represents the data and control dependences between statements in the program. For example, if the value computed at statement s depends on the values computed at statement t, the user may trace back from the vertex corresponding to s to the vertex for t. Recently, Choi et al. [116, 38] have made an efficient implementation of flowback analysis for parallel programs.

[1]Unless stated otherwise, "slice" will denote "backward slice".

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)
```

(a)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
           ;
  i := i + 1
end;
write(x)
```

(b)

Figure 3.2: **(a)** Another example program. **(b)** Dynamic slice w.r.t. criterion (`n = 2`, 8^1, `x`).

In the case of dynamic program slicing, only the dependences that occur in a *specific* execution of the program are taken into account. A *dynamic slicing criterion* specifies the input, and distinguishes between different occurrences of a statement in the execution history; typically, it consists of triple $\langle$input, occurrence of a statement, variable$\rangle$. An alternate view of the difference between static and dynamic slicing is that dynamic slicing assumes *fixed* input for a program, whereas static slicing does not make assumptions regarding the input. A number of hybrid approaches, where a combination of static and dynamic information is used to compute slices, can be found in the literature. Choi et al. [38], Duesterwald et al. [51], and Kamkar [87] use static information in order to decrease the amount of computations that have to be performed at run-time. Venkatesh [137], Ning et al. [117], and Field, Ramalingam, and Tip (see Chapters 4 and 5) consider situations where only a *subset* of the inputs to program are constrained.

Figure 3.2 shows an example program, and its dynamic slice w.r.t. the criterion (`n = 2`, 8^1, `x`), where 8^1 denotes the first occurrence of statement 8 in the execution history of the program. Note that for input `n = 2`, the loop is executed twice, and that the assignments `x := 17` and `x := 18` are each executed once. In this example, the **else** branch of the **if** statement may be omitted from the dynamic slice since the assignment of 18 to variable `x` in the first iteration of the loop is "killed" by the assignment of 17 to `x` in the second iteration[2]. By contrast, the *static* slice of the program in Figure 3.2 **(a)** w.r.t. criterion (8, `x`) consists of the entire program.

3.1.3 Applications of slicing

The main application that Weiser had in mind for slicing was debugging [144, 145, 147]: if a program computes an erroneous value for some variable x at some program point, the

[2]In fact, one might argue that the **while** construct may be replaced by the **if** statement in its body. This type of slice will be discussed in Section 3.6.

bug is likely to be found in the slice with respect to x at that point. The use of slicing for debugging was further explored by Lyle and Weiser [109], Choi et al. [38], Agrawal et al. [5], Fritzson et al. [59], and Pan and Spafford [121, 122].

A number of other applications has since been proposed: parallelization [146], program differencing and integration [70, 74], software maintenance [61], testing [51, 88, 65, 20], reverse engineering [21, 82, 81], and compiler tuning [106]. Section 3.5 contains an overview of how slicing is used in each of these application areas.

3.1.4 Related work

There are a number of earlier frameworks for comparing slicing methods, as well as some earlier surveys of slicing methods.

Venkatesh [137] presents formal definitions of several types of slices in terms of denotational semantics. He distinguishes three independent dimensions according to which slices can be categorized: static vs. dynamic, backward vs. forward, and closure vs. executable. Some of the slicing methods in the literature are classified according to these criteria [147, 120, 74, 6, 77, 100].

Lakhotia [102] restates a number of static slicing methods [147, 120, 74] as well as the program integration algorithm of Horwitz, Prins, and Reps [74] in terms of operations on directed graphs. He presents a uniform framework of *graph slicing*, and distinguishes between *syntactic* properties of slices that can be obtained solely through graph-theoretic reasoning, and *semantic* properties, which involve interpretation of the graph representation of a slice. Although the paper only addresses static slicing methods, it is stated that some dynamic slicing methods [6, 100] may be modeled in a similar way.

Gupta and Soffa present a generic algorithm for *static slicing* and the solution of related dataflow problems (such as determining reaching definitions) that is based on performing a traversal of the control flow graph [66]. The algorithm is parameterized with: (i) the *direction* in which the CFG should be traversed (backward or forward), (ii) the *type* of dependences under consideration (data and/or control dependence), (iii) the *extent* of the search (i.e., should only immediate dependences be taken into account, or transitive dependences as well), and (iv) whether only the dependences that occur along *all* CFG-paths paths, or dependences that occur along *some* CFG-path should be taken into account. A slicing criterion is either a *set of variables* at a certain program point or a *set of statements*. For slices that take data dependences into account, one may choose between the values of variables *before* or *after* a statement.

Horwitz and Reps [76] present a survey of the work that has been done at the University of Wisconsin-Madison on slicing, differencing, and integration of single-procedure and multi-procedure programs as operations on PDGs [72, 74, 130, 70, 77, 75]. In addition to presenting an overview of the most significant definitions, algorithms, theorems, and complexity results, the motivation for this research is discussed in considerable detail.

An earlier classification of static and dynamic slicing methods was presented by Kamkar [86, 87]. The differences between Kamkar's work and ours may be summarized as follows. First, this work is more up-to-date and complete; for instance, Kamkar does not address any

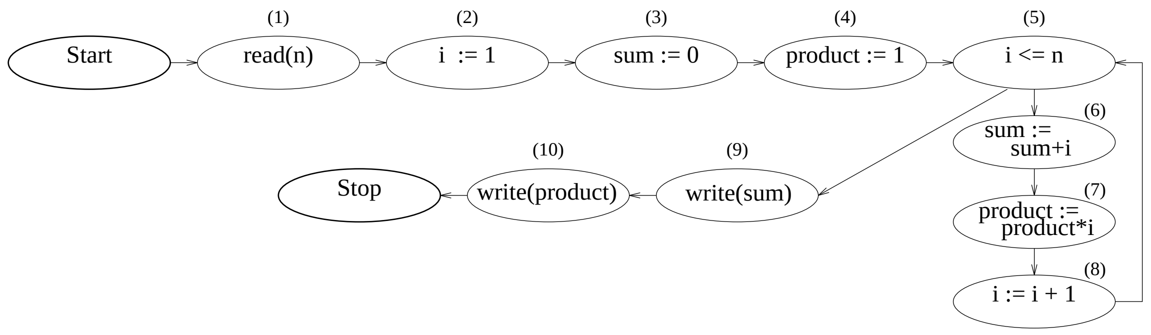


Figure 3.3: CFG of the example program of Figure 3.1 **(a)**.

of the papers that discuss slicing in the presence of unstructured control flow [12, 13, 3, 37] or methods for computing slices that are based on information-flow relations [22, 62]. Second, the organization of our work and Kamkar's is different. Whereas Kamkar discusses each slicing method and its applications separately, this survey is organized in terms of a number of "orthogonal" dimensions, such as the problems posed by procedures, or composite variables, aliasing, and pointers. This approach enables us to consider combinations of solutions to different dimensions. Third, unlike Kamkar we compare the accuracy and efficiency of slicing methods (by applying them to the same or similar example programs), and attempt to determine their fundamental strengths and weaknesses (i.e., irrespective of the original presentation). Finally, Kamkar does not discuss any of the recent approaches (see Chapter 5 and [52]) for improving the accuracy of slicing by employing compiler-optimization techniques.

3.1.5 Organization of this chapter

The remainder of this chapter is organized as follows. Section 3.2 introduces the cornerstones of most slicing algorithms: the notions of data dependence and control dependence. Readers familiar with these concepts may skip this section and consult it when needed. Section 3.3 contains an overview of static slicing methods. First, we consider the simple case of slicing structured programs with only scalar variables. Then, algorithms for slicing in the presence of procedures, unstructured control flow, composite variables and pointers, and concurrency are considered. Section 3.3.6 compares and classifies methods for static slicing. Section 3.4 addresses dynamic slicing methods, and is organized in a similar way as Section 3.3. Applications of program slicing are discussed in Section 3.5. Section 3.6 discusses recent work on the use of compiler-optimization techniques for obtaining more accurate slices. Finally, Section 3.7 summarizes the main conclusions of this survey.

3.2 Data dependence and control dependence

Data dependence and control dependence are defined in terms of the CFG of a program. A CFG contains a node for each statement and control predicate in the program; an edge from node i to node j indicates the possible flow of control from the former to the latter. CFGs contain special nodes labeled START and STOP corresponding to the beginning and the end of the program, respectively.

The sets $\text{DEF}(i)$ and $\text{REF}(i)$ denote the sets of variables defined and referenced at CFG node i, respectively. Several types of data dependences can be distinguished, such as flow dependence, output dependence and anti-dependence [53]. Flow dependences can be further classified as being loop-carried or loop-independent, depending whether or not they arise as a result of loop iteration. For the purposes of slicing, only flow dependence is relevant, and the distinction between loop-carried and loop-independent flow dependences can be ignored. Intuitively, a statement j is *flow dependent* on statement i if a value computed at i is used at j in some program execution. In the absence of aliasing [105, 104], flow dependence may be defined formally as follows: there exists a variable x such that: (i) $x \in \text{DEF}(i)$, (ii) $x \in \text{REF}(j)$, and, (iii) there exists a path from i to j without intervening definitions of x. Alternatively stated, the definition of x at node i is a *reaching definition* for node j.

Control dependence is usually defined in terms of post-dominance. A node i in the CFG is *post-dominated* by a node by j if all paths from i to STOP pass through j. A node j is *control dependent* on a node i if there exists a path P from i to j such that j post-dominates every node in P, excluding i and j. Determining control dependences in a program with arbitrary control flow is studied by Ferrante et al. [53]. For programs with structured control flow, control dependences can be determined in a simple syntax-directed manner [73]: the statements immediately[3] inside the branches of an **if** or **while** statement are control dependent on the control predicate.

As an example, Figure 3.3 shows the CFG for the example program of Figure 3.1 **(a)**. Node 7 is flow dependent on node 4 because: (i) node 4 defines variable `product`, (ii) node 7 references variable `product`, and (iii) there exists a path $4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ without intervening definitions of `product`. Node 7 is control dependent on node 5 because there exists a path $5 \rightarrow 6 \rightarrow 7$ such that: (i) node 6 is post-dominated by node 7, and (ii) node 5 is not post-dominated by node 7.

Many of the slicing approaches that will be discussed in the sequel use the Program Dependence Graph (PDG) representation of a program [101, 53]. The vertices of the PDG correspond to the statements and control predicates of the program, and the edges of a PDG correspond to data and control dependences between them. The key issue is that the partial ordering of the PDG vertices induced by the dependence edges must be obeyed to preserve the semantics of the program.

In the PDGs of Horwitz et al. [73, 74, 75, 77], a distinction is made between loop-carried and loop-independent flow dependences, and there is an additional type of data dependence edges named *def-order* dependence edges. Horwitz et al. argue that their PDG variant is *adequate*: if two programs have isomorphic PDGs, they are strongly equivalent. This means

[3]A statement in a branch of an **if** statement that occurs within another **if** statement is only control dependent on the predicate of the inner **if** statement.

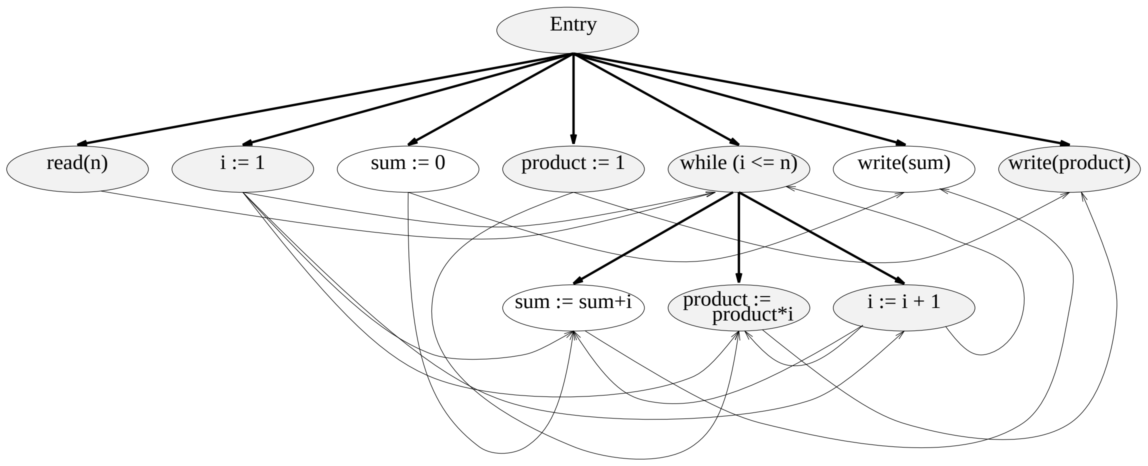


Figure 3.4: PDG of the program in Figure 3.1 **(a)**.

that, when started with the same input state, they either compute the same values for all variables, or they both diverge. It is argued that the PDG variant of [73] is minimal in the sense that removing any of the types of dependence edges, or disregarding the distinction between loop-carried and loop-independent flow edges would result in inequivalent programs having isomorphic PDGs. However, for the computation of program slices, only flow dependences and control dependences are necessary. Therefore, only these dependences will be considered in the sequel.

As an example, Figure 3.4 shows the PDG of the program of Figure 3.1 **(a)**. In this figure, the PDG variant of Horwitz, Reps, and Binkley [77] is used. Thick edges represent control dependences[4] and thin edges represent flow dependences. The shading of certain vertices in the PDG of Figure 3.4 will be explained in Section 3.3.1.3.

3.3 Methods for static slicing

3.3.1 Basic algorithms

In this section, we will study basic algorithms for static slicing of structured, single-procedure programs with scalar variables. These algorithms essentially compute the same information, but in different ways.

[4]The usual labeling of control dependence edges is omitted here, as this is irrelevant for the present discussion. Furthermore, loop-carried flow dependence edges from a vertex to itself will be omitted, as such edges are irrelevant for the computation of slices.

For each edge $i \rightarrow_{\text{CFG}} j$ in the CFG:

$$\begin{aligned} R_C^0(i) &= R_C^0(i) \cup \{\, v \mid v \in R_C^0(j),\ v \notin \text{DEF}(i) \,\} \cup \{\, v \mid v \in \text{REF}(i),\ \text{DEF}(i) \cap R_C^0(j) \neq \emptyset \,\} \\ S_C^0 &= \{i \mid (\text{DEF}(i) \cap R_C^0(j)) \neq \emptyset, i \rightarrow_{\text{CFG}} j\} \end{aligned}$$

Figure 3.5: Equations for determining *directly* relevant variables and statements.

3.3.1.1 Dataflow equations

Weiser's original definition of program slicing [147] is based on iterative solution of dataflow equations[5]. Weiser defines a *slice* as an *executable* program that is obtained from the original program by deleting zero or more statements. A *slicing criterion* consists of a pair $\langle n, V \rangle$ where n is a node in the CFG of the program, and V a subset of the program's variables. In order to be a slice with respect to criterion $\langle n, V \rangle$, a subset S of the statements of program P must satisfy the following properties: (i) S must be a valid program, and (ii) whenever P halts for a given input, S also halts for that input, computing the same values for the variables in V whenever the statement corresponding to node n is executed. At least one slice exists for any criterion: the program itself. A slice is *statement-minimal* if no other slice for the same criterion contains fewer statements. Weiser argues that statement-minimal slices are not necessarily unique, and that the problem of determining statement-minimal slices is undecidable.

Weiser describes an iterative algorithm for computing approximations of statement-minimal slices. It is important to realize that this algorithm uses *two* distinct "layers" of iteration. These can be characterized as follows:

1. Tracing transitive data dependences. This requires iteration in the presence of loops.
2. Tracing control dependences, causing the inclusion in the slice of certain control predicates. For each such predicate, step 1 is repeated to include the statements it is dependent upon.

The algorithm determines consecutive sets of *relevant variables* from which sets of *relevant statements* are derived; the computed slice is defined as the fixpoint of the latter set. First, the *directly relevant variables* are determined: this is an instance of step 1 of the iterative process outlined above. The set of directly relevant variables at node i in the CFG is denoted $R_C^0(i)$. The iteration starts with the initial values $R_C^0(n) = V$, and $R_C^0(m) = \emptyset$ for any node $m \neq n$. Figure 3.5 shows a set of equations that define how the set of relevant variables at the *end* j of a CFG edge $i \rightarrow_{\text{CFG}} j$ affects the set of relevant variables at the

[5]Weiser's definition of branch statements with indirect relevance to a slice contains an error [148]. Therefore, the modified definition proposed in [107] is followed here. However, we do not agree with the statement in [107] that 'It is not clear how Weiser's algorithm deals with loops'.

$$
\begin{aligned}
B_C^k &= \{b \mid i \in S_C^k,\ i \in \textsc{Infl}(b)\} \\
R_C^{k+1}(i) &= R_C^k(i) \cup \textstyle\bigcup_{b \in B_C^k} R^0_{(b,\textsc{Ref}(b))}(i) \\
S_C^{k+1} &= B_C^k \cup \{i \mid \textsc{Def}(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}
\end{aligned}
$$

Figure 3.6: Equations for determining *indirectly* relevant variables and statements.

NODE #	DEF	REF	INFL	R_C^0	R_C^1
1	{ `n` }	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	{ `i` }	$\emptyset$	$\emptyset$	$\emptyset$	{ `n` }
3	{ `sum` }	$\emptyset$	$\emptyset$	{ `i` }	{ `i, n` }
4	{ `product` }	$\emptyset$	$\emptyset$	{ `i` }	{ `i, n` }
5	$\emptyset$	{ `i, n` }	{ 6, 7, 8 }	{ `product, i` }	{ `product, i, n` }
6	{ `sum` }	{ `sum, i` }	$\emptyset$	{ `product, i` }	{ `product, i, n` }
7	{ `product` }	{ `product, i` }	$\emptyset$	{ `product, i` }	{ `product, i, n` }
8	{ `i` }	{ `i` }	$\emptyset$	{ `product, i` }	{ `product, i, n` }
9	$\emptyset$	{ `sum` }	$\emptyset$	{ `product` }	{ `product` }
10	$\emptyset$	{ `product` }	$\emptyset$	{ `product` }	{ `product` }

Table 3.1: Results of Weiser's algorithm for the example program of Figure 3.1 **(a)** and slicing criterion $\langle 10, \{$ `product` $\}\rangle$.

beginning i of that edge. The least fixed point of this process is the set of directly relevant variables at node i. From R_C^0, a set of *directly relevant statements*, S_C^0, is derived. Figure 3.5 shows how S_C^0 is defined as the set of all nodes i that define a variable v that is a relevant at a CFG-successor of i.

As mentioned, the second "layer" of iteration in Weiser's algorithm consists of taking control dependences into account. Variables referenced in the control predicate of an **if** or **while** statement are *indirectly* relevant, if (at least) one of the statements in its body is relevant. To this end, the *range of influence* $\textsc{Infl}(b)$ of a branch statement b is defined as the set of statements control dependent on b. Figure 3.6 shows a definition of the branch statements B_C^k that are indirectly relevant due to the influence they have on nodes i in S_C^k. Next, the sets of *indirectly relevant variables* $R_C^{k+1}(i)$ are determined. In addition to the variables in $R_C^k(i)$, $R_C^{k+1}(i)$ contains variables that are relevant because they have a transitive data dependence on statements in B_C^k. This is determined by performing the first type of iteration again (i.e., tracing transitive data dependences) with respect to a set of criteria $\langle b, \textsc{Ref}(b)\rangle$, where b is a branch statement in B_C^k (see Figure 3.6). Figure 3.6 also shows a definition of the sets S_C^{k+1} of *indirectly relevant statements* in iteration $k+1$. This set consists of the the nodes in B_C^k together with the nodes i that define a variable that is R_C^{k+1}-relevant to a CFG-successor j.

$$\begin{array}{lll|lll|lll}
\lambda_\epsilon & = & \emptyset & \lambda_{S_1;S_2} & = & \lambda_{S_1} \cup (\rho_{S_1} \cdot \lambda_{S_2}) & \lambda_{v:=e} & = & \textsc{Vars}(e) \times \{\, e \,\} \\
\mu_\epsilon & = & \emptyset & \mu_{S_1;S_2} & = & (\mu_{S_1} \cdot \rho_{S_2}) \cup \mu_{S_2} & \mu_{v:=e} & = & \{\, \langle e, v \rangle \,\} \\
\rho_\epsilon & = & \textsc{Id} & \rho_{S_1;S_2} & = & \rho_{S_1} \cdot \rho_{S_2} & \rho_{v:=e} & = & (\textsc{Vars}(e) \times \{\, v \,\}) \cup (\textsc{Id} - \langle v, v \rangle)
\end{array}$$

$$\begin{array}{lll}
\lambda_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\textsc{Vars}(e) \times \{\, e \,\}) \cup \lambda_{S_1} \cup \lambda_{S_2} \\
\mu_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\{\, e \,\} \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \mu_{S_1} \cup \mu_{S_2} \\
\rho_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\textsc{Vars}(e) \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \rho_{S_1} \cup \rho_{S_2} \cup \textsc{Id} \\
\\
\lambda_{\textbf{while } e \textbf{ do } S} & = & \rho_S^* \cdot ((\textsc{Vars}(e) \times \{\, e \,\}) \cup \lambda_S) \\
\mu_{\textbf{while } e \textbf{ do } S} & = & (\{\, e \,\} \times \textsc{Defs}(S)) \cup \mu_S \cdot \rho_S^* \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id}) \\
\rho_{\textbf{while } e \textbf{ do } S} & = & \rho_S^* \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id})
\end{array}$$

Figure 3.7: Definition of information-flow relations.

The sets R_C^{k+1} and S_C^{k+1} are nondecreasing subsets of the program's variables and statements, respectively; the fixpoint of the computation of the S_C^{k+1} sets constitutes the desired program slice.

As an example, consider slicing the program of Figure 3.1 **(a)** with respect to criterion $\langle 10, \{\, \texttt{product} \,\} \rangle$. Table 3.1 summarizes the DEF, REF, INFL sets, and the sets of relevant variables computed by Weiser's algorithm. The CFG of the program was shown earlier in Figure 3.3. From the information in the table, and the definition of a slice, we obtain $S_C^0 = \{2, 4, 7, 8\}$, $B_C^0 = \{5\}$, and $S_C^1 = \{1, 2, 4, 5, 7, 8\}$. For our example, the fixpoint of the sets of indirectly relevant variables is reached at set S_C^1. The corresponding slice w.r.t. criterion $C \equiv (10, \{\, \texttt{product} \,\})$ as computed by Weiser's algorithm is identical to the program shown in Figure 3.1 **(b)** apart from the fact that the output statement `write(product)` is not contained in the slice.

Lyle [108] presents a modified version of Weiser's slicing algorithm. Apart from some minor changes in terminology, this algorithm is essentially the same as Weiser's [147].

Hausler [67] restates Weiser's algorithm in a functional style. For each type of statement (empty statement, assignment, statement composition, **if**, and **while**) he defines two functions δ and α. Roughly speaking, these functions express how a statement transforms the set of relevant variables R_C^i, and the set of relevant statements S_C^i, respectively. The functions δ and α are defined in a compositional manner. For empty statements and assignments, δ and α can be derived from the statement in a syntax-directed manner. The δ and α functions for statement sequences and **if** statements, can be inferred from the δ and α functions for their components, respectively. The functions for a **while** statement are obtained by effectively transforming it into an infinite sequence of **if** statements.

EXPRESSION #[a]	POTENTIALLY AFFECTED VARIABLES
1	{ `n, sum, product, i` }
2	{ `sum, product, i` }
3	{ `sum` }
4	{ `product` }
5	{ `sum, product, i` }
6	{ `sum` }
7	{ `product` }
8	{ `sum, product, i` }
9	$\emptyset$
10	$\emptyset$

[a]Expression numbers correspond to line numbers in Figure 3.1 **(a)**.

Figure 3.8: Information-flow relation μ for the example program of Figure 3.1 **(a)**.

3.3.1.2 Information-flow relations

Bergeretti and Carré [22] define a number of *information-flow relations* that can be used to compute slices. For a statement (or sequence of statements) S, a variable v, and an expression (i.e., a control predicate or the right-hand side of an assignment) e that occurs in S, the relations λ_S, μ_S, and ρ_S are defined. These information-flow relations possess the following properties: $\langle v, e \rangle \in \lambda_S$ iff the value of v on entry to S potentially affects the value computed for e, $\langle e, v \rangle \in \mu_S$ iff the value computed for e potentially affects the value of v on exit from S, and $\langle v, v' \rangle \in \rho_S$ iff the value of v on entry to S may affect the value of v' on exit from S. The set E_S^v of *all* expressions e for which $\langle e, v \rangle \in \mu_S$ can be used to construct *partial statements*. A partial statement of statement S associated with variable v is obtained by replacing all statements in S that do not contain expressions in E_S^v by empty statements.

Information-flow relations are computed in a syntax-directed, bottom-up manner. For an empty statement, the relations λ_S and μ_S are empty, and ρ_S is the identity. For an assignment $v := e$, λ_S contains $\langle v', e \rangle$ for all variables v' that occur in e, μ_S consists of $\langle e, v \rangle$, and ρ_S contains $\langle v', v \rangle$ for all variables that occur in e as well as $\langle v'', v'' \rangle$ for all variables $v'' \neq v$. Figure 3.7 shows how information-flow relations for sequences of statements, conditional statements and loop statements are constructed from the information-flow relations of their constituents. In the figure, ϵ denotes an empty statement, "$\cdot$" relational join[6], ID the identity relation, $\text{VARS}(e)$ the set of variables occurring in expression e, and $\text{DEFS}(S)$ the set of variables that may be defined in statement S. The convoluted definition for **while** constructs is obtained by effectively transforming it into an infinite sequence of nested one-branch **if** statements. The relation ρ^* used in this definition is the transitive and reflexive closure of ρ.

A slice w.r.t. the value of a variable v at an arbitrary location can be computed by inserting a dummy assignment $v' := v$ at the appropriate place, where v' is a variable that

[6]The join of two relations R_1 and R_2 contains all pairs $\langle e_1, e_3 \rangle$ for which there exists an e_2 such that $\langle e_1, e_2 \rangle \in R_1$ and $\langle e_2, e_3 \rangle \in R_2$.

did not previously occur in S. The slice w.r.t. the final value of v' in the modified program is equivalent to a slice w.r.t. v at the selected location in the original program.

Static *forward* slices can be derived from relation λ_S in a way that is similar to the method for computing static backward slices from the μ_S relation.

Figure 3.8 shows the information-flow relation μ for the (entire) program of Figure 3.1 **(a)**[7]. From this relation it follows that the set of expressions that potentially affect the value of `product` at the end of the program are $\{\ 1,\ 2,\ 4,\ 5,\ 7,\ 8\ \}$. The corresponding partial statement is obtained by omitting all statements from the program that do not contain expressions in this set, i.e., both assignments to `sum` and both `write` statements. The result is identical to the slice computed by Weiser's algorithm (see Section 3.3.1.1).

3.3.1.3 Dependence graph based approaches

Ottenstein and Ottenstein [120] were the first of many to define slicing as a reachability problem in a dependence graph representation of a program. They use the PDG [101, 53] for static slicing of single-procedure programs.

In dependence graph based approaches, the slicing criterion is identified with a vertex v in the PDG. In Weiser's terminology, this corresponds to a criterion $\langle n, V\rangle$ where n is the CFG node corresponding to v, and V the set of *all* variables defined or used at v. Consequently, slicing criteria of PDG-based slicing methods are less general than those of methods based on dataflow equations or information-flow relations (the fine-grained PDGs of Jackson and Rollins, discussed below, are a notable exception here). However, in Section 3.3.6.2, it will be shown how more precise slicing criteria can be "simulated" by PDG-based slicing methods. For single-procedure programs, the slice w.r.t. v consists of all vertices that can reach v. The related parts of the source text of the program can be found by maintaining a mapping between vertices of the PDG and the source text during the construction of the PDG.

The PDG variant of Ottenstein and Ottenstein [120] shows considerably more detail than that by Horwitz, Reps, and Binkley [77]. In particular, there is a vertex for each (sub)expression in the program, and file descriptors appear explicitly as well. As a result, `read` statements involving irrelevant variables are not "sliced away", and slices will execute correctly with the full input of the original program.

In Figure 3.4 the PDG of the program of Figure 3.1 **(a)** was shown. Shading is used to indicate the vertices in the slice w.r.t. `write(product)`.

Jackson and Rollins [82] introduce a variation on the PDG that is distinguished by fine-grained dependences between individual variables defined or used at program points. An important advantage of this approach is that it allows one to determine more accurately which variables are responsible for the inclusion of a particular statement in a slice.

Each vertex consists of a *box* that contains a separate *port* for each variable defined at that

[7]Bergeretti and Carré do not define information-flow relations for I/O statements. For the purposes of this example, it is assumed that the statement `read(n)` can be treated as an assignment `n :=` *SomeConstant*, and that the statements `write(sum)` and `write(product)` should be treated as empty statements.

program point, as well as for each variable used at that point. Dependence relations between variables used at a program point p, and variables defined at p are represented by *internal* dependence edges *inside* the box for p. Data dependences between statements are defined in the usual way, in terms of reaching definitions. Control dependences between statements, however, are modeled as mock data dependences. To this end, each box has an ϵ port that represents the "execution of" the associated statement. Control predicates are assumed to define a temporary value that is represented by a τ port. If a statement with box p is control dependent on a statement with box q, this is modeled by a dependence edge from p's τ port to q's ϵ port. Finally, dependences on constant values and input values are represented by γ ports—the role of these ports is irrelevant for the present discussion.

Jackson and Rollins generalize the traditional notion of a slicing criterion to a pair $\langle$*source*, *sink*$\rangle$, where *source* is a set of definition ports and *sink* of a set of use ports. Slicing is generalized to *chopping*: determining the subset of the program's statements that cause influences of *source* elements on *sink* elements. It is argued that conventional notions slicing of backward and forward slicing can be expressed in terms of chopping. Rather than defining chopping algorithms in the usual way, as a graph-reachability algorithm, Jackson and Rollins formally define their algorithm in a relational fashion, as a number of relations between ports.

3.3.2 Procedures

The main problem posed by interprocedural static slicing is that, in order to compute accurate slices, the call-return structure of interprocedural execution paths must be taken into account. Simple algorithms that include statements in a slice by traversing (some representation of) the program in a single pass have the property that they consider *infeasible* execution paths, causing slices to become larger than necessary. Several solutions to this problem, often referred to as the "calling-context" problem, will be discussed below.

3.3.2.1 Dataflow equations

Weiser's approach for interprocedural static slicing [147, 148] involves three separate tasks.

- First, interprocedural *summary information* is computed, using previously developed techniques [19]. For each procedure P, a set $\textsc{Mod}(P)$ of variables that may be modified by P is computed, and a set $\textsc{Use}(P)$ of variables that may be used by P. In both cases, the effects of procedures transitively called by P are taken into account.
- The second component of Weiser's algorithm is an *intra*procedural slicing algorithm. This algorithm was discussed previously in Section 3.3.1.1. However, it is slightly extended in order to determine the effect of call-statements on the sets of relevant variables and statements that are computed. This is accomplished using the summary information computed in step (1). A call to procedure P is treated as a conditional assignment statement '**if** <SomePredicate> **then** $\textsc{Mod}(P)$:= $\textsc{Use}(P)$' where actual parameters are substituted for formal parameters [148]. Worst-case assumptions

```
      program Main;
      ...
        while ( ··· ) do
          P(x1, x2, ···, xn);
          x1 := z;
          x1 := x2;
          x2 := x3;
          ...
          x(n-1) := xn
        end;
(L)     write(z)
      end
```

```
     procedure P( y1, y2, ···, yn );
     begin
       write(y1);
       write(y2);
       ...
(M)    write(yn)
     end
```

Figure 3.9: A program where procedure P is sliced n times by Weiser's algorithm for criterion $\langle \texttt{L}, \{ \texttt{z} \} \rangle$.

have to be made when a program calls external procedures, and the source-code is unavailable.

- The third part is the actual interprocedural slicing algorithm that iteratively generates new slicing criteria with respect to which intraprocedural slices are computed in step (2). For each procedure P, new criteria are generated for (i) procedures Q called by P, and (ii) procedures R that call P. The new criteria of (i) consist of all pairs $(n_Q,\ V_Q)$ where n_Q is the last statement of Q and V_Q is the set of relevant variables in P in the scope of Q (formals are substituted for actuals). The new criteria of (ii) consist of all pairs $(N_R,\ V_R)$ such that N_R is a call to P in R, and V_R is the set of relevant variables at the first statement of P that is in the scope of R (actuals are substituted for formals).

Weiser formalizes the generation of new criteria by way of functions $\textsc{Up}(\mathcal{S})$ and $\textsc{Down}(\mathcal{S})$ that map a set $\mathcal{S}$ of slicing criteria in a procedure P to a set of criteria in procedures that call P, and a set of criteria in procedures called by P, respectively. The set of *all* criteria with respect to which intraprocedural slices are computed consists of the transitive and reflexive closure of the Up and Down relations; this is denoted $(\textsc{Up} \cup \textsc{Down})*$. Thus, for an initial criterion C, slices will be computed for all criteria in the set $(\textsc{Up} \cup \textsc{Down}) * (\{ C \})$.

Weiser determines the criteria in this set "on demand" [148]: the generation of new criteria in step (3) and the computation of intraprocedural slices in step (2) are intermixed; the iteration stops when no new criteria are generated. Although the number of intraprocedural slices computed in step (2) could be reduced by combining "similar" criteria (e.g., replacing two criteria $\langle n, V_1 \rangle$ and $\langle n, V_2 \rangle$ by a single criterion $\langle n, V_1 \cup V_2 \rangle$), Weiser writes that "no speed-up tricks have been implemented" [147, page 355, col.2]. In fact, one would expect that such speed-up tricks would affect the performance of his algorithm dramatically. The main issue is that the computation of the Up and Down sets requires that the sets of relevant variables are known at all call sites. In other words, the computation of these sets relies on *slicing* these procedures. In the course of doing this, new variables may become relevant

```
    program Example;
    begin
(1)   a := 17;
(2)   b := 18;
(3)   P(a,b,c,d);
(4)   write(d)
    end

    procedure P(v,w,x,y);
(5)   x := v;
(6)   y := w
    end
```

(a)

```
program Example;
begin
  a := 17;
  b := 18;
  P(a,b,c,d);

end

procedure P(v,w,x,y);
        ;
  y := w
end
```

(b)

```
program Example;
begin
          ;
  b := 18;
  P(a,b,c,d);
  write(d)
end

procedure P(v,w,x,y);
        ;
  y := w
end
```

(c)

Figure 3.10: **(a)** Example program. **(b)** Weiser's slice with respect to criterion $\langle 4, \{$ d $\}\rangle$. **(a)** A slice with respect to the same criterion computed by the Horwitz-Reps-Binkley algorithm.

at previously encountered call sites, and new call sites may be encountered. Consider for example, the program shown in Figure 3.9. In the subsequent discussion, L denotes the program point at statement `write(z)` and M the program point at the last statement in procedure `P`. Computing the slice w.r.t. criterion $\langle L, \{$ z $\}\rangle$ requires n iterations of the body of the **while** loop. During the i^{th} iteration, variables $\mathtt{x}_1, \cdots, \mathtt{x}_i$ will be relevant at the call site, causing the inclusion of criterion $\langle M, \{ \mathtt{y}_1, \cdots, \mathtt{y}_i \}\rangle$ in DOWN(`Main`). If no precaution is taken to combine the criteria in DOWN(`Main`), procedure `P` will be sliced n times.

The fact that Weiser's algorithm does not take into account *which* output parameters are dependent on *which* input parameters is a source of imprecision. Figure 3.10 **(a)** shows an example program that manifests this problem. For criterion $\langle 4, \{$ d $\}\rangle$, Weiser's interprocedural slicing algorithm [147] will compute the slice shown in Figure 3.10 **(b)**. This slice contains the statement `a := 17` due to the spurious dependence between variable `a` before the call, and variable `d` after the call. The Horwitz-Reps-Binkley algorithm that will be discussed in Section 3.3.2.3 will compute the more accurate slice shown in Figure 3.10 **(c)**.

Horwitz, Reps, and Binkley [77] report that Weiser's algorithm for interprocedural slicing is unnecessarily inaccurate, because of what they refer to as the "calling context" problem. In a nutshell, the problem is that when the computation "descends" into a procedure Q that is called from a procedure P, it will "ascend" to *all* procedures that call Q, not only P. This includes *infeasible* execution paths that enter Q from P and exit Q to a different procedure. Traversal of such paths gives rise to inaccurate slices.

Figure 3.11 shows a program that exhibits the calling-context problem. For example, assume that a slice is to be computed w.r.t. criterion $\langle 10,$ `product`$\rangle$. Using summary information to approximate the effect of the calls, the initial approximation of the slice will consist of the entire main procedure except lines 3 and 6. In particular, the procedure calls `Multiply(product, i)` and `Add(i, 1)` are included in the slice,

```
      program Example;
      begin
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     Add(sum, i);
(7)     Multiply(product, i);
(8)     Add(i, 1)
      end;
(9)   write(sum);
(10)  write(product)
      end
```

```
      procedure Add(a; b);
      begin
(11)  a := a + b
      end

      procedure Multiply(c; d);
      begin
(12)  j := 1;
(13)  k := 0;
(14)  while j <= d do
      begin
(15)    Add(k, c);
(16)    Add(j, 1);
      end;
(17)  c := k
      end
```

Figure 3.11: Example of a multi-procedure program.

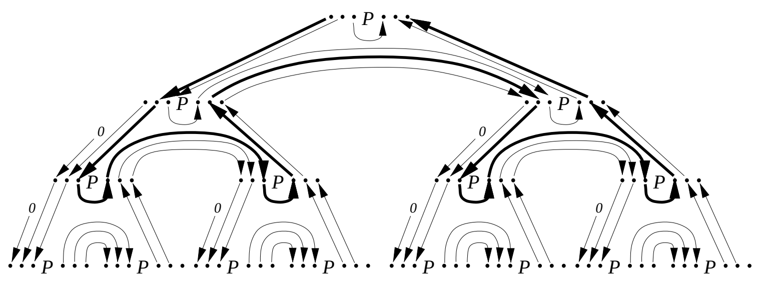
because: (i) the variables `product` and `i` are deemed relevant at those points, and (ii) using interprocedural data flow analysis it can be determined that MOD(`Add`) $= \{$ `a` $\}$, USE(`Add`) $= \{$ `a, b` $\}$, MOD(`Multiply`) $= \{$ `c` $\}$, and USE(`Multiply`) $= \{$ `c, d` $\}$. As the initial criterion is in the main program, we have that UP($\{ \langle 10,$ `product` $\rangle \}$) $= \emptyset$, and that DOWN($\{ \langle 10,$ `product` $\rangle \}$) contains the criteria $\langle 11, \{$ `a` $\}\rangle$ and $\langle 17, \{$ `c, d` $\}\rangle$. The result of slicing procedure `Add` for criterion $\langle 11, \{$ `a` $\}\rangle$ and procedure `Multiply` for criterion $\langle 17, \{$ `c, d` $\}\rangle$ will be the inclusion of these procedures in their entirety. Note that the calls to `Add` at lines 15 and 16 cause the generation of a new criterion $\langle 11, \{$ `a, b` $\}\rangle$ and thus re-slicing of procedure `Add`. It can now be seen that the example program exhibits the "calling context" problem: Since line (11) is in the slice, new criteria are generated for *all* calls to `Add`. These calls include the (already included) calls at lines 8, 15, and 16, but also the call `Add(sum, i)` at line 6. The new criterion $\langle 6, \{$ `sum, i` $\}\rangle$ that is generated will cause the inclusion of lines 6 and 3 in the slice. Consequently, the slice consists of the entire program.

It is our conjecture that the calling context problem of Weiser's algorithm can be fixed by observing that the criteria in the UP sets are only needed to include procedures that (transitively) call the procedure containing the initial criterion[8]. Once this is done, *only* DOWN sets need to be computed. For an initial criterion C, this corresponds to determining the set of criteria DOWN $*$ (UP $*$ ($\{ C \}$)), and computing the intraprocedural slices with respect to each of these criteria. Reps [126] suggested that this essentially corresponds to the two passes of the Horwitz-Reps-Binkley algorithm (see Section 3.3.2.3) if all UP sets are computed before determining any DOWN sets.

[8]A similar observation was made by Jiang et al. [83]. However, they do not explain that this approach only works when a call to procedure p is treated as a conditional assignment **if** <SomePredicate> **then** MOD(P) := USE(P).

```
     program P³(x₁,x₂,x₃);
     begin
       t := 0;
       P³(x₂,x₃,t);
       P³(x₂,x₃,t);
(L)    x₁ := x₁ + 1
     end;
```

(a)



(b)

Figure 3.12: **(a)** Example program. **(b)** Exponentially long path traversed by the Hwang-Du-Chou algorithm for interprocedural static slicing for criterion $\langle \texttt{L}, \texttt{x}_3 \rangle$.

Hwang, Du, and Chou [79] propose an iterative solution for interprocedural static slicing based on replacing (recursive) calls by instances of the procedure body. From a conceptual point of view, each iteration comprises of the following two steps. First, procedure calls are inlined, substituting actual parameters for formal parameters. Then, the slice is re-computed, where any remaining procedure call is treated as if it were an empty statement (i.e., it is assumed to have no effect on the flow dependences between its surrounding statements). This iterative process terminates when the resulting slice is identical to the slice computed in the previous iteration—i.e., until a fixed point is reached. It is assumed that some mapping is maintained between the statements in the various expanded versions of the program, and in the original program.

The approach of Hwang et al. does not suffer from the calling context problem because expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps [125, 128] has shown recently that for a certain family $\texttt{P}^k$ of recursive programs, this algorithm takes time $O(2^k)$, i.e., exponential in the length of the program. An example of such a program is shown in Figure 3.12 **(a)**. Figure 3.12 **(b)** shows the exponentially long path that is effectively traversed by the Hwang-Du-Chou algorithm.

3.3.2.2 Information-flow relations

Bergeretti and Carré [22] explain how the effect of procedure calls can be approximated in the absence of recursion. Exact dependences between input and output parameters are determined by slicing the called procedure with respect to each output parameter (i.e., computation of the μ relation for the procedure). Then, each procedure call is replaced by a set of assignments, where each output parameter is assigned a fictitious expression that contains the input parameters it depends upon. As only feasible execution paths are considered, this approach does not suffer from the calling context problem. A call to a side-effect free function can be modeled by replacing it with a fictitious expression containing all actual parameters. Note that the computed slices are not truly interprocedural since no attempt is done to slice procedures other than the main program.

For the example program of Figure 3.11, the slice w.r.t. the final value of `product` would include all statements except `sum := 0`, `Add(sum,i)`, and `write(sum)`.

3.3.2.3 Dependence graphs

Horwitz, Reps, and Binkley [77] present an algorithm for computing precise interprocedural static slices, which consists of the following three components:

1. The *System Dependence Graph* (SDG), a graph representation for multi-procedure programs.
2. The computation of interprocedural summary information. This takes the form of precise dependence relations between the input and output parameters of each procedure call, and is explicitly present in the SDG in the form of *summary edges*.
3. A two-pass algorithm for extracting interprocedural slices from an SDG.

We will begin with a brief overview of SDGs. In the discussion that follows it is important to realize that parameter passing by value-result[9] is modeled as follows: (i) the calling procedure copies its actual parameters to *temporary* variables before the call, (ii) the formal parameters of the called procedure are initialized using the corresponding temporary variables, (iii) before returning, the called procedure copies the final values of the formal parameters to the temporary variables, and (iv) after returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.

An SDG contains a program dependence graph for the main program, and a procedure dependence graph for each procedure. There are several types of vertices and edges in SDGs that do not occur in PDGs. For each call statement, there is a *call-site vertex* in the SDG as well as *actual-in* and *actual-out* vertices that model the copying of actual parameters to/from temporary variables. Each procedure dependence graph has an entry vertex, and *formal-in* and *formal-out* vertices to model copying of formal parameters to/from temporary variables[10]. Actual-in and actual-out vertices are control dependent on the call-site vertex; formal-in and formal-out vertices are control dependent on the procedure's entry vertex. In addition to these *intra*procedural dependence edges, an SDG contains the following *inter*procedural dependence edges: (i) a control dependence edge between a call-site vertex and the entry vertex of the corresponding procedure dependence graph, (ii) a *parameter-in* edge between corresponding actual-in and formal-in vertices, (iii) a *parameter-out* edge between corresponding formal-out and actual-out vertices, and (iv) *summary edges* that represent *transitive inter*procedural data dependences.

[9]The Horwitz-Reps-Binkley algorithm [77] is also suitable for call-by-reference parameter passing provided that aliases are resolved. To this end, two approaches are proposed: transformation of the original program into an equivalent alias-free program, or the use of a generalized flow dependence notion that takes possible aliasing patterns into account. The first approach yields more precise slices, whereas the second one—further explored by Binkley [31]—is more efficient. For a discussion of parameter passing mechanisms the reader is referred to [7, Section 7.5].

[10]Using interprocedural data flow analysis [16], the sets of variables that can be referenced or modified by a procedure can be determined. This information can be used to eliminate actual-out and formal-out vertices for parameters that will never be modified, resulting in more precise slices.

The second part of the Horwitz-Reps-Binkley algorithm, computation of summary dependences, involves the construction of an attribute grammar that models the calling relationships between the procedures (as in a call graph). Then, the subordinate characteristic graph for this grammar is computed. For each procedure in the program, this graph contains edges that correspond to precise transitive flow dependences between its input and output parameters. The summary edges of the subordinate characteristic graph are copied to the appropriate places at each call site in the SDG. Details of the Horwitz-Reps-Binkley algorithm for determining summary edges are outside the scope of this survey—for details, the reader is referred to [77].

The third phase of the Horwitz-Reps-Binkley algorithm consists of a two-pass traversal of the SDG. The summary edges of an SDG serve to circumvent the calling context problem. Assume that slicing starts at some vertex s. The first phase determines all vertices from which s can be reached *without descending into procedure calls*. The transitive interprocedural dependence edges guarantee that calls can be side-stepped, without descending into them. The second phase determines the remaining vertices in the slice by descending into all previously side-stepped calls.

Figure 3.13 shows the SDG for the program of Figure 3.11, where interprocedural dataflow analysis is used to eliminate the vertices for the second parameters of the procedures `Add` and `Multiply`. In the figure, thin solid arrows represent flow dependences, thick solid arrows correspond to control dependences, thin dashed arrows are used for call, parameter-in, and parameter-out dependences, and thick dashed arrows represent transitive interprocedural flow dependences. The vertices in the slice w.r.t. statement `write(product)` are shown shaded; light shading indicates the vertices identified in the first phase of the algorithm, and dark shading indicates the vertices identified in the second phase. Clearly, the statements `sum := 0`, `Add(sum, i)`, and `write(sum)` are not in the slice.

Slices computed by Horwitz-Reps-Binkley algorithm [77] are not necessarily executable programs. Cases where only a subset of the vertices for actual and formal parameters are in the slice correspond to procedures where *some* of the arguments are "sliced away"; for different calls to the procedure, different arguments may be omitted. Two approaches are proposed for transforming such a non-executable slice an executable program. First, several variants of a procedure may be incorporated in a slice [77]; this has the disadvantage that the slice is no longer a restriction of the original program. The second solution consists of extending the slice with all parameters that are present at *some* call to *all* calls that occur in the slice. In addition, all vertices on which the added vertices are dependent must be added to the slice as well. This second approach is pursued by Binkley [30]. Clearly the second approach yields larger slices than the first one.

Finally, it is outlined how interprocedural slices can be computed from partial SDGs (corresponding to programs under development, or programs containing library calls) and how, using the SDG, interprocedural *forward* slices can be computed in a way that is very similar to the previously described method for interprocedural (backward) slicing.

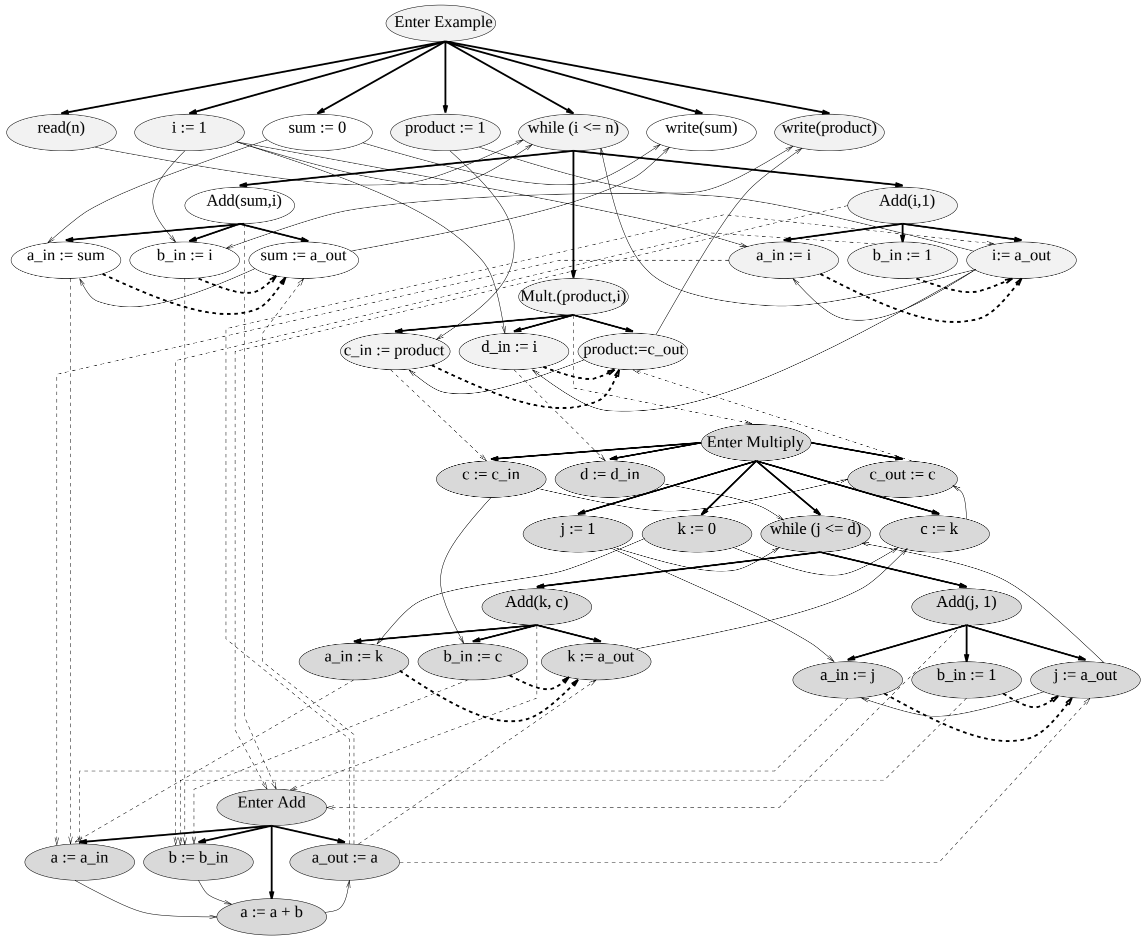


Figure 3.13: SDG of the program in Figure 3.11.

Recently, Reps et al. [129, 128] proposed a new algorithm for computing the summary edges of an SDG, which is asymptotically more efficient than the Horwitz-Reps-Binkley algorithm [77] (the time requirements of these algorithms will be discussed in Section 3.3.6.3). Input to the algorithm is an SDG where no summary edges have been added yet, i.e., a collection of procedure dependence graphs connected by call, parameter-in, and parameter-out edges. The algorithm uses a worklist to determine *same-level* realizable paths. Intuitively, a same-level realizable path obeys the call-return structure of procedure calls, and it starts and ends at the same level (i.e., in the same procedure). Same-level realizable paths between formal-in and formal-out vertices of a procedure P induce summary edges between the corresponding actual-in and actual-out vertices for any call to P. The algorithm starts by asserting that a same-level realizable path of length zero exists from any formal-out vertex to itself. A worklist is used to select a path, and extend it by adding an edge to its beginning. Reps et al. [129] also present a demand version of their algorithm, which *incrementally* determines the summary edges of an SDG.

Lakhotia [103] presents an algorithm for computing interprocedural slices that is also based on SDGs. This algorithm computes slices that are identical to the slices computed by the Horwitz-Reps-Binkley algorithm. Associated with every SDG vertex v is a three-valued tag; possible values for this tag are: "$\perp$" indicating that v has not been visited, "$\top$" indicating that v has been visited, and *all* vertices from which v can be reached should be visited, and "β" indicating that v has been visited, and *some* of the vertices from which v can be reached should be visited. More precisely, an edge from an entry vertex to a call vertex should only be traversed if the call vertex is labeled $\top$. A worklist algorithm is used to visit all vertices labeled $\top$ before visiting any vertex labeled β. When this process ends, vertices labeled either $\top$ or β are in the slice. Lakhotia's algorithm traverses performs a single pass through the SDG. However, unlike the Horwitz-Reps-Binkley algorithm, the value of a tag may change *twice*. Therefore it is unclear if Lakhotia's algorithm is really an improvement over the Horwitz-Reps-Binkley two-pass traversal algorithm.

The dependence graph model of Jackson and Rollins [82] (see Section 3.3.1.3) is "modular", in the sense that a single box is used for each procedure call. Instead of linking the individual dependence graphs for the procedures of the program, Jackson and Rollins represent procedure calls in a more abstract fashion: the internal dependence edges inside a procedure's box effectively correspond to the summary edges of Horwitz et al. [77, 128]. Unlike the previously discussed methods, this algorithm side-steps the calling context problem by only extending slices to *called* procedures, and not to *calling* procedures (unless explicitly requested by the user). Here, 'extending a slice to a called procedure' involves slicing the (separate) dependence graph for that procedure with respect to the appropriate ports of its exit node (i.e., corresponding to the ports at the point of call that occur in the slice).

Whereas for simple statements the internal dependence edges between ports of the associated box in the dependence graph can be computed in a simple syntax-directed manner, a more elaborate scheme is required for procedures. In the absence of recursion, the internal summary dependence edges for a procedure are derived from the dependences inside and between the boxes for the statements that constitute the procedure body. For recursive procedures, Jackson and Rollins briefly discuss a simple iterative scheme for determining internal dependence edges, and state that their algorithm is essentially an adaptation of the solution presented by Ernst [52] (see Section 3.6). The essence of their scheme is that the internal dependence edges for non-recursive calls are determined in the manner sketched above, and that there are initially no internal dependence edges for calls in a recursive cycle. In each subsequent step, the transitive dependences between the input parameters and the output parameters of a recursive procedure are recomputed by slicing in a graph that contains the summary edges determined in the previous cycle. Then, summary edges are added to the graph for those dependences that did not occur in the previous cycle. This iteration process terminates when no more additional transitive dependences can be found.

3.3.3 Unstructured control flow

3.3.3.1 Dataflow equations

Lyle [108] reports that (his version of) Weiser's algorithm for static slicing yields incorrect slices in the presence of unstructured control flow: the behavior of the slice is not necessarily a projection of the behavior of the program. He presents a conservative solution for dealing with **goto** statements: any **goto** that has a non-empty set of relevant variables associated with it is included in the slice.

Gallagher [60] and Gallagher and Lyle [61] also use a variation of Weiser's method. A **goto** statement is included in the slice if it jumps to a label of an included statement[11]. Agrawal [3] shows that this algorithm does not produce correct slices in all cases.

Jiang et al. [83] extend Weiser's slicing method to C programs with unstructured control flow. They introduce a number of additional rules to "collect" the unstructured control flow statements such as **goto**, **break**, and **continue** that are part of the slice. Unfortunately, no formal justification is given for the treatment of unstructured control flow constructs in [83]. Agrawal [3] shows that this algorithm may also produce incorrect slices.

3.3.3.2 Dependence graphs

Ball and Horwitz [12, 13] and Choi and Ferrante [37] discovered independently that conventional PDG-based slicing algorithms produce incorrect results in the presence of unstructured control flow: slices may compute values at the criterion that differ from what the original program does. These problems are due to the fact that the algorithms do not determine correctly when unconditional jumps such as **break**, **goto**, and **continue** statements are required in a slice.

As an example, Figure 3.14 **(a)** shows a variant of our example program, which uses a **goto** statement. Figure 3.14 **(b)** shows the PDG for this program. The vertices that have a transitive dependence on statement `write(product)` are highlighted. Figure 3.14 **(c)** shows a textual representation of the program thus obtained. Clearly, this "slice" is incorrect because it does not contain the **goto** statement, causing non-termination. In fact, the previously described PDG-based algorithms will *only* include a **goto** if it is the slicing criterion itself, because no statement is either data or control dependent on a **goto**.

The solution of [12, 13] and the first solution presented in [37] are remarkably similar: unconditional jumps are regarded as *pseudo-predicate* vertices where the "true" branch consists of the statement that is being jumped to, and the "false" branch of the *textually* next statement. Correspondingly, there are two outgoing edges in the *augmented* control flow graph (ACFG). Only one of these edges can actually be traversed during execution; the other outgoing edge is "non-executable". In constructing the (augmented) PDG, data dependences are computed using the (original) CFG, and control dependences are computed

[11] Actually, this is a slight simplification. Each basic block is partitioned into labeled blocks; a *labeled block* is a subsequence of the statements in a basic block starting with a labeled statement, and containing no other labeled statements. A **goto** is included in the slice if it jumps to a label for which there is some included statement in its block.

```
read(n);
i := 1;
sum := 0;
product := 1;
while true do
begin
  if (i > n) then
    goto L;
  sum := sum + i;
  product := product * i;
  i := i + 1
end;
L: write(sum);
write(product)
```

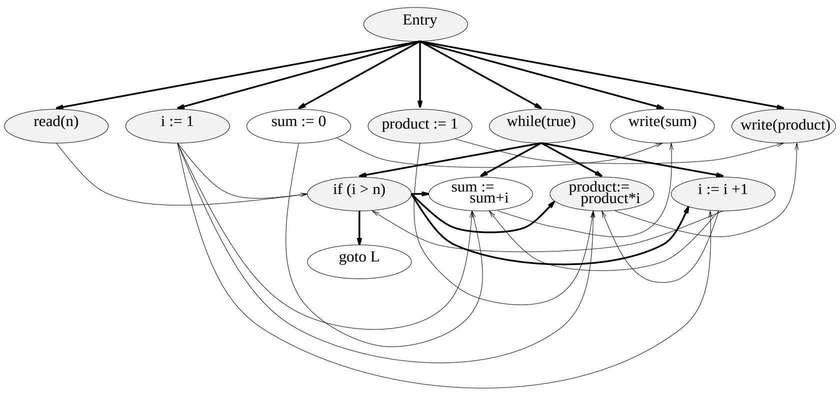
(a)

```
read(n);
i := 1;

product := 1;
while true do
begin
  if (i > n) then
          ;

  product := product * i;
  i := i + 1
end;

write(product)
```

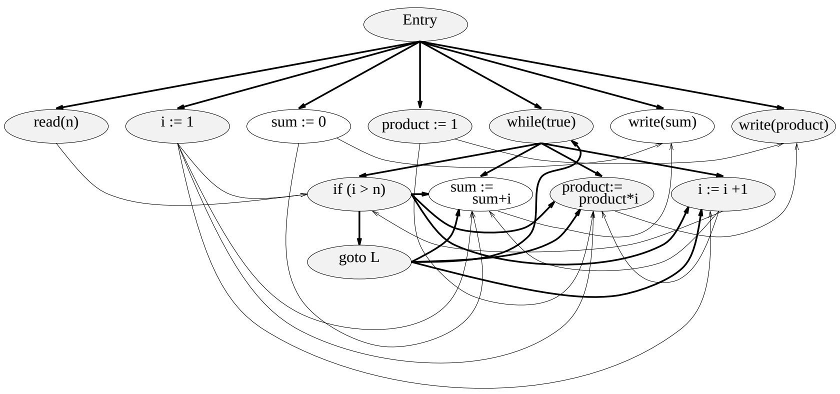
(c)

```
read(n);
i := 1;

product := 1;
while true do
begin
  if (i > n) then
    goto L;

  product := product * i;
  i := i + 1
end;
L:
write(product)
```

(e)



(b)



(d)

Figure 3.14: **(a)** Program with unstructured control flow, **(b)** PDG for program of **(a)**, **(c)** incorrect slice w.r.t. statement `write(product)`, **(d)** Augmented PDG for program of **(a)**, **(e)** correct slice w.r.t. statement `write(product)`.

using the ACFG. Slicing is defined in the usual way, as a graph reachability problem. Labels pertaining to statements excluded from the slice are moved to the closest post-dominating statement that occurs in the slice.

The main difference between the approach by Ball and Horwitz and the first approach of Choi and Ferrante is that the latter use a slightly more limited example language: conditional and unconditional **goto**'s are present, but no structured control flow constructs. Although Choi and Ferrante argue that these constructs can be transformed into conditional and unconditional **goto**'s, Ball and Horwitz show that, for certain cases, this results in overly large slices. Both groups present a formal proof that their algorithms compute correct slices.

Figure 3.14 **(d)** shows the augmented PDG for the program of Figure 3.14 **(a)**; vertices from which the vertex labeled `write(product)` can be reached are indicated by shading. The (correct) slice corresponding to these vertices is shown in Figure 3.14 **(e)**.

Choi and Ferrante distinguish two disadvantages of the slicing approach based on augmented PDGs. First, APDGs require more space than conventional PDGs, and their construction takes more time. Second, non-executable control dependence edges give rise to spurious dependences in some cases. In their second approach, Choi and Ferrante utilize the "classical" PDG. As a first approximation, the standard algorithm for computing slices is used, which by itself produces incorrect results in the presence of unstructured control flow. The basic idea is that for each statement that is *not* in the slice, a new **goto** to its immediate post-dominator is added. In a separate phase, redundant cascaded **goto** statements are removed. The second approach has the advantage of computing smaller slices than the first. A disadvantage of it, however, is that slices may include **goto** statements that do not occur in the original program.

Yet another PDG-based method for slicing programs with unstructured control flow was recently proposed by Agrawal [3]. Unlike the methods by Ball and Horwitz [12, 13] and Choi and Ferrante [37], Agrawal uses *unmodified* PDGs. He observes that a *conditional* jump statement of the form **if** `P` **then goto** `L` must be included in the slice if predicate `P` is in the slice because another statement in the slice is control dependent on it. The terminology "conventional slicing algorithm" is adopted to refer to the standard PDG-based slicing method, with the above extension to conditional jump statements.

Agrawal's key observation is that an unconditional jump statement J should be included in the slice if and only if the immediate postdominator of J that is included in the slice differs from the immediate lexical successor of J that is included in the slice. Here, a statement S' is a *lexical successor* of a statement S if S textually precedes S' in the program[12]. The statements on which the newly added statement is transitively dependent must also be added to the slice. The motivation for this approach can be understood by considering a sequence of statements $S_1; S_2; S_3$ where S_1 and S_3 are in the slice, and where S_2 contains an unconditional jump statement to a statement that does not have S_3 as its lexical successor. Suppose that S_2 were not included in the slice. Then the flow of control in the slice would pass unconditionally from S_1 to S_3, though in the original program this need not

[12]As Agrawal observes, this notion is equivalent to the non-executable edges in the augmented control flow graphs used by Ball and Horwitz, and Choi and Ferrante.

always be the case, because the jump might transfer the control elsewhere. Therefore the jump statement must be included, together with all statements it depends upon. Agrawal's algorithm traverses the postdominator tree of a program in pre-order, and considers jump statements for inclusion in this order. The algorithm iterates until no jump statements can be added; this is necessary because adding a jump (and the statements it depend upon) may change the lexical successors and postdominators *in the slice* of other jump statements, which may therefore need to be included as well. Although no proof is stated, Agrawal claims that his algorithm computes correct slices identical to those computed by the Ball-Horwitz and Choi-Ferrante algorithms.

Agrawal's algorithm [3] may be simplified significantly if the only type of jump that occurs in a program is a *structured jump*, i.e., a jump to a lexical successor. C **break**, **continue**, and **return** statements are all structured jumps. First, only a single traversal of the post-dominator tree is required. Second, jump statements have to be added only if they are control dependent on a predicate that is in the slice. In this case, the statements they are dependent upon are already included in the slice. For programs with structured jumps, the algorithm can be further simplified to a *conservative* algorithm by including in the slice all jump statements that are control dependent on a predicate that is in the slice.

Agrawal's algorithm will include the **goto** statement of the example program of Figure 3.14 **(a)** because it is control dependent on the (included) predicate of the **if** statement.

3.3.4 Composite data types and pointers

Lyle [108] proposes a conservative solution to the problem of static slicing in the presence of arrays. Essentially, any update to an element of an array is regarded as an update and a reference of the entire array.

The PDG variant of Ottenstein and Ottenstein [120] contains a vertex for each subexpression; special *select* and *update* operators serve to access elements of an array.

In the presence of pointers (and procedures), situations may occur where two or more variables refer to the same memory location—a phenomenon commonly called *aliasing*. Aliasing complicates the computation of slices because the notion of flow dependence depends on which variables are (potential) aliases. Even in the *intra*procedural case, the problem of determining potential aliases in the presence of multiple-level pointers is an NP-hard problem [105]. However, slices may be computed using conservative approximations of data dependences that are based on approximate alias information. Conservative algorithms for determining potential aliases were presented by Landi and Ryder [104], and Choi, Burke, and Carini [36].

Horwitz, Pfeiffer, and Reps [71] present a slightly different approach for computing flow dependences in the presence of pointers. Instead of defining (approximate) flow dependences in terms of definitions and uses of variables that are potentially aliased, the notion of flow dependence is defined in terms of potential definitions and uses of *abstract memory locations*. An algorithm is presented that computes approximations of the memory layouts that may occur at each program point during execution.

The PDG-based static slicing algorithm proposed by Agrawal, DeMillo and Spafford [4] implements a similar idea to deal with both composite variables and pointers. Their solution consists of determining reaching definitions for a scalar variable v at node n in the flowgraph by finding all paths from nodes corresponding to a definition of v to n that do not contain other definitions of v. When composite data types and pointers are considered, definitions involve *l-valued expressions* rather than variables. An l-valued expression is any expression that may occur as the left-hand side of an assignment. Agrawal et al. present a new definition of reaching definitions that is based on the layout of memory locations potentially denoted by l-valued expressions. Memory locations are regarded as abstract quantities (e.g., the array a corresponds to "locations" $a[1]$, $a[2]$,$\cdots$). Whereas a definition for a scalar variable either does or does not reach a use, the situation becomes more complex when composite data types and pointers are allowed. For a def-expression e_1 and a use-expression e_2, the following situations may occur:

Complete Intersection The memory locations corresponding to e_1 are a superset of the memory locations corresponding to e_2. An example is the case where e_1 defines the whole of record b, and e_2 is a use of $b.f$.

Maybe Intersection It cannot be determined statically whether or not the memory locations of a e_1 coincide with those of e_2. This situation occurs when e_1 is an assignment to array element $a[i]$ and e_2 is a use of array element $a[j]$. Pointer dereferencing may also give rise to Maybe Intersections.

Partial Intersection The memory locations of e_1 are a subset of the memory locations of e_2. This occurs for example when some array a is used at e_2, and some element $a[i]$ of a is defined at e_1.

Given these concepts, an extended reaching definition function is defined that traverses the flowgraph until it finds Complete Intersections, makes worst-case assumptions in the case of Maybe Intersections, and continues the search for the array or record parts that have not been defined yet in the case of Partial Intersections.

Lyle and Binkley [110] present an approach for slicing in the presence of pointers that is based on a variation of symbolic execution. Their algorithm consists of two phases. First, all CFG nodes are determined that introduce addresses (either due to a use of the C '&' operator, or due to the dynamic allocation of a new object). These addresses are propagated through the CFG yielding a set of address values for each pointer at each program point. A number of propagation rules defines how addresses are propagated by assignments statements[13]. In the second phase, the information collected in the first phase is used to determine which statements should be included in a slice. This second phase is essentially a generalization of Lyle's slicing algorithm [108].

Jiang, Zhou and Robson [83] present an algorithm for slicing C programs with pointers and arrays that is based on Weihl's notion of *dummy variables* [142]. The basic idea is that for each pointer p, the dummy variable $(1)p$ denotes the value pointed to by p, and for each variable x, $(-1)x$ denotes the address of q. Jiang et al. define data dependences in the usual way, in terms of definitions and uses of (dummy) variables. Unfortunately, this approach

[13]In their definitions, Lyle and Binkley only address straight-line code, and argue that control-dependence issues are "orthogonal" to the data-dependence issues raised by pointers.

```
(1) p = &x;
(2) *p = 2;
(3) q = p;
(4) write(*q)
```

(a)

#	DEF	REF	R_C^0
1	{p}	{(-1)x}	$\emptyset$
2	{(1)p}	{p}	{p, (1)q}
3	{q}	{p}	{p, (1)q}
4	$\emptyset$	{q, (1)q}	{q, (1)q}

(b)

```
(1) p = &x;
(2)       ;
(3) q = p;
(4)
```

(c)

Figure 3.15: **(a)** Example program. **(b)** Defined variables, used variables, and relevant variables for this program. **(c)** Incorrect slice w.r.t. criterion $C = \langle 4, \{$ q, (1)q $\}\rangle$.

appears to be flawed[14]. Figure 3.15 shows an example program, the DEF, REF, and R_C^0 sets for each statement, and the incorrect slice computed for criterion $C = \langle 4, \{$ q, (1)q $\}\rangle$. The second statement is incorrectly omitted because it does not define any variable that is relevant at statement 3.

3.3.5 Concurrency

Cheng [35] considers static slicing of concurrent programs using dependence graphs. He generalizes the notions of a CFG and a PDG to a *nondeterministic parallel control flow net*, and a *program dependence net* (PDN), respectively. In addition to usual PDG edges, PDNs also contain edges for selection dependences, synchronization dependences, and communication dependences. *Selection* dependence is similar to control dependence but involves nondeterministic selection statements, such as the ALT statement of Occam-2. *Synchronization* dependence reflects the fact that the start or termination of the execution of a statement depends on the start or termination of the execution of another statement. *Communication* dependence corresponds to situations where a value computed at one program point influences the value computed at another point through interprocess communication. Static slices are computed by solving a reachability problem in a PDN. Unfortunately, Cheng does not clearly state the semantics of synchronization and communication dependence, nor does he state or prove any property of the slices computed by his algorithm.

An interesting point is that Cheng uses a notion of *weak* control dependence [123] for the construction of PDNs. This notion subsumes the standard notion of control dependence; the difference is that weak control dependences exist between the control predicate of a loop, and the statements that follows it. For example, the statements on lines 9 and 10 of the program of Figure 3.1 **(a)** are weakly (but not strongly) control dependent on the control predicate on line 5.

[14]The counterexample of Figure 3.15 was provided by Susan Horwitz.

3.3.6 Comparison

3.3.6.1 Overview

In this section, the static slicing methods that were presented earlier are compared and classified. The section is organized as follows: Section 3.3.6.1 summarizes the problems that are addressed in the literature. Sections 3.3.6.2 and 3.3.6.3 compare the *accuracy* and *efficiency* of slicing methods that address the same problem, respectively. Finally, in Section 3.3.6.4 possibilities for combining algorithms that deal with different problems are discussed.

Table 3.2 provides an overview of the most significant slicing algorithms that can be found in the literature. For each paper, the table lists the computation method used and indicates: (i) whether or not interprocedural slices can be computed, (ii) the control flow constructs under consideration, (iii) the data types under consideration, and (iv) whether or not concurrency is considered. It is important to realize that the entries of Table 3.2 only indicate the *problems that have been addressed*; the table does *not* indicate the "quality" of the solutions (with the exception that incorrect solutions are indicated by footnotes). Moreover, the table also does *not* indicate which algorithms may be combined. For example, the Horwitz-Reps-Binkley interprocedural slicing algorithm [77] could in principle be combined with any of the dependence graph based slicing methods for dealing with unstructured control flow [13, 3, 37]. Possibilities for such combinations are investigated to some extent in Section 3.3.6.4. The work by Ernst [52] and by Field et al. (see Chapters 4 and 5) that occurs in Table 3.2 relies on substantially different techniques than those used for the static slicing algorithms discussed previously, and will therefore be studied separately in Section 3.6.

Kamkar [86] distinguishes between methods for computing slices that are executable programs, and those for computing slices that consist of a set of "relevant" statements. We agree with the observation by Horwitz et al. [77], that for *static* slicing of *single-procedure* programs this is merely a matter of presentation. However, for multi-procedure programs, the distinction *is* significant, as was remarked in Section 3.3.2.3. Nevertheless, we believe that the distinction between executable and non-executable interprocedural slices can be ignored in this case as well, because the problems are strongly related: Binkley [30] describes how precise executable interprocedural static slices can be obtained from the non-executable interprocedural slices computed by the algorithm of Horwitz et al. [77].

A final remark here concerns I/O statements. The slices computed by Weiser's algorithm [147] and the algorithm by Bergeretti and Carré [22] never contain output statements because: (i) the DEF set of an output statement is empty so that no other statement is data dependent on it, and (ii) no statement is control dependent on an output statement. Horwitz and Reps [76] suggest a way for making an output value dependent on all previous output values by treating a statement `write(v)` as an assignment `output := output || v`, where `output` is a string-valued variable containing all output of the program, and '`||`' denotes string concatenation. Output statements can be included in the slice by including `output` in the set of variables in the criterion.

	Computation Method [a]	Interprocedural Solution	Control Flow [b]	Data Types [c]	Concurrency
Weiser [147, 107]	D	yes	S	S	no
Lyle [108]	D	no	A	S, A	no
Gallagher, Lyle [60, 61]	D	no	A[d]	S	no
Jiang et al. [83]	D	yes	A[d]	S, A, P[e]	no
Lyle, Binkley [110]	D	no	S[f]	S, P	no
Hausler [67]	F	no	S	S	no
Bergeretti, Carré [22]	I	yes[g]	S	S	no
Ottenstein [120]	G	no	S	S, A	no
Horwitz et al. [74, 130, 75]	G	no	S	S	no
Horwitz et al. [77]	G	yes	S	S	no
Binkley [31]	G	yes[h]	S	S	no
Binkley [32]	G	yes[i]	S	S	no
Jackson, Rollins [82, 81]	G	yes	S	S	no
Reps et al. [129, 128]	G	yes	S	S	no
Lakhotia [103]	G	yes	S	S	no
Agrawal et al. [4]	G	no	S	S, A, P	no
Ball, Horwitz [12, 13]	G	no	A	S	no
Choi, Ferrante [37]	G	no	A	S	no
Agrawal [3]	G	no	A	S	no
Cheng [35]	G	no	S	S	yes
Ernst [52]	O	yes	A	S, A, P	no
Field et al. (Chap. 5)	R	no	S	S, P	no

[a] D = dataflow equations, F = functional/denotational semantics, I = information-flow relations, G = reachability in a dependence graph, O = dependence graphs in combination with optimization techniques (see Section 3.6). R = dependence tracking in term graph rewriting systems (see Section 3.6).
[b] S = structured, A = arbitrary.
[c] S = scalar variables, A = arrays/records, P = pointers.
[d] Solution incorrect (see [3]).
[e] Solution incorrect (see Section 3.3.4).
[f] Only straight-line code is considered.
[g] Non-recursive procedures only.
[h] Takes parameter aliasing into account.
[i] Produces slices that are executable programs.

Table 3.2: Overview of static slicing methods.

3.3.6.2 Accuracy

An issue that complicates the comparison of the static slicing methods discussed previously is the fact that some methods allow more general slicing criteria than others. For slicing methods based on dataflow equations and information-flow relations, a slicing criterion consists of a pair $\langle s, V \rangle$, where s is a statement and V an arbitrary set of variables. In contrast, with the exception of the "modular" PDGs of Jackson and Rollins [82], the slicing criteria of PDG-based slicing methods effectively correspond to a pair $\langle s, \textsc{Vars}(s) \rangle$, where s is a statement and $\textsc{Vars}(s)$ the set of *all* variables *defined or used at* s.

However, a PDG-based slicing method can compute a slice with respect to a criterion $\langle s, V \rangle$ for arbitrary V by performing the following three steps. First, the CFG node n corresponding to PDG vertex s is determined. Second, the set of CFG nodes N corresponding to all definitions that reach a variable in V at node n are determined. Third, the set of PDG vertices S corresponding to the set of CFG nodes N is determined; the desired slice consists of all vertices from which a vertex in S can be reached. Alternatively, one could insert a statement $v{:=}e$ at the point of interest, where v is some dummy variable that did not occur previously in the program, and e is some expression containing all variables in V, re-construct the PDG, and slice with respect to the newly added statement. Having dealt with this issue, some conclusions regarding the accuracy of static slicing methods can now be stated:

basic algorithms. For *intra*procedural static slicing, the accuracy of methods based on dataflow equations [147] (see Section 3.3.1.1) information-flow relations [22] (see Section 3.3.1.2), and PDGs [120] (see Section 3.3.1.3) is essentially the same, although the presentation of the computed slices differs: Weiser defines his slice to be an executable program, whereas in the other two methods slices are defined as a subset of statements of the original program.

procedures. Weiser's *inter*procedural static slicing algorithm [147] is inaccurate for two reasons, which can be summarized as follows. First, the interprocedural summary information used to approximate the effect of a procedure call establishes relations between the set of *all* input parameters, and the set of *all* output parameters; by contrast, the approaches of [22, 77, 79, 129, 128] determine for each output parameter precisely which input parameters it depends upon. Second, the algorithm fails to take the call-return structure of interprocedural execution paths into account. These problems are addressed in detail in Section 3.3.2.1.

The algorithm by Bergeretti and Carré [22] does not compute truly interprocedural slices because only the main program is being sliced. Moreover, the it is not capable of handling recursive programs. Bergeretti-Carré slices are accurate in the sense that: (i) exact dependences between input and output parameters are used, and (ii) the calling-context problem does not occur.

The solutions of [79, 77, 129, 128] compute accurate interprocedural static slices, and are capable of handling recursive programs (see Sections 3.3.2.2 and 3.3.2.3). Ernst [52] and

Jackson and Rollins [82] also present a solution for accurate interprocedural static slicing, but do not present of proof of correctness.

Binkley extended the Horwitz-Reps-Binkley algorithm [77] in two respects: a solution for interprocedural static slicing in the presence of parameter aliasing [31], and a solution for obtaining executable interprocedural static slices [30].

unstructured control flow. Lyle's method for computing static slices in the presence of unstructured control flow is very conservative (see Section 3.3.3.1). Agrawal [3] has shown that the solutions proposed by Gallagher and Lyle [60, 61] and by Jiang et al. are incorrect [83]. Precise solutions for static slicing in the presence of unstructured control flow have been proposed by Ball and Horwitz [12, 13], Choi and Ferrante [37], and Agrawal [3] (see Section 3.3.3.2). It is our conjecture that these three approaches are equally accurate.

composite variables and pointers. A number of solutions for slicing in the presence of composite variables and pointers were discussed in Section 3.3.4. Lyle [108] presented a very conservative algorithm for static slicing in the presence of arrays. The approach by Jiang et al. [83] for slicing in the presence of arrays and pointers was shown to be incorrect. Lyle and Binkley [110] present an approach for computing accurate slices in the presence of pointers, but only consider straight-line code. Agrawal et al. propose an algorithm for static slicing in the presence of arrays and pointers that is more accurate than Lyle's algorithm [108].

concurrency. The only approach for static slicing of concurrent programs was proposed by Cheng (see Section 3.3.5).

3.3.6.3 Efficiency

Below, the efficiency of the static slicing methods that were studied earlier will be addressed:

basic algorithms. Weiser's algorithm for *intra*procedural static slicing based on dataflow equations [147] can determine a slice in $O(v \times n \times e)$ time[15], where v is the number of variables in the program, n the number of vertices in the CFG, and e the number of edges in the CFG.

Bergeretti and Carré [22] report that the μ_S relation for a statement S can be computed in $O(v^2 \times n)$. From μ_S, the slices for all variables at S can be obtained in constant time.

Construction of a PDG essentially involves computing all data dependences and control

[15]Weiser [147] states a bound of $O(n \times e \times \log(e))$. However, this is a bound on the number of "bit-vector" steps performed, where the length of each bit-vector is $O(v)$. We have multiplied the cost by $O(v)$ to account for the cost of such bit-vector operations. The problem of determining relevant variables is similar to that of determining possibly-uninitialized variables. Using the transformation technique of Reps et al. [129] this information can be computed in $O(v \times e)$ time. The process of determining relevant variables is repeated at most $O(n)$ times due to the inclusion in the slice of branch statements with indirect relevance. Hence, an improved bound for Weiser's algorithm is $O(v \times n \times e)$.

dependences in a program. For structured programs, control dependences can be determined in a syntax-directed fashion, in $O(n)$. In the presence of unstructured control flow, the control dependences of a single-procedure program can be computed in $O(e)$ in practice [40, 84]. Computing data dependences essentially corresponds to determining the reaching definitions for each use. For scalar variables, this can be accomplished in $O(e \times d)$, where d is the number of definitions in the program (see, e.g., [129]). From $d \leq n$ it follows that a PDG can be constructed in $O(e \times n)$ time.

One of the self-evident advantages of PDG-based slicing methods is that, once the PDG has been computed, slices can be extracted in linear time, $O(V + E)$, where V and E are the number of vertices and edges in the *slice*, respectively. This is especially useful if several slices of the same program are required. In the worst case, when the slice consists of the entire program, V and E are equal to the number of vertices and edges of the PDG, respectively. In certain cases, there can be a quadratic blowup in the number of flow dependence edges of a PDG, e.g., $E = O(V^2)$. We are not aware of any slicing algorithms that use more efficient program representations such as the SSA form [8]. However, Yang et al. [149] use Program Representation Graphs as a basis for a program integration algorithm that accommodates semantics-preserving transformations. This algorithm is based on techniques similar to slicing.

procedures. In the discussion below, *Visible* denotes the maximal number of parameters and variables that are visible in the scope of any procedure, and *Params* denotes the maximum number of formal-in vertices in any procedure dependence graph of the SDG. Moreover, *TotalSites* is the total number of call sites in the program; N_p and E_p denote the number of vertices and edges in the CFG of procedure p, and *Sites*$_p$ the number of call sites in procedure p.

Weiser does not state an estimate of the complexity of his interprocedural slicing algorithm [147]. However, one can observe that for an initial criterion C, the set of criteria in (UP ∪ DOWN)*(C) contains at most $O(\mathit{Visible})$ criteria in each procedure p. An intraprocedural slice of procedure p takes time $O(\mathit{Visible} \times N_p \times E_p)$. Furthermore, computation of interprocedural summary information can be done in $O(\mathit{Globals} \times \mathit{TotalSites})$ time [39]. Therefore, the following expression constitutes an upper bound for the time required to slice the entire program:

$$O(\mathit{Globals} \times \mathit{TotalSites} + \mathit{Visible}^2 \times \Sigma_p(\mathit{Sites}_p \times N_p \times E_p))$$

The approach by Bergeretti and Carré requires that, in the worst case, the μ relation is computed for each procedure. Each call site is replaced by at most *Visible* assignments. Therefore, the cost of slicing a procedure p is $O(\mathit{Visible}^2 \times (n + \mathit{Visible} \times \mathit{Sites}_p))$, and the total cost of computing a slice of the main program is:

$$O(\mathit{Visible}^2 \times \Sigma_p(n + \mathit{Visible} \times \mathit{Sites}_p))$$

As was discussed in Section 3.3.2.1, the approach by Hwang, Du, and Chou may require time exponential in the size of the program.

Construction of the individual procedure dependence graphs of an SDG takes time $O(\Sigma_p(E_p \times N_p))$. The Horwitz-Reps-Binkley algorithm for computing summary edges takes time:

$$O(\mathit{TotalSites} \times E^{\mathrm{PDG}} \times \mathit{Params} + \mathit{TotalSites} \times \mathit{Sites}^2 \times \mathit{Params}^4)$$

where *Sites* is the maximum number of call sites in any procedure, and E^{PDG} is the maximum number of control and data dependence edges in any procedure dependence graph. (for details, see [77, 128]). The Reps-Horwitz-Sagiv-Rosay approach for computing summary edges requires

$$O(P \times E^{\mathrm{PDG}} \times \mathit{Params} + \mathit{TotalSites} \times \mathit{Params}^3)$$

time [128]. Here, P denotes the number of procedures in the program. Assuming that the number of procedures P is usually much less than the number of procedure calls *TotalSites*, both terms of the complexity measure of the Reps-Horwitz-Sagiv-Rosay approach are asymptotically smaller than those of the Horwitz-Reps-Binkley algorithm.

Once an SDG has been constructed, a slice can be extracted from it (in two passes) in $O(V + E)$, where V and E are the number of vertices and edges in the slice, respectively. In the worst case, V and E are the number of vertices and edges in the SDG, respectively.

Binkley does not state a cost estimate of his algorithm for interprocedural slicing in the presence of parameter aliasing [31]. The cost of his "extension" for deriving executable interprocedural slices [30] from "non-executable" interprocedural slices is linear in the size of the SDG.

Jackson and Rollins [82], who use an adaptation of Ernst's algorithm for determining summary dependences (see Section 3.3.2.3) claim a bound of $O(v \times n^2)$, where v denotes the number of variables, and n the number of ports in the dependence graph. Observe that each port is effectively a pair $\langle$variable, statement$\rangle$). In the approach by Jackson and Rollins, extraction of a slice is done in a single traversal of the dependence graph, which requires $O(V + E)$ time, where V and E denote the number of vertices (i.e., ports) and edges in the slice.

unstructured control flow. Lyle [108] presented a conservative solution for dealing with unstructured control flow. His algorithm is a slightly modified version of Weiser's algorithm for *structured* control flow [147], which requires $O(v \times n \times e)$ time.

No complexity estimates are stated in [3, 13, 37]. However, the differences between these algorithms and the "standard" PDG-based slicing algorithm are only minor: Ball and Horwitz [13] and Choi and Ferrante [37] use a slightly different control dependence subgraph in conjunction with the data dependence subgraph, and Agrawal [3] uses the standard PDG in conjunction with a lexical successor tree that can be constructed in linear time, $O(n)$. Therefore it is to be expected that the efficiency of these algorithms is roughly equivalent to that of the standard, PDG-based algorithm discussed above.

composite variables and pointers. Lyle's approach for slicing in the presence of arrays [108] has the same complexity bound as Weiser's solution [147] for scalar variables only,

	Procedures	Unstructured Control Flow	Composite Variables	Concurrency
Dataflow Equations	Weiser [147, 107]	Lyle [108]	Lyle [108]	—
Inf.-Flow Relations	Bergeretti, Carré [22]	—	—	—
PDG-based	Horwitz et al. [77] Lakhotia [102] Reps et al. [129, 128] Binkley [30]	Ball, Horwitz [12, 13] Choi, Ferrante [37] Agrawal [3]	Agrawal et al.[4][a]	Cheng [35]

[a] Algorithms for computing conservative approximations of data dependences in the presence of aliasing can be used. See Section 3.3.4.

Table 3.3: Orthogonal dimensions of static slicing.

because the worst-case length of reaching definitions paths remains the same.

The cost of constructing PDGs of programs with composite variables and pointers according to the algorithm proposed by Agrawal et al. [4] is the same as that of constructing PDGs of programs with scalar variables only. This is the case because the worst-case length of (potential) reaching definitions paths remains the same, and determining Maybe Intersections and Partial Intersections (see Section 3.3.4) can be done in constant time.

Lyle and Binkley do not state a cost estimate for their approach for slicing in the presence of pointers [110].

It should be remarked here that more accurate static slices can be determined in the presence of non-scalar variables if more advanced (but computationally expensive) data dependence analysis were performed (see, e.g., [113, 150]).

concurrency. Cheng [35] does not state any complexity estimate for determining selection, synchronization, and communication dependence. The time required for extracting slices is $O(V + E)$, where V and E denote the number of vertices and edges in the PDN, respectively.

3.3.6.4 Combining static slicing algorithms

Table 3.3 highlights "orthogonal" dimensions of static slicing: dealing with procedures, unstructured control flow, non-scalar variables, and concurrency. For each computation method, the table shows which papers present a solution for these problems. In principle, solutions to different problems could be combined if they appear in the same row of Table 3.3

1^{1} `read(n)`
2^{2} `i := 1`
3^{3} `i <= n` `/* 1 <= 2 /*`
4^{4} `(i` **mod** `2 = 0)` `/* 1` **mod** `2 = 1 /*`
6^{5} `x := 18`
7^{6} `i := i + 1`
3^{7} `i <= n` `/* 2 <= 2 /*`
4^{8} `(i` **mod** `2 = 0)` `/* 2` **mod** `2 = 0 /*`
5^{9} `x := 17`
7^{10} `i := i + 1`
3^{11} `i <= n` `/* 3 <= 2 /*`
8^{12} `write(x)`

(a)

$$\begin{array}{lcl}
\mathrm{DU} & = & \{ \; \langle 1^1,3^3\rangle,\ \langle 1^1,3^7\rangle,\ \langle 1^1,3^{11}\rangle, \\
& & \quad \langle 2^2,3^3\rangle,\ \langle 2^2,4^4\rangle,\ \langle 2^2,7^6\rangle, \\
& & \quad \langle 7^6,3^7\rangle,\ \langle 7^6,4^8\rangle,\ \langle 7^6,7^{10}\rangle, \\
& & \quad \langle 5^9,8^{12}\rangle,\ \langle 7^{10},3^{11}\rangle \quad \} \\
\\
\mathrm{TC} & = & \{ \; \langle 3^3,4^4\rangle,\ \langle 3^3,6^5\rangle,\ \langle 3^3,7^6\rangle, \\
& & \quad \langle 4^4,6^5\rangle,\ \langle 3^7,4^8\rangle,\ \langle 3^7,5^9\rangle, \\
& & \quad \langle 3^7,7^{10}\rangle,\ \langle 4^8,5^9\rangle \quad \} \\
\\
\mathrm{IR} & = & \{ \; \langle 3^3,3^7\rangle,\ \langle 3^3,3^{11}\rangle,\ \langle 3^7,3^3\rangle, \\
& & \quad \langle 3^7,3^{11}\rangle,\ \langle 3^{11},3^3\rangle,\ \langle 3^{11},3^7\rangle, \\
& & \quad \langle 4^4,4^8\rangle,\ \langle 4^8,4^4\rangle,\ \langle 7^6,7^{10}\rangle, \\
& & \quad \langle 7^{10},7^6\rangle \quad \}
\end{array}$$

(b)

Figure 3.16: **(a)** Trajectory for the example program of Figure 3.2 **(a)** for input `n = 2`. **(b)** Dynamic Flow Concepts for this trajectory.

(i.e., if they apply to the same computation method).

3.4 Methods for dynamic slicing

3.4.1 Basic algorithms

In this section, we will study basic algorithms for dynamic slicing of structured, single-procedure programs with scalar variables.

3.4.1.1 Dynamic flow concepts

Korel and Laski [99, 100] describe how dynamic slices can be computed. They formalize the execution history of a program as a *trajectory* consisting of a sequence of "occurrences" of statements and control predicates. Labels serve to distinguish between different occurrences of a statement in the execution history. As an example, Figure 3.16 shows the trajectory for the program of Figure 3.2 **(a)** for input `n = 2`.

A *dynamic slicing criterion* is specified as a triple $\langle x, I^q, V\rangle$ where x denotes the input of the program, statement occurrence I^q is the q^{th} element of the trajectory, and V is a subset of the variables of the program[16]. Korel and Laski define a *dynamic slice* with respect

[16]Korel and Laski's definition of a dynamic slicing criterion is somewhat inconsistent. It assumes that a trajectory is available although the input x uniquely defines this. A self-contained and minimal definition of a dynamic slicing criterion would consist of a triple $\langle x, q, V\rangle$ where q is the number of a statement occurrence in the trajectory induced by input x.

to a criterion $\langle x, I^q, V \rangle$ as an *executable* program S that is obtained from a program P by removing zero or more statements. Three restrictions are imposed on S. First, when executed with input x, the trajectory of S is identical to the trajectory of P from which all statement instances are removed that correspond to statements that do not occur in S. Second, identical values are computed by the program and its slice for all variables in V at the point specified in the criterion. Third, it is required that statement I corresponding to statement instance I^q specified in the slicing criterion occurs in S. Korel and Laski observe that their notion of a dynamic slice has the property that if a loop occurs in the slice, it is traversed the same number of times as in the original program.

In order to compute dynamic slices, Korel and Laski introduce three *dynamic flow concepts* that formalize the dependences between occurrences of statements in a trajectory. The *Definition-Use* (DU) relation associates a use of a variable with its last definition. Note that in a trajectory, this definition is uniquely defined. The *Test-Control* (TC) relation associates the most recent occurrence of a control predicate with the statement occurrences in the trajectory that are control dependent upon it. This relation is defined in a syntax-directed manner, for structured program constructs only. Occurrences of the same statement are related by the symmetric *Identity* (IR) relation. Figure 3.16 **(b)** shows the dynamic flow concepts for the trajectory of Figure 3.16 **(a)**.

Dynamic slices are computed in an iterative way, by determining successive sets S^i of directly and indirectly relevant statements. For a slicing criterion $\langle x, I^q, V \rangle$, the initial approximation S^0 contains the last definitions of the variables in V in the trajectory before statement instance I^q, as well as the test actions in the trajectory on which I^q is control dependent. Approximation S^{i+1} is defined as follows:

$$S^{i+1} = S^i \cup A^{i+1}$$

where A^{i+1} is defined as follows:

$$A^{i+1} = \{\, X^p \mid X^p \notin S^i,\ \langle X^p, Y^t \rangle \in (\mathrm{DU} \cup \mathrm{TC} \cup \mathrm{IR}) \text{ for some } Y^t \in S^i,\ p < q \,\}$$

where q is the "label" of the statement occurrence specified in the slicing criterion. The dynamic slice is easily obtained from the fixpoint S_C of this process (as q is finite, this always exists): any statement X for which an instance X^p occurs in S_C will be in the slice. Furthermore, statement I corresponding to criterion I^q is added to the slice.

As an example, the dynamic slice for the trajectory of Figure 3.16 and the criterion $\langle n = 2, 8^{12}, \{\, x \,\} \rangle$ is computed. Since the final statement is not control dependent on any other statement, the initial approximation of the slice consists of the last definition of x: $A^0 = \{\, 5^9 \,\}$. Subsequent iterations will produce $A^1 = \{\, 3^7, 4^8 \,\}$, $A^2 = \{\, 7^6, 1^1, 3^3, 3^{11}, 4^4 \,\}$, and $A^3 = \{\, 2^2, 7^{10} \,\}$. From this, it follows that:

$$S_C = \{\, 1^1,\ 2^2,\ 3^3,\ 4^4,\ 7^6,\ 3^7,\ 4^8,\ 5^9,\ 7^{10},\ 3^{11},\ 8^{12} \,\}$$

Thus, the dynamic slice with respect to criterion $\langle n = 2, 8^{12}, \{\, x \,\} \rangle$ includes every statement except statement 5, corresponding to statement 6^5 in the trajectory. This slice was shown earlier in Figure 3.2 **(b)**.

1^1 `read(n)`
2^2 `i := 1`
3^3 `i <= n`
4^4 `(i` **mod** `2 = 0)`
6^5 `x := 18`
7^6 `i := i + 1`
3^7 `i <= n`
8^8 `write(x)`

(a)

$$\begin{array}{lcll}
\text{DU} & = & \{ & \langle 1^1, 3^3\rangle,\ \langle 1^1, 3^7\rangle, \\
 & & & \langle 2^2, 3^3\rangle,\ \langle 2^2, 4^4\rangle, \\
 & & & \langle 2^2, 7^6\rangle,\ \langle 6^5, 8^8\rangle, \\
 & & & \langle 7^6, 3^7\rangle \quad \} \\
 & & & \\
\text{TC} & = & \{ & \langle 3^3, 4^4\rangle,\ \langle 3^3, 6^5\rangle, \\
 & & & \langle 3^3, 7^6\rangle,\ \langle 4^4, 6^5\rangle \quad \} \\
 & & & \\
\text{IR} & = & \{ & \langle 3^3, 3^7\rangle,\ \langle 3^7, 3^3\rangle \quad \}
\end{array}$$

(b)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
          ;
  i := i + 1
end;
write(x)
```

(c)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
          ;

end;
write(x)
```

(d)

Figure 3.17: **(a)** Trajectory of the example program of Figure 3.2 **(a)** for input n = 1. **(b)** Dynamic flow concepts for this trajectory. **(c)** Dynamic slice for criterion $\langle n = 1, 8^8, x\rangle$. **(d)** Non-terminating slice with respect to the same criterion obtained by ignoring the effect of the IR relation.

The role of the IR relation calls for some clarification. Consider the trajectory of the example program of Figure 3.2 **(a)** for input n = 1, shown in Figure 3.17 **(a)**. The dynamic flow concepts for this trajectory, and the slice with respect to criterion $\langle n = 1, 8^8, \{\ x\ \}\rangle$ are shown in Figure 3.17 **(b)** and **(c)**, respectively. Note that the slice thus obtained is a terminating program. However, computing the slice without taking the IR relation into account would yield the non-terminating program of Figure 3.17 **(d)**. The reason for this phenomenon (and thus for introducing the IR relation) is that the DU and TC relations only traverse the trajectory in the *backward* direction. The purpose of the IR relation is to traverse the trajectory in *both* directions, and to include all statements and control predicates that are necessary to ensure termination of loops in the slice. Unfortunately, no proof is provided that this is always sufficient.

Unfortunately, traversing the IR relation in the "backward" direction causes inclusion of statements that are not necessary to preserve termination. For example, Figure 3.18 **(a)** shows a slightly modified version of the program of Figure 3.2 **(a)**. Figure 3.18 **(b)**

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     z := x;
(8)     i := i + 1
      end;
(9)   write(z)
```

(a)

1^{1}	`read(n)`
2^{2}	`i := 1`
3^{3}	`i <= n`
4^{4}	`(i mod 2 = 0)`
6^{5}	`x := 18`
7^{6}	`z := x`
8^{7}	`i := i + 1`
3^{8}	`i <= n`
4^{9}	`(i mod 2 = 0)`
5^{10}	`x := 17`
7^{11}	`z := x`
8^{12}	`i := i + 1`
3^{13}	`i <= n`
9^{14}	`write(z)`

(b)

Figure 3.18: **(a)** Example program. **(b)** Trajectory for input n = 2.

shows the trajectory for this program. From this trajectory, it follows that $\langle 7^6, 7^{11} \rangle \in$ IR, $\langle 6^5, 7^6 \rangle \in$ DU, and $\langle 5^{10}, 7^{11} \rangle \in$ DU. Therefore, both statements (5) and (6) will be included in the slice, although statement (6) is neither needed to compute the final value of `z` nor to preserve termination.

We conjecture that restricting the IR relation to statement instances that correspond to control predicates in the program would yield smaller slices. Alternatively, it would be interesting to investigate if using a dynamic variation of Podgurski and Clarke's notion of weak control dependence [123] could be used instead of the IR relation.

3.4.1.2 Dynamic dependence relations

Gopal [62] describes an approach where *dynamic dependence relations* are used to compute dynamic slices. He introduces dynamic versions of Bergeretti and Carré's information-flow relations[17] λ_S, μ_S, and ρ_S (see Section 3.3.1.2). The $\overline{\lambda}_S$ relation contains all pairs $\langle v, e \rangle$ such that statement e depends on the input value of v when program S is executed. Relation $\overline{\mu}_S$ contains all pairs $\langle e, v \rangle$ such that the output value of v depends on the execution of statement e. A pair $\langle v, v' \rangle$ is in relation $\overline{\rho}_S$ if the output value of v' depends on the input value of v. In these definitions, it is presumed that S is executed for some fixed input.

For empty statements, assignments, and statement sequences Gopal's dependence relations are exactly the same as for the static case. The (static) information-flow relations for a conditional statement are derived from the statement itself, and from the statements that constitute its branches. For *dynamic* dependence relations, however, only the dependences that arise in the branch that is actually executed are taken into account. As in [22], the

[17]Gopal uses the notation s_v^S, v_v^S, and v_s^S. In order to avoid confusion and to make the relation with Bergeretti and Carré's work explicit (see Section 3.3.1.2), we will use $\overline{\lambda}_S$, $\overline{\mu}_S$, and $\overline{\rho}_S$ instead.

$$\begin{array}{lcl|lcl|lcl}
\overline{\lambda}_{\epsilon} & = & \emptyset & \overline{\lambda}_{S_1;S_2} & = & \overline{\lambda}_{S_1} \cup \overline{\rho}_{S_1} \cdot \overline{\lambda}_{S_2} & \overline{\lambda}_{v:=e} & = & \textsc{Vars}(e) \times \{\, e \,\} \\
\overline{\mu}_{\epsilon} & = & \emptyset & \overline{\mu}_{S_1;S_2} & = & \overline{\mu}_{S_1} \cdot \overline{\rho}_{S_2} \cup \overline{\mu}_{S_2} & \overline{\mu}_{v:=e} & = & \{\, \langle e, v \rangle \,\} \\
\overline{\rho}_{\epsilon} & = & \textsc{Id} & \overline{\rho}_{S_1;S_2} & = & \overline{\rho}_{S_1} \cdot \overline{\rho}_{S_2} & \overline{\rho}_{v:=e} & = & (\textsc{Vars}(e) \times \{\, v \,\}) \cup (\textsc{Id} - \langle v, v \rangle)
\end{array}$$

$$\overline{\lambda}_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} = \begin{cases} (\textsc{Vars}(e) \times \{\, e \,\}) \cup \overline{\lambda}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\textsc{Vars}(e) \times \{\, e \,\}) \cup \overline{\lambda}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

$$\overline{\mu}_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} = \begin{cases} (\{\, e \,\} \times \textsc{Defs}(S_1)) \cup \overline{\mu}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\{\, e \,\} \times \textsc{Defs}(S_2)) \cup \overline{\mu}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

$$\overline{\rho}_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} = \begin{cases} (\textsc{Vars}(e) \times \textsc{Defs}(S_1)) \cup \overline{\rho}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\textsc{Vars}(e) \times \textsc{Defs}(S_2)) \cup \overline{\rho}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

Figure 3.19: Definition of dynamic dependence relations.

EXPRESSION #[a]	AFFECTED VARIABLES
1	{ i, n, x }
2	{ i, x }
3	{ i, x }
4	{ i, x }
5	{ x }
6	$\emptyset$
7	{ i, x }
8	$\emptyset$

[a]Expressions are indicated by the line numbers in Figure 3.2.

Figure 3.20: The $\overline{\mu}$ relation for the example program of Figure 3.2 **(a)** and input `n = 2`.

dependence relation for a **while** statement (omitted here) is expressed in terms of dependence relations for nested conditionals with equivalent behavior. However, whereas in the static case loops are effectively replaced by their infinite unwindings, the dynamic case only requires that a loop be unwound k times, where k is the number of times the loop executes. The resulting definitions are very convoluted because the dependence relations for the body of the loop may differ in each iteration. Hence, a simple transitive closure operation, as was used in the static case, is insufficient.

Figure 3.19 summarizes Gopal's dynamic dependence relations. Here, $\textsc{Defs}(S)$ denotes the set of variables that is actually modified by executing statement S. Note that this definition of DEFS is "dynamic" in the sense that it takes into account which branch of an **if** statement is executed. Using these relations, a dynamic slice w.r.t. the final value of a variable v is defined as:

$$\sigma_v^P \equiv \{e \mid \langle e, v \rangle \in \overline{\mu}_P\}$$

Figure 3.20 **(a)** shows the information-flow relation $\overline{\mu}$ for the (entire) program of Fig-

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then

  else
    x := 18;

end;
```

(a)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
           ;
  z := x;
  i := i + 1
end;
```

(b)

Figure 3.21: **(a)** Non-terminating slice computed for example program of Figure 3.2 **(a)** with respect to the final value of `x`, for input `n = 1`. **(b)** Slice for the example program of Figure 3.18 **(a)** with respect to the final value of `x`, for input `n = 2`.

ure 3.2 **(a)**[18]. From this relation it follows that the set of expressions that affect the value of `x` at the end of the program for input `n = 2` are $\{ 1, 2, 3, 4, 5, 7 \}$. The corresponding dynamic slice is almost identical to the one shown in Figure 3.1 **(b)**, the only difference being the fact that Gopal's algorithm excludes the final statement `write(x)` on line 8.

For certain cases, Gopal's algorithm may compute a non-terminating slice of a terminating program. Figure 3.21 **(a)** shows the slice for the program of Figure 3.2 and input `n = 1` as computed according to Gopal's algorithm.

An advantage of using dependence relations is that, for certain cases, smaller slices are computed than by Korel and Laski's algorithm. For example, Figure 3.21 **(b)** shows the slice with respect to the final value of `z` for the example program of Figure 3.18 **(a)**, for input `n = 2`. Observe that the assignment `x := 18`, which occurs in the slice computed by the algorithm of Section 3.4.1.1, is not included in Gopal's slice.

3.4.1.3 Dependence graphs

Miller and Choi [116] were the first to introduce a dynamic variation of the PDG, called the *dynamic program dependence graph*. These graphs are used by their parallel program debugger to perform *flowback analysis* [15] and are constructed incrementally, on demand. Prior to execution, a (variation of a) static PDG is constructed, and the object code of the program is augmented with code that generates a log file. In addition, an emulation package is generated. Programs are partitioned into so-called emulation blocks (typically, a subroutine). During debugging, the debugger uses the log file, the static PDG, and the

[18]Gopal does not define information-flow relations for I/O statements. For the purposes of this example, it is assumed that the statement `read(n)` can be treated as an assignment `n :=` *SomeConstant*, and that the statements `write(sum)` and `write(product)` should be treated as empty statements.

emulation package to re-execute an emulation block, and obtain the information necessary to construct the part of the dynamic PDG corresponding to that block. In case the user wants to perform flowback analysis to parts of the graph that have not been constructed yet, more emulation blocks are re-executed.

Agrawal and Horgan [6] develop an approach for using dependence graphs to compute dynamic slices. Their first two algorithms for computing dynamic slices are inaccurate, but useful for understanding their final approach. The initial approach uses the PDG as it was discussed in Section 3.3.1.3[19], and marks the *vertices* that are executed for a given test set. A dynamic slice is computed by computing a static slice in the subgraph of the PDG that is induced by the marked vertices. By construction, this slice only contains vertices that were executed. This solution is imprecise because it does not detect situations where there exists a flow edge in the PDG between a marked vertex v_1 and a marked vertex v_2, but where the definitions of v_1 are not actually used at v_2.

For example, Figure 3.22 **(a)** shows the PDG of the example program of Figure 3.2 **(a)**. Suppose that we want to compute the slice w.r.t. the final value of `x` for input `n = 2`. All vertices of the PDG are executed, causing all PDG vertices to be marked. The static slicing algorithm of Section 3.3.1.3 will therefore produce the entire program as the slice, even though the assignment `x := 18` is irrelevant. This assignment is included in the slice because there exists a dependence edge from vertex `x := 18` to vertex `write(x)` even though this edge does not represent a dependence that occurs during the second iteration of the loop. More precisely, this dependence only occurs in iterations of the loop where the control variable `i` has an odd value.

The second approach consists of marking PDG *edges* as the corresponding dependences arise during execution. Again, the slice is obtained by traversing the PDG, but this time only along marked edges. Unfortunately, this approach still produces imprecise slices in the presence of loops because an edge that is marked in some loop iteration will be present in all subsequent iterations, even when the same dependence does not recur. Figure 3.22 **(b)** shows the PDG of the example program of Figure 3.18 **(a)**. For input `n = 2`, all dependence edges will be marked, causing the slice to consist of the entire program. It is shown in [6] that a potential refinement of the second approach, consisting of *un*marking edges of previous iterations, is incorrect.

Agrawal and Horgan point out the interesting fact that their second approach for computing dynamic slices produces results that are identical to those of the algorithm proposed by Korel and Laski (see Section 3.4.1.1). However, the PDG of a program (with optionally marked edges) requires only $O(n^2)$ space (n denotes the number of statements in the program), whereas Korel and Laski's trajectories require $O(N)$ space, where N denotes the number of executed statements.

Agrawal and Horgan's second approach computes overly large slices because it does not account for the fact that different occurrences of a statement in the execution history may be (transitively) dependent on different statements. This observation motivates their third

[19]The dependence graphs of [6] do not have an entry vertex. The absence of an entry vertex does not result in a different slice. For reasons of uniformity, all dependence graphs shown in this thesis have an entry vertex.

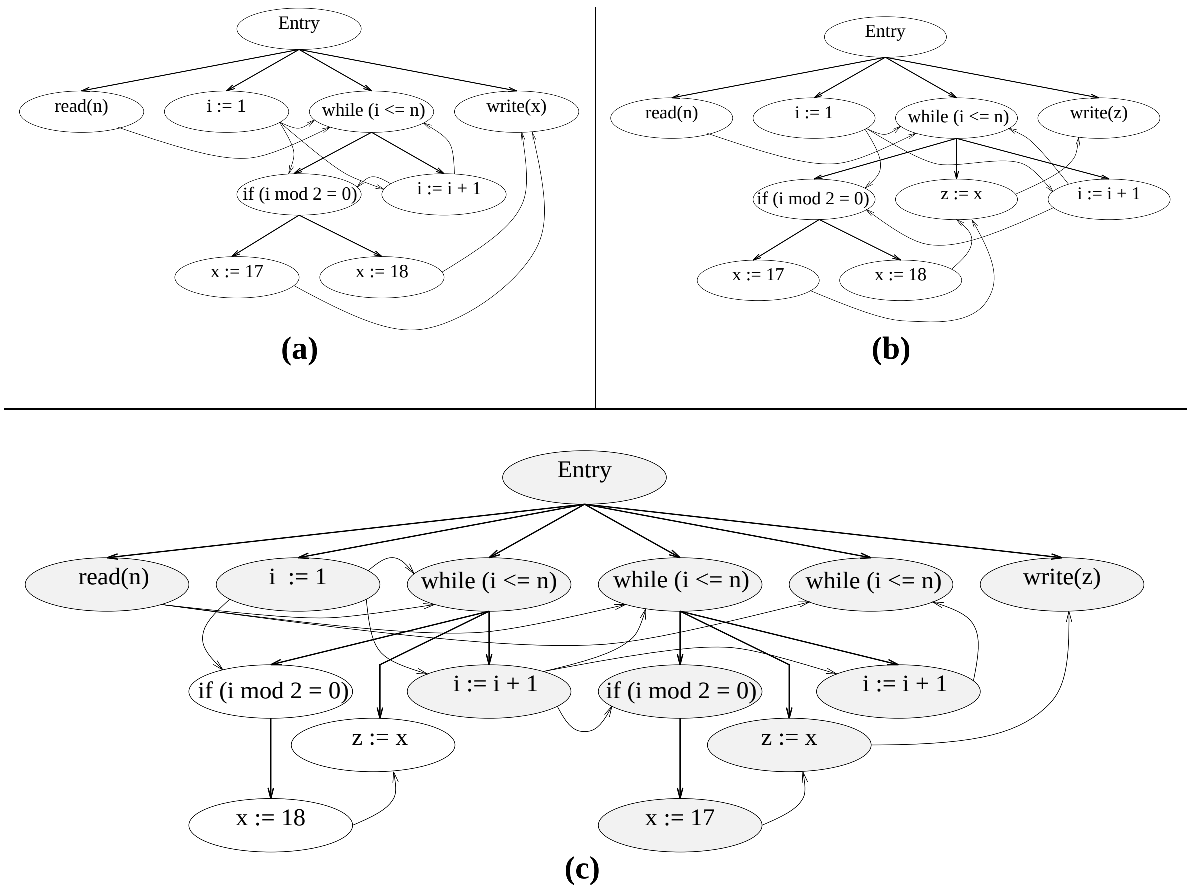


Figure 3.22: **(a)** PDG of the program of Figure 3.2 **(a)**. **(b)** PDG of the program of Figure 3.18 **(a)**. **(c)** DDG of the program of Figure 3.18 **(a)**.

solution: create a distinct vertex in the dependence graph for each occurrence of a statement in the execution history. This kind of graph is referred to as a *Dynamic Dependence Graph* (DDG). A dynamic slicing criterion is identified with a vertex in the DDG, and a dynamic slice is computed by determining all DDG vertices from which the criterion can be reached. A statement or control predicate is included in the slice if the criterion can be reached from at least one of the vertices for its occurrences.

Figure 3.22 **(c)** shows the DDG for the example program of Figure 3.18 **(a)**. The slicing criterion corresponds to the vertex labeled `write(z)`, and all vertices from which this vertex can be reached are indicated by shading. Observe that the criterion cannot be reached from the vertex labeled `x := 18`. Therefore the corresponding assignment is not in the slice.

The disadvantage of using DDGs is that the number of vertices in a DDG is equal to the number of executed statements. The number of dynamic slices, however, is in the worst case $O(2^n)$, where n is the number of statements in the program being sliced. Figure 3.23

```
program Q^n;
  read(x_1);
  ...
  read(x_n);
  MoreSubsets := true;
  while MoreSubsets do
  begin
    Finished := false;
    y := 0;
    while not(Finished) do
    begin
      read(i);
      case (i) of
        1:  y := y + x_i;
        ...
        n:  y := y + x_n;
      end;
      read(Finished);
    end;
    write(y);
    read(MoreSubsets);
  end
end.
```

Figure 3.23: Program $\mathtt{Q}^n$ with $O(2^n)$ different dynamic slices.

shows a program $\mathtt{Q}^n$ that has $O(2^n)$ dynamic slices. The program reads a number of values in variables $\mathtt{x}_i$ $(1 \le i \le n)$, and allows one to compute the sum $\sum_{x \in S} x$, for any number of subsets $S \subseteq \{ \mathtt{x}_1, \cdots, \mathtt{x}_n \}$. The crucial observation here is that, in each iteration of the outer loop, the slice with respect to statement `write(y)` will contain exactly the statements `read(`$\mathtt{x}_i$`)` for $\mathtt{x}_i \in S$. Since a set with n elements has 2^n different subsets, program $\mathtt{Q}^n$ has $O(2^n)$ different dynamic slices.

Agrawal and Horgan propose to reduce the number of vertices in the DDG by merging vertices for which the transitive dependences map to the same set of statements. In other words, a new vertex is only introduced if it can create a new dynamic slice. Obviously, this check involves some run-time overhead. The resulting graph is referred to as the *Reduced Dynamic Dependence Graph* (RDDG) of a program. Slices computed using RDDGs have the same precision as those computed using DDGs.

In the DDG of Figure 3.22 **(c)**, the vertices labeled `i := i + 1` and the rightmost two vertices labeled `i <= n` have the same transitive dependences; these vertices depend on statements 1, 2, 3, and 8 of the program of Figure 3.18 **(a)**. Hence, the RDDG for this program (given input `n = 2`) is obtained by merging these four DDG vertices into one vertex.

Agrawal and Horgan [6] present an algorithm for the construction of an RDDG without having to keep track of the entire execution history. The information that needs to be maintained is: (i) for each variable, the vertex corresponding to its last definition, (ii) for each predicate, the vertex corresponding to its last execution, and (iii) for each vertex in the RDDG, the dynamic slice w.r.t. that vertex.

3.4.2 Procedures

Agrawal, DeMillo and Spafford [4] consider dynamic slicing of procedures with call-by-value, call-by-reference, call-by-result, and call-by-value-result parameter-passing. A key property of their method is that dynamic data dependences are defined in terms of definitions and uses of *memory locations*; this has the advantage that global variables do not require special treatment, and that no alias analysis is necessary. Agrawal et al. describe how each parameter passing mechanism can be modeled by a set of mock assignments that is inserted before and/or after each procedure call. In the subsequent discussion, it is assumed that a procedure P with formal parameters $f_1, \cdots, f_n$ is called with actual parameters $a_1, \cdots, a_n$. *Call-by-value* parameter-passing can be modeled by a sequence of assignments $f_1{:=}a_1$; $\cdots$; $f_n{:=}a_n$ that is executed before the procedure is entered. In order to determine the memory cells for the correct activation record, the USE sets for the actual parameters a_i are determined *before* the procedure is entered, whereas the DEF sets for the formal parameters f_i are determined *after* the procedure is entered. For *Call-by-value-result* parameter-passing, additional assignments of formal parameters to actual parameters have to be performed upon exit from the procedure. *Call-by-reference* parameter-passing does not require any actions specific to dynamic slicing, as the same memory cell is associated with corresponding actual and formal parameters a_i and f_i.

An alternative approach for interprocedural dynamic slicing was presented by Kamkar, Shahmehri, and Fritzson [90, 89]. This work distinguishes itself from the solution by Agrawal et al. by the fact that the authors are primarily concerned with procedure-level slices. That is, they study the problem of determining the set of *call sites* in a program that affect the value of a variable at a particular call site.

During execution, a *(dynamic dependence) summary graph* is constructed. The vertices of this graph, referred to as *procedure instances*, correspond to procedure activations annotated with their parameters[20]. The edges of the summary graph are either activation edges corresponding to procedure calls, or summary dependence edges. The latter type reflects transitive data and control dependences between input and output parameters of procedure instances.

A *slicing criterion* is defined as a pair consisting of a procedure instance, and an input or output parameter of the associated procedure. After constructing the summary graph, a slice with respect to a slicing criterion is determined in two steps. First, the parts of the summary graph from which the criterion can be reached is determined; this subgraph is referred to as an *execution slice*. Vertices of an execution slice are *partial* procedure instances, because

[20]More precisely, Kamkar refers to the *incoming* and *outgoing* variables of a procedure. This notion also applies to global variables that are referenced or modified in a procedure.

some parameters may be “sliced away”. An interprocedural program slice consists of all call sites in the program for which a partial instance occurs in the execution slice.

Three approaches for constructing summary graphs are considered. In the first approach, *intra*procedural data dependences are determined statically: this may result in inaccurate slices in the presence of conditionals. In the second approach, all dependences are determined at run-time. While this results in accurate dynamic slices, the dependences for a procedure P have to be re-computed every time P is called. The third approach attempts to combine the efficiency of the “static” approach with the accuracy of the “dynamic” approach by computing the dependences inside basic blocks statically, and the inter-block dependences dynamically. In all approaches control dependences[21] are determined statically. It is unclear how useful this third approach is in the presence of composite variables and pointers, where the run-time intra-block dependences cannot be determined statically: additional alias analysis would have to be performed at run-time.

Kamkar [87] adapts the interprocedural slicing method by Kamkar et al. [90, 89] to compute statement-level interprocedural slices (i.e., slices consisting of a set of statements instead of a set of call sites). In essence, this is accomplished by introducing a vertex for each *statement instance* (instead of each procedure instance) in the summary graph. The same three approaches (static, dynamic, combined static/dynamic) for constructing summary graphs can be used.

Choi, Miller and Netzer [38] discuss an approach for interprocedural flowback analysis. Initially, it is assumed that a procedure call may modify every global variable; to this end, the static PDG is augmented with *linking* edges indicating *potential* data dependences. In a second phase, interprocedural summary information is used to either replace linking edges by data dependence edges, or delete them from the graph. Some linking edges may remain; these have to be resolved at run-time.

3.4.3 Composite data types and pointers

3.4.3.1 Dynamic flow concepts

Korel and Laski [100] consider slicing in the presence of composite variables by regarding each element of an array, or field of a record as a distinct variable. Dynamic data structures are treated as *two* distinct entities, namely the pointer itself and the object being pointed to. For dynamically allocated objects, they propose a solution where a unique name is assigned to each object.

3.4.3.2 Dependence graphs

Agrawal, DeMillo, and Spafford [4] present a dependence graph based algorithm for dynamic slicing in the presence of composite data types and pointers. Their solution consists of

[21] Kamkar et al. use a notion of *termination-preserving* control dependence that is similar to Podgurski and Clarke’s *weak* control dependence [123].

expressing DEF and USE sets in terms of *actual memory locations* provided by the compiler. The algorithm of [4] is similar to that for *static* slicing in the presence of composite data types and pointers by the same authors (see Section 3.3.4). However, during the computation of *dynamic* reaching definitions, no *Maybe* Intersections can occur—only *Complete* and *Partial* Intersections.

Choi, Miller, and Netzer [38] extend the flowback analysis method by Miller and Choi [116] (see Section 3.4.1.3) in order to deal with arrays and pointers. For arrays, *linking edges* are added to their static PDGs; these edges express *potential* data dependences that are either deleted, or changed into genuine data dependences at run-time. Pointer accesses are resolved at run-time, by recording all uses of pointers in the log file.

3.4.4 Concurrency

3.4.4.1 Dynamic flow concepts

Korel and Ferguson [98] extend the dynamic slicing method of Korel and Laski [99, 100] to distributed programs with Ada-type rendezvous communication (see, e.g., [18]). For a distributed program, the execution history is formalized as a *distributed program path* that, for each task, comprises: (i) the sequence of statements (trajectory) executed by it, and (ii) a sequence of triples $\langle A, C, B \rangle$ identifying each rendezvous the task is involved in. Here, A identifies the **accept** statement in the task, B identifies the other task that participated in the communication, and C denotes the entry call statement in the task that was involved in the rendezvous.

A dynamic slicing criterion of a distributed program specifies: (i) the input of each task, (ii) a distributed program path P, (iii) a task w, (iv) a statement occurrence q in the trajectory of w, and (v) a variable v. A dynamic slice with respect to such a criterion is an executable projection of the program that is obtained by deleting statements from it. However, the program is only guaranteed to preserve the behavior of the program if the rendezvous in the slice occur in the same relative order as in the program. (Note that not all rendezvous of the program need to be in the slice.)

The Korel-Ferguson method for computing slices of distributed programs of is basically a generalization of the Korel-Laski method, though stated in a slightly different manner. In addition to the previously discussed dynamic flow concepts (see Section 3.4.1.1), a notion of *communication influence* is introduced, to capture the interdependences between tasks. The authors also present a distributed version of their algorithm that uses a separate slicing process for each task.

3.4.4.2 Dependence graphs

Duesterwald, Gupta, and Soffa [50] present a dependence graph based algorithm for computing dynamic slices of distributed programs. They introduce a Distributed Dependence

Graph (DDG)[22] for representing distributed programs.

A distributed program P consists of a set of processes $P_1, \cdots, P_n$. Communication between processes is assumed to be synchroneous and nondeterministic and is expressed by way of **send** and **receive** statements. A *distributed dynamic slicing criterion* $\langle I_1, X_1 \rangle, \cdots, \langle I_n, X_n \rangle$ specifies for each process P_i its input I_i, and a set of statements X_i. A *distributed dynamic slice* S is an executable set of processes $P'_1, \cdots, P'_n$ such that the statements of P'_i are a subset of those of P_i. Slice S computes the same values at statements in each X_i as program P does, when executed with the same input. This is accomplished by: (i) including *all* input statements in the slice, and (ii) replacing nondeterministic communication statements in the program by deterministic communication statements in the slice.

A DDG contains a single vertex for each statement and control predicate in the program. Control dependences between statements are determined statically, prior to execution. Edges for data and communication dependences are added to the graph at run-time. Slices are computed in the usual way by determining the set of DDG vertices from which the vertices specified in the criterion can be reached. Both the construction of the DDG and the computation of slices is performed in a distributed manner; a separate DDG construction process and slicing process is assigned to each process P_i in the program; these processes communicate when a **send** or **receive** statement is encountered.

Due to the fact that a single vertex is used for all occurrences of a statement in the execution history, inaccurate slices may be computed in the presence of loops (see Section 3.4.1.1). For example, the slice with respect to the final value of `z` for the program of Figure 3.18 with input `n = 2` will be the entire program.

Cheng [35] presents an alternative dependence graph based algorithm for computing dynamic slices of distributed and concurrent programs. The PDN representation of a concurrent program (see Section 3.3.5) is used for computing dynamic slices. Cheng's algorithm is basically a generalization of the initial approach proposed by Agrawal and Horgan [6]: the PDN vertices corresponding to executed statements are marked, and the static slicing algorithm of Section 3.3.5 is applied to the PDN subgraph induced by the marked vertices. As was discussed in Section 3.4.1.3, this yields inaccurate slices.

Choi et al. [116, 38] describe how their approach for flowback analysis can be extended to parallel programs. Shared variables with semaphores, message-passing communication, and Ada-type rendezvous mechanisms are considered. To this end, a parallel dynamic graph is introduced that contains synchronization vertices for synchronization operations (such as P and V on a semaphore) and synchronization edges that represent dependences between concurrent processes. Choi et al. explain how, by analysis of the parallel dynamic graph, read/write and write/write conflicts between concurrent processes can be found.

[22]This abbreviation 'DDG' used in Section 3.4.4.2 should not be confused with the notion of a Dynamic Dependence Graph that was discussed earlier in Section 3.4.1.

	Computation Method [a]	Executable	Interprocedural Solution	Data Types [b]	Concurrency
Korel, Laski [99, 100]	D	yes	no	S, A, P	no
Korel, Ferguson [98]	D	yes	no	S, A	yes
Gopal [62]	I	no	no	S	no
Agrawal, Horgan [6]	G	no	no	S	no
Agrawal et al. [2, 4]	G	no	yes	S, A, P	no
Kamkar et al. [90, 89]	G	no	yes	S	no
Duesterwald et al. [50]	G	yes	no	S, A, P	yes
Cheng [35]	G	no	no	S	yes
Choi et al. [116, 38]	G	no	yes	S, A, P	yes
Field et al. (Chap. 5, 6)	R	yes	no	S, A, P	no

[a] D = dynamic flow concepts, I = dynamic dependence relations, G = reachability in a dependence graph. R = dependence tracking in term graph rewriting systems (see Section 3.6).
[b] S = scalar variables, A = arrays/records, P = pointers.

Table 3.4: Overview of dynamic slicing methods.

3.4.5 Comparison

In this section, we compare and classify the dynamic slicing methods that were presented earlier. The section is organized as follows: Section 3.4.5.1 summarizes the problems that are addressed in the literature. Sections 3.4.5.2 and 3.4.5.3 compare the *accuracy* and *efficiency* of slicing methods that address the same problem, respectively. Finally, Section 3.4.5.4 investigates the possibilities for “combining” algorithms that deal with different problems.

3.4.5.1 Overview

Table 3.4 lists the dynamic slicing algorithms discussed earlier, and summarizes the issues studied in each paper. For each paper, the table shows: (i) the computation method, (ii) whether or not the computed slices are executable programs, (iii) whether or not an interprocedural solution is supplied, (iv) the data types under consideration, and (v) whether or not concurrency is considered. Similar to Table 3.2, the table only shows problems that *have been addressed*. It does *not* indicate how various algorithms may be combined, and it also does not give an indication of the quality of the work. The work by Field et al. (see Chapters 4, 5, and 6) mentioned in Table 3.4 relies on substantially different techniques than those used for the dynamic slicing algorithms discussed previously, and will therefore be studied separately in Section 3.6.

Unlike in the static case, there exists a significant difference between methods that compute executable slices [99, 100, 50, 98], and approaches that compute slices merely consisting of sets of statements [6, 4, 62]. The latter type of slice may not be executable due to the absence of assignments for incrementing loop counters[23]. For convenience, we will henceforth refer to such slices as "non-executable" slices. As was discussed in Section 3.4.1.1, the algorithms that compute executable dynamic slices may produce inaccurate results in the presence of loops.

Apart from the work by Venkatesh [137], there is very little semantic justification for any of the methods for computing "non-executable" slices. The algorithms of [116, 6, 90, 35, 89] are graph-reachability algorithms that compute a set of statements that directly or indirectly "affect" the values computed at the criterion. Besides the algorithms themselves, little or no attention is paid to formal characterization of such slices.

3.4.5.2 Accuracy

basic algorithms. The slices computed by Korel and Laski's algorithm [99, 100] (see Section 3.4.1.1) are larger than those computed by the algorithms by Agrawal and Horgan [6] (see Section 3.4.1.3) and Gopal [62] (see Section 3.4.1.2). This is due to Korel and Laski's constraint that their slices should be executable.

procedures. Dependence graph based algorithms for interprocedural dynamic slicing were proposed by Agrawal, DeMillo, and Spafford [4], and by Kamkar et al. [90, 89] (see Section 3.4.2). It is unclear if one of these algorithms produces more accurate slices than the other.

composite variables and pointers. Korel and Laski [100] (see Section 3.4.1.1), and Agrawal, DeMillo, and Spafford (see Section 3.4.1.3) proposed methods for dynamic slicing in the presence of composite variables and pointers. We are unaware of any difference in accuracy.

concurrency. Korel and Ferguson [98] (see Section 3.4.4.1) and Duesterwald, Gupta, and Soffa [50] (see Section 3.4.4.2) compute executable slices, but deal with nondeterminism in a different way: the former approach requires a mechanism for replaying rendezvous in the slice in the same relative order as they appeared in the original program, whereas the latter approach replaces nondeterministic communication statements in the program by deterministic communication statements in the slice. Cheng [35] and Choi et al. [116, 38] (see Section 3.4.4.2) do not address this problem because the slices they compute are not necessarily executable. The dynamic slicing methods by Cheng and Duesterwald et al. are inaccurate because they are based on "static" dependence graphs in which no distinction

[23]Of course, such a slice may be executed anyway; however, it may not terminate.

is made between the different occurrences of a statement in the execution history (see the discussion in Section 3.4.1.3).

3.4.5.3 Efficiency

Since dynamic slicing involves run-time information, it is not surprising that all dynamic slicing methods discussed in this section have time requirements that depend on the number of executed statements (or procedure calls in the case of [90, 89]) N. All algorithms spend at least $O(N)$ time during execution in order to store the execution history of the program, or to update dependence graphs. Certain algorithms (e.g., [99, 100, 98]) traverse the execution history in order to extract the slice and thus require again *at least* $O(N)$ time *for each slice*, whereas other algorithms require less (sometimes even constant) time. Whenever time requirements are discussed below, the time spent during execution for constructing histories or dependence graphs will be ignored. Space requirements will always be discussed in detail.

basic algorithms. Korel and Laski's solution [99, 100] (see Section 3.4.1.1) requires $O(N)$ space to store the trajectory, and $O(N^2)$ space to store the dynamic flow concepts. Construction of the flow concepts requires $O(N \times (v + n))$ time, where v and n are the number of variables and statements in the program, respectively. Extracting a single slice from the computed flow concepts can be done in $O(N)$ time.

The algorithm by Gopal [62] (see Section 3.4.1.2) requires $O(N)$ space to store the execution history and $O(n \times v)$ space to store the $\overline{\mu}_S$ relation. The time required to compute the $\overline{\mu}_S$ relation for a program S is bounded by $O(N^2 \times v^2)$. From this relation, slices can be extracted in $O(v)$ time.

As was discussed in Section 3.4.1.3, the slicing method proposed by Agrawal and Horgan requires at most $O(2^n)$ space, where n is the number of statements in the program. Since vertices in an RDDG are annotated with their slice, slices can be extracted from it in $O(1)$.

procedures. The interprocedural dynamic slicing method proposed by Kamkar et al. [90, 89] (see Section 3.4.2) requires $O(P^2)$ space to store the summary graph, where P is the number of executed procedure calls. A traversal of this graph is needed to extract a slice; this takes $O(P^2)$ time.

The time and space requirements of the method by Agrawal, DeMillo, and Spafford [4] are essentially the same as those of the Agrawal-Horgan basic slicing method discussed above.

composite variables and pointers. The algorithms by Korel and Laski [100] (see Section 3.4.3.1) and Agrawal, DeMillo, and Spafford [4] (see Section 3.4.3.2) for slicing in the presence of composite variables and pointers are adaptations of the basic slicing algorithms by Korel and Laski and Agrawal and Horgan, respectively (see the discussion above). These adaptations, which essentially consist of a change in the reaching definitions functions that

	Procedures	Composite Variables	Concurrency
Dyn. Flow Concepts	—	Korel, Laski [99, 100]	Korel, Ferguson [98]
Dyn. Dependence Relations	Gopal [62]	—	—
Dependence Graphs	Agrawal et al. [4] Kamkar et al. [90, 89]	Agrawal et al. [4]	Duesterwald et al. [50] Cheng [35] Choi et al. [116, 38]

Table 3.5: Orthogonal dimensions of dynamic slicing.

are used to determine data dependences, do not affect the worst-case behavior of the algorithms. Therefore, we expect the time and space requirements to be the same as in the scalar variable case.

concurrency. The algorithms by Cheng [35] and Duesterwald et al. [50] are based on *static* PDGs. Therefore, only $O(n^2)$ space is required to store the dependence graph, and slices can be extracted in $O(n^2)$ time. The distributed slicing algorithm by Duesterwald et al. [50] uses a separate slicing process for each process in the program; the slicing process for process P_i requires time $O(e_i)$, where e_i is the number of edges in the PDG for process P_i. The communication overhead between the slicing processes requires at most $O(e)$ time, where e is the number of edges in the entire graph.

3.4.5.4 Combining dynamic slicing algorithms

Table 3.5 displays solutions to “orthogonal” dimensions of dynamic slicing: dealing with procedures, composite variables and pointers, and communication between processes. The algorithms based on dynamic flow concepts for dealing with composite variables/pointers [100], and concurrency [98] may be integrated with little problems. For dependence graphs, however, the situation is slightly more complicated because:

- Different graph representations are used. Agrawal et al. [4], Kamkar et al. [90, 89] and Choi et al. [116, 38] use dynamic dependence graphs with distinct vertices for different occurrence of statements in the execution history. In contrast, Duesterwald et al. [50] and Cheng [35] use variations of static PDGs.
- The dynamic slicing by Agrawal et al. [4] is based on definition and use of memory locations. All other dependence graph based slicing methods are based on definitions

and uses of variable names.

Furthermore, it is unclear if the combined static/dynamic interprocedural slicing approach by Kamkar et al. [90, 89] is practical in the presence of composite variables and pointers, because the intra-block dependences cannot be determined statically in this case, and additional alias analysis would be required at run-time.

3.5 Applications of program slicing

3.5.1 Debugging and program analysis

Debugging can be a difficult task when one is confronted with a large program, and few clues regarding the location of a bug. Program slicing is useful for debugging, because it potentially allows one to ignore many statements in the process of localizing a bug [108]. If a program computes an erroneous value for a variable x, only the statements in the slice w.r.t. x have (possibly) contributed to the computation of that value. In this case, it is *likely* that the error occurs in the one of the statements in the slice. However, it need not *always* be the case that the error occurs in the slice, as an error may consist of a statement that is missing inadvertently. However, in situations like this it is probable that more, or different statements show up in the slice than one would expect.

Forward slices are also useful for debugging. A forward slice w.r.t. a statement s can show how a value computed at s is being used subsequently, and can help the programmer ensure that s establishes the invariants assumed by the later statements. For example, this can be useful in catching off-by-one errors. Another purpose of forward slicing is to inspect the parts of a program that may be affected by a proposed modification, to check that there are no unforeseen effects on the program's behavior.

Lyle and Weiser [109] introduce *program dicing*, a method for combining the information of different slices. The basic idea is that, when a program computes a correct value for variable x and an incorrect value for variable y, the bug is *likely* to be found in statements that are in the slice w.r.t. y, but not in the slice w.r.t. x. This approach is not fail-safe in the presence of multiple bugs, and when computations that use incorrect values produce correct values (referred to as *coincidental correctness* by Agrawal [2]). The authors claim that program dicing still produces useful results when these assumptions are relaxed.

Bergeretti and Carré [22] explain how static slicing methods can detect "dead" code, i.e., statements that cannot affect any output of the program. Often, such statements are not executable due to the presence of a bug. Static slicing can also be employed to determine uses of uninitialized variables, another symptom of an error in the program. However, there exist previous techniques for detection of dead code and uses of uninitialized variables [7, 150] that do not rely on slicing.

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior. *Dynamic* slices are particularly useful here, because they only reflect the actual dependences of that execution, resulting in smaller slices than static ones. Agrawal's thesis [2] contains a detailed discussion how static and dynamic slicing can be utilized for semi-automated debugging of programs [6, 4]. He proposes an approach where the user gradually “zooms out” from the location where the bug manifested itself by repeatedly considering larger data and control slices. A *data slice* is obtained by only taking (static or dynamic) data dependences into account; a *control slice* consists of the set of control predicates surrounding a language construct. The closure of all data and control slices w.r.t. an expression is the (static or dynamic) slice w.r.t. the set of variables used in the expression. The information of several dynamic slices can be combined to gain some insight into the location of a bug. Several operations on slices are proposed to this end, such as union, intersection, and difference. The difference operation is a dynamic version of the program “dicing” notion of Lyle and Weiser [109]. Obviously, these operations for combining slices may produce false leads in the presence of multiple bugs or coincidental correctness. Agrawal, DeMillo, and Spafford [5] discuss the implementation of a debugging tool that is based on ideas in previous papers by the same authors [2, 4, 6].

Pan and Spafford [121, 122] present a number of heuristics for fault localization. These heuristics describe how dynamic slices (variations on the type proposed by Agrawal et al. [6]) can be used for selecting a set of suspicious statements that is likely to contain a bug. The approach of Pan and Spafford consists of two phases. First, the program is executed for an extensive number of test cases, and each test case is classified as being *error-revealing* or *non-error-revealing*, depending on the fact whether or not its reveals the presence of a bug. The second step consists of the actual *heuristic rules* for combining the information contained in dynamic slices for these test cases in various ways. As an example, one might think of displaying the set of statements that occur in *every* dynamic slice for an error-revealing test-case—such statements are likely to contain the bug. Other heuristics depend on the *inclusion frequency* or the *influence frequency* of statements in dynamic slices. The former denotes the number of slices in which a particular statement occurs, whereas the latter notion indicates the number of times that a statement in a particular dynamic slice is “referred to” in terms of data dependence and control dependence. For example, one of the heuristics given by Pan and Spafford consists of selecting the statements with “high” influence frequency in a slice for a selected error-revealing test case. Note that this requires a *threshold* to be specified by the user that determines the boundary between “high” and “low” frequencies. It is argued that this boundary can be shifted interactively, thereby gradually increasing the number of statements under consideration.

Choi, Miller and Netzer [38] describe the design and efficient implementation of a debugger for parallel programs that incorporates *flowback analysis*, a notion introduced in the seminal paper by Balzer [15]. Intuitively, flowback analysis reveals how the computation of values depends on the earlier computation of other values. The difference between flowback analysis and (dependence graph based) dynamic slices is that the former notion allows one to interactively browse through a dependence graph, whereas the latter consists of the set of all program parts corresponding to vertices of the graph from which a designated vertex—the criterion—can be reached.

Fritzson et al. use interprocedural static [59] and dynamic [90, 87] slicing for algorithmic debugging [132, 131]. An algorithmic debugger partially automates the task of localizing a bug by comparing the *intended* program behavior with the *actual* program behavior. The intended behavior is obtained by asking the user whether or not a program unit (e.g., a procedure) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level. By applying the algorithmic debugging process to a *slice* w.r.t. an incorrectly valued variable instead of the entire program, many irrelevant questions can be skipped.

3.5.2 Program differencing and program integration

Program *differencing* [70] is the task of analyzing an old and a new version of a program in order to determine the set of program components of the new version that represent syntactic and semantic changes. Such information is useful because only the program components reflecting changed behavior need to be tested. The key issue in program differencing consists of partitioning the components of the old and new version in a way that two components are in the same partition only if they have equivalent behaviors. The program integration algorithm of Horwitz, Prins, and Reps [74] discussed below, compares slices in order to detect equivalent behaviors. However, an alternative partitioning technique by Yang et al. [149, 70], which is not based on comparing slices but on comparing smaller units of code, produces more accurate results because semantics-preserving transformations (e.g., copy propagation) can be accommodated.

Horwitz, Prins, and Reps [74] use the static slicing algorithm for single-procedure programs by Horwitz, Reps, and Binkley [77] as a basis for an algorithm that integrates changes in variants of a program. The inputs of their algorithm consist of a program *Base*, and two variants *A* and *B* that have been derived from *Base*. The algorithm consists of the following steps:

1. The PDGs G_{Base}, G_A, and G_B are constructed. Correspondences between "related" vertices of these graphs are assumed to be available.
2. Sets of *affected points* of G_A and G_B w.r.t. G_{Base} are determined; these consist of vertices in G_A (G_B) that have a different slice in G_{Base}[24].
3. A merged PDG G_M is constructed from G_A, G_B, and the sets of affected points determined in (2).
4. Using G_A, G_B, G_M, and the sets of affected points computed in (2), the algorithm determines whether or not the behaviors of A and B are preserved in G_M. This is accomplished by comparing the slices w.r.t. the affected points of G_A (G_B) in G_M and G_A (G_B). If different slices are found, the changes interfere and the integration cannot be performed.
5. If the changes in A and B do not interfere, the algorithm tests if G_M is a feasible PDG, i.e., if it corresponds to some program. If this is the case, program M is constructed from G_M. Otherwise, the changes in A and B cannot be integrated.

A semantic justification for the single-procedure slicing algorithm of Horwitz, Reps,

[24]These sets of affected points can be computed efficiently by way of a *forward* slice w.r.t. all *directly* affected points, i.e., all vertices in G_A that do not occur in G_{Base} and all vertices in that have a different set of incoming edges in G_A and in G_{Base} [76].

and Binkley [77] and the program integration algorithm of Horwitz, Prins, and Reps [74] is presented by Reps and Yang [130]. This paper formalizes the relationship between the execution behaviors of programs, slices of those programs, and between variants of a program and the corresponding integrated version. The comparison of slices (in step 4) relies on the existence of a mapping between the different components. If such a mapping were not available, however, the techniques of Horwitz and Reps [75] for comparing two slices in time that is linear in the sum of their sizes could be used.

Reps [124] presents an alternative formulation of the Horwitz-Prins-Reps program integration algorithm that is based on Brouwerian algebras. The algebraic laws that hold in such algebras are used to restate the algorithm and to prove properties such as associativity of consecutive integrations.

Binkley, Horwitz and Reps [33] generalize the integration algorithm of Horwitz, Prins, and Reps [74] to multi-procedure programs. It is shown that such programs cannot be integrated on a per-procedure basis (program behavior would not be preserved in all cases), and that a straightforward extension using the Horwitz-Reps-Binkley interprocedural slicing algorithm is insufficiently powerful (it reports "interference" in too many cases). While a complete discussion of the theory that underlies the Binkley-Horwitz-Reps multi-procedure integration algorithm is outside the scope of this survey, it can be remarked here that the algorithm relies on backward and forward interprocedural slices on the SDG representation of the program.

3.5.3 Software maintenance

One of the problems in software maintenance consists of determining whether a change at some place in a program will affect the behavior of other parts of the program. Gallagher and Lyle [60, 61] use static slicing for the decomposition of a program into a set of components (i.e., reduced programs), each of which captures part of the original program's behavior. They present a set of guidelines for the maintainer of a component that, if obeyed, preclude changes in the behavior of other components. Moreover, they describe how changes in a component can be merged back into the complete program in a semantically consistent way.

Gallagher and Lyle use the notion of a *decomposition slice* for the decomposition of programs. Intuitively, a decomposition slice captures part of the behavior of a program, and its complement captures the behavior of the rest of the program. A decomposition slice w.r.t. a variable v is defined as the set of all statements that may affect the "observable" value of v at some point; it is defined as the union of the slices w.r.t. v at any statement that outputs v, and the last statement of the program. An *output-restricted* decomposition slice (ORD slice) is a decomposition slice from which all output statements are removed. Two ORD slices are *independent* if they have no statements in common; an ORD slice is *strongly dependent* on another ORD slice if it is a subset of the latter. An ORD slice that is not strongly dependent on any other ORD slice is *maximal*. A statement that occurs in more than one ORD slice is *dependent*; otherwise it is *independent*. A variable is *dependent* if it is assigned to in some dependent statement; it is *independent* if it is only assigned to in independent statements. Only maximal ORD slices contain independent statements, and the union of all maximal

ORD slices is equal to the original program (minus output statements). The *complement* of an ORD slice is defined as the original program minus all independent statements of the ORD slice and all output statements.

The essential observation by Gallagher and Lyle [61] is that independent statements in a slice do not affect the data and control flow in the complement. This results in the following guidelines for modification:

- Independent statements may be deleted from a decomposition slice.
- Assignments to independent variables may be added anywhere in a decomposition slice.
- Logical expressions and output statements may be added anywhere in a decomposition slice.
- New control statements that surround any dependent statements will affect the complement's behavior.

New variables may be considered as independent variables, provided that there are no name clashes with variables in the complement. If changes are required that involve a dependent variable v, the user can either extend the slice so that v is independent (in a way described in the paper), or introduce a new variable. Merging changes to components into the complete program is a trivial task. Since it is guaranteed that changes to an ORD slice do not affect its complement, only testing of the modified slice is necessary.

3.5.4 Testing

A program satisfies a "conventional" *data flow testing* criterion if all def-use pairs occur in a successful test-case. Duesterwald, Gupta, and Soffa [51] propose a more rigorous testing criterion, based on program slicing: each def-use pair must be exercised in a successful test-case; moreover it must be *output-influencing*, i.e., have an influence on at least one output value. A def-use pair is output-influencing if it occurs in an *output slice*, i.e., a slice w.r.t. an output statement. It is up to the user, or an automatic test-case generator to construct enough test-cases such that all def-use pairs are tested. Three slicing approaches are utilized, based on different dependence graphs. Static slices are computed using *static dependence graphs* (similar to the PDGs of Horwitz, Reps, and Binkley [77]), dynamic slices are computed using *dynamic dependence graphs* (similar to DDGs of Agrawal and Horgan [6], but instances of the same vertex are merged, resulting in a slight loss of precision), and *hybrid slices* are computed using dependence graphs that are based on a combination of static and dynamic information. In the hybrid approach, the set of variables in the program is partitioned into two disjoint subsets in a way that variables in one subset do not refer to variables in the other subset. Static dependences are computed for one subset (typically scalar variables), dynamic dependences for the other subset (typically arrays and pointers). The advantage of this approach is that it combines reasonable efficiency with reasonable precision.

Kamkar, Shahmehri, and Fritzson [88] extend the work of Duesterwald, Gupta, and Soffa to multi-procedure programs. To this end, they define appropriate notions of interprocedural def-use pairs. The interprocedural dynamic slicing method by Kamkar et al. [90, 89] is used to determine which interprocedural def-use pairs have an effect on a correct output value, for a given test case. The summary graph representation that was discussed in Section 3.4.2 is slightly modified by annotating vertices and edges with def-use information. This way, the set of def-use pairs exercised by a slice can be determined efficiently.

Regression testing consists of re-testing only the parts affected by a modification of a previously tested program, while maintaining the "coverage" of the original test suite. Gupta, Harrold, and Soffa [65] describe an approach to regression testing where slicing techniques are used. Backward and forward static slices serve to determine the program parts affected by the change, and only test cases that execute "affected" def-use pairs need to be executed again. Conceptually, slices are computed by backward and forward traversals of the CFG of a program, starting at the point of modification. However, the algorithms by Gupta, Harrold, and Soffa [65] are designed to determine the information necessary for regression testing only (i.e., affected def-use pairs).

Binkley [29] describes an approach for reducing the cost of regression testing of *multi-procedure* programs by (i) reducing the number of tests that must be re-run, and (ii) decreasing the size of the program that they must run on. This is accomplished by determining the set of program points *affected* by the modification, and the set of *preserved* program points (see Section 3.5.2). The set of affected points is used to construct a smaller and more efficient program that only captures the modified behavior of the original program; all test-cases that need to be re-run can be applied to this program. The set of preserved points is used to infer which test-cases need not be re-run.

Bates and Horwitz [20] use a variation of the PDG notion of Horwitz, Prins, and Reps [74] for incremental program testing. Testing criteria are defined in terms of PDG notions: i.e., the "all-vertices" testing criterion is satisfied if each vertex of the PDG is exercised by a test set (i.e., each statement and control predicate in the program is executed). An "all-flow-edges" criterion is defined in a similar manner. Given a tested and subsequently modified program, slicing is used to determine: (i) the statements affected by the modification, and (ii) the test-cases that can be reused for the modified program. Roughly speaking, the former consists of the statements that did not occur previously as well as any statements that have different slices. The latter requires partitioning the statements of the original and the modified program into equivalence classes; statements are in the same class if they have the same "control" slice (a slightly modified version of the standard notion). Bates and Horwitz prove that statements in the same class are exercised by the same test cases.

3.5.5 Tuning compilers

Larus and Chandra [106] present an approach for tuning of compilers where dynamic slicing is used to detect potential occurrences of redundant common subexpressions. Finding such a common subexpression is an indication of sub-optimal code being generated.

Object code is instrumented with trace-generating instructions. A trace-regenerator reads a trace and produces a stream of events, such as the read and load of a memory location. This stream of events is input for a compiler-auditor (e.g., a common-subexpression elimination auditor) that constructs dynamic slices w.r.t. the current values stored in registers. Larus and Chandra use a variation of the approach by Agrawal and Horgan [6]: a dynamic slice is represented by directed acyclic graph (DAG) containing all operators and operands that produced the current value in a register. A common subexpression occurs when isomorphic DAGs are constructed for two registers. However, the above situation only indicates that a common subexpression occurs in a *specific* execution. A common subexpression occurs in *all* execution paths if its inputs are the same in all executions. This is verified by checking that: (i) the program counter PC1 for the first occurrence of the common subexpression dominates the program counter PC2 for the second occurrence, (ii) the register containing the first occurrence of the common subexpression is not modified along any path between PC1 and PC2, and (iii) neither are the inputs to the common subexpression modified along any path between PC1 and PC2. Although the third condition is impossible to verify in general, it is feasible to do so for a number of special cases. In general, it is up to the compiler writer to check condition (iii).

3.5.6 Other applications

Weiser [146] describes how slicing can be used to *parallelize* the execution of a sequential program. Several slices of a program are executed in parallel, and the outputs of the slices are *spliced* together in such a way that the I/O behavior of the original program is preserved. In principle, the splicing process may take place in parallel with the execution of the slices. A natural requirement of Weiser's splicing algorithm is that the set of all slices should "cover" the execution behavior of the original program. Splicing does not rely on a particular slicing technique; any method for computing executable static slices is adequate. Only programs with structured control flow are considered, because Weiser's splicing algorithm depends on the fact that execution behavior can be expressed in terms of a so-called program regular expression. The main reason for this is that reconstruction of the original I/O behavior becomes unsolvable in the presence of irreducible control flow.

Ott and Thus [119] view a module as a set of processing elements that act together to compute the outputs of a module. They classify the *cohesion class* of a module (i.e, the kind of relationships between the processing elements) by comparing the slices w.r.t. different output variables. *Low* cohesion corresponds to situations where a module is partitioned into disjoint sets of unrelated processing elements. Each set is involved in the computation of a different output value, and there is no overlap between the slices. *Control* cohesion consists of two or more sets of disjoint processing elements each of which depends on a common input value; the intersection of slices will consist of control predicates. *Data* cohesion corresponds to situations where data flows from one set of processing elements to another; slices will have non-empty intersection and non-trivial differences. *High* cohesion situations resemble pipelines. The data from a processing element flows to its successor; the slices of high cohesion modules will overlap to a very large extent. The paper does not rely on any specific slicing method, and no quantitative measures are presented.

Binkley [32] presents a graph rewriting semantics for System Dependence Graphs that is used for performing interprocedural constant propagation. The Horwitz-Reps-Binkley interprocedural slicing algorithm is used to extract slices that may be executed to obtain constant values.

Beck and Eichmann [21] consider the case where a “standard” module for an abstract data type module is used, and where only part of its functionality is required. Their objective is to “slice away” all unnecessary code in the module. To this end, they generalize the notion of static slicing to modular programs. In order to compute a reduced version of a module, an *interface dependence graph* (IDG) is constructed. This graph contains vertices for all definitions of types and global variables, and subprograms inside a module. Moreover, the IDG contains edges for every def-use relation between vertices. An *interface slicing criterion* consists of a module and a subset of the operations of the ADT. Computing interface slices corresponds to solving a reachability problem in an IDG. Inter-module slices, corresponding to situations where modules import other modules, can be computed by deriving new criteria for the imported modules.

Jackson and Rollins present a reverse engineering tool called “Chopshop” in [81] that is based on the techniques of [82] (see Sections 3.3.1.3 and 3.3.2.3). This tool provides facilities for visualizing program slices in a graphical manner as diagrams. In addition to “chopping” (see Section 3.3.1.3), their tool is capable of “abstracting” slices by eliminating all non-call-site nodes in a graph and resulting in a graph with only call site vertices and transitive dependence edges between these vertices.

Ning, Engberts, and Kozaczynski [117] discuss a set of tools for extracting components from large Cobol systems. These tools include facilities for *program segmentation*, i.e., distinguishing pieces of functionally related code. In addition to backward and forward static slices, *condition-based* slices can be determined. For a condition-based slice, the criterion specifies a constraint on the values of certain variables.

3.6 Recent developments

This section is concerned with recent work on improving the precision of slicing methods, which relies on the removal of two important restrictions characteristic of the slicing algorithms discussed previously:

1. The fact that a slice consists of a subset of the statements of the original program, sometimes with the additional constraint that a slice must constitute a syntactically valid program.
2. The fact that slices are computed by tracing data and control dependences.

Both of these “restrictions” adversely affect the accuracy of the computed slices. Moreover, it is important to realize that these issues are strongly interrelated in the sense that, in many cases, dismissing the former constraint is a prerequisite for being able to dismiss the latter one.

Weiser already observed some problems caused by the first constraint in his dissertation [144, page 6], where he states that ‘good source language slicing requires transformations beyond statement deletion’. This remark can easily be understood by considering a situation where a programming language does not allow **if** statements with empty branches, but where

```
read(n);
i := 1;
if (i > 0) then
  n := n + 1
else
  n := n * 2;
write(n)
```

(a)

```
read(n);
i := 1;
if (i > 0) then
  n := n + 1
else
                ;
write(n)
```

(b)

```
read(n);


  n := n + 1

                ;
write(n)
```

(c)

Figure 3.24: **(a)** Example program = static slice with respect to statement `write(n)`. **(b)** Accurate slice obtained by employing constant propagation. **(c)** Minimal slice.

a slicing algorithm would exclude all statements in such a branch. Taken to the extreme, such statements can never be removed from a slice because the result would not be a syntactically valid program. Hwang et al. [80] discuss a number of related problems and conclude that, in practice, statement deletion alone is an inadequate method for deriving slices.

The second constraint—the fact that slices are to be computed by tracing data and control dependences alone—has to be removed as well, if the singular objective is to compute slices that are as small as possible. To see this, consider the example program of Figure 3.24 **(a)**. Here, the static slice with respect to statement `write(n)` as computed by any of the "conventional" slicing algorithms consists of the entire program[25]. However, if constant propagation [141] or similar optimization techniques could be used in slicing, the resulting slices might be more accurate. In the program of Figure 3.24 **(a)**, for example, one can determine that the value of `i` is constant, and that the **else** branch of the conditional is never selected. Therefore, computation of the more accurate slice of Figure 3.24 **(b)** is conceivable. Moreover, if replacement of an entire **if** statement by one of the statements in its branches is allowed, one can imagine that the minimal slice of Figure 3.24 **(c)** is determined.

Other compiler optimization techniques[26], symbolic execution, and a variety of semantics-preserving operations can also be of use for obtaining more accurate slices. For example, Figure 3.25 **(a)** shows another example program, which is to be sliced with respect to its final statement `write(y)`. Once again, traditional slicing algorithms will fail to omit any statements. A more accurate slice for this example can be acquired by "merging" the two **if** statements. The effect of this semantics-preserving transformation is shown in Figure 3.25 **(b)**. Clearly, a slicing algorithm that can (conceptually) perform this transformation is in principle capable of determining the more accurate slice of Figure 3.25 **(c)**.

Approaches that use optimization techniques for obtaining more accurate slices, such as the ones shown in Figures 3.24 and 3.25, were presented by Field, Ramalingam, and Tip (see Chapter 5), and by Ernst [52]. At the conceptual level, these slicing approaches rely on the following components:

[25]Some algorithms [147, 22] would omit the `write` statement.

[26]See, e.g., [150] for a comprehensive overview.

```
read(p);
read(q);
if (p = q) then
  x := 18
else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

(a)

```
read(p);
read(q);
if (p = q) then
  begin
    x := 18;
    y := 2
  end
else
  begin
    x := 17;
    y := x
  end
write(y)
```

(b)

```
read(p);
read(q);
if (p = q) then
          ;
else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

(c)

Figure 3.25: **(a)** Example program = static slice with respect to the statement `write(y)`. **(b)** Transformed program. **(c)** More accurate slice obtained by slicing in the transformed program.

- Translation of the program to a suitable intermediate representation (IR).
- Transformation and optimization of the IR.
- Maintaining a mapping between the source-text, the original IR, and the optimized IR.
- Extraction of slices from the IR.

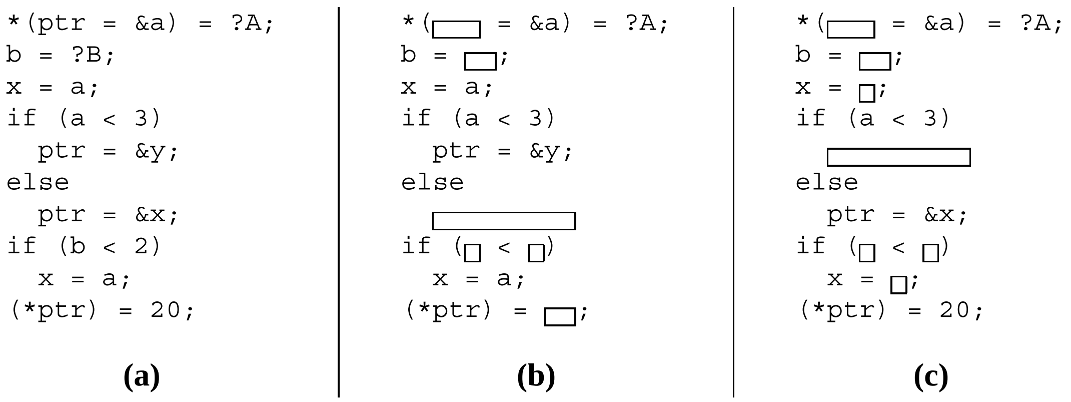
Field et al. (see Chapter 5) use an intermediate representation for imperative programs named PIM [55] as a basis for their slicing approach. Both the translation of a program to its PIM representation, and subsequent optimizations of PIM graphs are defined by an equational logic, which can be implemented by term rewriting [95] or graph rewriting [17]. Correspondences between the source text of a program, its initial PIM graph, and the subsequently derived optimized PIM graph are automatically maintained by a technique called *dynamic dependence tracking* (see Chapter 4). This technique, which is defined for arbitrary term rewriting systems, keeps track of the way in which new function symbols that are dynamically created in a rewriting process are *dependent* upon symbols that were previously present. These source correspondences are stored in PIM graphs as annotations of function symbols; in a sense this is similar to the way information is stored in the Reduced Dynamic Dependence Graphs of Agrawal et al. [6] (see Section 3.4.1.3). Extracting a slice with respect to a designated expression involves maintaining a pointer to the PIM-subgraph for that expression, and retrieving the dynamic dependence information stored in that PIM-subgraph. For details as to how this is accomplished, the reader is referred to Chapter 5.

Both PIM and dynamic dependence tracking have been implemented using the ASF+SDF Meta-environment, a programming environment generator [93] developed at CWI. Recent experiments have produced promising results. In particular, the (accurate) slices of Figures 3.24 **(b)** and 3.25 **(c)** have been computed. Recently, Tip has shown that dynamic dependence tracking can also be used to compute accurate dynamic slices from a simple algebraic specification [23] that specifies an interpreter (see Chapter 6).

Ernst [52] uses the Value Dependence Graph (VDG) [143] as an intermediate representation for his slicing technique. The nodes of a VDG correspond to computations, and the edges represent values that flow between computations. The most prominent characteristics of VDGs are: (i) control flow is represented as data flow, (ii) loops are modeled by recursive function calls, and (iii) all values and computations in a program, including operations on the heap and on I/O streams, are explicitly represented in the VDG. The transformation/optimization of VDGs is discussed in some detail in [143]. Ernst refers to the problem of maintaining a correspondence between the VDG and the source code graph throughout the optimization process, but no details are presented as to how this is accomplished. For the extraction of slices from a VDG, Ernst uses a simple and efficient graph reachability algorithm similar to the one used by Ottenstein and Ottenstein [120].

We are currently unable to provide an in-depth comparison of the approaches by Field et al. and by Ernst due to the elaborate optimizations that are involved, and the absence of any information regarding the "source correspondences" used by Ernst. A few differences between these works are obvious, however:

- The language considered by Ernst is substantially larger than the one studied in Chapter 5. Ernst has implemented a slicer for the full C language (including recursive procedures—see Section 3.3.2.3), whereas Field et al. do not (yet) address the problems posed by procedures and unstructured control flow.
- The approach by Field et al. permits the use of a number of variations of the PIM logic for treating loops, corresponding to different "degrees of laziness" in the language's semantics. Depending on the selected option, the computed slices will resemble the "non-executable" slices computed by Agrawal and Horgan [6], or the "executable" slices computed by Korel and Laski [100]. It is unclear from Ernst's paper if his approach provides the same degree of flexibility.
- Field et al. permit slices to be computed given any set of constraints on a program's inputs, and define the corresponding notion of a *constrained* slice, which subsumes the traditional concepts of static and dynamic slices. This is accomplished by rewriting PIM graphs that contain variables (corresponding to unknown values) according to PIM-rules that model symbolic execution. Ernst does not discuss a similar capability of his slicer.
- Field et al. define slices as a *subcontext* of (i.e., a "connected" set of function symbols in) a program's AST. Statements or expressions of the program that do not occur in the slice are represented by "holes" (i.e., missing subterms) in a context. Although this notion of a slice does not constitute an executable program in the traditional sense, the resulting slices *are* executable in the sense that such as slice can be rewritten to a PIM graph containing the same value for the expression specified in the slicing criterion, given the same constraints on the program's inputs.



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

(a)

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

(b)

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

(c)

Figure 3.26: **(a)** An example program. **(b)** Constrained slice with respect to the final value of `x` given the constraint `?A` := 2. **(c)** Conditional constrained slice with respect to the final value of `x` given the constraint `?A` $>$ 5.

Figure 3.26 shows an example program (taken from Chapter 5), and some constrained slices of it obtained using the approach by Field et al.[27]. The intuition behind these slices is quite simple: a "boxed" expression in a slice may be replaced by any other expression without affecting the computation of the value specified in the slicing criterion, given the specified constraints on the program's inputs. Although absurdly contrived, the example illustrates several important points. By not insisting that a slice be a syntactically valid program, distinctions can be made between assignment statements whose R-values are included but whose L-values are excluded and vice versa, as Figure 3.26 **(b)** shows. Observe that it is possible to determine that the values tested in a conditional are irrelevant to the slice, even though the body is relevant. In general, this permits a variety of fine distinctions to be made that traditional slicing algorithms cannot.

3.7 Conclusions

We have presented a survey of the static and dynamic slicing techniques that can be found in the present literature. As a basis for classifying slicing techniques we have used the computation method, and a variety of programming language features such as procedures, unstructured control flow, composite variables/pointers, and concurrency. Essentially, the problem of slicing in the presence of one of these features is "orthogonal" to solutions for each of the other features. For dynamic slicing methods, an additional issue is the fact whether

[27] In this figure, expressions that begin with a question mark, e.g., '`?A`', represent unknown values or inputs. Subterms of the program's AST that do not occur in the slices of Figure 3.26 **(b)** and **(c)** are replaced by a box.

or not the computed slices are *executable* programs that capture a part of the program's behavior. Wherever possible, different solutions to the same problem were compared by applying each algorithm to the same example program. In addition, the possibilities and problems associated with the integration of solutions for "orthogonal" language features were discussed.

3.7.1 Static slicing algorithms

In Section 3.3.6, algorithms for static slicing were compared and classified. Besides listing the specific slicing problems studied in the literature, we have compared the *accuracy* and, to some extent, the *efficiency* of static slicing algorithms. The most significant conclusions of Section 3.3.6 can be summarized as follows:

basic algorithms. For *intra*procedural static slicing in the absence of procedures, unstructured control flow, composite data types and pointers, and concurrency, the accuracy of methods based on dataflow equations [147], information-flow relations [22], and program dependence graphs [120] is essentially the same. PDG-based algorithms have the advantage that dataflow analysis has to be performed only once; after that, slices can be extracted in linear time. This is especially useful when several slices of the same program are required.

procedures. The first solution for *inter*procedural static slicing, presented by Weiser [147], is inaccurate for two reasons. First, this algorithm does not use exact dependence relations between input and output parameters. Second, the call-return structure of execution paths is not taken into account. The solution by Bergeretti and Carré [22] does not compute truly interprocedural slices because no procedures other than the main program are sliced. Moreover, the approach by Bergeretti and Carré is not sufficiently general to handle recursion. Exact solutions to the interprocedural static slicing problem have been presented by Hwang, Du, and Chou [79], Reps, Horwitz and Binkley [77], Reps, Horwitz, Sagiv, and Rosay [129, 128], Jackson and Rollins [82], and Ernst [52]. The Reps-Horwitz-Sagiv-Rosay algorithm for interprocedural static slicing is the most efficient of these algorithms. Binkley studied the issues of determining executable interprocedural slices [30], and of interprocedural static slicing in the presence of parameter aliasing [31].

unstructured control flow. Lyle was the first to present an algorithm for static slicing in the presence of unstructured control flow [108]. The solution he presents is conservative: it may include more **goto** statements than necessary. Agrawal [3] has shown that the solutions proposed by Gallagher and Lyle [60, 61] and by Jiang et al. [83] are incorrect. Precise solutions for static slicing in the presence of unstructured control flow have been proposed by Ball and Horwitz [12, 13], Choi and Ferrante [37], and Agrawal [3]. It is not clear how the efficiency of these algorithms compares.

composite data types/pointers. Lyle [108] presented a conservative algorithm for static slicing in the presence of arrays. The algorithm proposed by Jiang et al. in [83] is incorrect. Lyle and Binkley [110] presented an algorithm for static slicing in the presence of pointers, but only for straight-line code. Agrawal, DeMillo, and Spafford [4] propose a PDG-based algorithm for static slicing in the presence of composite variables and pointers.

concurrency. The only approach for static slicing of concurrent programs was proposed by Cheng [35]. Unfortunately, Cheng has not provided a justification of the correctness of his algorithm.

3.7.2 Dynamic slicing algorithms

Algorithms for dynamic slicing were compared and classified in Section 3.4.5. Due to differences in computation methods and dependence graph representations, the potential for integration of the dynamic slicing solutions for "orthogonal" dimensions is less clear than in the static case. The conclusions of Section 3.4.5 may be summarized as follows:

basic algorithms. Methods for *intra*procedural dynamic slicing in the absence of procedures, composite data types and pointers, and concurrency were proposed by Korel and Laski [99, 100], Agrawal and Horgan [6], and Gopal [62]. The slices determined by the Agrawal-Horgan algorithm and the Gopal algorithm are more accurate than the slices computed by the Korel-Laski algorithm, because Korel and Laski insist that their slices be executable programs. The Korel-Laski algorithm and Gopal's algorithm require an amount of space proportional to the number of statements that was executed because the entire execution history of the program has to be stored. Since slices are computed by traversing this history, the amount of time needed to compute a slice depends on the number of executed statements. A similar statement can be made for the flowback analysis algorithm by Choi et al. [116, 38]. The algorithm proposed by Agrawal and Horgan based on Reduced Dynamic Dependence Graphs requires at most $O(2^n)$ space, where n is the number of statements in the program. However, the time needed by the Agrawal-Horgan algorithm also depends on the number of executed statements because for each executed statement, the dependence graph may have to be updated.

procedures. Two dependence graph based algorithms for interprocedural dynamic slicing were proposed by Agrawal, DeMillo, and Spafford [4], and by Kamkar, Shahmehri, and Fritzson [90, 89]. The former method relies heavily on the use of memory cells as a basis for computing dynamic reaching definitions. Various procedure-passing mechanisms can be modeled easily by assignments of actual to formal and formal to actual parameters at the appropriate moments. The latter method is also expressed as a reachability problem in a (summary) graph. However, there are a number of differences with the approach of [4]. First, parts of the graph can be constructed at compile-time. This is more efficient, especially in cases where many calls to the same procedure occur. Second, Kamkar et al.

study procedure-level slices; that is, slices consisting of a set of procedure calls rather than a set of statements. Third, the size of a summary graph depends on the number of executed procedure calls, whereas the graphs of Agrawal et al. are more space efficient due to “fusion” of vertices with the same transitive dependences. It is unclear if one algorithm produces more precise slices than the other.

unstructured control flow. As far as we know, dynamic slicing in the presence of unstructured control flow has not been studied yet. However, it is our conjecture that the solutions for the static case [12, 13, 3, 37] may be adapted for dynamic slicing.

composite data types/pointers. Two approaches for dynamic slicing in the presence of composite data types and pointers were proposed, by Korel and Laski [100], and Agrawal, DeMillo, and Spafford [4]. The algorithms differ in their computation method: dynamic flow concepts vs. dependence graphs, and in the way composite data types and pointers are represented. Korel and Laski treat components of composite data types as distinct variables, and invent names for dynamically allocated objects and pointers whereas Agrawal, DeMillo, and Spafford base their definitions on definitions and uses of memory cells. It is unclear how the accuracy of these algorithms compares. The time and space requirements of both algorithms are essentially the same as in the case where only scalar variables occur.

concurrency. Several methods for dynamic slicing of distributed programs have been proposed. Korel and Ferguson [98] and Duesterwald, Gupta, and Soffa [50] compute slices that are executable programs, but have a different way of dealing with nondeterminism in distributed programs: the former approach requires a mechanism for replaying the rendezvous in the slice in the same relative order as they occurred in the original program whereas the latter approach replaces nondeterministic communication statements in the program by deterministic communication statements in the slice. Cheng [35] and Choi et al. [116, 38] do not consider this problem because the slices they compute are not executable programs. Duesterwald, Gupta, and Soffa [50] and Cheng [35] use *static* dependence graphs for computing *dynamic* slices. Although this is more space-efficient than the other approaches, the computed slices will be inaccurate (see the discussion in Section 3.4.1.1). The algorithms by Korel and Ferguson and by Choi et al. both require an amount of space that depends on the number of executed statements. Korel and Ferguson require their slices to be executable; therefore these slices will contain more statements than those computed by the algorithm of [116, 38].

3.7.3 Applications

Weiser [144] originally conceived of program slices as a model of the mental abstractions made by programmers when debugging a program, and advocated the use of slicing in debugging tools. The use of slicing for (automated) debugging was further explored by Lyle and Weiser [109], Choi et al. [38], Agrawal et al. [5], Fritzson et al. [59], and Pan and

Spafford [121, 122]. Slicing has also proven to be of use for a variety of other applications including: parallelization [146], program differencing and integration [70, 74], software maintenance [61], testing [51, 88, 65, 20], reverse engineering [21, 82, 81], and compiler tuning [106]. Section 3.5 contains a detailed overview of how slicing is used in each of these application areas.

3.7.4 Recent developments

Two important characteristics of conventional slicing algorithms adversely affect the accuracy of program slices:

- The fact that slices consist of a subset of the original program's statements, sometimes with the additional constraint that a slice must be a syntactically valid program.
- The fact that slices are computed by tracing data and control dependences.

Section 3.6 discusses recent work by Field, Ramalingam, and Tip (see Chapter 5) and by Ernst [52] for computing more accurate slices, where these "restrictions" are removed. In essence, these slicing algorithms compute more accurate slices due to the use of compiler-optimization techniques, symbolic execution, and a variety of semantics-preserving transformations for eliminating spurious dependences. At the conceptual level, the algorithms by Field et al. and Ernst consist of the following components:

- Translation of a program to a suitable intermediate representation (IR).
- Transformation and optimization of the IR.
- Maintaining a mapping between the source text, the original IR, and the optimized IR.
- Extraction of slices from the IR.

Although Field et al. and Ernst have reported promising results, much work remains to be done in this area.

Acknowledgements

Tom Reps provided the program and picture of Figure 3.12. Susan Horwitz provided the program of Figure 3.15. The programs shown in Figures 3.2 and 3.18 are adaptations of example programs in [2].

Chapter 4

Dynamic Dependence Tracking

(joint work with John Field)

Summary

Program slicing is a useful technique for debugging, testing, and analyzing programs. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. With rare exceptions, program slices have hitherto been computed and defined in ad-hoc and language-specific ways. The principal contribution of this chapter is to show that general and semantically well-founded notions of slicing and *dependence* can be derived in a simple, uniform way from *term rewriting systems* (TRSs). Our slicing technique is applicable to any language whose semantics is specified in TRS form. Moreover, we show that our method admits an efficient implementation.

In Chapter 5, dynamic dependence tracking is "applied to" PIM, an intermediate representation for imperative programs with an accompanying equational logic (which is implemented by rewriting). It will be shown that this permits the computation of various types of highly accurate program slices. Chapter 6 describes how the application of dynamic dependence tracking to algebraic specifications of interpreters yields a useful notion of dynamic program slicing in that context.

4.1 Introduction

4.1.1 Overview

Program slicing is a useful technique for debugging, testing, and analyzing programs. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as the *slicing criterion*. As originally defined by Weiser [147], a slicing criterion was the value of a variable at a particular program point and a slice consisted of an "executable" subset of the program's original statements. Numerous variations on the notion of slice have since been proposed, as well as many different techniques to compute them (see Chapter 3), but all reduce to determining *dependence* relations among program components. Unfortunately, with rare exceptions, the notion of "dependence" has been defined in an ad-hoc and language-specific manner, resulting in algorithms for computing slices that are notoriously difficult to understand, especially in the presence of pointers, procedures, and unstructured control flow. The contributions of this chapter are as follows:

- We define a general notion of slice that applies to any unconditional term rewriting system (TRS). Our definition uses a relation on *contexts* derived from the reduction relation on terms. This relation makes precise the *dynamic dependence* of function symbols in terms in a reduction sequence on symbols in previous terms in that sequence. Our notion of dependence does not require labeled terms [27, 28, 111, 112], and is distinguished by its ability to treat (normally problematic) TRSs with left-nonlinear rules.
- Our notion of slicing subsumes most of those defined in previous work on program slicing. The distinction traditionally made between "static" and "dynamic" slicing can be modeled by reduction of open or closed terms, respectively. Partial instantiation of open terms yields a useful intermediate notion of *constrained* slicing. Although Venkatesh defines a similar notion abstractly [137], he does not indicate how to *compute* such slices.
- We describe an algorithm by which slices can be efficiently computed in practice by systematically transforming the original TRS to gather dependence information. The overhead required to compute this information is linear in the size of the initial term. This algorithm produces minimal slices for left-linear systems, and sound (but not always minimal) slices for left-nonlinear systems.

Finally, for the case of left-linear systems, we present proofs that our definitions yield *minimal* and *sound* slices.

In Chapter 5, we will show how our techniques can be applied to standard programming languages, and compare these techniques to other algorithms in the literature. Chapter 6 discusses how the dynamic dependence relation defined in this chapter can be used for providing dynamic slicing facilities in generated source-level debugging tools. Here, we will concentrate primarily on technical foundations.

4.1.2 Motivating examples

Consider the program in Figure 4.1 **(a)** below, written in a tiny imperative programming language, **P**. The semantics of **P** are similar to those of many imperative programming languages with pointers. A **do** construct is executed by evaluating its statement list, and using the computed values to evaluate its **in** expression. Expressions of the form '$\overline{\mathtt{x}}$' are atoms, and play the dual role of basic values and addresses that may be assigned to using ':='. Addresses are explicitly dereferenced using '$\uparrow$'. The distinguished atoms $\overline{\mathtt{t}}$ and $\overline{\mathtt{f}}$ represent boolean values.

We evaluate **P** programs by applying the *rewriting rules* of Figure 4.2 to the *term* consisting of the program's syntax tree until no further rules are applicable. This reduction process produces a sequence of terms ending with a *normal form* that denotes the result of the evaluation. The program in Figure 4.1 **(a)** reduces to the normal form '**result** $\overline{\mathtt{t}}$'.

```
program                                          program
do $\overline{\mathtt{x}} := \overline{\mathtt{a}}$; $\overline{\mathtt{w}} := \overline{\mathtt{x}}$; $\overline{\mathtt{z}} := \overline{\mathtt{b}}$;          do $\overline{\mathtt{x}} := \overline{\bullet}$; $\overline{\mathtt{w}} := \overline{\mathtt{x}}$; $\overline{\mathtt{z}} := \bullet$;
    if $\overline{\mathtt{w}}\uparrow\uparrow = \overline{\mathtt{x}}\uparrow$                                 if $\overline{\mathtt{w}}\uparrow\uparrow = \overline{\mathtt{x}}\uparrow$
        then $\overline{\mathtt{y}} := \overline{\mathtt{x}}\uparrow$                                then $\overline{\mathtt{y}} := \overline{\mathtt{x}}\uparrow$
        else $\overline{\mathtt{y}} := \overline{\mathtt{b}}$                                  else $\bullet$
in $\overline{\mathtt{y}}\uparrow = \overline{\mathtt{x}}\uparrow$                                    in $\overline{\mathtt{y}}\uparrow = \overline{\mathtt{x}}\uparrow$

                (a)                                              (b)
```

Figure 4.1: **(a)** Example **P** Program. **(b)** Minimal slice with respect to the entire normal form of **(a)**.

[P1]	$\overline{\mathtt{X}} = \overline{\mathtt{X}}$	$\rightarrow$	$\overline{\mathtt{t}}$	
[P2]	$\overline{\mathtt{a}} = \overline{\mathtt{b}}$	$\rightarrow$	$\overline{\mathtt{f}}$	for all constants a, b such that $a \neq b$
[P3]	**if** $\overline{\mathtt{t}}$ **then** X **else** Y	$\rightarrow$	X	
[P4]	**if** $\overline{\mathtt{f}}$ **then** X **else** Y	$\rightarrow$	Y	
[P5]	**do** X **in** $\overline{\mathtt{Y}}$	$\rightarrow$	$\overline{\mathtt{Y}}$	
[P6]	**do** X **in** $Y = Z$	$\rightarrow$	$($**do** X **in** $Y) = ($**do** X **in** $Z)$	
[P7]	**do** $X; A := E$ **in** $(B\uparrow)$	$\rightarrow$	**if** $($**do** $X; A := E$ **in** $B) = ($**do** X **in** $A)$	
			then $($**do** X **in** $E)$	
			else do X **in** $(($**do** $A := E$ **in** $B)\uparrow)$	
[P8]	**do** $A := E$ **in** $(B\uparrow)$	$\rightarrow$	**if** $($**do** $A := E$ **in** $B) = A$	
			then E	
			else $(($**do** $A := E$ **in** $B)\uparrow)$	
[P9]	**do** $X;$ **if** A **then** B **else** C **in** E	$\rightarrow$	**if** $($**do** X **in** $A)$	
			then $($**do** $X; B$ **in** $E)$	
			else $($**do** $X; C$ **in** $E)$	
[P10]	**do if** A **then** B **else** C **in** E	$\rightarrow$	**if** A **then** $($**do** B **in** $E)$ **else** $($**do** C **in** $E)$	
[P11]	**do** X **in if** A **then** B **else** C	$\rightarrow$	**if** $($**do** X **in** $A)$	
			then $($**do** X **in** $B)$	
			$($**do** X **in** $C)$	
[P12]	**program** $\overline{\mathtt{X}}$	$\rightarrow$	**result** $\overline{\mathtt{X}}$	

Figure 4.2: Rewriting Semantics of **P**.

[B1]	$X \wedge (Y \oplus Z) \longrightarrow (X \wedge Y) \oplus (X \wedge Z)$	**[B3]**	$X \wedge \mathtt{ff} \longrightarrow \mathtt{ff}$
[B2]	$X \wedge \mathtt{tt} \longrightarrow X$	**[B4]**	$X \oplus X \longrightarrow \mathtt{ff}$

Figure 4.3: Boolean TRS **B**.

$$\underline{\mathtt{ff} \wedge (\mathtt{tt} \oplus \mathtt{tt})} \equiv T_0 \xrightarrow{[B1]} (\underline{\mathtt{ff} \wedge \mathtt{tt}}) \oplus (\mathtt{ff} \wedge \mathtt{tt}) \equiv T_1 \xrightarrow{[B2]} \mathtt{ff} \oplus (\underline{\mathtt{ff} \wedge \mathtt{tt}}) \equiv T_2$$
$$\xrightarrow{[B2]} \underline{(\mathtt{ff} \oplus \mathtt{ff})} \equiv T_3 \xrightarrow{[B4]} \mathtt{ff} \equiv T_4$$

Figure 4.4: A **B**-reduction; redexes are underlined.

Figure 4.1 **(b)** depicts the slice of the example program with respect to this normal form. The symbol '●' represents subterms of the program that do not affect its result.

It should be clear that a program slice is valuable for understanding which program components depend critically on the slicing criterion—even in the small example of Figure 4.1, this is not immediately obvious. Slicing information can be used to determine what statements might have to be changed in order to correct an error or to alter the value of the criterion. The techniques we describe also allow the programmer the option of binding various inputs to values or leaving them undefined, allowing the effects of various initial conditions to be precisely traced. This significant capability is unique to our approach, and derives from its generality. In addition, by defining different (TRS-based) semantics for the same language, different sorts of slices can be derived. For instance, by using variants of the semantics in [54], we can compute both traditional "static" and "dynamic" (see Chapter 3 for a more thorough discussion of the distinction between these notions) slices for the same language.

We believe that our notion of a slice should also prove useful as an adjunct to theorem-proving systems, since it yields certain universally quantified equations from derivations of equations on closed terms. Consider, for example, the simple TRS **B** in Figure 4.3, which defines a few boolean identities ('$\wedge$' denotes conjunction, '$\oplus$' exclusive-or). Figure 4.4 shows how **B**-term $\mathtt{ff} \wedge (\mathtt{tt} \oplus \mathtt{tt})$ can be reduced to $\mathtt{ff}$. Observe that in deriving the theorem $\mathtt{ff} \wedge (\mathtt{tt} \oplus \mathtt{tt}) = \mathtt{ff}$, we actually derive the *more general* theorem $P \wedge (\mathtt{tt} \oplus \mathtt{tt}) = \mathtt{ff}$, for arbitrary P. From the point of view of slicing, the slice with respect to the normal form $\mathtt{ff}$ is the subcontext $\bullet \wedge (\mathtt{tt} \oplus \mathtt{tt})$ of the initial term. To determine such a slice, we must pay careful attention to the behavior of *left-nonlinear* rules such as **[B4]** and **[P1]**, which many authors on reduction-theoretic properties of TRSs do not treat. In the sequel, we show how slices can be obtained by examining the manner in which rules *create* new function symbols, and *residuate*, or "move around" old ones.

4.1.3 Definition of a slice

In general, we will define a slice as a certain *context* contained in the initial term of some reduction. Intuitively, a context may be viewed as a connected (in the sense of a tree) subset of function symbols taken from a term. For instance, if $f(g(a, b),\ c) \equiv T$ is a term, then one of several contexts contained in T is $g(\bullet, b) \equiv C$. C contains an omitted subterm, or *hole*[1], denoted by '$\bullet$'. This hole results from deleting the subterm 'a' of T. We denote the fact that C is a subcontext of T by $C \sqsubseteq T$; contexts as well as terms may contain subcontexts.

In a slice, holes denote subterms that are irrelevant to the computation of the criterion. Figure 4.1 **(b)** depicts the *minimal subcontext of the original program that yields the slicing criterion via a "subreduction" of the original reduction*. Informally, the holes in the slice could be replaced by *any* **P**-expression and the same criterion could be produced by a **P**-reduction.

Definition 4.1 below makes precise our notion of slice. We will formalize the notion of "subreduction" of a sequence of reduction steps ρ using a set *Project*$*^{\rho}$, which is a collection of triples of the form $\langle C, \rho', C' \rangle$. Informally, such a triple denotes the fact that context C reduces to a context that is isomorphic to C' by a reduction ρ' derived from rule applications that also occur in ρ. We discuss *Project*$*^{\rho}$ further in Section 4.5. Two contexts are isomorphic if they have the same "structure" (though they may appear at different locations). This notion will be formalized in Section 4.2.

Definition 4.1 (Slice) *Let $\rho : T \longrightarrow^* T'$ be a reduction. Then a* <u>*slice*</u> *with respect to a subcontext C' of T' is a subcontext C of T with the property that there exists a reduction ρ' such that (i) $\rho' : C \longrightarrow^* D'$ for some $D' \sqsupseteq E'$, (ii) E' and C' are isomorphic, and (iii) $\langle C, \rho', D' \rangle \in$ Project$*^{\rho}$. Slice C is* <u>*minimal*</u> *if there is no slice with respect to criterion C' that contains fewer function symbols.*

Definition 4.1 is rendered pictorially in Figure 4.5.

The notion of TRS-based slice we define in the sequel can be used for any language whose operational semantics is defined by a TRS. Many languages whose semantics are traditionally defined via extended lambda-calculi or using structural operational semantics also have corresponding rewriting semantics [1, 54]. In [55], it is shown how many traditional program constructs may be modeled by an appropriate TRS.

4.1.4 Relation to origin tracking

At this point, the reader might wonder why any additional machinery is required beyond the origin relation that was defined in Chapter 2. Unfortunately, the origin relation does not provide information that is appropriate for computing program slices. The main reasons for this being the case are:

- The origin relation was designed with a different objective in mind: to trace recurrences of the "same" subterm in a term rewriting process. This notion is too "weak" for

[1]Some authors require that contexts contain exactly one hole; we will not.

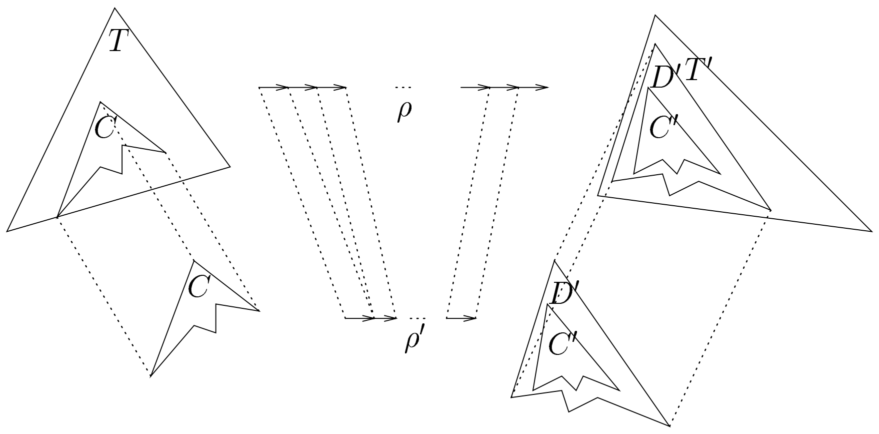


Figure 4.5: Depiction of the definition of a slice.

computing slices, where one needs to determine those parts of the initial term that are necessary for producing some designated subterm. The problem is that the slice (i.e., the parts of the initial term are not necessarily the same as (or even similar in structure to) the slicing criterion.

- A related issue is that not every subterm has a non-empty origin. In fact, all subterms that are "created" by the rewriting process have empty origins. It is evident that any notion of dependence in a rewriting process should take into account the creation of new function symbols as well as the "residuation" of symbols that occurred previously; otherwise, "empty" origins would occur frequently.

In Section 7.3, the connections between the origin relation and the dynamic dependence relation are discussed at somewhat greater length.

4.2 Basic definitions

In this section, we make precise the notion of a *context* introduced informally in the previous section. This notion will be the cornerstone of our formalization of slicing and dependence. Instead of deriving contexts from the usual definition of a term, we view terms as a special class of contexts. Contexts will be defined as connected fragments of *trees* decorated with function symbols and variables. We begin with a few preliminary definitions, most of which are standard.

4.2.1 Signatures, paths, context domains

A *signature* Σ is a finite set of *function symbols*; associated with each function symbol $f \in \Sigma$ is a natural number $arity(f)$, its number of arguments. We will assume the existence of a denumerable set of *variables* $\mathcal{V}$ such that $\Sigma \cap \mathcal{V} = \emptyset$. By convention, for each variable $X \in \mathcal{V}$, $arity(X) = 0$. Lower-case letters of the form $f, g, h, \cdots$ will denote function symbols and upper-case letters of the form $X, Y, Z, \cdots$ will represent variables.

A *path* is a sequence of positive integers that designates a particular function symbol or subtree by encoding a walk from the tree's root. The empty path, '$()$', designates the root of a tree; path $(i_1\, i_2 \cdots i_m)$ designates the i_m^{th} subtree (counted from left to right) of the subtree indicated by path $(i_1\, i_2 \cdots i_{(m-1)})$. The operation '$\cdot$' denotes path concatenation. Path p is a *prefix* of path q, denoted by $p \preceq q$, if there exists an r such that $q = p \cdot r$; if $r \neq ()$ then $p \prec q$.

A *context domain* P is a set of paths designating a connected fragment of a tree. This means that P must (i) possess a unique root, *root*(P), such that for all $p \in P$, *root*$(P) \preceq p$, and (ii) have no "gaps," i.e., for all p, q, r such that $p \prec q \prec r$ and $p, r \in P$ it must be the case that $q \in P$.

4.2.2 Contexts

We can now define a context as a total mapping from a context domain to function symbols and variables:

Definition 4.2 (Context) *Let Σ be a signature, $\mathcal{V}$ be a set of variables, and $\mathcal{P}$ be a context domain. Let μ be a total mapping from $\mathcal{P}$ to $(\Sigma \cup \mathcal{V})$ and p be a path. Then a pair $\langle p, \mu \rangle$ is a $\Sigma\mathcal{V}$-<u>context</u> if and only if:*

(i) *For all $q \in \mathcal{P}$ and $s \in \Sigma \cup \mathcal{V}$ such that $\mu(q) = s$, $q \cdot i \in \mathcal{P}$ for some i implies that $i \leq$ arity(s).*
(ii) *If $\mathcal{P} \neq \emptyset$, then $p =$ root$(\mathcal{P})$.*

Clause (i) of Definition 4.2 ensures that every child of a function symbol f must have an ordinal number less than or equal to the arity of f. Clause (ii) ensures that the root of the context is the same as the root of its underlying domain, except when the domain is empty; in the latter case, we will say that the context is *empty*. The definition is specifically designed to admit empty contexts, which will be important in the sequel for describing the behavior of *collapse rules*, i.e., rewriting rules whose right hand sides are single variables. Given context $C \equiv \langle p, \mu \rangle$, *root*$(C)$ denotes the path p, and $\mathcal{O}(C)$ the domain of μ. The path corresponding to a "missing child" in a context will be referred to as a *hole occurrence*; an empty context is also a hole. The set of hole occurrences in a context C will be denoted by $\mathcal{O}_\bullet(C)$. We will use $Cont(\Sigma, \mathcal{V})$ to denote the set of all $\Sigma\mathcal{V}$-contexts.

For any context C and a path p, $p \leftarrow C$ denotes an isomorphic context rooted at p obtained by *rerooting* C. This notation is used to represent contexts textually; e.g., $p \leftarrow f(\bullet, g(a, \bullet))$ represents a context rooted at p with two holes ('$\bullet$'), binary function symbols f and g and a constant a. $p \leftarrow \bullet$ represents an empty context rooted at p. We will say that contexts C and D are *isomorphic*, written $C \doteq D$, if $((\ ) \leftarrow C) \equiv ((\ ) \leftarrow D)$

A context C is a *term* if: (i) C has no hole occurrences, and (ii) *root*$(C) = ()$. Although the restriction of *root*(C) to be $()$ is not strictly necessary, it results in a definition that agrees most closely with that used by other authors. We will use $Term(\Sigma)$ to denote the set of terms over signature Σ. Letters $C, D, \cdots$ will generally denote arbitrary contexts, and $S, T, \cdots$ terms. Whenever convenient, we ignore the distinction between a variable X and the term consisting of that variable. Some convenient operations on contexts are introduced next.

For a context C, and $\mathcal{S}$ a subset of $\Sigma \cup \mathcal{V}$, $\mathcal{O}_{\mathcal{S}}(C)$ denotes the set of paths to elements of $\mathcal{S}$ in C; $\mathcal{O}_{\{s\}}(C)$ is abbreviated by $\mathcal{O}_s(C)$. The set of variable occurrences in a $\Sigma\mathcal{V}$-context C, i.e., $\mathcal{O}_{\mathcal{V}}(C)$, is denoted $\textsc{Vars}(C)$, and *Vars*$_1(C)$ is the set of variables that occur exactly once in C.

Two contexts are *compatible* if all paths common to both of their domains are mapped to the same symbol. If C and D are compatible, C is a *subcontext* of D, denoted by $C \sqsubseteq D$, if and only if one of the following holds: (i) C and D are nonempty and $\mathcal{O}(C) \subseteq \mathcal{O}(D)$, (ii) C and D are empty and $C \equiv D$, or (iii) C is empty, D is nonempty, *root*$(C) = q \cdot i \in \mathcal{O}(D)$, and $q \in \mathcal{O}(D)$. The third clause states that an empty context C is a subcontext of a nonempty context D *only* if its root is “sandwiched” between adjacent nodes in D. This property will greatly simplify a number of definitions in the sequel. Contexts D and E are *disjoint* if and only if there exists no context C such that $C \sqsubseteq D$ and $C \sqsubseteq E$. If C and D are contexts such that *root*$(D) \in (\mathcal{O}(C) \cup \mathcal{O}_\bullet(C))$, $C[D]$ denotes the context C where the subcontext or hole at *root*(D) is *replaced* by D. Note that for all $C \sqsubseteq D$, $D[C] \equiv D$. A context C is *elementary* iff $|\mathcal{O}(C)| = 1$.

A context *forest* is a set of mutually disjoint contexts. Forest $\mathcal{S}$ is a *subforest* of forest $\mathcal{T}$, denoted $\mathcal{S} \sqsubseteq \mathcal{T}$, if and only if for all contexts $C \in \mathcal{S}$, there exists a context $D \in \mathcal{T}$ such that $C \sqsubseteq D$. Some convenient set-like operations on context forests can be defined as follows: Let $\mathcal{S}$ and $\mathcal{T}$ be compatible context forests. Then their *union*, denoted by $\mathcal{S} \sqcup \mathcal{T}$, is the smallest forest $\mathcal{U}$ such that $\mathcal{S} \sqsubseteq \mathcal{U}$ and $\mathcal{T} \sqsubseteq \mathcal{U}$; their *difference*, denoted $\mathcal{S} - \mathcal{T}$, is the smallest forest $\mathcal{U}$ such that $\mathcal{U} \sqsubseteq \mathcal{S}$ and $\mathcal{S} \sqsubseteq (\mathcal{T} \sqcup \mathcal{U})$. If $\mathcal{P}$ is a set of paths, $C \,/\, P$ is the forest containing subcontexts of C rooted at paths in $\mathcal{P}$. The notion of context replacement is easily generalized to a forest $\mathcal{S}$. In the sequel, we allow simple contexts to be used as operands of context forest operations; such contexts are coerced to singleton forests. For example, ‘$C \sqcup D$’ denotes ‘$\{ C \} \sqcup \{ D \}$’.

4.3 Term rewriting and related relations

In this section, we formalize standard term rewriting-related notions using operations on contexts; we then define the important related ideas of *creation* and *residuation*, which are derived from the rewriting relation. We will first consider only *left-linear* TRSs; this restriction will be removed in in Section 4.7.

4.3.1 Substitutions and term rewriting systems

A *substitution* is a finite partial map from $\mathcal{V}$ to $Cont(\Sigma, \mathcal{V})$, where Σ is a signature and $\mathcal{V}$ a set of variables. A substitution σ is extended to a mapping on contexts by replacing each subcontext $C_X \sqsubseteq C$ consisting solely of a variable X by the context (*root*$(C_X) \leftarrow \sigma(X)$), for all X on which σ is defined. A *term rewriting system* $\mathcal{R}$ over a signature Σ is a set of pairs $\langle L, R\rangle$ such that L and R are terms over Σ, L does not consist of a sole variable, and $\textsc{Vars}(R) \subseteq \textsc{Vars}(L)$; $\langle L, R\rangle$ is called a *rewrite rule* and is commonly denoted by $L \to R$. For $\alpha \equiv L \to R \in \mathcal{R}$ we define $L_\alpha = L$ and $R_\alpha = R$. A rewrite rule α is *left-linear* if $\textsc{Vars}(L_\alpha) = \mathit{Vars}_1(L_\alpha)$. If $\mathcal{R}$ is a TRS, then we define an *$\mathcal{R}$-contraction* $\mathcal{A}$ to be a triple $\langle p, \alpha, \sigma\rangle$, where p is a path, α is a rule of $\mathcal{R}$, and σ is a substitution.

We use $p_\mathcal{A}$, $\alpha_\mathcal{A}$, $L_\mathcal{A}$, $R_\mathcal{A}$, and $\sigma_\mathcal{A}$ to denote p, α, $L_{\alpha_\mathcal{A}}$, $R_{\alpha_\mathcal{A}}$, and σ, respectively. Moreover, $\overline{L_\mathcal{A}}$ and $\overline{R_\mathcal{A}}$ will denote the contexts $(p_\mathcal{A} \leftarrow L_\mathcal{A})$ and $(p_\mathcal{A} \leftarrow R_\mathcal{A})$, respectively. The *$\mathcal{R}$-contraction relation*, $\longrightarrow_\mathcal{R}$, is defined by requiring that $T \longrightarrow_\mathcal{R} T'$ if and only if a contraction $\mathcal{A}$ exists such that $T \equiv T[\sigma_\mathcal{A}(\overline{L_\mathcal{A}})]$ and $T' \equiv T[\sigma_\mathcal{A}(\overline{R_\mathcal{A}})]$, for terms T, T'. The subcontext $\sigma_\mathcal{A}(\overline{L_\mathcal{A}})$ of C is an *$\alpha_\mathcal{A}$-redex*, and the context $\sigma_\mathcal{A}(\overline{R_\mathcal{A}})$ is an *$\alpha_\mathcal{A}$-reduct*; these contexts are abbreviated respectively by $\mathit{Redex}_\mathcal{A}$ and $\mathit{Reduct}_\mathcal{A}$. We will feel free to drop the subscript $\mathcal{R}$ of a contraction $\longrightarrow_\mathcal{R}$ in cases where it is clear which TRS we are referring to, and simply write $\longrightarrow^*$. For clarity, contraction arrows will frequently be labeled explicitly with the contraction $\mathcal{A}$ that is involved: $\xrightarrow{\mathcal{A}}$. As usual, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. A *reduction* ρ is a sequence of contractions $\mathcal{A}_1\mathcal{A}_2 \ldots \mathcal{A}_n$ such that if ρ is nonempty, there exist terms $T_0, T_1, \ldots, T_n$ where:

$$T_0 \xrightarrow{\mathcal{A}_1} T_1 \xrightarrow{\mathcal{A}_2} T_2 \cdots T_{n-1} \xrightarrow{\mathcal{A}_n} T_n$$

This reduction is abbreviated by $\rho : T_0 \longrightarrow^* T_n$. A reduction ρ is a reduction *of* term T if there exists T' such that $\rho : T \longrightarrow^* T'$. The reduction of length 0 is denoted by ϵ; for all terms T, we adopt the convention that $\epsilon : T \longrightarrow^* T$.

Given the definitions above, the **B**-reduction depicted in Figure 4.4 may be described formally by the following sequence of contractions:

$\langle(), [\mathbf{B1}], [X := \mathtt{ff}, Y := \mathtt{tt}, Z := \mathtt{tt}]\rangle$; $\langle(1), [\mathbf{B2}], [X := \mathtt{ff}]\rangle$; $\langle(2), [\mathbf{B2}], [X := \mathtt{ff}]\rangle$; $\langle(), [\mathbf{B4}], [X := \mathtt{ff}]\rangle$

Most of the new relations defined in the sequel are parameterized with a reduction $\rho\mathcal{A}$, in which the final contraction is highlighted. Several definitions are concerned with the last contraction $\mathcal{A}$ only; however, when our definitions are generalized in Section 4.7, the "history" contained in ρ will become relevant. Whenever we define a truly *inductive* relation on $\rho\mathcal{A}$, we will append a '$*$' to the name of the relation.

4.3.2 Context rewriting

In order to generalize term rewriting to context rewriting, a few auxiliary definitions are needed. A *variable instantiation* of a context C is a term T that can be obtained from C by replacing each hole with a variable that does not occur in C. A variable instantiation is a *linear instantiation* if each hole is replaced by a *distinct* variable. A context C rewrites to a context C', denoted $C \longrightarrow^* C'$, if and only if $T \longrightarrow^* T'$, where T is a linear instantiation of C and T' is a variable instantiation of C'. Note that context reduction is *not* defined as the transitive closure of a single-step contraction relation on contexts; this is necessary to correctly account for the way in which a reduction causes distinct holes to be moved and copied, particularly in the case of left-nonlinear rules.

4.3.3 Residuation and creation

In order to formalize our notion of slice, we must first reformulate the standard notion of *residual* and the somewhat less standard notion of *creation* in terms of contexts. Each of these will use Definition 4.3, which formalizes how an application of a contraction $\mathcal{A}$ has the effect of "copying," "moving," or "deleting" contexts bound to variable instances in $L_{\mathcal{A}}$ when $R_{\mathcal{A}}$ is instantiated. The elements of the set $\mathit{VarPairs}^{\rho\mathcal{A}}$ are pairs $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle$ of context forests, such that contexts $C_1 \in \mathcal{S}_1$ and $C_2 \in \mathcal{S}_2$ are corresponding subcontexts of the context bound to some variable in $\alpha_{\mathcal{A}}$.

Definition 4.3 (*VarPairs*) *Let ρA be a reduction. Then*

$$
\begin{aligned}
\mathit{VarPairs}^{\rho\mathcal{A}} \triangleq \{ \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \mid\ & X \in \mathcal{V}, \\
& C \sqsubseteq (() \leftarrow \sigma_{\mathcal{A}}(X)) \ \mathit{or}\ C = (() \leftarrow \bullet), \\
& q = \mathit{root}(C), \\
& \mathcal{S}_1 = \{ (p_L \cdot q \leftarrow C) \mid p_L \in \mathcal{O}_X(\overline{L_{\mathcal{A}}}) \}, \\
& \mathcal{S}_2 = \{ (p_R \cdot q \leftarrow C) \mid p_R \in \mathcal{O}_X(\overline{R_{\mathcal{A}}}) \} \ \}
\end{aligned}
$$

In left-linear systems, for any pair $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$, $\mathcal{S}_1$ is always a singleton. This will not, however, be the case when we generalize the definition for left-nonlinear systems.

As mentioned, the ρ parameter of relation *VarPairs* (and of the relations *Resid*, *Creating*, *Created*, and *CreateResid* that follow) is irrelevant for left-linear systems. This parameter *is* relevant in the definition of *VarPairs* for left-nonlinear systems (Definition 4.23). The ρ parameter is included in the definitions of this section solely for reasons of uniformity.

Definition 4.4 is the standard notion of *residual*, in relational form. For a contraction $\mathcal{A} : C \longrightarrow C'$, *Resid* associates each subcontext of C that is not affected by $\mathcal{A}$ with the corresponding subcontext of C'. Moreover, for each $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$, $C_1 \in \mathcal{S}_1$, and $C_2 \in \mathcal{S}_2$, C_1 is related to C_2. If $\mathcal{S}_2$ is empty, this will have the effect that no pairs are added to $\mathit{Resid}^{\rho\mathcal{A}}$.

Definition 4.4 (*Resid*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$
\begin{aligned}
\mathit{Resid}^{\rho\mathcal{A}} \triangleq\ & \{ \langle D_1, D_2 \rangle \mid D_1 \in \mathcal{S}_1,\ D_2 \in \mathcal{S}_2,\ \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}} \} \cup \\
& \{ \langle D, D \rangle \mid D \ \mathit{and}\ \mathit{Redex}_{\mathcal{A}} \ \mathit{are\ disjoint} \}
\end{aligned}
$$

The reflexive, transitive closure of Resid is defined by

$$
\begin{aligned}
\mathit{Resid}{*}^{\epsilon} &\triangleq \{ \langle C, C \rangle \mid C \in \mathit{Cont}(\Sigma) \} \\
\mathit{Resid}{*}^{\rho\mathcal{A}} &\triangleq \mathit{Resid}{*}^{\rho} \cdot \mathit{Resid}^{\rho\mathcal{A}}
\end{aligned}
$$

Here, the operation '$\cdot$' denotes relational join.

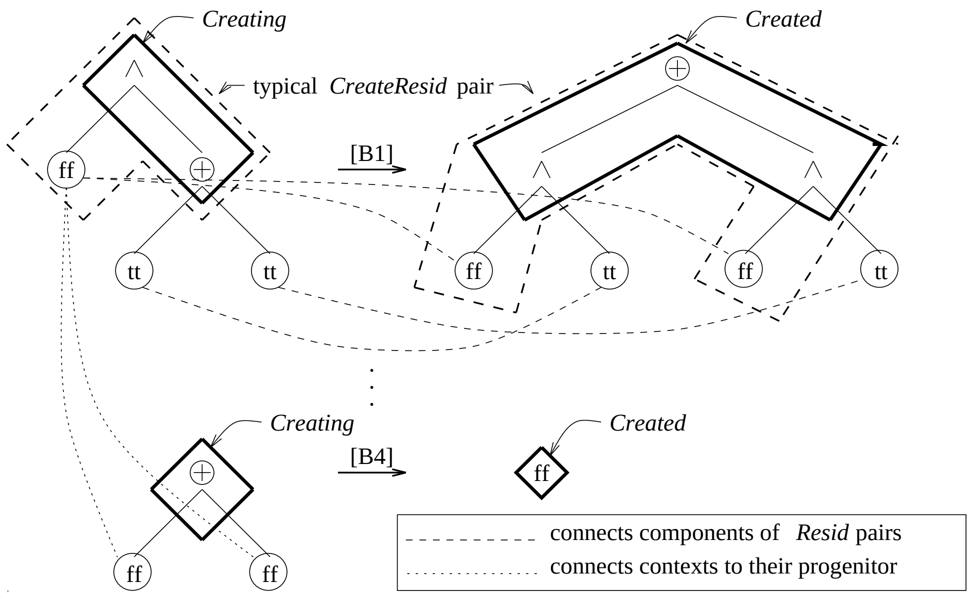


Figure 4.6: Illustration of selected relations and contexts derived from the **B**-reduction of Figure 4.4.

Figure 4.6 depicts *Resid* and several other definitions we will encounter in the sequel, as they apply to the initial and final contractions of the reduction in Figure 4.4, involving the left-linear rule **[B1]** and the left-*non*linear rule **[B4]** of TRS **B**, respectively.

Definition 4.5 describes the *creating* and the *created* contexts associated with a contraction $\mathcal{A}$. Intuitively, if contraction $\mathcal{A}$ is applied to term T, the creating context is the minimal subcontext of T needed for the left-hand side of $\mathcal{A}$'s rule to match; the created context is the corresponding minimal context "constructed" by the right-hand side of the rule. The former is defined as the context derived by subtracting from $\mathit{Redex}_{\mathcal{A}}$ all contexts $D_1 \in \mathcal{S}_1$ such that $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$. The latter is the context derived by subtracting from $\mathit{Reduct}_{\mathcal{A}}$ all contexts $D_2 \in \mathcal{S}_2$ such that $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$.

Definition 4.5 (*Creating* and *Created*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\mathit{Creating}^{\rho\mathcal{A}} \triangleq \mathit{Redex}_{\mathcal{A}} - \bigsqcup\{\mathcal{S}_1 \mid \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}\}$$

$$\mathit{Created}^{\rho\mathcal{A}} \triangleq \begin{cases} \mathit{Reduct}_{\mathcal{A}} - \bigsqcup\{\mathcal{S}_2 \mid \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}\} & \textit{when } R_{\mathcal{A}} \notin \mathcal{V} \\ p_{\mathcal{A}} \leftarrow \bullet & \textit{otherwise} \end{cases}$$

While $\mathit{Creating}^{\rho\mathcal{A}}$ and $\mathit{Created}^{\rho\mathcal{A}}$ could have been defined in a more direct way from the structure of $L_{\mathcal{A}}$, $R_{\mathcal{A}}$, and $p_{\mathcal{A}}$ without using $\mathit{VarPairs}^{\rho\mathcal{A}}$ at all, the approach we take here will be much easier to generalize when we consider left-nonlinear systems.

Combining Definitions 4.4 and 4.5, we arrive at the relation *CreateResid*, formalized in Definition 4.6. Every pair of *terms* $\langle T, T' \rangle \in \mathit{CreateResid}$ has the property that $T \longrightarrow T'$.

The difference between the contraction relation '$\longrightarrow$' and relation *CreateResid* is that the latter is *derived* from the former. Roughly speaking, $\mathit{CreateResid}^{\rho\mathcal{A}}$ may be regarded as the "restriction" of '$\longrightarrow$' to the specific contraction $\mathcal{A}$.

Definition 4.6 (*CreateResid*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{lll} \mathit{CreateResid}^{\rho\mathcal{A}} \triangleq & \{ \langle C_1, C_2 \rangle \mid & R \subseteq \mathit{Resid}^{\rho\mathcal{A}}, \\ & & \langle C, D \rangle \in R \text{ and } \langle C, D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}} \\ & & \qquad \text{imply } \langle C, D' \rangle \in R, \\ & & C_1 \text{ and } C_2 \text{ are contexts such that:} \\ & & C_1 = \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup \{ C \mid \langle C, C' \rangle \in R \}, \\ & & C_2 = \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup \{ C' \mid \langle C, C' \rangle \in R \} \quad \} \end{array}$$

Note that it is *impossible* to have both $\langle C_1, D\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C_2, D\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$, for any nonempty $C_1,\ C_2,\ D$; these relations may, however, overlap on empty contexts.

4.4 A dynamic dependence relation

In this section, we will derive our dynamic dependence relation, *Slice*$*$, using the concepts introduced in Section 4.3. For the empty reduction, *Slice*$*$ is defined as the identity relation. For a criterion D, the inductive case determines the minimal super-context $D' \sqsupseteq D$ for which there is a C such that $\langle C, D'\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$; then the slice for this C in reduction ρ is determined.

Definition 4.7 (*Slice*$*$) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{lll} \mathit{Slice}*^{\epsilon} & \triangleq & \{ \langle C, C \rangle \mid C \in \mathit{Cont}(\Sigma) \} \\ \mathit{Slice}*^{\rho\mathcal{A}} & \triangleq & \mathit{Slice}*^{\rho} \cdot \{ \langle C, D \rangle \mid \text{there exists a minimal } D' \sqsupseteq D \\ & & \qquad \text{such that } \langle C, D' \rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}}) \} \end{array}$$

Since $\mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C_2, D\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ only overlap for empty contexts, it is easy to see that the slice with respect to any nonempty criterion is uniquely defined. Empty contexts may have multiple slices, which arise from the application of collapse rules.

4.4.1 Example

In the example that follows, we will frequently use set comprehension to avoid unwieldy notation. We will consider the following **B**-reduction $\rho = \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3$:

$$S = (\mathtt{ff} \wedge \underline{(\mathtt{ff} \wedge \mathtt{tt})}) \wedge \mathtt{tt} \xrightarrow{\mathcal{A}_1} \underline{(\mathtt{ff} \wedge \mathtt{ff})} \wedge \mathtt{tt} \xrightarrow{\mathcal{A}_2} \underline{\mathtt{ff} \wedge \mathtt{tt}} \xrightarrow{\mathcal{A}_3} \mathtt{ff} = T$$

Note that for contraction $\mathcal{A}_1$, we have $p_{\mathcal{A}_1}$ = (1 2), $\overline{L_{\mathcal{A}_1}} = (1\,2) \leftarrow X \wedge \mathtt{tt}$, $\overline{R_{\mathcal{A}_1}} = (1\,2) \leftarrow X$, $\mathit{Redex}_{\mathcal{A}_1}$ = (1 2) $\leftarrow \mathtt{ff} \wedge \mathtt{tt}$, and $\mathit{Reduct}_{\mathcal{A}_1}$ = (1 2) $\leftarrow \mathtt{ff}$. This results in the following relations for $\mathcal{A}_1$:

$$\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1} & = & \{\, \langle \{\, (1\,2\,1) \leftarrow \texttt{ff} \,\}, \{\, (1\,2) \leftarrow \texttt{ff} \,\} \rangle,\ \langle \{\, (1\,2\,1) \leftarrow \bullet \,\}, \{\, (1\,2) \leftarrow \bullet \,\} \rangle \,\} \\
\mathit{Resid}^{\mathcal{A}_1} & = & \{\ \ \langle (1\,2\,1) \leftarrow \texttt{ff}, (1\,2) \leftarrow \texttt{ff} \rangle,\ \langle (1\,2\,1) \leftarrow \bullet, (1\,2) \leftarrow \bullet \rangle, \\
 & & \quad \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle (1\,2) \leftarrow \bullet, (1\,2) \leftarrow \bullet \rangle \,\} \cup \\
 & & \{\, C \ \mid\ C \sqsubseteq () \leftarrow (\texttt{ff} \wedge \bullet) \wedge \texttt{tt} \,\} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1} & = & (1\,2) \leftarrow (\bullet \wedge \texttt{tt}) \\
\mathit{Created}^{\mathcal{A}_1} & = & (1\,2) \leftarrow \bullet \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1} & = & \{\ \ \langle (1\,2) \leftarrow (\bullet \wedge \texttt{tt}), (1\,2) \leftarrow \bullet \rangle,\ \langle (1) \leftarrow \bullet \wedge (\texttt{ff} \wedge \texttt{tt}), (1) \leftarrow \bullet \wedge \texttt{ff} \rangle, \\
 & & \quad \langle (1) \leftarrow \texttt{ff} \wedge (\texttt{ff} \wedge \texttt{tt}), (1) \leftarrow \texttt{ff} \wedge \texttt{ff} \rangle, \\
 & & \quad \langle () \leftarrow (\bullet \wedge (\texttt{ff} \wedge \texttt{tt})) \wedge \bullet, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge \bullet \rangle, \\
 & & \quad \langle () \leftarrow (\bullet \wedge (\texttt{ff} \wedge \texttt{tt})) \wedge \texttt{tt}, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge \texttt{tt} \rangle, \\
 & & \quad \langle () \leftarrow (\texttt{ff} \wedge (\texttt{ff} \wedge \texttt{tt})) \wedge \bullet, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge \bullet \rangle, \\
 & & \quad \langle () \leftarrow (\texttt{ff} \wedge (\texttt{ff} \wedge \texttt{tt})) \wedge \texttt{tt}, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge \texttt{tt} \rangle \,\}
\end{array}$$

For contraction $\mathcal{A}_2$, we have $p_{\mathcal{A}_2} = (1)$, $\overline{L_{\mathcal{A}_2}} = (1) \leftarrow X \wedge \texttt{ff}$, $\overline{R_{\mathcal{A}_2}} = (1) \leftarrow \texttt{ff}$, $\mathit{Redex}_{\mathcal{A}_2} = (1) \leftarrow \texttt{ff} \wedge \texttt{ff}$, and $\mathit{Reduct}_{\mathcal{A}_2} = (1) \leftarrow \texttt{ff}$. Therefore, we have:

$$\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\, \langle \{\, (1\,1) \leftarrow \texttt{ff} \,\}, \emptyset \rangle,\ \langle \{\, (1\,1) \leftarrow \bullet \,\}, \emptyset \rangle \,\} \\
\mathit{Resid}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\, \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle (1) \leftarrow \bullet, (1) \leftarrow \bullet \rangle \,\} \cup \\
 & & \{\, \langle C, C \rangle \ \mid\ C \sqsubseteq () \leftarrow \bullet \wedge \texttt{ff} \,\} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1\mathcal{A}_2} & = & (1) \leftarrow (\bullet \wedge \texttt{ff}) \\
\mathit{Created}^{\mathcal{A}_1\mathcal{A}_2} & = & (1) \leftarrow \texttt{ff} \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\ \ \langle (1) \leftarrow (\bullet \wedge \texttt{ff}), (1) \leftarrow \texttt{ff} \rangle,\ \langle () \leftarrow (\bullet \wedge \texttt{ff}) \wedge \bullet, () \leftarrow \texttt{ff} \wedge \bullet \rangle, \\
 & & \quad \langle () \leftarrow (\bullet \wedge \texttt{ff}) \wedge \texttt{tt}, () \leftarrow \texttt{ff} \wedge \texttt{tt} \rangle \,\}
\end{array}$$

For the third contraction, $\mathcal{A}_3$, we have $p_{\mathcal{A}_3} = ()$, $\overline{L_{\mathcal{A}_3}} = () \leftarrow X \wedge \texttt{tt}$, $\overline{R_{\mathcal{A}_3}} = () \leftarrow X$, $\mathit{Redex}_{\mathcal{A}_3} = () \leftarrow \texttt{ff} \wedge \texttt{tt}$, and $\mathit{Reduct}_{\mathcal{A}_3} = () \leftarrow \texttt{ff}$. The following relations are computed for $\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3$:

$$\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\, \langle \{\, (1) \leftarrow \texttt{ff} \,\}, \{\, () \leftarrow \texttt{ff} \,\} \rangle,\ \langle \{\, (1) \leftarrow \bullet \,\}, \{\, () \leftarrow \bullet \,\} \rangle \,\} \\
\mathit{Resid}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\, \langle (1) \leftarrow \texttt{ff}, () \leftarrow \texttt{ff} \rangle,\ \langle (1) \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle \,\} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & () \leftarrow (\bullet \wedge \texttt{tt}) \\
\mathit{Created}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & () \leftarrow \bullet \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\, \langle () \leftarrow (\bullet \wedge \texttt{tt}), () \leftarrow \bullet \rangle \,\}
\end{array}$$

From the above and Definition 4.7 it follows that we have the following dynamic dependence relations between subcontexts of S and T:

$$\mathit{Slice}{*}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} = \{\, \langle (1) \leftarrow \bullet \wedge (\texttt{ff} \wedge \texttt{tt}), () \leftarrow \texttt{ff} \rangle, \quad \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle \,\}$$

Thus, the slice with respect to $() \leftarrow \texttt{ff} \sqsubseteq T$ is $(1) \leftarrow \bullet \wedge (\texttt{ff} \wedge \texttt{tt}) \sqsubseteq S$. This is the minimal context for which there exists a subreduction of ρ that yields the criterion. In this case, the projection consists of the first two contractions.

The above example also illustrates why *Slice*$*$ is defined on contexts rather than on context domains: collapse rules require special treatment in order to produce minimal slices. Note that the example exhibits two applications of collapse rule **[B2]**. Intuitively, the first one created the criterion, whereas the second one merely affects its *location*. We achieve this differentiation by: (i) having a collapse rule create an empty context $p_{\mathcal{A}} \leftarrow \bullet$ instead of the context consisting of the function symbol at path $p_{\mathcal{A}}$ (the approach of [94]), and (ii) defining an empty context $p \leftarrow \bullet$ to be a subcontext of a nonempty context *only* if the latter "surrounds" the former.

4.5 Projections, subreductions

In this section, we formalize the notion of a *projection* of a reduction on a subcontext of its initial term. It will be convenient to define simultaneously the initial context C and the final context D to which a projection corresponds along with the projected "subreduction" σ itself. We therefore define the set of *projection triples* as follows:

Definition 4.8 (Projection Triples) *Let $\mathcal{R}$ be a TRS over signature Σ. Then the set of $\mathcal{R}$ projection triples is inductively defined as follows:*

$$\mathit{Project}*^{\epsilon} \triangleq \{ \langle B, \epsilon, B \rangle \mid B \in \mathit{Cont}(\Sigma) \}$$

$$\begin{aligned}\mathit{Project}*^{\rho\mathcal{A}} \triangleq\ & \{\langle B, \sigma\mathcal{A}, D' \rangle \mid \langle B, \sigma, C \rangle \in \mathit{Project}*^{\rho}, && \text{(i)}\\ & \qquad \langle C, D \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}, \\ & \qquad D' \sqsubseteq D \} \cup \\ & \{\langle B, \sigma, D' \rangle \mid \langle B, \sigma, C \rangle \in \mathit{Project}*^{\rho}, && \text{(ii)}\\ & \qquad \langle C, D \rangle \in \mathit{Resid}^{\rho\mathcal{A}}, \\ & \qquad D' \sqsubseteq D \}\end{aligned}$$

The interesting cases in Definition 4.8 are numbered. Intuitively, these cases behave as follows:

- In case (i), the context D' that constitutes the third element of the triple is entirely contained in a context D that is involved in a *CreateResid*$^{\rho\mathcal{A}}$–relation. In this case, contraction $\mathcal{A}$ is deemed applicable to D', and the construction continues recursively with the context C that contracted to D, and reduction ρ.
- In case (ii), D' is a subcontext of some context D that residuated from a context C. In this case, contraction $\mathcal{A}$ was not applicable to D', and the construction continues recursively with the context C from which D residuated, and reduction ρ.

$$T_0 \equiv \underline{\underline{\boxed{(\texttt{ff} \wedge \texttt{ff})} \wedge (\texttt{tt} \oplus \texttt{tt})}} \xrightarrow{\mathcal{A}_{[B1]}} (\underline{(\texttt{ff} \wedge \texttt{ff})} \wedge \texttt{tt}) \oplus ((\texttt{ff} \wedge \texttt{ff}) \wedge \texttt{tt}) \xrightarrow{\mathcal{A}_{[B3]}} (\boxed{\texttt{ff}} \wedge \texttt{tt}) \oplus (\boxed{(\texttt{ff} \wedge \texttt{ff})} \wedge \texttt{tt}) \equiv T_2$$

Figure 4.7: Example of projections.

Note that *each* residual D of a context C gives rise to the construction of a new projection triple. This reflects the fact that different residuals of a context may be reduced differently, causing the construction of different subreductions.

Informally, the occurrence of a triple $\langle B, \sigma, D' \rangle$ in relation *Project*$*^{\rho}$ indicates that context B reduces to a context D that "contains" D'. Moreover, it does so by a reduction that is derived from the contractions of ρ specified in σ[1]. Therefore σ can justifiably be deemed a subreduction of ρ. In Section 4.6, we will prove this property of projections. In addition, we will show that *Slice*$*^{\rho}$ computes slices that correspond to minimal projections by effectively selecting the minimal supercontext D of D' (in each construction step) for which there exists a pair $\langle C, D \rangle$ in $(\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$.

As an example of the behavior of *Project*$*$, consider the B-reduction in Figure 4.7. As usual, we have underlined each redex. We use $\mathcal{A}_{[B1]}$ and $\mathcal{A}_{[B3]}$ to denote the contractions that use rules [B1] and [B3], respectively. Some typical, minimal elements of the set *Project*$*^{\mathcal{A}_{[B1]}\mathcal{A}_{[B3]}}$ are:

$$\begin{array}{l}
\langle\, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge (\bullet \oplus \bullet)\,,\ \mathcal{A}_{[B1]}\mathcal{A}_{[B3]}\,,\ () \leftarrow (\texttt{ff} \wedge \bullet) \oplus ((\texttt{ff} \wedge \texttt{ff}) \wedge \bullet)\,\rangle \\
\langle\, () \leftarrow \bullet \wedge (\bullet \oplus \bullet)\,,\ \mathcal{A}_{[B1]}\,,\ () \leftarrow (\bullet \wedge \bullet) \oplus (\bullet \wedge \bullet)\,\rangle \\
\langle\, (1) \leftarrow \texttt{ff} \wedge \texttt{ff}\,,\ \mathcal{A}_{[B3]}\,,\ (1\,1) \leftarrow \texttt{ff}\,\rangle \\
\langle\, (1) \leftarrow \texttt{ff} \wedge \texttt{ff}\,,\ \epsilon\,,\ (2\,1) \leftarrow \texttt{ff} \wedge \texttt{ff}\,\rangle
\end{array}$$

Observe that the last two of these projection triples "apply to" the subcontext $(1) \leftarrow (\texttt{ff} \wedge \texttt{ff})$ of T_0; this subcontext is shown boxed in Figure 4.7. The projections of the boxed subcontext of T_0 are also shown boxed (in T_2). Clearly, these triples correspond to the two different "paths through the reduction" taken by the boxed subterm of T_0. One residual is contracted in a subsequent step, the other is not.

The difference between the *Slice*$*$ and *Project*$*$ relations is best illustrated by a *non*-minimal element of the set *Project*$*^{\mathcal{A}_{[B1]}\mathcal{A}_{[B3]}}$ that does not occur in *Slice*$*^{\mathcal{A}_{[B1]}\mathcal{A}_{[B3]}}$, such as:

$$\langle\, () \leftarrow (\texttt{ff} \wedge \texttt{ff}) \wedge (\bullet \oplus \bullet)\,,\ \mathcal{A}_{[B1]}\mathcal{A}_{[B3]}\,,\ (1\,1) \leftarrow \texttt{ff} \rangle$$

[1]It is important to realize that the occurrence of a triple $\langle B, \sigma, D' \rangle$ in relation *Project*$*^{\rho}$ does *not* imply that that $B \xrightarrow{\sigma} D'$. This is the case because the root and substitution components of ρ are copied unmodified in the process of constructing σ. While σ could in principle be constructed in such a way that it "applies" directly to (i.e., is a reduction of) context B, this is not necessary for our purpose of proving that *some* reduction exists (see Lemma 4.18); moreover it would substantially complicate the construction process of σ in the definition of *Project*$*$.

4.6 Formal properties of slices

We can now state some theorems describing the most important properties of slices. In the sequel, all TRSs are assumed to be left-linear.

4.6.1 Uniqueness of slices

In order to prove that *Slice*$*$ is a many-to-one mapping for non-empty contexts (that is, each context has a unique slice), we will first prove a few lemmas.

Lemma 4.9 *Let $B \overset{\rho}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$ be a reduction. Then for any non-empty $D' \sqsubseteq D$ there is at most one $C' \sqsubseteq C$ such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$. Moreover, if it exists, this C' will be non-empty.*

Proof. Let $D' \sqsubseteq D$ be a non-empty context such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ for some $C' \sqsubseteq C$.

There are two cases:

1. $\mathit{root}(D') \preceq \mathit{root}(\mathit{Created}^{\rho\mathcal{A}})$ and D' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint.
   Then it follows from Definition 4.4 that $C' \equiv D'$ is the unique subcontext of C such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$. This C' is non-empty because D' is non-empty.
2. $D' \equiv (p_R \cdot q \leftarrow A)$ where $p_R = \mathcal{O}_X(\overline{R_{\mathcal{A}}})$, $A \sqsubseteq (() \leftarrow \sigma_{\mathcal{A}}(X))$, and $q = \mathit{root}(A)$ for some variable X.
   From left-linearity it follows that there is a *unique* path p_L such that $\{ p_L \} = \mathcal{O}_X(\overline{L_{\mathcal{A}}})$. From Definitions 4.3 and 4.4 it follows that $C' \equiv (p_L \cdot q \leftarrow A)$ is the unique subcontext of C such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$. Since rerooting a context does not affect its (non-)emptyness, C' will be a non-empty context. □

Lemma 4.10 *Let $B \overset{\rho}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$ be a reduction. Then for any non-empty $D' \sqsubseteq D$ there is at most one $C' \sqsubseteq C$ such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$. Moreover, if it exists, this C' will be non-empty.*

Proof. Let $D' \sqsubseteq D$ be a non-empty context such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ for some $C' \sqsubseteq C$.

From Definition 4.6 it follows that there exists a unique subset R of $\mathit{Resid}^{\rho\mathcal{A}}$ such that:

$$D' \equiv \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup \{ E' \mid \langle E, E' \rangle \in R \}$$

and also that there exists a unique context

$$C' \equiv \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup \{ E \mid \langle E, E' \rangle \in R \}$$

such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$. Since the left-hand side of a rewrite rule is not a single variable, $\mathit{Creating}^{\rho\mathcal{A}}$ is non-empty, causing this C' to be non-empty as well. □

Lemma 4.11 *Let $B \overset{\rho}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$ be a reduction. It is impossible to have $\langle C_1, D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C_2, D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ for any $C_1,\ C_2 \sqsubseteq C$ and any non-empty $D' \sqsubseteq D$.*

Proof. Assume that $\langle C_2, D' \rangle \in \textit{CreateResid}^{\rho\mathcal{A}}$ for some $C_2 \sqsubseteq C$, and some non-empty $D' \sqsubseteq D$. From Definition 4.6 it follows that $\textit{Creating}^{\rho\mathcal{A}} \sqsubseteq C_2$ and $\textit{Created}^{\rho\mathcal{A}} \sqsubseteq D'$.

From Definitions 4.4 and 4.5 it follows that for any pair $\langle C_1, D_1 \rangle \in \textit{Resid}^{\rho\mathcal{A}}$ with $C_1 \sqsubseteq C$, $D_1 \sqsubseteq D$, we have that C_1 and $\textit{Creating}^{\rho\mathcal{A}}$ are disjoint and D_1 and $\textit{Created}^{\rho\mathcal{A}}$ are disjoint.

From $\textit{Creating}^{\rho\mathcal{A}} \sqsubseteq C_2$ and $\textit{Creating}^{\rho\mathcal{A}} \not\sqsubseteq C_1$ it follows that $C_2 \not\equiv C_1$. □

Lemma 4.12 *Let* $\rho : B \overset{\rho'}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$ *be a reduction, and let* D'' *be a non-empty subcontext of* D*. Then there exists a unique minimal* D' *such that* $D'' \sqsubseteq D' \sqsubseteq D$ *and* $\langle C', D' \rangle \in (\textit{Resid}^{\rho} \cup \textit{CreateResid}^{\rho})$ *for some non-empty* $C' \sqsubseteq C$*. Moreover,*

$$\langle B', C' \rangle \in \textit{Slice}*^{\rho'} \Leftrightarrow \langle B', D' \rangle \in \textit{Slice}*^{\rho} \Leftrightarrow \langle B', D'' \rangle \in \textit{Slice}*^{\rho}$$

where $B' \sqsubseteq B$*.*

Proof. The theorem holds trivially if $\langle C', D'' \rangle \in (\textit{Resid}^{\rho} \cup \textit{CreateResid}^{\rho})$, for some $C' \sqsubseteq C$.

Assume that there exists no $C' \sqsubseteq C$ such that $\langle C', D'' \rangle \in (\textit{Resid}^{\rho} \cup \textit{CreateResid}^{\rho})$. From Definitions 4.3 and 4.4, it follows that D'' and $\textit{Created}^{\rho\mathcal{A}}$ are not disjoint—otherwise, D'' would be involved in a $\textit{Resid}^{\rho}$-relation. From the fact that D'' is not involved in a $\textit{CreateResid}^{\rho}$ relation either, we have that one or both of the following hold:

- $\textit{Created}^{\rho} \not\sqsubseteq D''$,
- $D'' - \textit{Created}^{\rho\mathcal{A}} \equiv \bigsqcup\{ E' \mid \langle E, E' \rangle \in R \}$, for some $R \subset \textit{Resid}^{\rho}$ such that there exist $\langle A, B \rangle \in R$, $\langle A, B' \rangle \in \textit{Resid}^{\rho}$ for which $\langle A, B' \rangle \notin R$.

Define:

$$\begin{aligned} R' &\triangleq R \cup \{ \langle A, B' \rangle \mid \langle A, B \rangle \in R,\ \langle A, B' \rangle \in \textit{Resid}^{\rho} \} \\ D' &\triangleq D' \sqcup \textit{Created}^{\rho} \sqcup \{ E' \mid \langle E, E' \rangle \in R' \} \end{aligned}$$

Clearly, D' is the minimal supercontext of D'' for which $\langle C', D' \rangle \in \textit{CreateResid}^{\rho}$, where

$$C' \equiv \textit{Creating}^{\rho} \sqcup \bigsqcup\{ E \mid \langle E, E' \rangle \in R' \} \sqsubseteq C$$

Since $\textit{Creating}^{\rho}$ is always non-empty, C' is non-empty as well.

From Definition 4.7 it follows that

$$\langle B', C' \rangle \in \textit{Slice}*^{\rho'} \Leftrightarrow \langle B', D' \rangle \in \textit{Slice}*^{\rho} \Leftrightarrow \langle B', D'' \rangle \in \textit{Slice}*^{\rho}$$

where $B' \sqsubseteq B$. □

Theorem 4.13 (Uniqueness of Slices) *Let* $\rho : B \overset{\rho}{\longrightarrow}{}^{*} D$ *be a reduction, and let* $D' \sqsubseteq D$ *be non-empty. Then there exists a unique non-empty* $B' \sqsubseteq B$ *such that* $\langle B', D' \rangle \in \textit{Slice}*^{\rho}$*.*

Proof. By induction on the length of the reduction ρ.

For $\rho \equiv \epsilon$, we have $\{ B' \mid \langle B', D' \rangle \in \textit{Slice}*^{\epsilon} \} = \{ D' \}$ according to Definition 4.7.

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$ such that $B \overset{\rho'}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$, and let $D' \sqsubseteq D$ be a non-empty context. According to Lemma 4.12, we may assume without loss of generality that $\langle C', D' \rangle \in (\textit{Resid}^{\rho'\mathcal{A}} \cup \textit{CreateResid}^{\rho'\mathcal{A}})$, for some $C' \sqsubseteq C$.

According to Lemma 4.11, we have *either* $\langle C', D'\rangle \in \mathit{Resid}^{\rho'\mathcal{A}}$ *or* $\langle C', D'\rangle \in \mathit{CreateResid}^{\rho'\mathcal{A}}$.

Lemmas 4.9 and 4.10 state that both *Resid* and *CreateResid* map any non-empty context $D' \sqsubseteq D$ to a unique non-empty $C' \sqsubseteq C$. By induction, there exists a unique non-empty $B' \sqsubseteq B$ such that $\langle B', C'\rangle \in \mathit{Slice}{*}^{\rho'}$.

From Definition 4.7 it follows that this B' is the unique non-empty subcontext of B such that $\langle B', D'\rangle \in \mathit{Slice}{*}^{\rho}$. □

Given Theorem 4.13, we will be able to write $C = \mathit{Slice}{*}^{\rho}(D)$ instead of $\langle C, D\rangle \in \mathit{Slice}{*}^{\rho}$, for non-empty D.

4.6.2 Preservation of topology

The following lemma states that slices may be computed by repeatedly determining a unique related subcontext of the previous context in the reduction.

Lemma 4.14 *Let* $\rho : B \overset{\rho'}{\longrightarrow^*} C \xrightarrow{\mathcal{A}} D$ *be a reduction, and let* $D'' \sqsubseteq D$ *be a non-empty context. Moreover, let* D' *be the unique minimal super-context of* D'' *for which there exists a non-empty context* $C' \sqsubseteq C$ *such that* $\langle C', D'\rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$. *Then*

$$\mathit{Slice}{*}^{\rho'}(C') = \mathit{Slice}{*}^{\rho}(D'') = \mathit{Slice}{*}^{\rho}(D')$$

Proof. Follows immediately from Definition 4.7, Lemma 4.12, and Theorem 4.13. □

Lemma 4.15 states that any pair of contexts $\langle C, D\rangle$ in the *Resid* relation can be “split” into a set $\mathcal{S}$ of pairs of elementary contexts in a way that each pair of elementary contexts $\langle C', D'\rangle$ in $\mathcal{S}$ also occurs in the *Resid* relation. This result will be used in the proof of the Inclusion Lemma, which follows below.

Lemma 4.15 *Let* C *and* D *be contexts such that* $\langle C, D\rangle \in \mathit{Resid}^{\rho}$, *for some reduction* ρ. *Then there exists a set* $\mathcal{S}$ *of pairs of elementary contexts such that all of the following hold:*

1. $\bigsqcup\{\, C' \mid \langle C', D'\rangle \in \mathcal{S} \,\} \equiv C$,
2. $\bigsqcup\{\, D' \mid \langle C', D'\rangle \in \mathcal{S} \,\} \equiv D$,
3. $\langle C', D'\rangle \in \mathcal{S}$ *implies that* $\langle C', D'\rangle \in \mathit{Resid}^{\rho}$.

Proof. Follows trivially from Definition 4.4. □

The next lemma and theorem demonstrate that slices effectively preserve the topology of their corresponding criteria. This is important in showing that slices are minimal projections.

Lemma 4.16 (Inclusion Lemma) *Let* $\rho\mathcal{A}$ *be a reduction such that* $B \overset{\rho}{\longrightarrow^*} C \xrightarrow{\mathcal{A}} D$, *and let* $D'' \sqsubseteq D' \sqsubseteq D$ *be non-empty contexts such that there exist pairs* $\langle C', D'\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$, *and* $\langle C'', D''\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$. *Then* $C'' \sqsubseteq C'$.

Proof. There are three cases:

1. $\langle C', D'\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C'', D''\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$.
   From Definition 4.4 it follows that there are two cases:

(a) $\mathit{root}(D') \preceq \mathit{Created}^{\rho\mathcal{A}}$, D' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint, $\mathit{root}(D'') \preceq \mathit{Created}^{\rho\mathcal{A}}$, and D'' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint. Then $C' \equiv D'$ and $C'' \equiv D''$, and therefore $C'' \sqsubseteq C'$.

(b) $D' \equiv (p_R \cdot q' \leftarrow A')$, $D'' \equiv (p_R \cdot q'' \leftarrow A'')$ where $p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})$, $A' \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X))$, $A'' \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X))$, $q' = \mathit{root}(A')$, $q'' = \mathit{root}(A'')$, for some variable X.
From Definitions 4.3 and 4.4 and left-linearity, it follows that there exists a unique occurrence p_L of X in $L_\mathcal{A}$ such that $C' \equiv (p_L \cdot q' \leftarrow A')$, $C'' \equiv (p_L \cdot q'' \leftarrow A'')$. $D'' \sqsubseteq D'$ implies $q' \preceq q''$, $A'' \sqsubseteq A'$ and therefore that $C'' \sqsubseteq C'$.

2. $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ and $\langle C'', D'' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$.
From Definition 4.6 it follows that

$$\begin{array}{lcl}
D' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E' \mid \langle E, E' \rangle \in R' \,\} \\
D'' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E' \mid \langle E, E' \rangle \in R'' \,\} \\
C' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E \mid \langle E, E' \rangle \in R' \,\} \\
C'' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E \mid \langle E, E' \rangle \in R'' \,\}
\end{array}$$

for $R',\ R'' \subseteq \mathit{Resid}^{\rho\mathcal{A}}$. According to Lemma 4.15, we may assume without loss of generality that for all $\langle E, E' \rangle \in R', R''$, both E and E' are elementary. From $D'' \sqsubseteq D'$ it follows that $R'' \subseteq R'$ and therefore that $C'' \sqsubseteq C'$.

3. $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ and $\langle C'', D'' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$
According to Definition 4.6, we have that

$$\begin{array}{lcl}
D' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E' \mid \langle E, E' \rangle \in R \,\} \\
C' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, E \mid \langle E, E' \rangle \in R \,\}
\end{array}$$

for some $R \subseteq \mathit{Resid}^{\rho\mathcal{A}}$. According to Lemma 4.15, we may assume without loss of generality that for all $\langle E, E' \rangle \in R$, both E and E' are elementary. From $D'' \sqsubseteq D'$, and the disjointness of $\mathit{Created}^{\rho\mathcal{A}}$ and D'' it follows that there exists a subset $R' \subseteq R$ such that

$$D'' \equiv \bigsqcup\{\, E' \mid \langle E, E' \rangle \in R' \,\}$$

Using an argument similar to that in case 1, it follows that

$$C'' \equiv \bigsqcup\{\, E \mid \langle E, E' \rangle \in R' \,\}$$

Consequently $C'' \sqsubseteq C'$.

Note that the case where $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C'', D'' \rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ is impossible, given $D'' \sqsubseteq D'$. □

Theorem 4.17 (Inclusion Theorem) *Let $\rho : B \overset{\rho}{\longrightarrow}{}^{*} D$ be a reduction, let $D'' \sqsubseteq D' \sqsubseteq D$, and let D'' be non-empty. Then $\mathit{Slice}{*}^{\rho}(D'') \sqsubseteq \mathit{Slice}{*}^{\rho}(D')$.*

Proof. By induction on the length of the reduction ρ.

For $\rho \equiv \epsilon$, we trivially have:

$$\mathit{Slice}{*}^{\epsilon}(D'') \equiv D'' \sqsubseteq D' \equiv \mathit{Slice}{*}^{\epsilon}(D')$$

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$ such that $B \overset{\rho'}{\longrightarrow^*} C \xrightarrow{\mathcal{A}} D$, and let D', D'' be subcontexts of D such that $D'' \sqsubseteq D' \sqsubseteq D$ and D'' is non-empty. According to Lemmas 4.12 and 4.14 we may assume without loss of generality that there exist non-empty contexts C', $C'' \sqsubseteq C$ such that

$$\langle C', D' \rangle \in (\mathit{Resid}^{\rho} \ \cup \ \mathit{CreateResid}^{\rho}),\ \mathit{Slice}{*}^{\rho}(D') = \mathit{Slice}{*}^{\rho'}(C')$$

and

$$\langle C'', D'' \rangle \in (\mathit{Resid}^{\rho} \ \cup \ \mathit{CreateResid}^{\rho}),\ \mathit{Slice}{*}^{\rho}(D'') = \mathit{Slice}{*}^{\rho'}(C'')$$

According to Lemma 4.16, $D'' \sqsubseteq D'$ implies $C'' \sqsubseteq C'$. By induction, $C'' \sqsubseteq C'$ implies $\mathit{Slice}{*}^{\rho'}(C'') \sqsubseteq \mathit{Slice}{*}^{\rho'}(C')$. Consequently, it follows that $\mathit{Slice}{*}^{\rho}(D'') \sqsubseteq \mathit{Slice}{*}^{\rho}(D')$. □

4.6.3 The relation between *Slice∗* and *Project∗*

Lemma 4.18 formally justifies the relationship between a reduction and the components of a projection triple.

Lemma 4.18 *Let ρ be a reduction, and let $\langle B, \sigma, D'' \rangle \in \mathit{Project}{*}^{\rho}$. Then there exist contexts E' and D such that $B \longrightarrow^* D$, $E' \sqsubseteq D$, and $E' \doteq D''$.*

Proof. By induction on the length of ρ.

According to Definition 4.8, $\langle B, \sigma, D'' \rangle \in \mathit{Project}{*}^{\epsilon}$ implies that $\sigma \equiv \epsilon$ and that $B \equiv D''$. From this it follows trivially that $B \longrightarrow^* D$, for $D \equiv D''$.

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$, and $\langle B, \sigma, D'' \rangle \in \mathit{Project}{*}^{\rho}$. From Definition 4.8 it follows that two cases can be distinguished:

1. $D'' \sqsubseteq D'$, $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$, and $\langle B, \sigma', C' \rangle \in \mathit{Project}{*}^{\rho}$.
   By induction, there exists a reduction $B \longrightarrow^* C$ for some $C \sqsupseteq F'$, where $F' \doteq C'$.
   From Definition 4.6 it follows that $C' \xrightarrow{\mathcal{A}} D'$. Therefore we have that

   $$B' \longrightarrow^* C \equiv C[F'] \longrightarrow^* C[E'] = D$$

   where $E' \equiv (\mathit{root}(F') \leftarrow D')$. Since D' and E' are isomorphic, and $D'' \sqsubseteq D'$, it follows that $D'' \doteq E''$ for some $E'' \sqsubseteq E' \sqsubseteq D$.
2. $D'' \sqsubseteq D'$, $\langle C', D' \rangle \in \mathit{Resid}^{\rho}$, and $\langle B, \sigma', C' \rangle \in \mathit{Project}{*}^{\rho}$.
   By induction, there exists a reduction $B \longrightarrow^* C$ for some $C \sqsupseteq F'$, where $F' \doteq C'$.
   From Definition 4.4 it follows that $C' \doteq D'$. Therefore we have that

   $$B' \longrightarrow^* C \equiv C[F'] \equiv C[E'] = D$$

   where $E' \equiv (\mathit{root}(F') \leftarrow D')$. Since D' and E' are isomorphic, and $D'' \sqsubseteq D'$, it follows that $D'' \doteq E''$ for some $E'' \sqsubseteq E \sqsubseteq D$. □

The lemma below establishes a connection between the relations $\mathit{Slice}{*}^{\rho}$ and $\mathit{Project}{*}^{\rho}$.

Lemma 4.19 *Let ρ be a reduction such that $B \overset{\rho}{\longrightarrow}{}^{*} D$, and let $B' = \mathit{Slice}{*}^{\rho}(D')$ for some non-empty $D' \sqsubseteq D$. Then there exists a triple $\langle B', \sigma, D' \rangle \in \mathit{Project}{*}^{\rho}$.*

Proof. By induction on the length of reduction ρ.

Let $\rho \equiv \epsilon$. From Definition 4.7 it follows that $B' = \mathit{Slice}{*}^{\epsilon}(D')$ implies $B' \equiv D'$. Moreover, from Definition 4.8 it follows that $\langle B', \epsilon, D' \rangle \in \mathit{Project}{*}^{\epsilon}$ implies $B' \equiv D'$ as well, so that the lemma trivially holds.

For the inductive case, assume that $\rho \equiv \rho' \mathcal{A}$ such that $B \overset{\rho'}{\longrightarrow}{}^{*} C \overset{\mathcal{A}}{\longrightarrow} D$, and let $B' = \mathit{Slice}{*}^{\rho}(D'')$, for some non-empty $D'' \sqsubseteq D$. According to Lemmas 4.12 and 4.14 there exists a unique $D' \sqsupseteq D''$ such that $\langle C', D' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ and $\mathit{Slice}{*}^{\rho}(D'') = \mathit{Slice}{*}^{\rho}(D') = \mathit{Slice}{*}^{\rho'}(C') = B'$.

By induction there exists a triple $\langle B', \sigma', C' \rangle \in \mathit{Project}{*}^{\rho'}$. From Lemma 4.11, it follows that there are two cases:

1. $\langle C', D' \rangle \in \mathit{Resid}^{\rho}$. Since $D'' \sqsubseteq D'$ it follows from Definition 4.8 that that $\langle B', \sigma', D'' \rangle \in \mathit{Project}{*}^{\rho}$.
2. $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$. Since $D'' \sqsubseteq D'$ it follows from Definition 4.8 that $\langle B', \sigma' \mathcal{A}, D'' \rangle \in \mathit{Project}{*}^{\rho}$. □

4.6.4 Soundness and minimality

The soundness theorem states that the $\mathit{Slice}{*}$ relation computes slices that comply with Definition 4.1.

Theorem 4.20 (Soundness) *Let ρ be a reduction such that $B \overset{\rho}{\longrightarrow}{}^{*} D$. Moreover, let $B' = \mathit{Slice}{*}^{\rho}(D'')$ for some non-empty $D'' \sqsubseteq D$. Then there exists a reduction σ such that:*

1. *$\langle B', \sigma, D'' \rangle \in \mathit{Project}{*}^{\rho}$, and*
2. *$B' \overset{\sigma}{\longrightarrow}{}^{*} D'$ such that there exists an $E'' \sqsubseteq D'$ for which $E'' \doteq D''$.*

Proof. Follows immediately from Lemmas 4.18 and 4.19. □

Our final theorem states that a slice is the *minimal* initial component of some projection triple whose final component contains the slicing criterion:

Theorem 4.21 (Minimality) *Let ρ be a reduction, and let $B'_s = \mathit{Slice}{*}^{\rho}(D'_s)$ for some non-empty D'_s. Then $\langle B'_p, \sigma, D'_p \rangle \in \mathit{Project}{*}^{\rho}$ and $D'_p \sqsupseteq D'_s$ together imply that $B'_p \sqsupseteq B'_s$.*

Proof. By induction on the length of reduction ρ.

For $\rho \equiv \epsilon$, Definition 4.8 states that $\langle B'_p, \sigma, D'_p \rangle \in \mathit{Project}{*}^{\epsilon}$ implies $\sigma \equiv \epsilon$ and $B'_p \equiv D'_p$. Moreover, according to Definition 4.7 we have that $B'_s = \mathit{Slice}{*}^{\epsilon}(D'_s)$ implies $B'_s \equiv D'_s$. Therefore $D'_p \sqsupseteq D'_s$ implies that $B'_p \sqsupseteq B'_s$.

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$, let $B'_s = \mathit{Slice}{*}^{\rho}(D'_s)$ for some non-empty D'_s, and let $\langle B'_p, \sigma, D'_p\rangle \in \mathit{Project}{*}^{\rho}$ such that $D'_p \sqsupseteq D'_s$. Then by Definition 4.8, there exists a $D_p \sqsupseteq D'_p$ such that $\langle C_p, D_p\rangle \in (\mathit{Resid}^{\rho'\mathcal{A}} \cup \mathit{CreateResid}^{\rho'\mathcal{A}})$, and $\langle B'_p, \sigma', C_p\rangle \in \mathit{Project}{*}^{\rho'}$.

According to Lemmas 4.12 and 4.14, there exists a unique minimal super-context D_s of D'_s such that $\langle C_s, D_s\rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ and:

$$\mathit{Slice}{*}^{\rho'}(C_s) = \mathit{Slice}{*}^{\rho}(D_s) = \mathit{Slice}{*}^{\rho}(D'_s) = B_s$$

By induction, $\langle B'_p, \sigma', C_p\rangle \in \mathit{Project}{*}^{\rho'}$ and $C_p \sqsupseteq C_s$ together imply that $B_p \sqsupseteq B_s$. Consequently it suffices to show that $C_p \sqsupseteq C_s$.

From (i) the fact that D_s is the *minimal* super-context of D'_s that is related in a $\mathit{CreateResid}^{\rho}$-relation, (ii) the fact that D_p is *some* supercontext of D'_s that is involved in a $\mathit{CreateResid}^{\rho}$-relation, and (iii) Definition 4.6, it follows that $D_p \sqsupseteq D_s$. According to Lemma 4.16, we therefore have $C_p \sqsupseteq C_s$. This concludes the proof of the minimality theorem. □

Together, Theorems 4.20 and 4.21 imply that our construction of slices agrees with Definition 4.1.

4.7 Nonlinear rewriting systems

Unfortunately, our previous definitions do not extend trivially to left-nonlinear TRSs[2], because they do not account for the fact that nonlinearities in the left-hand side of a rule constrain the set of contexts for which the rule is applicable. For example, when rule **[B4]** of TRS **B** of Figure 4.3 is applied to $\texttt{ff} \oplus \texttt{ff}$, this results in a contraction

$$T \equiv \texttt{ff} \oplus \texttt{ff} \xrightarrow{\mathcal{A}} \texttt{ff} \equiv T'$$

Our previous definitions yield $C = () \leftarrow (\bullet \oplus \bullet) \sqsubseteq T$ as the slice with respect to criterion $D = () \leftarrow \texttt{ff} \sqsubseteq T'$. This is not a valid slice, because some instantiations of C do not reduce to a context containing D; e.g., $() \leftarrow \texttt{tt} \oplus \texttt{ff}$ does not. A related problem is that multiple contexts may be related to a single criterion in the presence of left-nonlinear *collapse* rules; this conflicts with our objective that a slice with respect to a context consist of a single context.

A simple solution for nonlinear TRSs would be to restrict *VarPairs* to variables that occur at most once in the left-hand side of a rule. However, this would yield larger slices than necessary. For instance, for the reduction of Figure 4.4 the non-minimal slice $\texttt{ff} \wedge (\texttt{tt} \oplus \texttt{tt})$ would be computed. The immediate cause for this inaccuracy is the fact that the subcontexts $(1) \leftarrow \texttt{ff}$ and $(2) \leftarrow \texttt{ff}$ of T_3 are deemed responsible for the creation of term T_4. However, they are *residuals of the same subcontext* $C = (1) \leftarrow \texttt{ff} \sqsubseteq T_0$. This being the case, C may be replaced by an *arbitrary* context without affecting the applicability of the left-nonlinear rule.

[2]The definition of the *Slice*$*$ relation for nonlinear systems in [58] contained an error. The definitions in this section therefore supersede the earlier ones.

We can account for this fact by modifying the *VarPairs* relation as follows: if, for a rule α, all occurrences of a variable X in L_α are matched against a set of "equivalent" contexts $\mathcal{S}$ that are residuals of a common context (one that occurs earlier in the reduction sequence), then the contexts in $\mathcal{S}$ are deemed to be residuated by α (assuming X occurs in R_α). All other cases cause *creation*: those subcontexts matched against X that are not residuals of a common context are deemed *creating*, and the corresponding subcontexts matched against X in R_α are *created*.

4.7.1 Formal definitions for nonlinear systems

If a context D is created at some point in a reduction, and D has a residual C that occurs later in the reduction, we will say that D is a *progenitor* of C. This concept will be useful for formulating an adequate notion of slice for nonlinear TRSs. Formally, we have:

Definition 4.22 (Progenitor) *Let T be a term, σ and τ be reductions such that $\sigma\tau : U \longrightarrow^* T$ for some term U, and D be a subcontext of T. Then we will say that a context C is a σ,τ-*progenitor *of D if $\langle C, D\rangle \in Resid{*}^{\tau}$, and either $C \sqsubseteq CreateResid^{\sigma}$ or $\sigma = \epsilon$.*

We will say that a context forest $\mathcal{S}$ has *common* σ,τ–progenitor C if for all $D \in \mathcal{S}$, D has σ,τ–progenitor C. Note that an empty context may have more than one progenitor, due to collapse rules, which have the effect of combining existing empty contexts as well as creating new ones. Also note that the progenitor of a context C created by the last step of reduction ρ has ρ,ϵ–progenitor C.

We can now revise Definition 4.3 to account for common residuals in subterms matched nonlinearly:

Definition 4.23 (*VarPairs* for nonlinear TRSs) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{lll} VarPairs^{\rho\mathcal{A}} \triangleq & \{ \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \mid & X \in \mathcal{V}, \\ & & C \sqsubseteq (() \leftarrow \sigma_{\mathcal{A}}(X)) \ \text{or} \ C = (() \leftarrow \bullet), \\ & & q = root(C), \\ & & \mathcal{S}_1 = \{(p_L \cdot q \leftarrow C) \mid p_L \in \mathcal{O}_X(\overline{L_{\mathcal{A}}})\}, \\ & & \mathcal{S}_2 = \{(p_R \cdot q \leftarrow C) \mid p_R \in \mathcal{O}_X(\overline{R_{\mathcal{A}}})\}, \\ & & \mathcal{S}_1 \ \textit{has a common } \sigma,\tau\textit{–progenitor}, \\ & & \sigma\tau = \rho \ \} \end{array}$$

For linear TRSs, Definition 4.23 reduces to Definition 4.3, since $\mathcal{S}_1$ is always a singleton and thus has a trivial common progenitor.

In nonlinear TRSs, certain empty contexts at the "edge" of *Creating* and *Resid* have a creating effect that does not occur in the linear case; the definition of *Slice*$*$ for nonlinear systems must therefore be modified accordingly. More specifically, in the linear case, the empty contexts between *Creating* and *Resid* are irrelevant to the applicability of the redex. However, in the nonlinear case, they are indeed relevant, since if these "glue" contexts were not empty, the nonlinear match would not occur (unless, as with other contexts matched nonlinearly, the edge contexts have a common progenitor).

The following definition computes the union of the slices with respect to relevant edge empty contexts:

Definition 4.24 (*EdgeSlices*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{rcll} \mathit{EdgeSlices}^{\rho\mathcal{A}} & \triangleq & \bigsqcup\{\, C \mid & \langle \mathcal{S}_1, \mathcal{S}_2\rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}, \\ &&& (p \leftarrow \bullet) \in (\mathcal{S}_1 \cap \mathcal{O}_\bullet(\mathit{Creating}^{\rho\mathcal{A}})), \\ &&& D \textit{ is a } \sigma, \tau\textit{–progenitor of } (p \leftarrow \bullet), \\ &&& D \textit{ is not a common } \sigma, \tau\textit{–progenitor of } \mathcal{S}_1, \\ &&& \langle C, D\rangle \in \mathit{CreatedSlice}{*}^{\sigma}, \\ &&& \sigma\tau = \rho \,\} \end{array}$$

(The relation *CreatedSlice*$*$, defined formally below, is a subrelation of *Slice*$*$ in which the second elements are *created* by the last step of the reduction; this yields a slice specific to the progenitor in the definition when more than one progenitor exists). Definition 4.24 yields the union of slices with respect to empty context criteria at the “edge” between *Creating* and *Resid* that are not derived from a progenitor common to all the contexts associated with a given variable. Note that *EdgeSlices*$^{\rho\mathcal{A}}$ is always empty for linear TRSs, since for such systems, the forest $\mathcal{S}_1$ in the definition is always a singleton.

Our definition of *Slice*$*$ in the nonlinear case is essentially the same as that for the linear case, except that we must add the information in *EdgeSlices* where appropriate:

Definition 4.25 (*Slice*$*$ for nonlinear TRSs) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{rcl} \mathit{Slice}{*}^{\epsilon} & \triangleq & \{\, \langle C, C\rangle \mid C \in \mathit{Cont}(\Sigma) \,\} \\ \mathit{Slice}{*}^{\rho\mathcal{A}} & \triangleq & \mathit{ResidSlice}{*}^{\rho\mathcal{A}} \cup \mathit{CreatedSlice}{*}^{\rho\mathcal{A}} \end{array}$$

where

$$\begin{array}{rcll} \mathit{ResidSlice}{*}^{\rho\mathcal{A}} & \triangleq & \mathit{Slice}{*}^{\rho} \cdot \mathit{Resid}^{\rho\mathcal{A}} & \\ \mathit{CreatedSlice}{*}^{\rho\mathcal{A}} & \triangleq & \{\, \langle C, E\rangle \mid & E \textit{ and } \mathit{Created}^{\rho\mathcal{A}} \textit{ are not disjoint}, \\ &&& \langle C', D\rangle \in \mathit{Slice}{*}^{\rho}, \\ &&& \textit{there exists a minimal } E' \sqsupseteq E \textit{ such that} \\ &&& \qquad \langle D, E'\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}, \\ &&& C = C' \sqcup \mathit{EdgeSlices}^{\rho\mathcal{A}} \,\} \end{array}$$

Definition 4.25 is complicated by the necessity of splitting the pure residuation case from the creation case—the two cases *both* apply only when created and residuated information overlap exactly; i.e., when $\mathcal{A}$ is a collapse rule application.

While Definition 4.25, along with the auxiliary definitions, may appear rather complicated, testing whether two contexts have a common progenitor can be performed cheaply in practice if reduction is implemented using *term graph rewriting* techniques [17]. Graph rewriting causes terms that are created by contraction of sets of residuals of previous reductions to be shared in a graphical data structure. If such an implementation is used, testing whether two contexts have a common progenitor reduces to determining whether the contexts are represented by a common shared subgraph.

4.7.2 Example: slicing in a nonlinear system

Recall the reduction used in the example of Figure 4.4:

$$\begin{array}{ll} \underline{\texttt{ff} \wedge (\texttt{tt} \oplus \texttt{tt})} \equiv T_0 & \xrightarrow{\mathcal{A}_1} \; (\underline{\texttt{ff} \wedge \texttt{tt}}) \oplus (\texttt{ff} \wedge \texttt{tt}) \equiv T_1 \xrightarrow{\mathcal{A}_2} \texttt{ff} \oplus (\underline{\texttt{ff} \wedge \texttt{tt}}) \equiv T_2 \\ & \xrightarrow{\mathcal{A}_3} \; \underline{(\texttt{ff} \oplus \texttt{ff})} \equiv T_3 \xrightarrow{\mathcal{A}_4} \texttt{ff} \equiv T_4 \end{array}$$

We have denoted the contractions in the reduction above by $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, and $\mathcal{A}_4$. In the sequel, we will abbreviate the reduction sequence $\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3$ by τ.

Applying the definitions of the previous section to this example, we find that the most interesting step is the contraction $\mathcal{A}_4$, which uses nonlinear rule [B4]. In Figure 4.6, the two ff subterms in the term matched by contraction $\mathcal{A}_4$ have the *same* progenitor in the initial term, indicated by dotted lines. Definition 4.23 thus implies that the ff subterms are components of *VarPairs*. Consequently, the *Creating* context for the **[B4]** contraction does not include the ff subterms. Taken together, these facts allow us to conclude that the final term of the reduction of Figure 4.6 does *not* depend on the ff subterm of the initial term of the reduction.

It is instructive to observe the effect of the formal definitions of Section 4.7.1 with respect to contraction $\mathcal{A}_4$. In order to determine whether the contexts bound to the nonlinearly matched variable X are derived from a common source, we must first consider the common progenitors of the contexts in $\mathit{VarPairs}^{\tau\mathcal{A}_4}$, which are:

$$\begin{array}{llll} \{\, (1) \leftarrow \texttt{ff},\; (2) \leftarrow \texttt{ff} \,\} \;\sqsubseteq\; T_3 & \text{has common } \epsilon, \tau\text{–progenitor} & (1) \leftarrow \texttt{ff} \;\sqsubseteq\; T_0 \\ \{\, (1) \leftarrow \bullet,\; (2) \leftarrow \bullet \,\} \;\sqsubseteq\; T_3 & \text{has common } \epsilon, \tau\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_0 \end{array}$$

Since the contexts bound to the nonlinearly matched variable X (namely, $(1) \leftarrow \texttt{ff}$, $(2) \leftarrow \texttt{ff}$, $(1) \leftarrow \bullet$, and $(2) \leftarrow \bullet$) have a common progenitor, they are included in *VarPairs*:

$$\begin{array}{lcl} \mathit{VarPairs}^{\tau\mathcal{A}_4} & = & \{\, \langle\{(1) \leftarrow \texttt{ff}, (2) \leftarrow \texttt{ff}\}, \emptyset\rangle,\; \langle\{(1) \leftarrow \bullet, (2) \leftarrow \bullet\}, \emptyset\rangle \,\} \\ \mathit{Resid}^{\tau\mathcal{A}_4} & = & \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle \end{array}$$

Using *VarPairs*, we can eliminate the nonlinearly matched contexts from *Creating* and *Created*:

$$\begin{array}{lcl} \mathit{Creating}^{\tau\mathcal{A}_4} & = & () \leftarrow \bullet \oplus \bullet \\ \mathit{Created}^{\tau\mathcal{A}_4} & = & () \leftarrow \texttt{ff} \end{array}$$

However, before we can compute the *Slice*∗ relation, we must consider slices with respect to the "edge" empty contexts $(1) \leftarrow \bullet$ and $(2) \leftarrow \bullet$, which separate *Creating* from elements of *Resid*. Their progenitor information is as follows:

$$(1) \leftarrow \bullet \;\sqsubseteq\; T_3 \;\left\{ \begin{array}{ll} \text{has } \mathcal{A}_1\mathcal{A}_2, \mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_2 \\ \text{has } \mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_1 \\ \text{has } \epsilon, \tau\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_0 \end{array} \right.$$

$$(2) \leftarrow \bullet \;\sqsubseteq\; T_3 \;\left\{ \begin{array}{ll} \text{has } \tau, \epsilon\text{–progenitor} & (2) \leftarrow \bullet \;\sqsubseteq\; T_3 \\ \text{has } \mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (2) \leftarrow \bullet \;\sqsubseteq\; T_1 \\ \text{has } \epsilon, \tau\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_0 \end{array} \right.$$

$$\rho_1 : \quad (\underline{\texttt{ff} \wedge \texttt{ff}}) \wedge (\texttt{tt} \oplus \texttt{tt}) \equiv T_0 \xrightarrow{[B3]} \texttt{ff} \wedge (\texttt{tt} \oplus \texttt{tt}) \equiv T_1 \quad \longrightarrow^* \quad \underline{\texttt{ff} \oplus \texttt{ff}} \equiv T_4 \xrightarrow{[B4]} \texttt{ff} \equiv T_5$$

$$\rho_2 : \quad \underline{(\texttt{ff} \wedge \texttt{ff}) \wedge (\texttt{tt} \oplus \texttt{tt})} \equiv T_0 \xrightarrow{[B1]} ((\texttt{ff} \wedge \texttt{ff}) \wedge \texttt{tt}) \oplus ((\texttt{ff} \wedge \texttt{ff}) \wedge \texttt{tt}) \equiv T_1'$$
$$\longrightarrow^* (\underline{\texttt{ff} \wedge \texttt{ff}}) \oplus (\texttt{ff} \wedge \texttt{ff}) \equiv T_2' \xrightarrow{[B3]} \texttt{ff} \oplus (\underline{\texttt{ff} \wedge \texttt{ff}}) \equiv T_3' \xrightarrow{[B3]} \underline{\texttt{ff} \oplus \texttt{ff}} \equiv T_4' \xrightarrow{[B4]} \texttt{ff} \equiv T_5'$$

Figure 4.8: Sensitivity of nonlinear slicing to reduction strategy.

$(1) \leftarrow \bullet$ and $(2) \leftarrow \bullet$ each have three progenitors because the collapse rule **[B2]** (applied in contractions $\mathcal{A}_2$ and $\mathcal{A}_3$) has the effect of combining the empty contexts above and below the matched part of the redex, as well as creating a "new" empty context.

For the purpose of computing $\mathit{EdgeSlices}^{\tau\mathcal{A}_4}$, we need consider only those progenitors not common to both $(1) \leftarrow \bullet \sqsubseteq T_3$ and $(2) \leftarrow \bullet \sqsubseteq T_3$. These are: $(1) \leftarrow \bullet \sqsubseteq T_1$, $(2) \leftarrow \bullet \sqsubseteq T_1$, $(1) \leftarrow \bullet \sqsubseteq T_2$, and $(2) \leftarrow \bullet \sqsubseteq T_3$. The *CreatedSlice*$*$ subrelations relevant to the latter contexts are as follows:

$$\begin{array}{lcl}
\{\, \langle () \leftarrow \bullet \wedge (\bullet \oplus \bullet), (1) \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet \wedge (\bullet \oplus \bullet), (2) \leftarrow \bullet \rangle \,\} & \subseteq & \mathit{CreatedSlice}{*}^{\mathcal{A}_1} \\
\langle () \leftarrow \bullet \wedge (\texttt{tt} \oplus \bullet), (1) \leftarrow \bullet \rangle & \subseteq & \mathit{CreatedSlice}{*}^{\mathcal{A}_1\mathcal{A}_2} \\
\langle () \leftarrow \bullet \wedge (\bullet \oplus \texttt{tt}), (2) \leftarrow \bullet \rangle & \subseteq & \mathit{CreatedSlice}{*}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3}
\end{array}$$

Taking the context union of the *CreatedSlice*$*$ information above, we get:

$$\begin{array}{lcl}
\mathit{EdgeSlices}^{\tau\mathcal{A}_4} & = & () \leftarrow \bullet \wedge (\bullet \oplus \bullet) \ \sqcup\ () \leftarrow \bullet \wedge (\texttt{tt} \oplus \bullet) \ \sqcup\ () \leftarrow \bullet \wedge (\bullet \oplus \texttt{tt}) \\
 & = & () \leftarrow \bullet \wedge (\texttt{tt} \oplus \texttt{tt})
\end{array}$$

Combining the information computed above and using Definition 4.25, we finally have:

$$\mathit{Slice}{*}^{\tau\mathcal{A}_4} \ = \ \{\, \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet \wedge (\texttt{tt} \oplus \texttt{tt}), () \leftarrow \texttt{ff} \rangle \,\}$$

Consequently, the slice $\bullet \wedge (\texttt{tt} \oplus \texttt{tt}) \sqsubseteq T_0$ is computed for criterion $() \leftarrow \texttt{ff} \sqsubseteq T_4$.

4.7.3 Nonlinear systems and optimality

Although the approach to nonlinear slicing developed in the previous section is sound, it does not always yield minimal slices. To see this, consider the $\mathbf{B}$ reductions in Figure 4.8.

Although both ρ_1 and ρ_2 start and end at the same term, using the definitions of Section 4.7.1, the slice with respect to criterion T_5 is $(\bullet \wedge \texttt{ff}) \wedge (\texttt{tt} \oplus \texttt{tt})$, whereas the slice with respect to criterion T_5' is $(\texttt{ff} \wedge \texttt{ff}) \wedge (\texttt{tt} \oplus \texttt{tt})$, i.e., the entire initial term.

The difference in the slices results from the order in which redexes were contracted in the two reductions. In ρ_1, the $(\texttt{ff} \wedge \texttt{ff}) \equiv S_0$ subterm of T_0 is contracted immediately, and two residuals of its contractum, $\texttt{ff}$, subsequently appear in term T_4. In ρ_2, however, S_0 is not immediately contracted. Instead, the reduction produces an intermediate term T_2' containing two residuals of S_0. These residuals are contracted in subsequent steps, ultimately yielding the term T'_4. However, unlike T_4, the two `ff` subterms of T'_4 are *not* residuals of any previous term. Since the `ff` subterms of T_4 have a common progenitor, the definitions of Section 4.7.1 allow information common to the slices of the `ff` subterms of T_4 to be omitted when the nonlinear rule **[B4]** is applied. In the case of T'_4, however, the `ff` subterms have no common progenitor, and thus no information can be omitted.

It should be clear from the example of Figure 4.8 that the notion of progenitor is dependent upon reduction order. One way to avoid the problems illustrated by Figure 4.8 is to use an *innermost* reduction strategy, in which all redexes are contracted before they are residuated. However, if we do not wish to impose restrictions on allowable reduction strategies, we must take into account the behavior of reductions such as ρ_2, in which terms that have no common progenitor *could have had* a common progenitor if the redexes were contracted in a different order.

Put another way, we must treat sets of terms that are all "derived in the same way" from a set of residuals with a common progenitor as equivalent to sets of terms with a true common progenitor. Maranget [111, 112] defines a notion of equivalence modulo permutation of redexes that could, if extended to non-orthogonal systems, be used for determining when classes of terms are or could have been residuals of a common term. However, if reduction is implemented using term graph rewriting techniques, terms that have common progenitors and terms that *could have* common progenitors are indistinguishable. In the case of the example in Figure 4.8, both term T_4 and term T'_4 would be represented by identical graphs in which the `ff` subterms would be shared.

Unfortunately, even graph rewriting does not eliminate the possibility of computing suboptimal slices for nonlinear systems. Consider, for instance, the following TRS $\mathbf{E}$:

[E1]	$\mathtt{f}(X) \longrightarrow \mathtt{eq}(\mathtt{g}(X),\ \mathtt{h}(X))$	**[E4]**	$\mathtt{k}(\mathtt{a}) \longrightarrow \mathtt{b}$
[E2]	$\mathtt{h}(X) \longrightarrow \mathtt{k}(X)$	**[E5]**	$\mathtt{eq}(X, X) \longrightarrow \mathtt{c}$
[E3]	$\mathtt{g}(X) \longrightarrow \mathtt{k}(X)$		

Note in particular that rule **[E5]** is nonlinear. Now consider the following $\mathbf{E}$-reduction:

$$\begin{array}{lllllll} \rho: & \mathtt{f(a)} & \xrightarrow{[E1]} & \mathtt{eq(g(a),\ h(a))} & \xrightarrow{[E2]} & \mathtt{eq(k(a),\ h(a))} & \xrightarrow{[E3]} \ \mathtt{eq(k(a),\ k(a))} \\ & & \xrightarrow{[E4]} & \mathtt{eq(b,\ k(a))} & \xrightarrow{[E4]} & \mathtt{eq(b,\ b)} \equiv T & \xrightarrow{[E5]} \ \mathtt{c} \end{array}$$

In principle, we ought to be able to determine that the slice with respect to the final term `c` of ρ is `f(●)`, since we can attain the same final term by omitting the fourth and fifth reduction steps entirely. However it is difficult to see how any information short of maintaining the entire reduction history could be used to determine that this is the case. In particular, note that the `b` subterms of the intermediate term T in ρ do not have a common progenitor, nor are they derived in an "equivalent" way from the sets of residuals. Therefore, we cannot use information about the derivations of the `b` subterms in isolation as a means for allowing common slice information to be omitted when rule **[E5]** is applied.

We are led to conclude that short of maintaining information about an entire reduction history, the only systematic way to treat nonlinear rules is to eliminate information associated with nonlinearly-matched subterms possessing a common progenitor (generalized using graph reduction techniques to account for "potential progenitors"). It is conceivable, however, that a restricted class of reduction systems or reduction strategies could eliminate the problems exhibited in the example of Figure 4.8. We leave it to future work to explore these possibilities further.

4.8 Implementation

In principle, one could implement slicing by storing information about every step of a reduction ρ, and then computing relation $\mathit{Slice}{*}^{\rho}$ based on this information. In practice, such an approach is infeasible since it would require space and time proportional to the length of ρ for each choice of criterion. Since our reasons for investigating dependence relations are eminently practical, we use an alternative method that allows slices to be computed as a "side-effect" of the reduction process, in a way that efficiently yields slices with respect to any chosen criterion. During the reduction process, our method maintains (i) the slices for all elementary subcontexts of a term, and (ii) the context union of the slices with respect to any empty context. (Recall that $\mathit{Slice}{*}^{\rho}$ is not necessarily single-valued on empty contexts.) The latter information is associated with the non-empty elementary context (i.e., function symbol) with the same root. Using this information, a slice with respect to any non-empty context D can be determined by computing the context union of the slices with respect to all elementary and empty contexts that are a subcontext of D.

This simple scheme has been implemented in the rewriting engine of the ASF+SDF Meta-environment [93]. Slices for elementary and empty contexts are stored as *annotations* of function symbols. More precisely, each function symbol has two slices associated with it: the slice with respect to the elementary context consisting of that function symbol, and the slice with respect to the empty context "above" that function symbol. Each rewrite step that is performed by the rewriting engine has the effect of propagating "slice" information from the symbols in the redex to the symbols in the reduct. These propagations effectively compute for each symbol in the reduct the context union of the slices of a set of symbols in the redex.

The implementation of the $\mathit{Slice}{*}$ relation for *nonlinear* rewriting systems is much simpler in practice than one might infer from the definitions in Section 4.7. Recall that the main problem addressed in that section consists of defining the notion of a progenitor, and determining whether or not two contexts have progenitors in common. As the term rewriting engine of the ASF+SDF Meta-environment actually performs term *graph* rewriting, the check for common progenitors corresponds to testing whether or not two subterms are *shared*. For certain ill-behaved rewriting systems that feature a combination of nonlinear and collapse rules, our implementation produces sub-optimal results. In practice, this does not pose any problems.

Slices are efficiently represented by bit-vectors, whose size is proportional to the size of the initial term. Using bit-vectors, set union operations can be performed in time linear in the length of the vector. The number of unions per reduction step is bounded by the number of function symbols that need to be matched. Consequently, the overhead per reduction step is *linear* in the size of the initial term. In other words, performing dynamic dependence tracking has the effect of slowing down the execution (i.e., the term rewriting process) by a factor proportional to the size of the initial term. We have implemented dynamic dependence tracking in the ASF+SDF system [93], and conducted some performance measurements. In our experiments, the term rewriting engine of the ASF+SDF system is never slowed down by more than one order of magnitude.

4.9 Related work

The term "slice" was first coined by Weiser [147], and defined for imperative programming languages using dataflow analysis. Subsequent work, beginning with that of Ottenstein and Ottenstein [120], has focused on use of *program dependence graphs* [53] for computing slices. Cartwright and Felleisen [34] and Venkatesh [137] discuss the denotational foundations of dependence and slicing, respectively for similar classes of languages; however, they do not provide an operational means to *compute* slices. Chapter 3 provides a survey of current work on program slicing.

A number of authors have considered various "labeling" or "tracking" schemes that propagate auxiliary information in conjunction with reduction systems; these schemes are similar in some respects to the method we will use to implement slicing. Bertot [27, 28] defines an *origin function*, which is a generalization of the classic notions of residual and descendant in the lambda-calculus and TRSs. He applies this idea to the implementation of source-level program debuggers for languages implemented using *natural semantics* [85]. Van Deursen, Klint and Tip addressing similar problems, define a slightly expanded class of "origin" information for the larger class of *conditional* TRSs (see Chapter 2). However, slicing is not considered in these works, nor do these "tracking" algorithms propagate information appropriate for computing slices.

In [94, page 85], Klop presents a "tracing relation" that is very similar to our dynamic dependence notion, and observes that it can be used to distinguish the *needed prefix* and the *non-needed part* of a term. In our terminology, the needed part is the slice with respect to the entire normal form, and the non-needed parts correspond to the "holes" in this slice. In other words, replacing the non-needed parts by arbitrary subterms will result in the same normal form. There are two main differences with our work. First, Klop's tracing relation is only defined for *orthogonal* TRSs. Second, for collapse rules the top symbol of the reduct is considered to be "created". As we discussed earlier (see the last paragraph of Section 4.4), this gives rise to slices being non-minimal. Finally, Klop does not study the use of tracing relations for program slicing, nor does he give an algorithm to compute his relation efficiently in practice.

In certain respects, our technique is the dual of *strictness analysis* in lazy functional programming languages, particularly the work of Wadler and Hughes [138] using *projections*. Strictness analysis is used to characterize those subcomponents of a function's input domain that are always needed to compute a result; we instead determine subcomponents of a *particular* input that are *not* needed. However, there are significant differences: strictness analysis is concerned with domain-theoretic *approximations* of values, usually requires computation by fixpoint iteration, and rarely addresses more than a few core functional primitives. By contrast, we perform exact analysis on a particular input (although we can effectively perform some approximate analyses by reduction of open terms), compute our results algebraically, and can address any construct expressible in TRS form.

Maranget [111, 112] provides a comprehensive study of lazy and optimal reductions in orthogonal TRSs using labeled terms. Although Maranget's label information could in principle be used to compute slices, he does not discuss such an application, nor does he provide any means by which such labels could be used to implement slicing. Like Klop, Maranget also only considers *orthogonal* TRSs. Our approach covers a larger class of TRSs, and provides a purely *relational* definition of slice that does not require labeling.

4.10 Future work

An important question for future work is to define classes of TRSs for which slices are independent of the reduction actually used. While orthogonal systems certainly have this property, we believe it should be possible to characterize non-orthogonal systems for which this property also holds.

Chapter 5

Parametric Program Slicing

(joint work with John Field and G. Ramalingam)

Summary

Program slicing is a technique for isolating computational threads in programs. In this chapter, we show how to mechanically extract a family of practical algorithms for computing slices directly from semantic specifications. These algorithms are based on combining the notion of *dynamic dependence tracking* in term rewriting systems, which was introduced in Chapter 4, with a program representation whose behavior is defined via an equational logic [55]. Our approach is distinguished by the fact that changes to the behavior of the slicing algorithm can be accomplished through simple changes in rewriting rules that define the semantics of the program representation. Thus, e.g., different notions of dependence may be specified, properties of language-specific data types can be exploited, and various time, space, and precision tradeoffs may be made. This flexibility enables us to generalize the traditional notions of static and dynamic slices to that of a *constrained* slice, where any subset of the inputs of a program may be supplied.

5.1 Introduction

Program *slicing* is an important technique for program understanding and program analysis. Informally, a program slice consists of the program parts that (potentially) affect the values of specified variables at some designated program point—the *slicing criterion*. Although originally proposed as a means for program debugging [147], it has subsequently been used for performing such diverse tasks as program integration and "differencing" [74], software maintenance and testing [61, 51], compiler tuning [106], and parallelization of sequential code [146].

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

- We generalize the traditional notions of static and dynamic slices to that of a *constrained* slice. Static and dynamic slices have previously been computed by different techniques. By contrast, our approach provides a generic algorithm for computing constrained slices.

- Given a well-defined specification of a translation from a programming language to a common intermediate representation called PIM [55], we automatically extract a semantically well-founded language-*specific* algorithm for computing constrained slices. An advantage of this approach is that only the PIM translation is language dependent; the mechanics of slicing itself are independent of the language.
- PIM's semantics (and thus that of the source language via translation) is defined by a set of *rewriting* rules. These rules implicitly carry out many techniques used in optimizing compilers, e.g., conditional constant propagation and dead code elimination. The slices we obtain are thus often more precise than those computed by previous algorithms. By choosing different subsets of rules or adding additional rules, the precision of the analysis, as well as its time and space complexity, may be readily varied. We illustrate the flexible nature of our approach by defining several extensions to PIM's core logic. These variants describe differing treatments of loop semantics, and consequently define differing slice behaviors.

One of the primary contributions of this chapter is an algorithm for computing *constrained slices*. Despite the myriad variations on the theme of slicing that can be found in the literature (see Chapter 3), almost all existing slicing algorithms fall into one of two classes: *static slicing algorithms*, which make no assumptions about the inputs to the program, and consequently compute slices that are valid for all possible input instances, and *dynamic slicing algorithms*, which accept a specific instantiation of *all* inputs, and compute slices valid only for that specific case. A constrained slice is valid for all instantiations of the inputs that satisfy a given set of constraints. In the sequel, we will primarily consider constraints that specify the values of some subset of the input parameters of the program.

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

While Venkatesh has defined a notion of a *quasi-static* slice [137] similar to that of a constrained slice, we know of no previous work that describes how such slices may be *computed*. In a recent paper [117], Ning et al. describe a reverse engineering tool that permits users to specify constraints on variables and extract *conditional* slices, but they do not specify how these slices are computed or how powerful the constraints can be. One might consider combining partial evaluation of programs with static slicing to compute constrained slices, but, as will be explained later, this does not lead to satisfactory results.

The feasibility of the ideas in this chapter has been demonstrated by a successful prototype implementation of the PIM logic and translators for significant subsets of such disparate languages as C and Cobol using the ASF+SDF Meta-environment [93], a programming environment generator based on algebraic specifications.

5.2 Overview

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

5.2.1 Motivating example

Figure 5.1 **(a)** shows an example program written in μC, a C subset that we will use for all the examples in this chapter. μC has the standard C syntax and semantics, with one extension: *meta-variables* like `?P` and `?Q` are used to represent unknown values or inputs. All data in μC are assumed to be integers or pointers; we also assume that no address arithmetic is used. When we discuss loops in Section 5.5, we will for simplicity further restrict our analysis to programs containing only constant L-values.

The example of Figure 5.1 **(a)** is not entirely trivial, due to manipulation of pointers in a conditional statement. The static slice with respect to the final value of `result` consists of the entire program. The dynamic slice with respect to the final value of `result` for input `p = 5, q = 3` is shown in Figure 5.1 **(d)**; note that it does not immediately reveal the effect of each input. The effect of input `p = 5` is illustrated by the constrained slice of Figure 5.1 **(b)**; clearly it causes the aliasing of `*ptr` to `y`, and thereby makes both assignments to `x` obsolete. In Figure 5.1 **(c)**, the effect of the other input, `q = 3` is shown: the statements in the first branch of the second `if` statement become irrelevant. Note that in general, it is not the case that a slice with respect to multiple constraints consists of the "intersection" of the slices with respect to each constraint.

In examples in the sequel, we will use the double box notation of Figure 5.1 to denote a slicing criterion and the constraints, if any, on meta-variables. We will also use the terminology "slice of P *at* `x` [given C]" to denote the slice of P with respect to the final value of variable `x` [given meta-variable constraints C]. Slicing with respect to arbitrary expressions at intermediate program points will be discussed in Section 5.4.6.

5.2.2 Slicing via rewriting

PIM [55] consists of a rooted directed acyclic graph program representation[1] and an equational logic that operates on PIM graphs. These graphs can also be interpreted (or depicted) as *terms* after "flattening". A subsystem of the full PIM logic defines a *rewriting* semantics for a program's PIM representation. Rewriting rules can be used not only to execute programs, but also to perform various kinds of analysis by simplification of a program's PIM representation; each simplification step consists of the application of a rule of PIM's logic.

To compute the slice of a program with respect to the final value of a variable x, we begin with a term that "encodes" (i) the abstract syntax tree (AST) of the program, (ii) the variable x that represents the slicing criterion, and (iii) a (possibly empty) set of additional

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

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

result

(a)

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

result
given ?P := 5

(b)

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

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

result
given ?Q := 3

(c)

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

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

result
given ?P := 5, ?Q := 3

(d)

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

if (q < 0) {

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

result
given ?P := 5, ?Q := 3

(d′)

Figure 5.1: **(a)** Example program (= static slice). **(b)** Constrained slice with `?P` := 5. **(c)** Constrained slice with `?Q` := 3. **(d)** Constrained slice with `?P` := 5, `?Q` := 3 (= dynamic slice). **(d′)** Non-postprocessed term slice corresponding to **(d)**.

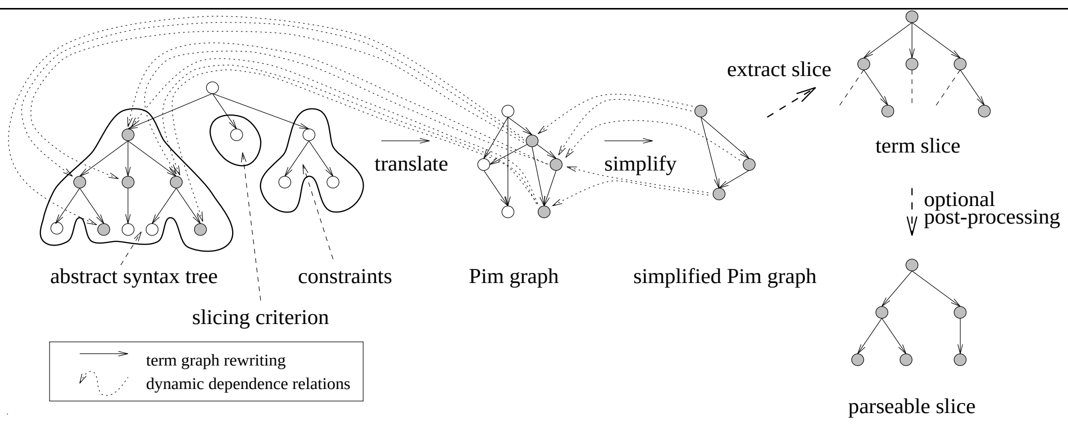


Figure 5.2: Overview of our approach.

constraints. Next, we translate the AST to a graph comprising its PIM representation. This translation is assumed to be defined by a rewriting system (although it need not necessarily be implemented that way). The resulting graph is then simplified by repeated application of sets of rewriting rules derived from the PIM logic. This *reduction* process is carried out using the technique of *term graph rewriting* [17]. The graph that results from the reduction process represents the final value of variable x (in terms of the unconstrained meta-variables). During the reduction process, we maintain *dynamic dependence relations* (see Chapter 4) that relate nodes of the graph being manipulated to the AST. These relations are defined in a simple way directly from the structure of each rewriting rule, and will be discussed in more detail in Section 5.3. By tracing the dynamic dependence relations from the simplified PIM-graph back to the AST, we finally derive the slice of the AST with respect to x. The steps involved in the slicing process are depicted in Figure 5.2.

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

5.2.3 Term slices and parseable slices

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

It is often the case, however, that one wishes obtain a *parseable* representation of the slice (i.e., a syntactically well-formed AST without missing subtrees). Therefore, term slices may be optionally postprocessed in various ways to obtain parseable programs with identical behavior[3].

Figure 5.1 **(d′)** depicts the term slice corresponding to Figure 5.1 **(d)** before postprocessing. Certain fine details are present in this term slice that do not appear in Figure 5.1 **(d)**, e.g., the L-values but not the R-values of certain assignment statements appear in the term slice.

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

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

5.2.4 More examples

The example in Figure 5.3 illustrates the flexibility of our technique by showing some of the differing treatments of loops that are possible (loops will be further studied in Section 5.5). Figure 5.3 **(b)** depicts what we will call a *pure dynamic slice* at `result`, given `?N` := 5 and `?P` := 1. Note that this slice includes the `while` loop though it computes no value relevant to the criterion. This is the case because the underlying slicing algorithm faithfully reflects the standard semantics, under which there *is* a dependence between the `while` loop and the subsequent assignment to `result`. This phenomenon is noted in Cartwright and Felleisen's discussion of demand and control dependence [34]. This notion of dependence is also closely related to the notion of *weak control dependence* discussed by Podgurski and Clarke [123]. The slice in Figure 5.3 **(c)**, similar to the kind computed by Agrawal and Horgan [6], results from adding some simple equational rules to be discussed later. The same variant of the slicing algorithm produces the result in Figure 5.3 **(d)**, though the program is non-terminating for the constraints specified under the standard semantics. Previous dynamic slicing algorithms [6, 100] will not terminate for this input constraint. In

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

[3]Whether or not this is always possible strongly depends on the language under consideration. E.g., in languages where an **if**-statement with empty branches is not allowed, removal of all statements from the **if**-branches is clearly a problem. We believe that, in cases like this, term slices are an improvement over "conventional" program slices, as they are capable of conveying more accurate information. A similar observation was made by Ernst [52].

this sense, our dynamic slicing algorithm is “more consistently lazy”.

As a final example, consider the program in Figure 5.4. Although absurdly contrived, the example illustrates several important points. By not insisting that the slice be parseable, we can make distinctions between assignment statements whose R-values are included but whose L-values are excluded and vice versa, as Figure 5.4 **(b)** shows. We also see that it is possible to determine that the values tested in a conditional are irrelevant to the slice, even though the body is relevant. In general, our approach can make a variety of fine distinctions that other algorithms cannot.

Figure 5.4 **(c)** gives an example of a *conditional constraint*. Such constraints can be handled by straightforward extensions to our basic algorithm, and will be discussed briefly in Section 5.4.7.

5.3 Term rewriting and dynamic dependence tracking

Our approach to slicing is based on extending the generic notion of *dynamic dependence tracking* in term rewriting systems (see Chapter 4) to realistic programming languages. In this section, we review dynamic dependence tracking and the basic ideas behind term and graph rewriting. For further details on term rewriting, the reader is referred to the excellent tutorial survey of Klop [95].

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

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

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

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

may be rewritten as follows (subterms affected by rule applications are underlined):

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

Observe in the example above that the outer context $\wedge\langle \bullet\,,\,\mathbf{F}\rangle$ (‘$\bullet$’ denotes a missing subterm) is not affected at all, and therefore occurs in T_0, T_1, and T_2. Furthermore, the occurrence of variables p_1, p_2, and p_3 in both the left-hand side and the right-hand side of (B14) causes the subterms **T**, **F**, and $\wedge\langle \mathbf{F}\,,\,\mathbf{T}\rangle$ of the underlined subterm of T_0 to reappear in T_1. Also note that variable p occurs only in the left-hand side of (B10): consequently, the subterm (of T_1) $\vee\langle \mathbf{F}\,,\,\wedge\langle \mathbf{F}\,,\,\mathbf{T}\rangle\rangle$ matched against p does not reappear in T_2. Thus, the subterm matched against p is *irrelevant* for producing the constant **T** in T_2: the “creation”

```
n = ?N;                 n = ?N;
i = 1;                  i = 1;
sum = 0;                       ;
while (i != n) {        while (i != n) {
  sum = sum + i;                       ;
  i = i + 1;              i = i + 1;
}                       }
if (?P)                 if (?P)
  result = n*(n-1)/2;     result = n*(n-1)/2;
else                    else
  result = sum;                       ;
                               result
                        given ?N := 5, ?P := 1

          (a)                     (b)
```

```
n = ?N;                 n = ?N;
     ;                       ;
       ;                       ;



;                       ;
if (?P)                 if (?P)
  result = n*(n-1)/2;     result = n*(n-1)/2;
else                    else
              ;                       ;
       result                  result
given ?N := 5, ?P := 1  given ?N := 0, ?P := 1

          (c)                     (d)
```

Figure 5.3: **(a)** An example program. **(b)** Pure dynamic slice at `result` given `?N` := 5, `?P` := 1. **(c)** Lazy dynamic slice at `result` given `?N` := 5, `?P` := 1. **(d)** Lazy dynamic slice at `result` given `?N` := 0, `?P` := 1.

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

	x given ?A $:= 2$	x given ?A > 5

(a) **(b)** **(c)**

Figure 5.4: **(a)** An example program. **(b)** Constrained slice at x given ?A $:= 2$. **(c)** Conditional constrained slice at x given ?A > 5.

of this subterm **T** only requires the presence of the matched symbols '$\vee$' and '**T**'. This observation is the keystone of our reduction-based slicing technique: We "track" those subterms that are relevant to each reduction step; subterms that are relevant to *no* reduction step can then be eliminated from the slice.

The tracking process determines not only which subterms are relevant to a given reduction step, but also how subterms are combined and propagated by the reduction as a whole. To accomplish this task, we define for each reduction step that takes a term T_i and yields a new term T_{i+1} the notions of *creation* and *residuation*. These are binary relations between the nodes of T_i and the nodes of T_{i+1}. The creation relation relates the new symbols in T_{i+1} produced by the rewriting step to the nodes of T_i that matched the symbols in the left-hand side of the rewriting rule (making the rewriting step possible). The residuation relation relates every other node in T_{i+1} to the corresponding occurrence of the same node in the T_{i+1}. The dynamic dependence relation for a multi-step reduction r then consists, roughly speaking[4], of the transitive closure of creation and residuation relations for the rewriting steps in r. Figure 5.5 shows all the relations for the example reduction discussed above.

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

[4]The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. The exact problems posed by these rules are extensively discussed in Chapter 4.

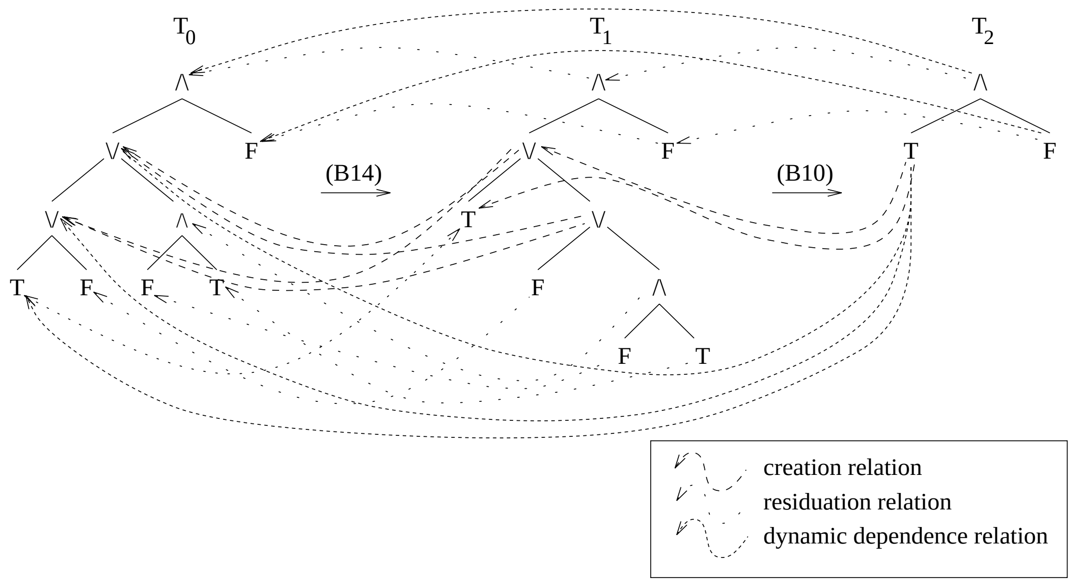


Figure 5.5: Example of creation and residuation relations.

S reduces to a term C' containing context C via a reduction r', and (ii) r' is a subreduction of r. These properties are rendered pictorially in Figure 5.6, and have the important implication that all the slices computed by our technique are effectively "executable" with respect to the rewriting semantics.

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

Returning to the example of Figure 5.5, we can determine the term slice with respect to the constant **T** in T_2 by tracing back all creation and residuation relations to T_0. By following the transitive relations in Figure 5.5; the reader may verify that this slice consists of the subcontext $\vee\langle\vee\langle\mathbf{T}\ ,\ \bullet\rangle\ ,\ \bullet\rangle$.

5.3.1 Efficient implementation of term rewriting

We implement term rewriting using the technique of *term graph rewriting* [17]. This technique extends the basic idea of term rewriting from labeled trees to rooted, labeled graphs, or *term graphs*. A term graph may be viewed as a term by traversing it from its root and replacing all shared subgraphs by separate copies of their term representations. For clarity, we will frequently depict PIM term graphs or subgraphs in "flattened" form

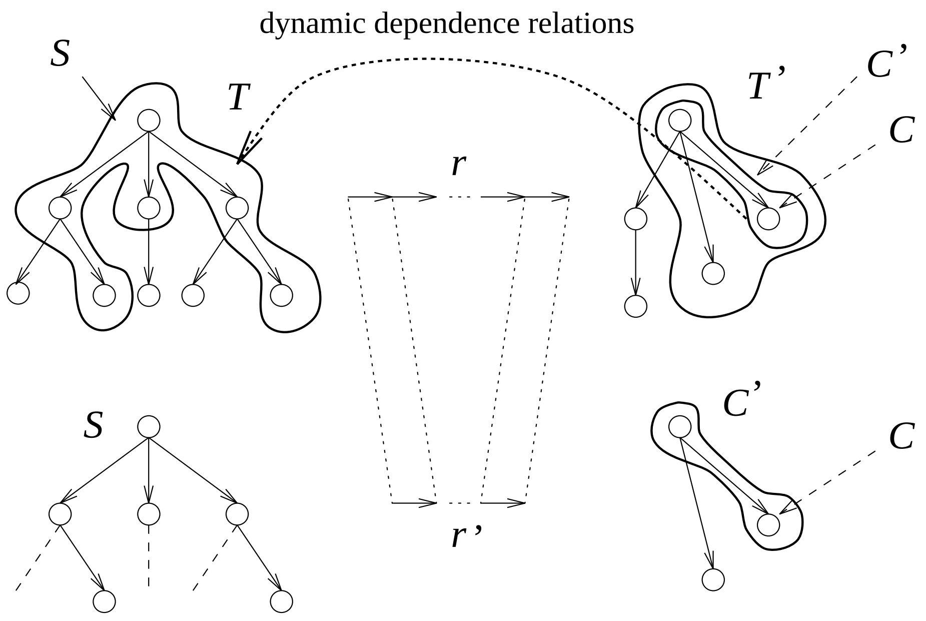


Figure 5.6: The concept of dynamic dependence.

as terms. (The flattened representation of the graph T_2 in Figure 5.7, for instance, is $\vee\langle\wedge\langle \mathbf{T}\ ,\ \mathbf{F}\rangle\ ,\ \wedge\langle \mathbf{T}\ ,\ \mathbf{T}\rangle\rangle$.)

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

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

In rule (B22), the variable p_1 appears *twice* on the right-hand side. Although the left-hand side instance of p_1 in (B22) matches only a single subterm, the result of the rule application must contain two instances of the subterm matched by p_1. Rather than duplicating such a term, it can be shared, as illustrated by the example in Figure 5.7, in which rule (B22) is applied to term $T_0 \equiv \wedge\langle\vee\langle \mathbf{T}\ ,\ \mathbf{F}\rangle\ ,\ \vee\langle \mathbf{F}\ ,\ \mathbf{T}\rangle\rangle$. We see also from Figure 5.7 that the result of a single application of reduction rule (here, rule (B10)) *inside* a shared subterm can also be shared, thus giving the effect of multiple reductions for the price of one.

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

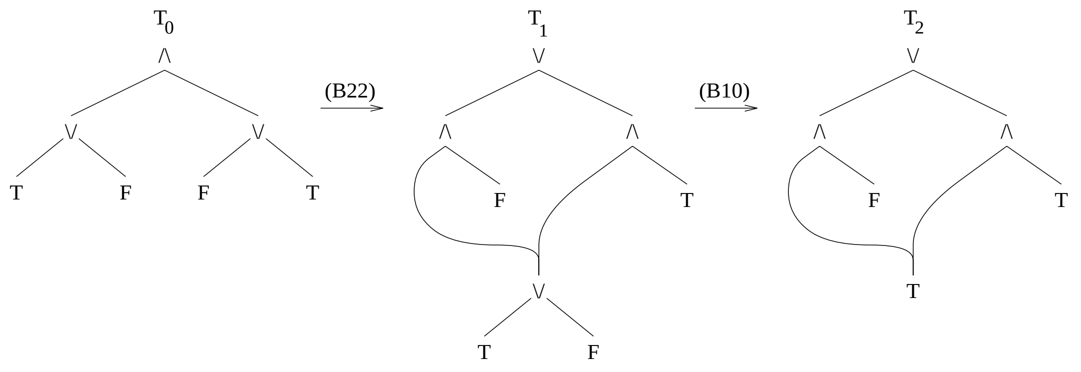


Figure 5.7: Creation of shared subgraphs and a shared reduction step using a graph rewriting implementation.

5.4 PIM + dynamic dependence tracking = slicing

PIM was designed to generalize and rationalize many of the properties of commonly used graphical representations for imperative programs such as SSA-form [40] and PDGs [53], and to provide a semantically sound but mechanizable framework for performing program analysis and optimization. PIM's formal progenitor is Cartwright and Felleisen's notion of *lazy store* [34], interpreted operationally rather than denotationally. Unlike SSA-form and PDGs, computations on addresses required for arrays or pointers are "first-class citizens," and procedures and functions are integral parts of the formalism.

5.4.1 μC-to-PIM translation

Figure 5.8 depicts a very simple μC program, P_1, its corresponding PIM representation, and several slicing-related structures.

The graph depicted in Figure 5.8, denoted by $\mathit{Slice}(P_1, \mathtt{x}, \langle\rangle)$, is generated by translating P_1 to its corresponding PIM representation and embedding the resulting graph (labeled S_{P_1}) in a graph corresponding to the slicing criterion x. $\mathit{Slice}(P_1, \mathtt{x}, \langle\rangle)$ is simply the PIM expression denoting the final value of the variable x. Only a small number of graph edges, primarily those connecting shared subgraphs to multiple parents are shown explicitly in Figure 5.8; we have flattened most other subgraphs for clarity. Parent nodes in the graph are depicted *below* their children to emphasize the correspondence between program constructs and corresponding PIM subgraphs.

S_{P_1} is generated by a simple syntax-directed translation. A representative subset of the translation rules appears in Figure 5.9. The translation is specified in the Natural Semantics style [85] for clarity; however, the translation is implemented by a pure rewriting system[5].

[5]A rewriting system can be derived from simple classes of Natural Semantics specifications such as the one in Figure 5.8 in a purely mechanical fashion.

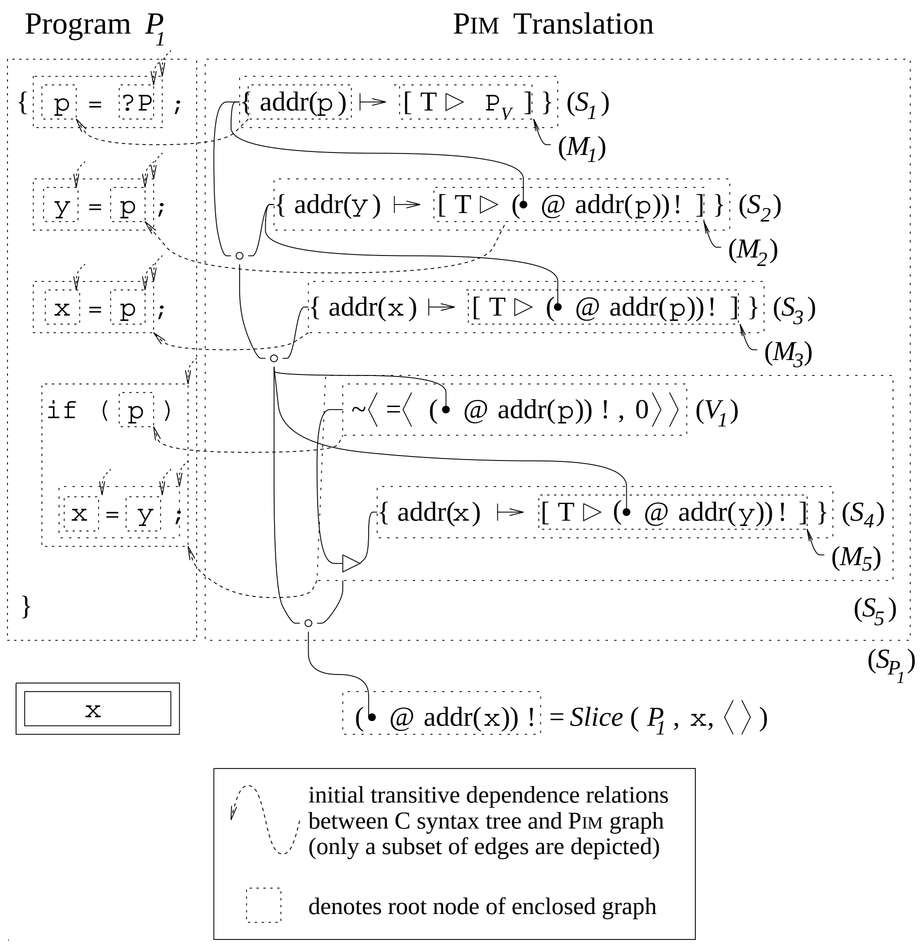

Program P_1
PIM Translation
{ p = ?P ;
y = p ;
x = p ;
if (p)
x = y ;
}
{ addr(p) ↦ [T ▷ P_V] } (S_1)
(M_1)
{ addr(y) ↦ [T ▷ (• @ addr(p))!] } (S_2)
(M_2)
{ addr(x) ↦ [T ▷ (• @ addr(p))!] } (S_3)
(M_3)
~⟨ =⟨ (• @ addr(p)) ! , 0⟩⟩ (V_1)
{ addr(x) ↦ [T ▷ (• @ addr(y))!] } (S_4)
(M_5)
(S_5)
(S_{P_1})
x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

Figure 5.8: P_1 and its PIM representation, S_{P_1}. Major corresponding structures in P_1 and S_{P_1} are located side-by-side.

The translation uses several sequent forms corresponding to the principal C syntactic components. The general form for these sequents is:

$$s \vdash c \Rightarrow t$$

Such a sequent may be read as "μC construct c translates to PIM term t, given initial (PIM) store s. '$\Rightarrow$' is subscripted by 'Pgm', 'Exp', or 'LValue', depending on whether a statement, expression, or L-value (address), respectively, is being translated. Pure expressions (those having no side-effects) and unpure expressions are distinguished in the translation process; subscripts p and u are used to denote the two types. The shared subgraphs in S_{P_1} arise from repeated instances of store variables in the antecedents of the translation rules in Figure 5.9, as illustrated in Figure 5.7.

The translation process establishes transitive dependence relations between nodes of the program's AST and the PIM graph S_{P_1}, as described in Section 5.3. Figure 5.8 depicts a representative subset of these relations for the root nodes of certain subtrees of the syntax tree of P_1. We have used vestigial arrows in the syntax tree to indicate that nodes are referred to by *some* set of nodes in the PIM graph. We have also depicted statements of P_1 and their corresponding PIM subgraphs side-by-side.

5.4.2 Overview of PIM

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

The graph S_{P_1} as a whole is a PIM *store* structure[6], essentially an abstract term representation of memory. S_{P_1} is constructed from the sequential composition (using the '$\circ$' operator) of substores corresponding to the statements comprising P_1. The subgraphs accessible from boxes labeled S_1–S_4 in Figure 5.8 correspond to the four assignment statements in P_1. The simplest form of store is a *cell* such as

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

A store cell associates an *address expression* (here addr($\mathtt{p}$)) with a *merge structure,* (here $[\mathbf{T} \rhd \mathtt{P}_{\mathcal{V}}]$). *Constant* addresses such as addr($\mathtt{p}$) represent ordinary variables. More generally, address *expressions* are used when addresses are computed, e.g., in pointer references. '$\emptyset_s$' is used to denote the empty store.

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

[6]For clarity, Figure 5.8 does not depict certain *empty* stores created by the translation process; this elision will be irrelevant in the sequel.

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

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

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

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

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

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

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

$$(E_{p1})\qquad s \vdash \text{ Id } \Rightarrow_{\text{Exp}_p} (s \mathbin{@} \text{addr(Id)})\ !$$

$$(E_{p2})\qquad s \vdash \text{ ?Id } \Rightarrow_{\text{Exp}_p} \text{Id}_{\mathcal{V}}$$

$$(E_{u1})\qquad \frac{s \vdash \text{ Exp } \Rightarrow_{\text{Exp}} \langle v,\ u\rangle}{s \vdash \texttt{ * } \text{Exp} \Rightarrow_{\text{Exp}_u} \langle((s \circ u) \mathbin{@} v)\ !,\ u\rangle}$$

$$(E_{u2})\qquad \frac{s \vdash \text{ LValue } \Rightarrow_{\text{LValue}} \langle v,\ u\rangle}{s \vdash \texttt{ \& } \text{LValue} \Rightarrow_{\text{Exp}_u} \langle v,\ u\rangle}$$

$$(E_{u3})\qquad \frac{\begin{array}{c} s \vdash \text{ Exp } \Rightarrow_{\text{Exp}_u} \langle v_E,\ u_E\rangle, \\ s \circ u_E \vdash \text{LValue} \Rightarrow_{\text{LValue}} \langle v_L,\ u_L\rangle \end{array}}{\begin{array}{c} s \vdash \text{LValue} \texttt{ = } \text{Exp} \Rightarrow_{\text{Exp}_u} \\ \langle v_E,\ u_E \circ u_L \circ \{v_L \mapsto [\mathbf{T} \rhd v_E]\}\rangle \end{array}}$$

$$(E_{u4})\qquad \frac{\begin{array}{c} s \vdash \text{Exp}_1 \Rightarrow_{\text{Exp}_u} \langle v_1,\ u_1\rangle, \\ s \circ u_1 \vdash \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \langle v_2,\ u_2\rangle \end{array}}{\begin{array}{c} s \vdash \text{Exp}_1 \texttt{ + } \text{Exp}_2 \Rightarrow_{\text{Exp}_u} \\ \langle +\langle v_1\ ,\ v_2\rangle,\ u_1 \circ u_2\rangle \end{array}}$$

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

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

Figure 5.9: Representative translation rules for μC.

Merge expressions m_1 and m_2 may be composed into *ordered lists* of the form $m_1 \circ_m m_2$, in which the *rightmost* guarded cell takes precedence. Such lists correspond roughly to Lisp **cond** expressions, and represent information similar to SSA-form ϕ nodes [40], particularly the *gated* SSA variant of [14]. Unlike normal conditional expressions, however, merges cannot evaluate to values unless they are referred to in a special context represented by the *selection* operation, '!'. Among other places, this operator is used in the translation of every variable reference. S_{P_1} contains no non-trivial merge structures, but such structures will arise in the simplification process. $\emptyset_m$ denotes the null merge structure. In the sequel, we will often drop subscripts distinguishing related store and merge constructs when no confusion will arise.

In addition to guards in merge cells, stores such as S_5 (which corresponds to the '`if`' statement as a whole) may also be guarded. The guard expression V_1 corresponds to the `if`'s predicate expression. Consistent with standard C semantics, the guard V_1 tests whether the value of the variable `p` is nonzero.

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

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

is

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

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

5.4.3 PIM rewriting and elimination of dependences

PIM's equational logic consists of an "operational" subsystem, PIM$^{\rightarrow}$, plus a set of additional non-oriented equational rules for reasoning about operational equivalences in PIM$^{\rightarrow}$, instances of which can also be oriented for use in analysis. PIM$^{\rightarrow}$ is confluent and normalizing (assuming an appropriate strategy), thus it can be viewed as defining an operational semantics or interpreter for PIM terms. An important subsystem of PIM$^{\rightarrow}$ that defines the semantics of programs without loops or procedures, PIM$_t^{\rightarrow}$, is *canonical*, that is, *strongly* normalizing as well as confluent. PIM$^{\rightarrow}$ can be enriched with certain oriented instances of rules in (PIM$-$PIM$^{\rightarrow}$) in such a way that confluence is preserved on closed terms, and such that unique normal forms for open terms exist up to certain trivial permutations. In the sequel, we will refer to the enriched rewriting system as COREPIM$^{\rightarrow}$. PIM's rules and subsystem structure are described in detail in [55]; key subsystems are reviewed in Appendix A.

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

5.4.4 Reduction of unconstrained and constrained slices

Figure 5.10 depicts key steps in the reductions of $\mathit{Slice}(P_1, \texttt{x}, \langle \texttt{?P} := 0\rangle)$ and $\mathit{Slice}(P_1, \texttt{x}, \langle\rangle)$, the slices of P_1 that result from these reductions, and certain dependence relations for reduction steps that are critical to producing the slices. These reductions share a common initial subsequence that is independent of the substitution generated in the constrained case. We have numbered certain important intermediate graphs in the reductions. The interpretation of several of these graphs (depicted in flattened form) is as follows:

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

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

which have the effect of distributing the reference to the variable x, $\mathrm{addr}(\texttt{x})$, to the sequence of substores S_1, S_2, S_3, S_5. Graph (3) results from applications of the rule

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

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

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

5.4.4.1 Constrained case: *Slice*$(P_1, \texttt{x}, \langle \texttt{?P} := 0\rangle)$

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

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

has the effect of eliminating the body of the `if` from the final slice. This can be seen in detail in the "exploded" (L6) rule application in Figure 5.10. In this case, the only transitive

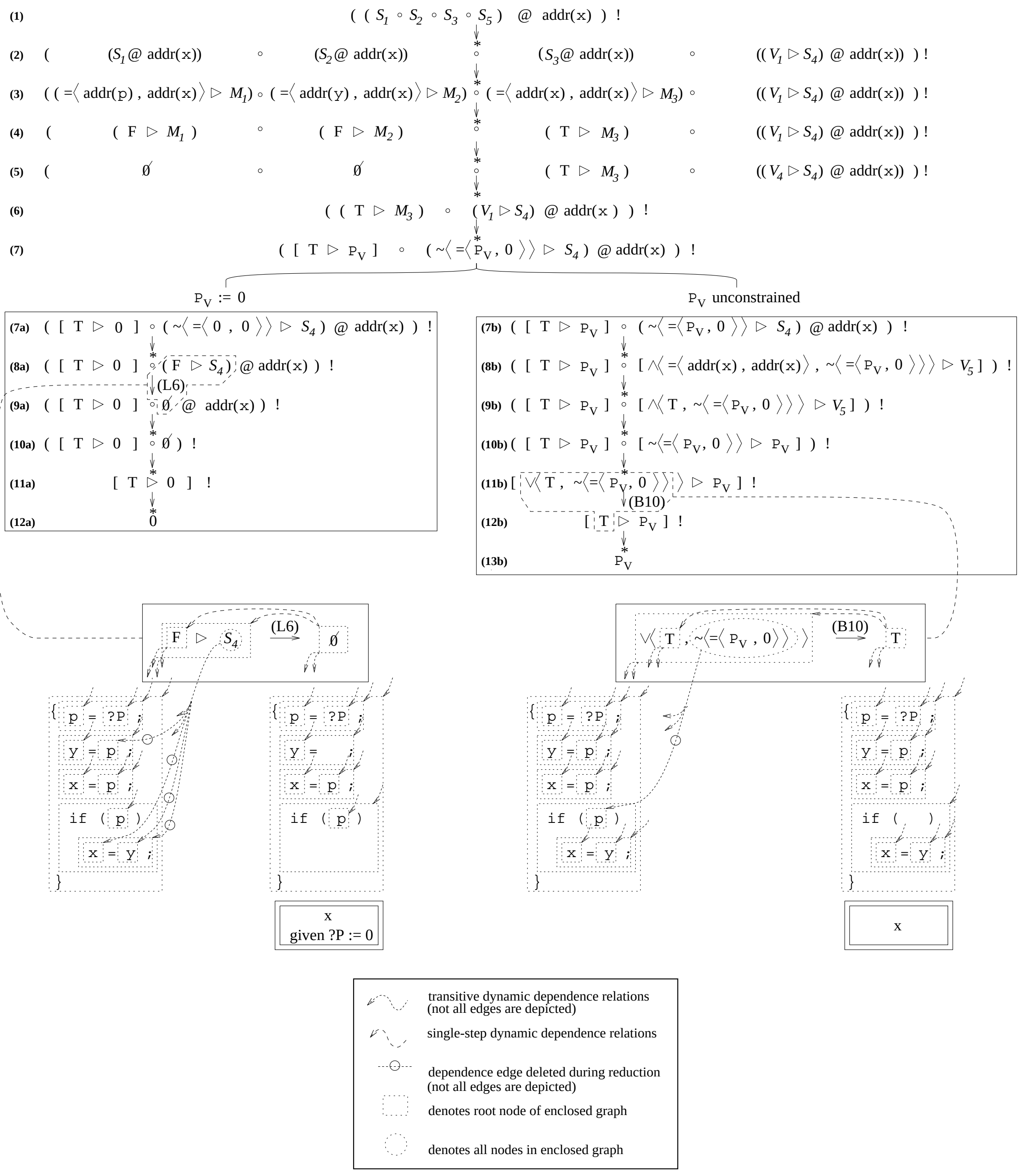

(1) ((S1 ∘ S2 ∘ S3 ∘ S5) @ addr(x)) !
(2) ((S1@ addr(x)) ∘ (S2@ addr(x)) ∘ (S3@ addr(x)) ∘ ((V1 ▷ S4) @ addr(x))) !
(3) ((=⟨ addr(p) , addr(x)⟩ ▷ M1) ∘ (=⟨ addr(y) , addr(x)⟩ ▷ M2) ∘ (=⟨ addr(x) , addr(x)⟩ ▷ M3) ∘ ((V1 ▷ S4) @ addr(x))) !
(4) ((F ▷ M1) ∘ (F ▷ M2) ∘ (T ▷ M3) ∘ ((V1 ▷ S4) @ addr(x))) !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

Figure 5.10: Reduction of $\mathit{Slice}(P_1, \texttt{x}, \langle\rangle)$ and $\mathit{Slice}(P_1, \texttt{x}, \langle \texttt{?P} := 0\rangle)$. Steps in which removal of dependence edges eliminates constructs from slices are highlighted along with the resulting slices.

dependence edges linking the constructs in the body of the `if` statement and the identifier `p` in the assignment `y = p;` have their origin in the subgraph S_4. When this subgraph is eliminated by the application of (L6), the constructs effectively disappear from the slice.

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

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

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

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

Slices that contain statements from the arms of a conditional statement but not its predicate, are unusual enough to deserve some discussion. Such slices indicate that the value of the predicate itself is irrelevant, even though the conditional statement contains some relevant statement, e.g., an assignment to some relevant variable. Such situations *can* arise in realistic programs. Consider, for example, the statement

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

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

5.4.5 Slicing and reduction strategies

As PIM$^{\rightarrow}$ is a confluent rewriting system, reductions may be performed anywhere in a graph without affecting the final term produced (assuming the reduction terminates at all). This "stateless" property of reduction systems accommodates a variety of performance tradeoffs derived from varying the reduction *strategy*. We use an outermost or "lazy" strategy, which ensures that only steps that contribute to a final result are performed. (Note, however, that the reduction depicted in Figure 5.10 uses a strategy that is not strictly outermost to better illustrate the properties of certain intermediate terms). Alternatively, the PIM representation of the entire program could be normalized "eagerly" prior to the specification of any slicing criterion; those steps specific to the criterion or constraints could be performed later.

Reduction strategies can also have an effect on slices. In the constrained case, both of the reductions depicted in Figure 5.10 are valid, and, consequently *both* of the slices depicted are also valid. Slices are therefore not necessarily unique, even when the underlying reduction system is confluent. However, our reduction strategy favors the left reduction over the right one in the constrained case. Intuitively, this favors a "standard" execution semantics that corresponds most closely with results of traditional program slicing algorithms.

5.4.6 Slicing at intermediate program points

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

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

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

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

5.4.7 Conditional constraints

A slice with respect to a *conditional constraint* such as that depicted in Figure 5.4 **(c)** can be computed by constructing a PIM graph roughly equivalent to that which would be produced by inserting the body of the program in a conditional statement where the predicate is the conjunction of all such constraints.

The effectiveness of our slicing algorithm in handling conditional constraints depends primarily on its ability to reason about the operations allowed in such constraints. The extensible nature of our approach makes it easy to augment the slicing algorithm by incorporating sophisticated reasoning capabilities about particular domains into the slicing algorithm, as it only involves adding rewrite rules characterizing the appropriate domains. For example, in the case of Figure 5.4 **(c)**, rudimentary rules for reasoning about arithmetic inequalities suffice to compute the slice shown.

5.4.8 Complexity tradeoffs

Use of different normalizing subsets of PIM equations allows various accuracy/time tradeoffs in the analysis process. For instance, pointer-induced alias analysis is NP-complete even in the absence of loops and procedures [104], although such analysis is usually tractable in practice. By including or excluding appropriate PIM rules, one can effectively choose more precise (but potentially slow) or more conservative (but guaranteed fast) pointer analysis. For instance, eliminating rule (M3) (see Figure A.1) effectively inhibits propagation of symbolic addresses representing pointer values, thus preventing these expressions involving these addresses from being resolved or simplified. Rule (L11) has the effect of joining common results of common expression propagation (including address expressions) in different branches of a conditional, and can be enabled, disabled, or restricted to prevent or allow such propagation.

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

5.5 Variations on a looping theme

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

5.5.1 loop execution rules: pure dynamic slicing

Loops are represented in PIM by terms of the form

$$Loop(\lambda x_{\mathcal{S}}.body(u_E, v_E, u_S), s)$$

Informally, u_E is a store representing the side-effects of evaluating the loop predicate, v_E represents the value of the predicate, u_S is a store representing the side-effects of the loop

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

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

Figure 5.11: Loop execution rules.

body, all as functions of the store $x_{\mathcal{S}}$ at the beginning of a loop iteration. The second argument s is the incoming store. The term $Loop(\lambda x_{\mathcal{S}}.body(u_E, v_E, u_S), s)$ itself denotes the store representing the side-effects of executing the loop until the predicate evaluates to false. The rewriting rules in Figure 5.11, which we will refer to as loop execution rules, specify this behavior formally. Consider the underlined subterm of the right-hand side of the rule (loop). The underlined subterm may be read as: the side-effects of evaluating the loop predicate and—if the predicate evaluates to true, the side-effects of executing the loop body once—composed with the side-effects of executing the same loop with an appropriately updated store, namely, $x_{\mathcal{S}} \circ_s u_E \circ_s u_S$. The rest of the term serves to express the recursion using the recursion combinator **Y**. $f[x := g]$ represents the result of substituting g for free occurrences of x in f (with the usual provisos about variable capture and renaming), $FV(\tau)$ is the set of free variables in a term τ, and 'S' is a technicality—a "sort coercion" operator that has no semantic content. [54] shows how the β rule and substitution can be encoded as pure rewriting rules.

Utilizing these rules and COREPIM$^{\rightarrow}$ during the simplification phase leads to a straightforward dynamic slicing algorithm that we call a *pure dynamic slicing algorithm*. Section 5.2.4 discusses an example (Figure 5.3 **(b)**) of this sort of slice.

5.5.2 ϕ rules: lazy dynamic slicing

Figure 5.12 depicts a set of rules, the ϕ rules, that statically simplify PIM stores generated by loops. The effect of ϕ rules on the PIM representation is essentially to introduce an SSA-form ϕ node [40] for every variable that might be assigned a value inside the loop. In terms of slicing, these rules have the effect of permitting loops to be removed from slices if it can be determined (statically) that the loop cannot assign to any "variable of interest".

We will refer to the slicing algorithm obtained by using both the loop execution rules and the ϕ rules in conjunction with COREPIM$^{\rightarrow}$ as a *lazy dynamic slicing algorithm*. This slicing algorithm computes traditional dynamic slices, such as Figure 5.3 **(c)**, as well as the somewhat more unusual result in Figure 5.3 **(d)**.

The slices produced by our lazy dynamic slicing algorithm are closer to the slices produced by the Agrawal-Horgan algorithm [6] than to the slices produced by the Korel-

$$
\begin{array}{rcll}
\mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s) & \longrightarrow & & \\
& & \mathit{Project}(\ \mathit{Loop}(\lambda x_{\mathcal{S}}.\mathit{body}(u_E, v_E, u_S), s), & \\
& & \quad \mathit{AssignedVars}(u_E \circ_s u_S)\ ) & \text{(LA1)} \\
\mathit{Project}(s, \{\}) & \longrightarrow & \emptyset_s & \text{(LA2)} \\
\mathit{Project}(s, \{v\} \cup r) & \longrightarrow & (v \mapsto s\ @\ v) \circ_s \mathit{Project}(s, r) & \text{(LA3)} \\
& & \vdots & \\
& & \text{rules for computing } \mathit{AssignedVars}(s), & \\
& & \text{the set of variables assigned to in the store s} & \\
& & \vdots &
\end{array}
$$

Figure 5.12: ϕ rules.

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

$$\boxed{\begin{array}{c}\texttt{z}\\ \text{given } \texttt{?X} := 5\end{array}}$$

(a) **(b)**

Figure 5.13: **(a)** Example program. **(b)** Dynamic slice at `z` given `?X` := 5.

Laski algorithm [100]. The Korel-Laski slices tend to be larger than the Agrawal-Horgan slices since they, unlike the Agrawal-Horgan slices, are executable. Our dynamic slices, though not executable under the "standard semantics", *are* executable with respect to the semantics specified by the rewriting rules.

Figure 5.13 **(a)** illustrates an important difference between our algorithm and previous dynamic slicing algorithms. Since the `if` predicate evaluates to false, previous dynamic slicing algorithms exclude the predicate (and any of the statements the predicate evaluation is dependent upon) from the dynamic slice with respect to `z`. However, as has been observed before [100, 121], the predicate does "affect" the final value of `z`, in the sense that the predicate must be executed in order to ensure that the value of `x`, and thus `z`, is *not* negated. In other words, the result of the predicate evaluation has an effect on the final result, even though the statement guarded by the predicate is not executed. In applications such as debugging it is useful to include these statements in the slice [121, 122]. In this sense, our slicing algorithm produces a slice that is semantically more consistent than existing dynamic slicing algorithms.

5.5.3 loop splitting rules: static loop slicing

Figure 5.14 contains the essential subset of a collection of rules that we will refer to as loop splitting rules, which can be used in conjunction with CorePim$^{\rightarrow}$ to compute a classical static slice. The goal of these rules is quite simple. Consider the Pim term

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

This reduces to a term of the form

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

where u_S, representing the loop body, is the store

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

Intuition suggests that this term should be reducible to

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

where ${u_S}'$ is the store

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

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

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

to a term

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

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

The rules in Figure 5.14 show the essence of what we need to do. Rule (SA1) simply transforms a dereference operation on a store computed by a loop into a corresponding dereference operation on a *restriction* of the loop-computed store. This leaves the bulk of the work to the operator '@@', whose purpose is to restrict a store to a set of addresses of interest. Rule (SA2) is the key rule defining the behavior of this operator. (The operator $\in$ may be interpreted as denoting the usual set-theoretic member function.)

The rule of primary interest is (SA6), which performs the restriction operation for a store generated by a loop. Restricting a loop-computed store with respect to a set l of addresses requires restricting the loop body store and the initial incoming store with respect to a set of addresses r. The set r is a superset of the set l, and effectively accounts for loop-carried

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

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

$\vdots$

rules for computing $\mathit{Demand}(s, x_{\mathcal{S}})$, the set of
addresses dereferencing free instances of $x_{\mathcal{S}}$ in s

$\vdots$

Figure 5.14: Loop splitting rules.

dependences. The antecedent of rule (SA6) specifies the condition this set r has to satisfy, namely that the set of variables r should include the set l, the set of variables required to compute the loop predicate, and the set of variables necessary to compute values assigned to the variables in r within the loop. Put another way, the antecedent of (SA6) ensures that the set of variables r is transitively closed with respect to loop-carried dependences. The auxiliary function $Demand(t, x_{\mathcal{S}})$, roughly, identifies "upwards exposed" variables in t. More formally, given a store or merge t, $Demand(t, x_{\mathcal{S}})$ identifies dereferences of the free store variable $x_{\mathcal{S}}$ in t and collects the address operands of such dereferences.

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

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

5.5.4 loop invariance rules: invariance-sensitive slicing

We now turn our attention to the final set of rules, which we will refer to as loop invariance rules. We will refer to the slicing algorithm obtained by using these rules, the CorePim$^{\rightarrow}$ rules, and the loop splitting rules, as an *invariance sensitive* slicing algorithm. If the loop execution rules are used as well, we obtain a β *invariance-sensitive* algorithm. The primary difference between the two algorithms is that the latter will execute (i.e., unfold) a loop as long as its predicate evaluates to a constant. Figure 5.15 illustrates this behavior. The β invariance-sensitive slicing algorithm, by executing the loop, discovers that *two* of the three assignments to `y` in the loop are irrelevant for the given input constraint `?N` $:= 5$. The simpler algorithm avoids unfolding the loop; however, by effectively performing constant propagation, it discovers that *one* of the three assignment statements is irrelevant.

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

$$Loop(\lambda x_{\mathcal{S}}.body(u_E, v_E, u_S), s) \ @ \ a$$

The store u_S represents the loop body, and free occurrences of $x_{\mathcal{S}}$ in u_S denote the store at the beginning of a specific loop iteration. In the presence of loop invariants, one can simplify the store u_S further. For instance, consider the example in Figure 5.15 **(c)**. The store u_S compiled for the loop body will contain subterms of the form $(x_{\mathcal{S}} \ @ \ \text{addr}(x))!$, denoting the value of variable `x` in a particular iteration. Since the value of `x` is a loop-invariant constant `6` (given the constraint `?N` $:= 5$), we would like to replace this term by `6`. This replacement, in turn, will allow further simplifications of the store, and ultimately lead to the slice depicted in the figure.

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

The rules in Figure 5.16 formalize these intuitions. The most important rule is (IA1), a conditional rule in the style of rule (SA6) (Figure 5.14). The consequent of the rule specializes the loop body and loop predicate of a loop-computed store with respect to the loop-invariant part of the store, namely s_{inv}. The antecedent guarding the applicability of the rule “defines” what it means for a part of the store to be loop invariant. This definition is stated in terms of a *subsumption* relationship ‘$\succeq$’ between program stores. A store s_1 subsumes a store s_2, if for every variable x assigned a value v in store s_2, x is also assigned the same value v in store s_1. The subsumption relation is concisely defined by the equational axiom (IA2). Rules (IA2.1) through (IA2.3) represent a conservative approximation to ‘$\succeq$’ that is more “directly computable,” since it is defined inductively. Less conservative approximations to (IA2) can also be defined that allow inference of more complex loop invariants.

Returning to the notion of a loop-invariant store, the store s_{inv} is considered to be loop invariant if (i) The incoming store s (the store before the loop begins its first iteration)

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

z
given ?N := 5

z
given ?N := 5

(a) **(b)** **(c)**

Figure 5.15: **(a)** Example program. **(b)** β invariance-sensitive slice at `z` given `?N` := 5. **(c)** Simple invariance-sensitive slice at `z` given `?N` := 5.

subsumes s_{inv}, and (ii) The loop body $u_E \circ_s u_S$, specialized for an incoming store $x_{\mathcal{S}} \circ_s s_{inv}$ that subsumes s_{inv}, and then composed with s_{inv} results in a store that subsumes s_{inv}.

As with rule (SA6) discussed in the section on static slicing, rule (IA1) cannot be used directly by the dependence tracking system. However, we can use the rule in conjunction with any algorithm for identifying loop invariants, such as the conditional constant propagation algorithm of Wegman et al. [141].

5.6 Pragmatics

A prototype implementation of our methods has been completed using the ASF+SDF Meta-environment [93]. The results obtained from this prototype have been encouraging, and we are now engaged in implementing a "free-standing" reduction-based slicing system using the most efficient possible implementation techniques. In this section, we briefly touch on several pragmatic issues that arise in implementing our approach.

5.6.1 Properties of graph reduction

Term graph reduction is a simple technique that can be implemented efficiently when an automaton-based matching algorithm and outermost reduction strategies are used. This leads us to believe that it should scale well to relatively large programs. Graph reduction also

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

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

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

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

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

Figure 5.16: Loop invariance rules.

has the advantage that results of reductions performed in shared subgraphs are immediately available to all supergraphs from which they are accessible. This means, e.g., that a slice can be computed in a subprogram (such as a procedure), and the results later used in computing the slice with respect to the entire program. It also means that reduction steps that are independent of a given criterion, but dependent on the program, can be shared and reused when new criteria are supplied.

5.6.2 Alternative translation algorithms

As alluded to in Section 5.2.2, it is not strictly necessary to use a rewriting system to translate a source program to PIM. Any algorithm to perform the translation suffices, *provided* that the dynamic dependence relations between the source AST and its PIM translation are correctly initialized. However, the correctness of these initial relations must be established by hand.

5.6.3 Chain rules

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

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

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

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

Figure 5.17: **(a)** Example program. **(b)** Example program after optimization using constant propagation and dead code elimination, given `?X` := 5. The static slice of the optimized program at `z` is the optimized program itself. **(c)** Our constrained slice at `z` given `?X` := 5.

5.7 Related work

PIM was introduced as a semantically sound internal representation for program analysis in [55]. The theoretical underpinnings of the notion of dynamic dependence were developed in Chapter 4 for arbitrary term rewriting systems. In this chapter, we have augmented PIM's logic (particularly for loop analysis), and applied the notion of dynamic dependence to it to develop a family of extensible slicing algorithms for standard programming languages, exploiting in particular the possibility of computing slices with respect to constraints.

Some previous algorithms [38, 51, 89] combine both static and dynamic information to compute slices, but primarily to combine the efficiency of static slicing algorithms with the accuracy of dynamic slicing algorithms. The notion of constrained slices is not studied in these papers.

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

Consider the example in Figure 5.17. Given the input constraint `?X` := 5, the program in Figure 5.17 **(a)** can be simplified using constant propagation and dead code elimination to yield the program shown in Figure 5.17 **(b)**. However, a static slice of this optimized program at `z` fails to provide the same information as our constrained slice of the original program with respect to the same criterion (Figure 5.17 **(c)**). This is due to the fact that the predicate of the `if` statement (which evaluates to false) is relevant to the computation of the final value of `z`, and should therefore be included in the slice. Slicing is intended to indicate *how* the value of a variable or expression is computed, not merely *what* its value may be. For further details on the relation between previous work on partial evaluation and PIM, see [55].

Ernst [52] presents an algorithm for static slicing that is similar to our algorithm in certain respects. In particular, Ernst describes how conventional program optimization techniques can be used to produce smaller and better slices. The internal representation that Ernst uses, the value dependence graph (VDG), has similarities with PIM, and the process of optimization itself consists of transforming the VDG. Ernst refers to the problem of maintaining a correspondence between the VDG and the source code graph throughout the optimization process, and mentions that this correspondence enables a history mechanism for explaining the transformations that were performed. No details are presented as to how this is done, but this aspect of Ernst's work appears to be the analogue of the notion of dynamic dependence used in our work to maintain a similar correspondence.

In Chapter 6, it is shown that dynamic dependence tracking can be used to compute accurate *dynamic* slices from a simple "interpreter-style" semantics for a language, and that these techniques are useful for debugging.

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

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

5.8 Future work

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

There are many areas in which our techniques could be generalized or expanded. Some subjects of ongoing research include:

Arbitrary control flow: We are investigating techniques for translating programs with arbitrary control flow to PIM, and the effects of such a translation on slicing.

Interprocedural and higher-order slicing: The techniques used for loop analysis (which are essentially a form of abstract interpretation in disguise) ought to be adaptable to the more general problem of analysis of procedures and functions.

Arrays, address arithmetic, and dynamic allocation: Proper treatment of these constructs requires a more complicated model of memory than simple symbolic addresses.

Other symbolic analysis techniques: It would be useful to be able to adapt our notion of dynamic dependence to more powerful theorem-proving techniques, such as those incorporating higher-order or equational unification or resolution.

Complexity of strategies and subsystems: Various PIM subsystems and reduction strategies need to be carefully studied in isolation to determine their worst-case complexity versus their ability to make slices precise. A set of stratified subsystems should be designed to enable users to make decisions interactively about tradeoffs between precision and speed.

Chapter 6

Generation of Source-Level Debugging Tools

Summary

Algebraic specifications have been used successfully as a formal basis for software development. The contribution of this chapter is to show that the *origin* relation, defined in Chapter 2, and the *dynamic dependence* relation, defined in Chapter 4, that are implicitly defined by an algebraic specification can be used to define powerful language-specific tools. In particular, it is shown how tools for source-level debugging and dynamic program slicing can be derived from algebraic specifications of interpreters.

6.1 Introduction

Algebraic specifications [23] have been used successfully for the generation of a variety of software development tools, such as type-checkers [42], interpreters [49], and program analysis tools [55] (see also Chapter 5). In this chapter, it is shown how two language-independent techniques, origin tracking (see Chapter 2) and dynamic dependence tracking (see Chapter 4), can be used to derive powerful language-specific debugging tools from algebraic specifications of interpreters. In particular, we show that—in addition to "standard" debugger features such as single-stepping, state inspection, and breakpoints—a variation of dynamic program slicing [6, 100, 89] can be defined with surprisingly little effort. The main contribution of our work is to show that the information required to construct such debugging tools is to a very large extent *language-independent* and *implicitly* present in a language's specification. Hence, we do not require the specification writer to add extensive descriptions for constructing them.

It is assumed that specifications are executed as conditional term rewriting systems [95]. Specifically, an algebraic specification of an interpreter expresses the execution of a program as the rewriting of a term consisting of a function `execute` applied to the abstract syntax tree of that program[1]. Rewriting this term will produce a sequence of terms that effectively represent the consecutive internal states of the interpreter. Origin tracking is a method

[1]Of course, interpreters can be specified in different "styles" than the one used here. However, the approach pursued here is highly suitable for the purpose of origin tracking, and experience has shown that realistic languages can easily be specified in this way [49, 134].

Program	::=	**declare** *DeclSeq* **begin** *StatSeq* **end**
DeclSeq	::=	*Decl; DeclSeq* | ϵ_d
StatSeq	::=	*Stat; StatSeq* | ϵ_s
Decl	::=	*Var*
Stat	::=	*Var* := *Exp* | **if** *Exp* **then** *StatSeq* **else** *StatSeq* **end** | **while** *Exp* **do** *StatSeq* **end**
Exp	::=	*Constant* | *Var* | (*Exp* + *Exp*) | (*Exp* - *Exp*) | (*Exp* * *Exp*) | (*Exp* = *Exp*)

Figure 6.1: Syntax of **L**.

for tracing occurrences of the *same* subterm in a sequence of terms, and will be used for the definition of single-stepping and breakpoints. Dependence tracking establishes certain minimal dependence relations between terms in a rewriting sequence, and will be used to obtain dynamic slices.

We illustrate our ideas by way of a very simple example language whose execution semantics are defined in Section 6.2. In Section 6.3, origin tracking and dependence tracking are presented in some detail. In Section 6.4 and 6.5, we discuss how *language-specific* tools for debugging and program slicing can be implemented using the language-independent techniques of Section 6.3. Practical experience with our approach is discussed in Section 6.6. Section 6.7 reviews other generic approaches for obtaining language-specific debuggers. Conclusions and directions for future work are reported in Section 6.8.

6.2 Specification of an interpreter

The techniques of this chapter will be illustrated by way of a simple imperative language **L** that features assignment statements, **if** statements, **while** statements, and statement sequences. **L**-expressions are constructed from constants, variables, arithmetic operators '`+`', '`-`', and '`*`', and the equality test operator '`=`'. Figure 6.1 shows a BNF grammar for the language **L**.

Figure 6.2 shows an algebraic specification that defines the execution of **L**-programs. The execution of an **L**-program P corresponds to the *rewriting* of the *term* `execute`(t_P) according to the specification of Figure 6.2, where t_P is the term that constitutes the abstract syntax tree (AST) of P. The result of this rewriting process is a term that represents a list containing the final value of each variable.

Term rewriting may be regarded as a cyclic process. Each cycle involves determining a subterm t and a rule $l = r$ such that t and l match. This is the case if a substitution σ can be found that maps every variable X in l to a term $\sigma(X)$ such that $t \equiv \sigma(l)$ (σ distributes over function symbols). For rewrite rules without conditions, the cycle is completed by replacing t by the instantiated right-hand side $\sigma(r)$. A term for which no rule is applicable for any of its subterms is called a *normal form*; the process of rewriting a term to its normal form (if it exists) is referred to as *normalizing*. A conditional rewrite rule (such as **[L16]**) is only applicable if all its conditions succeed; this is determined by instantiating and normalizing the left-hand side and the right-hand side of each condition. A positive condition (of the

/* top-level function for execution of programs */

[L1] `execute`(**declare** *DeclSeq* **begin** *StatSeq* **end**) = `exec`(*StatSeq*, `create`(*DeclSeq*, ϵ_e))

/* functions for creation and manipulation of environments */

[L2] `create`(ϵ_d, *Env*) = *Env*
[L3] `create`(*Var*;*DeclSeq*, *Env*) = `create`(*DeclSeq*, *Var* $\mapsto$ `0`; *Env*)
[L4] `lookup`(*Var* $\mapsto$ *Constant*;*Env*, *Var*) = *Constant*
[L5] `lookup`(*Var* $\mapsto$ *Constant*;*Env*, *Var*$'$) = `lookup`(*Env*, *Var*$'$) **when** *Var* $\neq$ *Var*$'$
[L6] `update`(*Var* $\mapsto$ *Constant*;*Env*, *Var*, *Constant*$'$) = *Var* $\mapsto$ *Constant*$'$;*Env*
[L7] `update`(*Var* $\mapsto$ *Constant*;*Env*, *Var*$'$, *Constant*$'$) = *Var* $\mapsto$ *Constant*; `update`(*Env*, *Var*$'$, *Constant*$'$)
when *Var* $\neq$ *Var*$'$

/* evaluation of expressions */

[L8] `eval`(*Constant*, *Env*) = *Constant*
[L9] `eval`(*Var*, *Env*) = `lookup`(*Var*, *Env*)
[L10] `eval`((*Exp* + *Exp*$'$), *Env*) = `intadd`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L11] `eval`((*Exp* - *Exp*$'$), *Env*) = `intsub`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L12] `eval`((*Exp* * *Exp*$'$), *Env*) = `intmul`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L13] `eval`((*Exp* = *Exp*$'$), *Env*) = `inteq`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))

/* execution of (lists of) statements */

[L14] `exec`(ϵ_s, *Env*) = *Env*
[L15] `exec`(*Var* := *Exp*;*StatSeq*, *Env*) = `exec`(*StatSeq*, `update`(*Env*, *Var*, `eval`(*Exp*, *Env*)))

[L16] `exec`(**if** *Exp* **then** *StatSeq* **else** *StatSeq*$'$ **end**;*StatSeq*$''$, *Env*) = `exec`(*StatSeq*$''$, `exec`(*StatSeq*, *Env*))
when `eval`(*Exp*, *Env*) $\neq$ `0`
[L17] `exec`(**if** *Exp* **then** *StatSeq* **else** *StatSeq*$'$ **end**;*StatSeq*$''$, *Env*) = `exec`(*StatSeq*$''$, `exec`(*StatSeq*$'$, *Env*))
when `eval`(*Exp*, *Env*) = `0`

[L18] `exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, *Env*) =
`exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, `exec`(*StatSeq*, *Env*))
when `eval`(*Exp*, *Env*) $\neq$ `0`
[L19] `exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, *Env*) = `exec`(*StatSeq*$'$, *Env*)
when `eval`(*Exp*, *Env*) = `0`

Figure 6.2: Algebraic specification of an **L**-interpreter.

```
declare
  i; s; p;
begin
  i := 5;
  s := 0;
  p := 1;
  while i do
    s := (s + i);
    p := (p * i);
    i := (i - 1);
  end;
end
```

(a)

```
p ↦ 120;
s ↦ 15;
i ↦ 0;
```

(b)

Figure 6.3: **(a)** Example **L**-program. **(b)** Environment obtained by executing the program of **(a)** according to the specification of Figure 6.2.

form $t_1 = t_2$) succeeds if and only if the resulting normal forms are syntactically equal, a negative condition ($t_1 \neq t_2$) if they are syntactically different.

The specification of Figure 6.2 is based on the manipulation of an environment, i.e., a list containing the current value for each variable in the program. Rule **[L1]** defines the top-level function `execute` in terms of two other functions, `create` and `exec`. The former, `create`, uses the declarations in the program to create an initial environment, where each variable is initialized with the value `0` (rules **[L2]** and **[L3]**)[2]. The latter, `exec`, specifies the execution of a list of statements; it "uses" the functions `lookup` (rules **[L4]**–**[L5]**) for retrieving a value from an environment, and `update` (rules **[L6]**–**[L7]**) for updating the value of a variable in an environment. Rules **[L8]**–**[L13]** define a recursive function `eval` for evaluating **L**-expressions. The specification of the operations `intadd`, `intsub`, `intmul`, and `inteq` on integer constants has been omitted. Rule **[L14]** states that executing an empty list of statements has the effect of leaving the environment unchanged. In rules **[L15]**–**[L19]**, the cases are specified where the list of statements is non-empty. Rule **[L15]** defines the execution of an assignment statement in terms of the evaluation of its right-hand side expression, and an update of the environment. In **[L16]**–**[L17]** the execution of a non-empty list of statements beginning with an **if**–**then**–**else** statement is defined by conditional rules; **[L16]** and **[L17]** correspond to situations where the control predicate evaluates to any non-zero value and zero, respectively. The execution of a **while** statement is specified in a similar way (rules **[L18]**–**[L19]**). Observe that in the case where the control predicate evaluates to a non-zero value, a new **while** statement is "generated" by the right-hand side of the rule.

Figure 6.3 **(a)** shows an example **L**-program. By applying the equations of Figure 6.2, the environment of Figure 6.3 **(b)** is produced.

[2]This specification assumes that every variable is properly declared.

6.3 Basic techniques

In this section, we will briefly present the origin and dependence relations that are implicitly defined by the specification of Figure 6.2. Figure 6.4 depicts some of the relations established by origin tracking and dynamic dependence tracking as a result of executing the program of Figure 6.3. The figure shows the initial term S, the final term T and an intermediate term U that occur in the process of executing the program according to the specification of Figure 6.2. The intermediate term U corresponds to the situation where the **while** loop is entered for the first time.

Subterms of U and S that are related by the origin relation are indicated by dashed lines in Figure 6.4.

Also shown in Figure 6.4 is a subcontext S' of S that is related to the subterm U' of U via the dynamic dependence relation. Observe that S' *excludes* the right-hand sides of two of the assignment statements in the program. One of the key properties of the dynamic dependence relation is that replacing these right-hand sides by *any* **L**-expression will yield a term that can be rewritten (via a subreduction of `r`) to a term that contains a subcontext `p` $\mapsto$ `1`.

Although origin and dependence relations are computed in a similar manner, using similar information as input, the nature of these relations is different. This is mainly due to the fact that these relations were designed with different objectives in mind. Origin information always involves *equal* terms. In the example of Figure 6.4, origin tracking establishes relations between a number of syntactically[3] equal terms; in this case corresponding to the statements of the program. Equality (via convertibility of terms) also plays an important role in the notion of dependence tracking. Dynamic dependence relations are in principle defined for any subcontext of any term that occurs in a rewriting process: associated with a subcontext s is the minimal subcontext of the initial term that was necessary for "creating" a term that contains s. In the sequel, we are primarily interested in the dynamic dependence relations for subcontexts that represent values computed by a program (such as the subterm U' in Figure 6.4). It will be shown below that for these subcontexts, the dynamic dependence relation will compute information that is similar to the notion of a dynamic program slice.

6.3.1 Origin tracking

In the discussion below, it is assumed that a term S is rewritten to a term T in zero or more steps: $S \rightarrow^* T$. In Chapter 2, the origin relation is formally defined as a relation between subterms of S and subterms of T; associated with every subterm T' of T is a set of subterms, *OriginOf*(T'), of the initial term S—the *origin of* T'. The principal properties of the origin relation may be summarized as follows:

[3]For the purpose of debugging, origin relations rarely involve terms that are not syntactically equal. Examples of origin relations involving terms not syntactically equal are mainly to be found in the area of error-reporting [49, 48, 44].

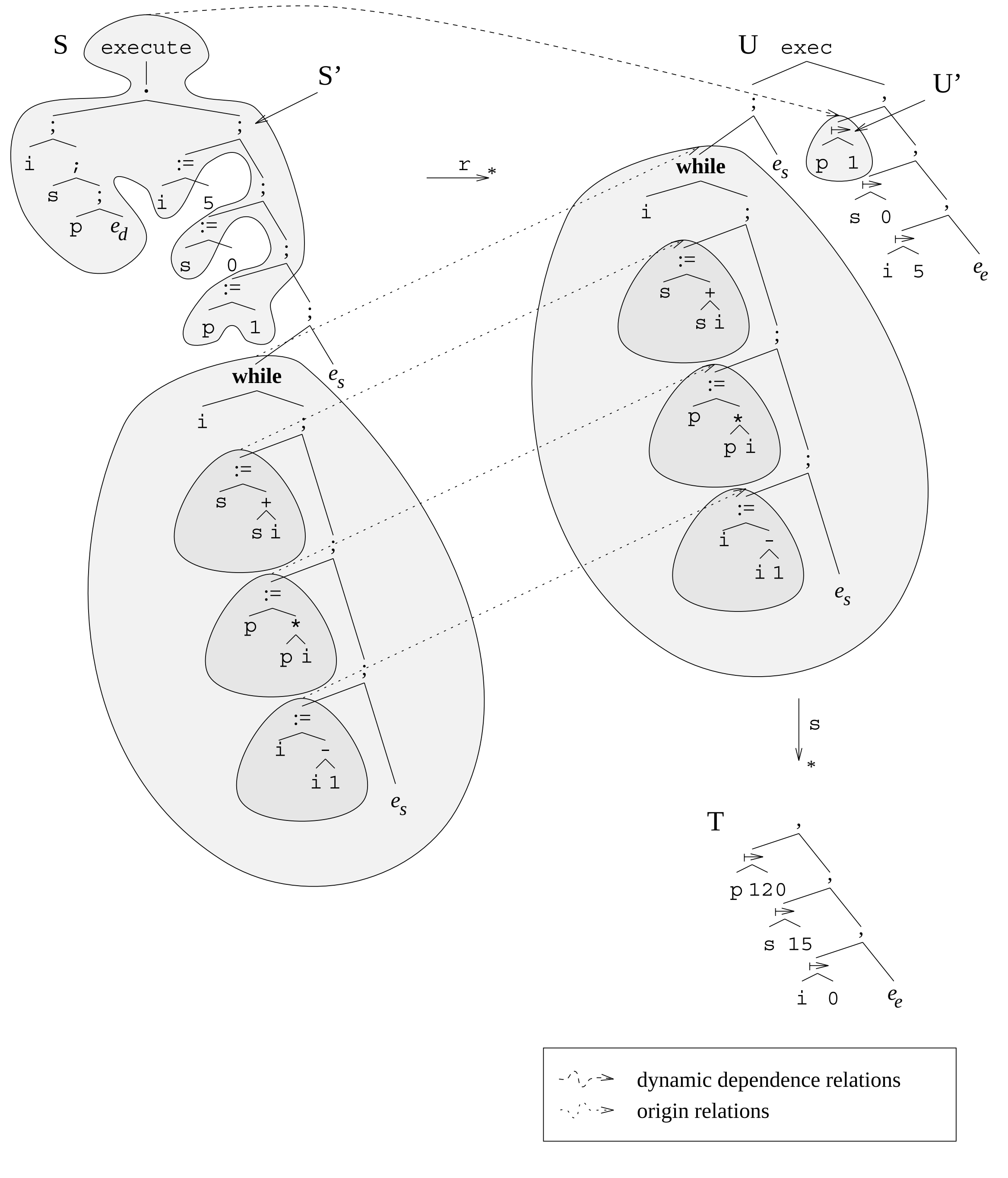


Figure 6.4: Illustration of origin and dynamic dependence relations.

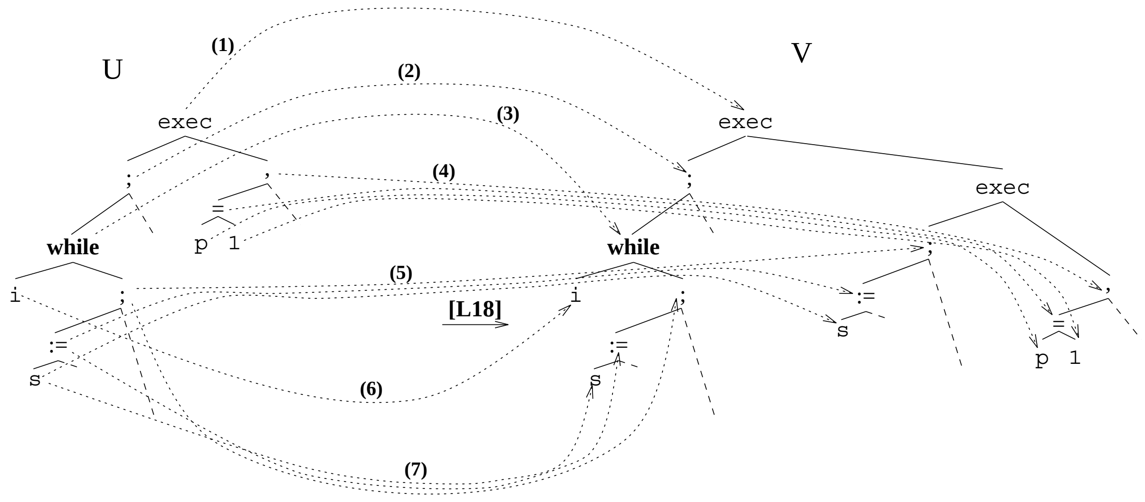


Figure 6.5: Origin relations.

- Relations involve *equal* terms (in the sense of rewriting): for each subterm $S' \in$ *OriginOf*(T') we have $S' \rightarrow^* T'$.
- Relations are defined in an inductive manner. For a reduction of length zero, the origin relation is the identity relation; for a multi-step reduction $S \rightarrow^* T \xrightarrow{r} U$, the origin of a subterm U' of U is defined in terms of the origins of subterms of T, and the structure of the applied rule, r.

As an example, Figure 6.5 partially shows the term U of Figure 6.4, and the term V it rewrites to via an application of **[L18]**. Dotted lines in the figure indicate origin relations. The relation labeled **(1)** is the relation between the root of U and V—such a relation is always present. Variables that occur in both the left-hand side and the right-hand side of **[L18]** cause more origin relations to appear—variable *Exp* gives rise to the relation labeled **(6)**, variable *StatSeq* to the sets of relations labeled **(5)** and **(7)**, and variable *Env* to the relations labeled **(4)**. The relation labeled **(3)** is caused by the occurrence of a subterm **while** *Exp* **do** *StatSeq* **end** in both the left-hand side and the right-hand side of **[L18]**. Relation **(2)** is also caused by a common subterm.

Note that the rightmost `exec` function symbol in term V is not related to any symbol in U—its origin is the empty set. In general, a term will have a non-empty origin if it was derived directly from a subterm of the initial term (here: the abstract syntax tree of a program). In Chapter 2, a number of sufficient constraints on specifications is stated that guarantee that origin sets of subterms with a specific root function symbol, or of a specific sort, contain at least one, or exactly one element. The specification of Figure 6.2 satisfies the constraints necessary to guarantee that each “statement” subterm will have an origin set containing exactly one element. For specifications that do not conform to these constraints, the origin relation of [43, Chapter 7] may be used, which is applicable to any specification of a compositional nature.

6.3.2 Dynamic dependence tracking

We introduce dynamic dependence tracking by considering a few simple rules for integer arithmetic:

$$\begin{array}{lll} \textbf{[A1]} & \texttt{intmul}(\texttt{0}, X) & = \texttt{0} \\ \textbf{[A2]} & \texttt{intmul}(\texttt{intmul(}X\texttt{, }Y\texttt{)}, Z) & = \texttt{intmul}(X, \texttt{intmul(}Y\texttt{, }Z\texttt{)}) \end{array}$$

By applying these rules, the term `intsub(3, intmul(intmul(0, 1), 2))` may be rewritten as follows (subterms affected by rule applications are underlined):

$$\begin{array}{lll} T_0 = & \texttt{intsub}(\texttt{3}, \underline{\texttt{intmul(intmul(0, 1), 2)}}) & \longrightarrow^{\textbf{[A2]}} \\ T_1 = & \texttt{intsub}(\texttt{3}, \underline{\texttt{intmul(0, intmul(1, 2))}}) & \longrightarrow^{\textbf{[A1]}} \\ T_2 = & \texttt{intsub}(\texttt{3}, \texttt{0}) & \end{array}$$

By carefully studying this example reduction, we can make the following observations:

- The outer context `intsub(3,` ●`)` of T_0 ('●' denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.
- The occurrence of variables *X*, *Y*, and *Z* in both the left-hand side and the right-hand side of **[A2]** causes the respective subterms `0`, `1`, and `2` of the underlined subterm of T_0 to reappear in T_1.
- Variable *X* only occurs in the left-hand side of **[A1]**. Consequently, the subterm (of T_1) `intmul(1, 2)` matched against *X* does not reappear in T_2. In fact, we can make the stronger observation that the subterm matched against *X* is *irrelevant* for producing the constant `0` in T_2: the "creation" of this subterm `0` only requires the presence of the context `intmul(0,` ●`)` in T_1.

The above observations are the cornerstones of the dynamic dependence relation that is defined in Chapter 4. Notions of *creation* and *residuation* are defined for single rewrite-steps. The former involves function symbols that are produced by rewrite rules whereas the latter corresponds to situations where symbols are copied, erased, or not affected by rewrite rules[4]. Figure 6.6 shows all residuation and creation relations for the example reduction discussed above.

Roughly speaking, the dynamic dependence relation for a multi-step reduction ρ consists of the transitive closure of creation and residuation relations for the individual rewrite steps in ρ. In Chapter 4, the dynamic dependence relation is defined as a relation on *contexts*, i.e., connected sets of function symbols in a term. The fact that C is a *subcontext* of a term T is denoted $C \sqsubseteq T$. For any reduction ρ that transforms a term T into a term T', a *term slice* with respect to some $C' \sqsubseteq T'$ is defined as the subcontext $C \sqsubseteq T$ that is found by tracing back the dynamic dependence relations from C'. The term slice C satisfies the following properties:

1. C can be rewritten to a term $D' \sqsupseteq C'$ via a reduction ρ', and

[4]The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. This is discussed at greater length in Chapter 4.

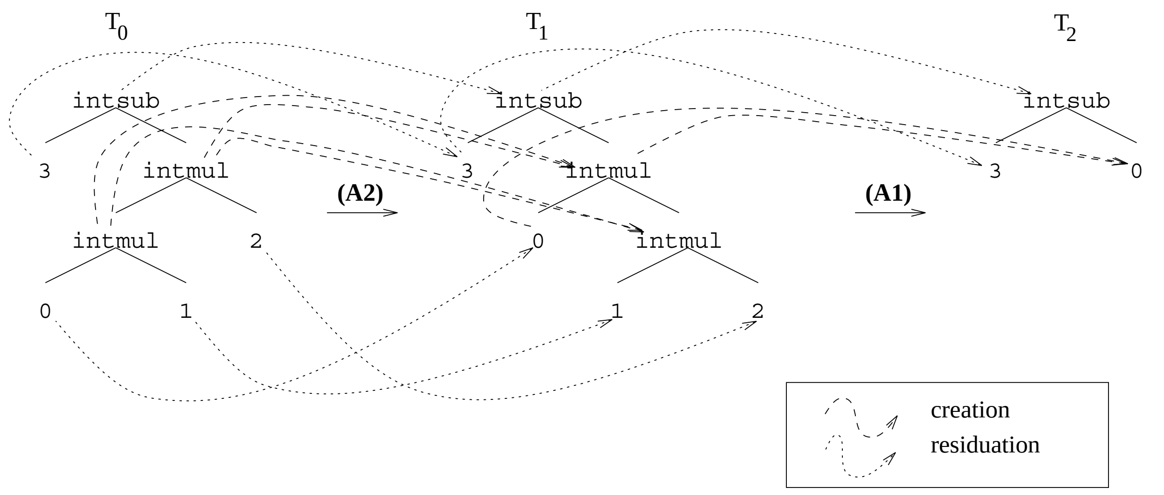


Figure 6.6: Example of creation and residuation relations.

2. ρ' is a subreduction of the original reduction ρ. Intuitively, ρ' contains a subset of the rule applications in ρ.

For precise definitions of contexts, subcontexts, and subreductions, the reader is referred to Chapter 4. The definition of a term slice is rendered pictorially in Figure 6.7.

In cases where no confusion arises, we will simply write $C = \mathit{SliceOf}(C')$ to indicate that C is the term slice with respect to C' for some reduction $\rho : T \to^* T'$, $C \sqsubseteq T$, and $C' \sqsubseteq T'$.

Returning to the example of Figure 6.6, we can determine the term slice with respect to the entire term T_2 by tracing back all creation and residuation relations to T_0; the reader may verify that `intsub(3, intmul(intmul(0, ●), ●))` = *SliceOf*(`intsub(3, 0)`).

6.3.3 Implementation

Origin tracking and dynamic dependence tracking have been implemented in the rewrite engine of the ASF+SDF Meta-environment [93]. All function symbols of all terms that arise in a rewriting process are *annotated* with their associated origin and dependence information; this information is efficiently represented by way of bit-vectors. Whenever a rewrite rule is applied to a term t, and a new term t' is created, origin and dependence information is *propagated* from t to t'. These propagations are expressed in terms of operations on sets. In Chapters 2 and 4, it is argued that the cost of performing these propagation steps is at worst linear in the size of the initial term of the reduction.

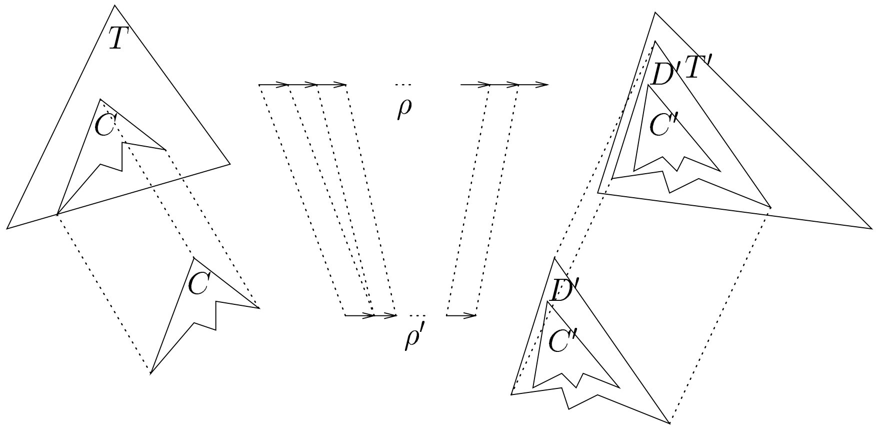


Figure 6.7: Depiction of the definition of a term slice.

6.4 Definition of debugger features

Below, we describe how a number of debugger features can be defined using the techniques of the previous section. We will primarily concentrate on the mechanisms needed for *defining* debugger features, and ignore issues related to a debugger's user-interface.

6.4.1 Single stepping/visualization

Step-wise execution of a program at the source code level is the basic feature of any debugger.

Observe that in the specification of Figure 6.2, the execution of a statement corresponds to the rewriting of a term of the following form:

$$\texttt{exec}(\textit{Stat};\textit{StatList}, \textit{Env})$$

where *Stat* represents any statement, *StatList* any list of statements, and *Env* any environment. Consequently, the fact that *some* statement is executed can be detected by matching the above *pattern* against the current redex[5].

Origin tracking can be used to determine *which* statement is currently being executed. We assume that the rewriting process is suspended whenever a redex T *matches* the above pattern. At this point, the subterm T' of T that is matched against variable *Stat* is determined. The origin of T', *OriginOf*(T'), will consist of the subtree of the program's AST that represents the currently executed statement. Thus, program execution can be visualized at the source-level by highlighting this subterm of the AST.

[5]We will use the term "redex" (short for <u>red</u>ucible <u>ex</u>pression) to denote the subterm that has been matched against some equation. For conditional rules, it is assumed that no conditions have been evaluated yet.

6.4.2 Breakpoints

Another standard feature of source-level debuggers is the *breakpoint*. The general idea is that the user selects a statement s in the program, and execution is continued until this statement is executed.

A breakpoint on a statement s can be implemented as follows. Let T_s be the subterm of the AST that corresponds to s. Then the rewriting process should be suspended when: (i) a redex T matches the pattern `exec`(*Stat*;*StatList*, *Env*) (indicating that *some* statement is being executed), and (ii) $T_s \in \mathit{OriginOf}(T')$, where T' is the subterm of T matched against variable *Stat*.

6.4.3 State inspection

At any moment that execution is suspended, either while single-stepping or due to a breakpoint, one may wish to inspect the values of variables or, more generally, arbitrary source-level expressions.

State inspection may be implemented as follows. We assume that execution was suspended at the moment that some statement was executed, i.e., a redex T matches the pattern `exec`(*Stat*;*StatList*, *Env*) Let T_z be the subterm of T that was matched against variable *Env*. Then an arbitrary source-level expression e (with an AST T_e) can be evaluated by *rewriting* the term `eval`(T_e, T_z) according to the specification of Figure 6.2. The result of this rewriting process will be a term representing the "current" value of expression e.

6.4.4 Watchpoints

Watchpoints [133] are a generalization of breakpoints. The user supplies a source-level expression e (with AST T_e), and execution continues until the value of that expression changes.

A watchpoint may be implemented as follows. First, an initial value u (with AST T_u) of expression e is computed (using the technique of Section 6.4.3) and stored by the debugger. Whenever a statement is executed, the current value v (with AST T_v) of e is determined and is compared with u by rewriting a term `inteq`(T_u, T_v). Execution (i.e., the rewriting process) is suspended when this test fails (i.e., yields the value zero).

6.4.5 Data breakpoints

A *data breakpoint* [139] is yet another variation on the breakpoint theme. A data breakpoint on a variable v (with AST T_v) is effective when that variable is referenced (or modified).

Data breakpoints can be implemented by suspending the rewriting process when a redex matches the pattern `lookup`($T_v \mapsto$ *Constant*;*Env*, T_v) (for a data breakpoint on a reference to v), or `update`($T_v \mapsto$ *Constant*;*Env*, T_v, *Constant*$'$) (for a data breakpoint on an update to v).

6.4.6 Call stack inspection

In the presence of procedures, the notion of an “environment” needs to be generalized to a stack of activation records, where each record contains the values of the local variables and parameters for a procedure call. Call-stack inspection can be defined in way that is similar to the techniques of Section 6.4.3, by visualizing the procedure calls in each record. One can easily imagine a tool that allows interactive traversal of the stack of activation records, and enables one to inspect the values of arbitrary source-level expressions in each scope.

6.5 Dynamic program slicing

Myriad variations on the notion of a *dynamic program slice* [6, 100, 89] can be found in the literature—see Chapters 3 and 5. For the purposes of this chapter, we define a *dynamic slice with respect to the current value of a variable* v to be the parts of the program that are necessary for obtaining the current value of v. To see why dynamic slicing is useful for debugging, consider a situation where an unintended value is computed for some variable v—only the statements in the dynamic slice with respect to v had an effect on the value of v. This allows one to ignore many statements in the process of localizing a bug[6].

Below we present a two-phase approach for computing dynamic slices. Section 6.5.1 discusses the nature of the “raw” information provided by the dynamic dependence relation we described in Section 6.3.2. In Section 6.5.2, we present an heuristic approach for post-processing this information, in order to obtain dynamic slices similar to those of [6, 100].

6.5.1 Pure term slices

We assume that execution was suspended at a moment that some statement was executed, i.e., a redex T matches the pattern `exec(`*Stat*`;`*StatList*`,`*Env*`)`. Let T_z be the subterm of T that was matched against *Env*, and let T_p be the subterm of T_z that constitutes the variable-value pair for variable x. Then, the dynamic dependence relation of Section 6.3.2 will associate with T_p a minimal set of function symbols, *SliceOf*(T_p), in the program's AST.

Figure 6.8 **(a)** shows a (textual representation of) the term slice that is determined for the final value of variable `p` as obtained by executing the example program of Figure 6.3. Observe that the two holes in this term slice can be replaced by *any* **L**-expression without affecting the computation of the value `120` for variable `p`.

One may wonder why the assignments to variable `s` are not completely omitted in the term slice of Figure 6.8 **(a)**. This is best understood by keeping in mind that *any* hole in a term slice may be replaced by *any* syntactically valid **L**-term. Note that the assignments to `s` cannot be replaced by any other assignment; e.g., they can certainly not be replaced by any assignment to `p`. Thus, informally stated, the left-hand sides of the assignments to `s` are in the slice because they cannot be replaced by assignments to `p`.

[6]Even in cases where a statement is missing inadvertently, dynamic slices may provide useful information. In such a case, it is likely that more statements show up in the slice than one would expect.

```
declare
  i; s; p;
begin
  i := 5;
  s := ●;
  p := 1;
  while i do
    s := ●;
    p := (p * i);
    i := (i - 1);
  end
end
```

(a)

```
declare
  i; s; p;
begin
  i := 5;
  p := 1;
  while i do
    p := (p * i);
    i := (i - 1);
  end;
end
```

(b)

Figure 6.8: **(a)** Term slice with respect to the final value of `p`. **(b)** Post-processed slice with respect to the final value of `p`.

[P1] *Var* := ● = ●

[P2] ● ; *StatSeq* = *StatSeq*

Figure 6.9: Specification for post-processing of term slices.

6.5.2 Post-processing of term slices

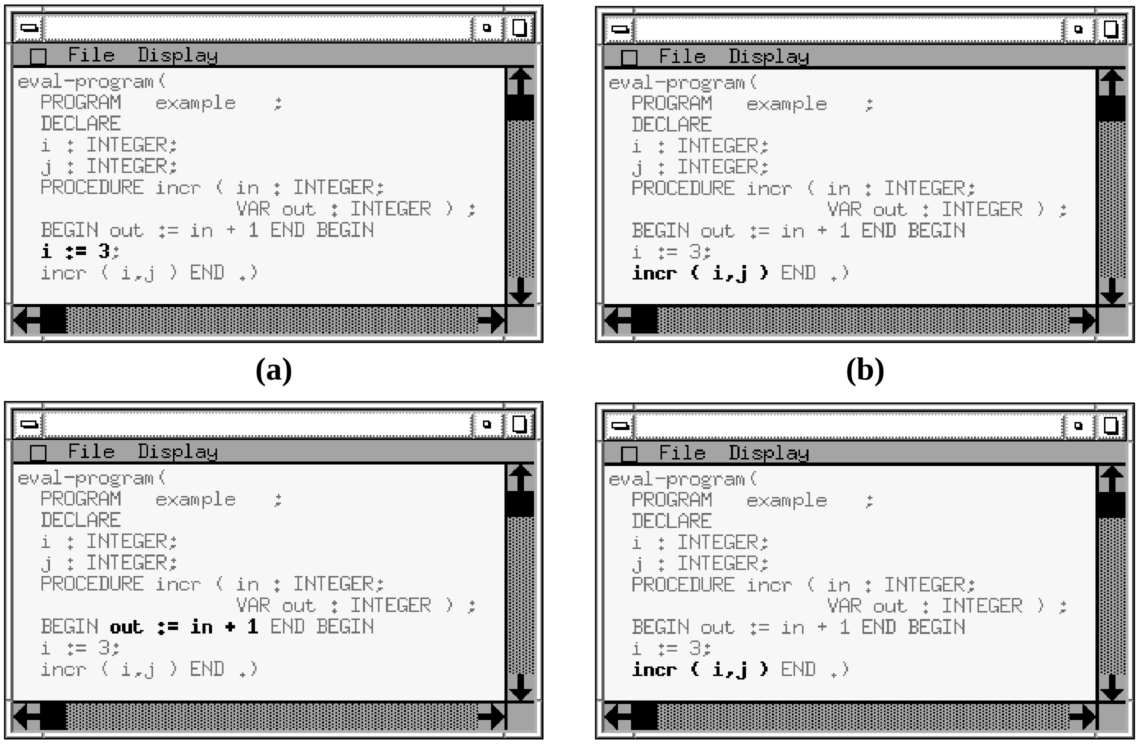
While term slices provide information that is semantically sound, they may contain a certain amount of "clutter", in the form of uninteresting information. An example of such information are the two partial assignments to variable `s` in the term slice of Figure 6.8 **(a)**.

In order obtain dynamic slices similar to those in [6, 100], one may *post-process* term slices by: (i) transforming any statement whose right-hand side is irrelevant into an irrelevant statement (rule **[P1]**), and (ii) removing irrelevant statements from statement lists (rule **[P2]**). A specification of this post-processing is shown in Figure 6.9. Rewriting the term slice of Figure 6.8 **(a)** according to this specification yields the slice of Figure 6.8 **(b)**.

The specification of Figure 6.9 is minimal—it only removes irrelevant assignments. In practice, one would like more sophisticated post-processing that, for example, removes all irrelevant declarations from the program. Post-processing becomes nontrivial in the presence of procedures, where situations may occur in which different parameters are omitted at different call sites.

6.6 Practical experience

To a large extent, the ideas in this chapter have been implemented using the ASF+SDF Meta-environment [93], a programming environment generator. In particular, origin tracking,



```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(a)

```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(b)

```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(c)

```
eval-program(
  PROGRAM   example   ;
  DECLARE
  i : INTEGER;
  j : INTEGER;
  PROCEDURE incr ( in : INTEGER;
                   VAR out : INTEGER ) ;
  BEGIN out := in + 1 END BEGIN
  i := 3;
  incr ( i,j ) END .)
```

(d)

Figure 6.10: Generated language-specific single-stepping tool.

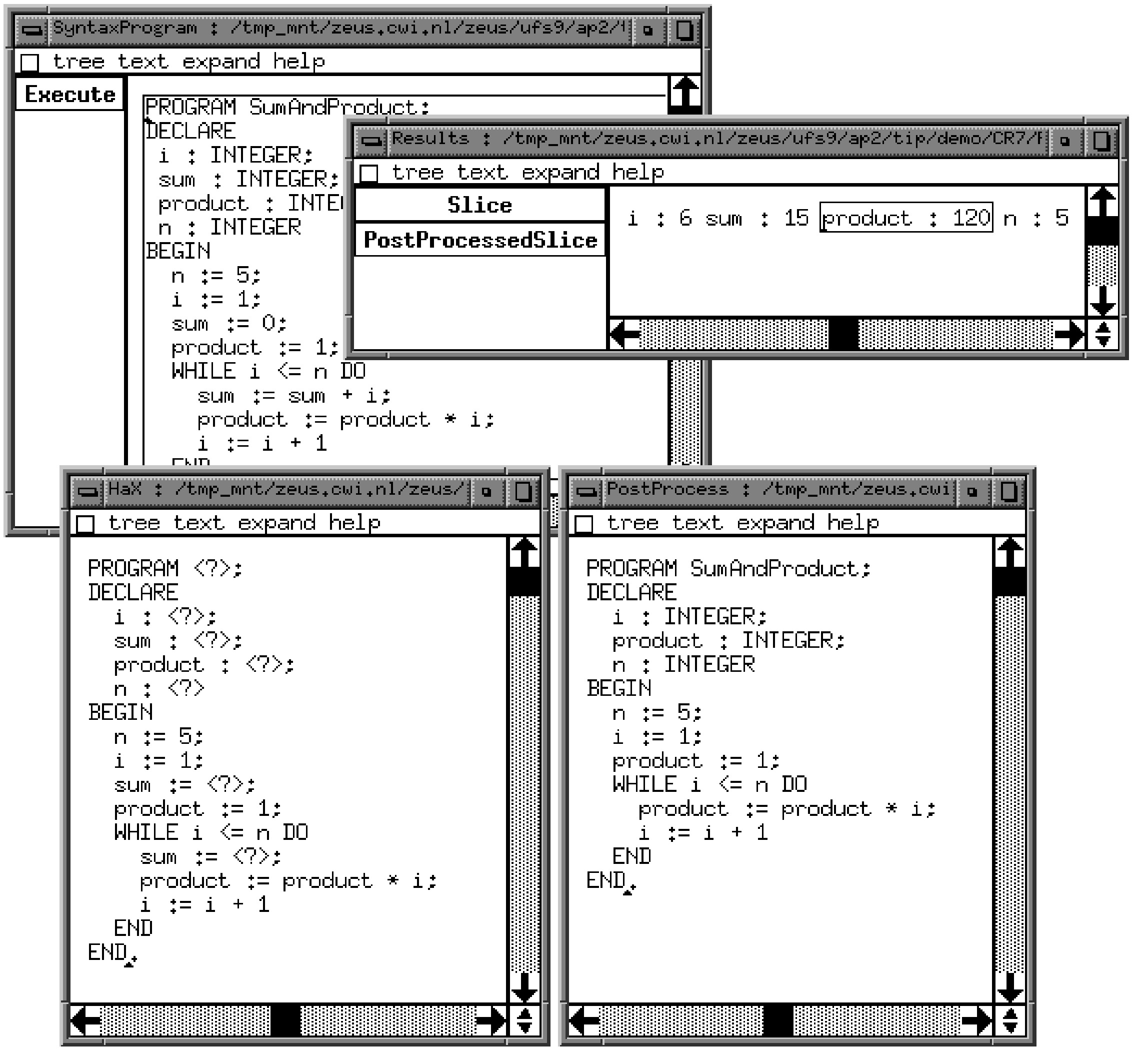
dynamic dependence tracking, and the matching of language-specific patterns have been implemented successfully.

Figure 6.10 shows a snapshots of a language-specific single-stepping tool for the language CLaX [49, 134], a substantial subset of Pascal that features procedures with nested scopes, unstructured control flow, and multi-dimensional arrays. This tool has been implemented according to the techniques of Section 6.4.1.

Figure 6.11 shows a screen dump of a dynamic slicing tool for the language CLaX, that was created using the technique of Section 6.5. In this figure, the dynamic slice with respect to the final value of variable ‘product’ is shown, both in pure “term slice” form (here, ‘<?>’ indicates a missing subterm), and in post-processed form.

6.7 Related work

The work that is most closely related to ours was done in the context of the PSG system [11]. A generator for language-specific debuggers was described in [10]. Language-specific compilers are generated by compiling denotational semantics definitions to a functional


SyntaxProgram : /tmp_mnt/zeus.cwi.nl/zeus/ufs9/ap2/
tree text expand help
Execute
Results : /tmp_mnt/zeus.cwi.nl/zeus/ufs9/ap2/tip/demo/CR7/
Slice
PostProcessedSlice
HaX : /tmp_mnt/zeus.cwi.nl/zeus/
PostProcess : /tmp_mnt/zeus.cwi

```
PROGRAM SumAndProduct;
DECLARE
 i : INTEGER;
 sum : INTEGER;
 product : INTE
 n : INTEGER
BEGIN
  n := 5;
  i := 1;
  sum := 0;
  product := 1;
  WHILE i <= n DO
    sum := sum + i;
    product := product * i;
    i := i + 1
```

```
i : 6 sum : 15 product : 120 n : 5
```

```
PROGRAM <?>;
DECLARE
  i : <?>;
  sum : <?>;
  product : <?>;
  n : <?>
BEGIN
  n := 5;
  i := 1;
  sum := <?>;
  product := 1;
  WHILE i <= n DO
    sum := <?>;
    product := product * i;
    i := i + 1
  END
END.
```

```
PROGRAM SumAndProduct;
DECLARE
  i : INTEGER;
  product : INTEGER;
  n : INTEGER
BEGIN
  n := 5;
  i := 1;
  product := 1;
  WHILE i <= n DO
    product := product * i;
    i := i + 1
  END
END.
```

Figure 6.11: Generated language-specific dynamic slicing tool.

language. A standard, language-independent interpreter is used to execute the generated functional language fragments. The behavior of a debugger is specified using a set of built-in debugging concepts. In particular, trace functions are provided for the visualization of execution. Other notions enable one to inspect the state of the interpreter, and to define breakpoints.

Bahlke et al. write that “correspondences between the abstract syntax tree and the terms of the functional language are established in both directions”. These correspondences are used to determine a language-specific notion of a step. However, the nature of these “correspondences” is not described, making it impossible to conclude how powerful these correspondences are, or what constraints on specifications they imply[7]. By contrast, our method for keeping track of correspondences, origin tracking (see Chapter 2, is well-defined, and has proven to be sufficiently powerful for realistic languages [134].

A second difference between the work by Bahlke et al. is the information that is used to define debugger features. In our approach, debugger features are defined in terms of specification-level patterns in conjunction with language-independent origin information. That is, the specification of the interpreter and the specification of debugger features are uniform. It is unclear to what extent the debugging concepts of [10] are similar to the interpreter’s specification.

Finally, Bahlke et al. do not consider more advanced debugger features such as watch-points, data breakpoints, and dynamic slices.

Bertot [26] contributes a technique called *subject tracking* to the specification language Typol [85, 41], for animation and debugging purposes. A key property of Typol specifications is that the meaning of a language construct is expressed in terms of its sub-constructs. A special variable, *Subject*, serves to indicate the language construct currently processed. This variable may be manipulated by the specification writer when different animation or debugging behavior is required.

Bertot does not consider other debugger features besides single-stepping, animation, and simple breakpoints.

Berry [25] presents an approach where animators are generated from structured operational semantics definitions. These specifications are augmented with *semantic display rules* that determine how to perform animation when a particular semantic rule is being processed. Various views of the execution of a program can be obtained by defining the appropriate display rules. Static views consist of parts of the abstract syntax tree of a program, and dynamic views are constructed from the program state during execution. As an example of a dynamic view, the evaluation of a control predicate may be visualized as the actual truth value it obtains during execution.

Although Berry considers highly sophisticated animation features, he does not consider debugger features such as breakpoints and dynamic program slices.

[7]The subset of Pascal that is considered in [10] does not contain **goto** statements. It is unclear what complications these statements would cause.

6.8 Conclusions and future work

We have presented a generic approach for deriving debugging and dynamic program slicing tools from algebraic specifications. The main conclusion of this chapter is that the information needed for implementing such tools is to a very large extent *language-independent* and *implicitly* present in the language's specification. The three "building blocks" we used to define debugger features are:

1. matching of patterns,
2. rewriting of terms, and
3. computation of origin/dependence information.

The first two items consist of functionality that is, at least in principle, already provided by any rewriting engine. As was described in Section 6.3, the information used in the third item can be computed automatically, as a side-effect of rewriting.

The only additional *language-dependent* information that is required to define debugging and slicing features consists of the specification of a set of language-specific patterns, and the actions that should be performed when a match with such a pattern occurs.

The emphasis of this chapter has been on generic techniques for constructing debugging tools; we have ignored all aspects that have to do with user-interfacing. In the future, we plan to develop a formalism in which one can *specify* such tools together with their user-interfaces.

Chapter 7

Conclusions

7.1 Summary

We have presented an approach to *generating* program analysis tools from formal specifications. In particular, the generation of tools for source-level debugging and for computing various types of program slices has been addressed in detail. We have discussed how such tools can be of practical use for software maintenance, reverse engineering, and various other applications.

Instead of "directly" implementing program analysis tools, we require that a language's semantics be formally specified by way of an executable algebraic specification. This permits us to view the execution of a program as a term rewriting process, or, more abstractly, as a sequence of terms. Two generic, specification-level techniques serve as the cornerstones of our approach for constructing program analysis tools: *origin tracking* and *dynamic dependence tracking*. Origin tracking establishes relations between recurrences of certain terms that occur in a rewriting process, and is used to formalize the notion of a "current locus" of program execution. Dynamic dependence tracking establishes certain dependence relations between terms that occur in a rewriting process, and is used to compute various notions of program slices. A crucial property of these techniques is the fact that:

> *All information needed for computing origin relations and dynamic dependence relations is implicitly present in the specification of the language's semantics.*

Moreover, these relations can be computed automatically and efficiently, as a side-effect of term rewriting.

Once origin and dynamic dependence relations have been computed, only a very small amount of additional language-specific information is needed to construct program analysis tools. This information mainly consists of a small set of language-specific patterns and associated actions. We have argued that the specification of these patterns and actions is very similar to the specification of the language's semantics itself, and therefore easy to write.

Some prototypes of program analysis tools have been generated according to our techniques using the ASF+SDF Meta-environment, a programming environment generator [93].

7.2 Main results

The main results of this thesis can be summarized as follows:

- We have presented a generic approach to generating program analysis tools that requires very little language-specific information other than a specification of a language's semantics, and the language-independent origin and dynamic dependence relations implicitly defined by that specification.

  Moreover, the additional information needed for constructing program analysis tools can be regarded as an *extension* to the language's specification (which itself may have been developed previously).
- The method for program slicing presented in Chapter 5 subsumes the traditional notions of static and dynamic program slicing, by allowing arbitrary sets of constraints on the inputs of a program. The precision of the (static) slices computed according to our approach compares favorably to that of previous slicing methods.
- We have conducted several experiments with automatically generated program analysis tools. Although current performance is rather poor (especially for programs over 50 lines), this is mainly due to a combination of two factors:
  1. The use of an interpreted term rewriting engine, and
  2. The fact that origin tracking and dynamic dependence tracking slow down execution (i.e., the term rewriting process) by a factor proportional to the size of the initial term.

  Current work on compiling specifications to efficient C code [92] is expected to overcome the first problem.

  Another effort towards greater efficiency is an efficient ML implementation of the slicing approach of Chapter 5 that is currently being developed at the IBM T.J. Watson Research Center.

The main contribution of this dissertation has been to explore semantic notions behind program slicing and debugging operations. One might justifiably remark that interpreters based on term rewriting are bound to be much slower than interpreters that are constructed by other means. However, given the recent advances in (rewriting) technology discussed above and the fact that efficiency is not the primary raison d'être of interpreters to begin with, we are mildly optimistic about the practicality of the work presented in this thesis.

7.3 Future work

We foresee a number of directions for extending and applying the work in this thesis:

- For certain applications, (e.g., determining positions of type-errors for type-checkers), different or stronger origin relations are desired [49, 48, 45, 43]. This is an area of research where much work remains to be done.
- Although defined in different ways, the origin relation and the dynamic dependence relation clearly have many similarities. A more systematic comparison of these

relations is required. We believe that a potential outcome of such a comparison could consist of a generic framework for specifying origin-like relations. One could imagine that, in such a framework, the origin relation is *parameterized* with a function defining the type of relations to be established, and that its properties are expressed in terms of this parameter.

- An issue related to the previous one is the fact that the origin relation and the dynamic dependence relation both rely on the combination of a notion of residuation [78] with "other information". In the case of origin tracking, the latter consists of the relations between redex and contractum, and between common subterms. For dependence tracking, it consists of a notion of "creation". We conjecture that it should be possible to redefine origin tracking and dependence tracking in a more consistent way, as relations that have a "residuation" subrelation in common.
- The dynamic dependence relation defined in Chapter 4 may produce slightly non-optimal results in the presence of left-nonlinear rewriting rules. Although this non-optimality does not seem to pose any problems in practice, we are currently investigating whether an approach based on graph rewriting (see, e.g., [17]) would yield more accurate results.
- The dynamic dependence relation is in certain respects similar to the information that is used by Field [54, 56] for the purpose of incremental rewriting. It would be interesting to determine exactly how these different concepts are related.
- We intend to experiment with the application of dynamic dependence tracking to specifications of the static semantics of a language (i.e., specifications of type-checkers). In principle, this would yield a "reduced" program that contains the same type-errors, but excludes all type-correct constructs.
- As was discussed earlier, the information needed for generating language-specific program analysis tools is to a very large extent language-independent. However, a small part of this information (e.g., the patterns and associated actions in Chapter 6) is *language-dependent*.

  A logical continuation of the work in this thesis would be the development of a formalism/language to express this language-specific information. Ideally, this formalism would also allow the description of the user-interface of the generated tools. The work by Koorn [96, 97] may be a good starting-point.
- In our current implementation, origin tracking and dynamic dependence tracking are implemented by way of a modification of the ASF+SDF system's rewrite engine. In order to gain performance, current work in this area includes a compiler of ASF+SDF specifications to efficient stand-alone C programs [92]. Rather than re-implementing origin tracking and dynamic dependence tracking in this new setting, it would be interesting to compute these relations by way of a systematic transformation on algebraic specifications.

  Some experiments in this area have been performed, but were unsuccessful due to the inefficient implementation of low-level set operations by way of "pure" rewriting. We conclude that an escape-mechanism (such as the hybrid specifications of [140]) is needed to compute these operations more efficiently.

- In this dissertation, we have mainly explored program slicing for purposes of program analysis and program understanding. It would be interesting to investigate whether slicing can be of use for program *restructuring* as well. The work by Griswold and Notkin [64, 63], who pursue a PDG-based method for program restructuring, may be relevant in this respect.

Appendix A

PIM details

In this appendix, we briefly review the PIM term structure and the most important subsets of PIM rules. Additional rules used primarily for performing induction are described in [55]; these rules are the foundation for the loop analysis rules presented in Section 5.5. In general, PIM is augmented with rules defining the semantics of language-specific data types such as integers.

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

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

The rules of PIM$_t^{\rightarrow}$ are given in Figure A.1. Variables v, m, s, and f will be used in the rules to refer to observable values, merge structures, store structures, and lambda-expressions, respectively. Equations (L1)–(L8) are generic to merge or store structures. Thus, each of the operators labeled ρ is to be interpreted as one of s or m. (E1) and (E2) are schemes for an infinite set of equations. Equation (C1) only applies if the argument of '$S(\cdot)$' is of sort $\mathcal{S}$. The rules of PIM$^{\rightarrow}$ consist of those of PIM$_t^{\rightarrow}$, along with the rules depicted in Figure 5.11.

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

The rules of PIM$_t^{=}$ are those of PIM$_t^{\rightarrow}$ along with those given in Figure A.2. As before, ρ in rules (L9)–(L11) is assumed to be one of m or s. In rule (M9), $C_{\mathcal{V}}[\ ]$ denotes an arbitrary strict *context* of value sort; this rule could also be less perspicuously rendered as a family of rules, one for each value-sorted function symbol.

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

Figure A.1: Equations of $\text{PIM}_t^{\rightarrow}$.

$$=\langle v\ ,\ v\rangle = \mathbf{T} \tag{E3}$$

$$l_2 \circ_\rho l_1 \circ_\rho l_2 = l_1 \circ_\rho l_2 \tag{L9}$$

$$\frac{\wedge\langle p_1\ ,\ p_2\rangle = \mathbf{F}}{(p_1\ \triangleright_\rho l_1) \circ_\rho (p_2\ \triangleright_\rho l_2) = (p_2\ \triangleright_\rho l_2) \circ_\rho (p_1\ \triangleright_\rho l_1)} \tag{L10}$$

$$(p_1\ \triangleright_\rho l) \circ_\rho (p_2\ \triangleright_\rho l) = (\vee\langle p_1\ ,\ p_2\rangle)\ \triangleright_\rho l \tag{L11}$$

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

$$\frac{=\langle v_1\ ,\ v_2\rangle = \mathbf{F}}{\{v_1 \mapsto m_1\} \circ_s \{v_2 \mapsto m_2\} = \{v_2 \mapsto m_2\} \circ_s \{v_1 \mapsto m_1\}} \tag{S7}$$

$$\begin{aligned}&(\neg\langle\ p\ \rangle\ \triangleright_m m_1 \circ_m \\ &m_2 \circ_m [p\ \triangleright v] = m_1 \circ_m m_2 \circ_m [p\ \triangleright v]\end{aligned} \tag{M6}$$

$$[p\ \triangleright m!] = [p\ \triangleright ?] \circ_m (p\ \triangleright_m m) \tag{M7}$$

$$([\mathbf{T}\ \triangleright ?] \circ_m m)! = m! \tag{M8}$$

$$C_{\mathcal{V}}[\ m!\ ] = (m \setminus (\lambda x_{\mathcal{V}}.C_{\mathcal{V}}[\ x_{\mathcal{V}}\ ]))\ !,\quad x_{\mathcal{V}} \notin FV(C_{\mathcal{V}}[\ ]) \tag{M9}$$

$$(m_1 \circ_m m_2) \setminus f = (m_1 \setminus f) \circ_m (m_2 \setminus f) \tag{M10}$$

$$[p\ \triangleright v] \setminus f = [p\ \triangleright (f\ \ v)] \tag{M11}$$

$$\emptyset_m \setminus f = \emptyset_m \tag{M12}$$

$$(p\ \triangleright_m m) \setminus f = p\ \triangleright_m (m \setminus f) \tag{M13}$$

$$(m \setminus \lambda x.v) \setminus f = m \setminus \lambda x.fv \tag{M14}$$

$$\wedge\langle p_1\ ,\ p_2\rangle = \wedge\langle p_2\ ,\ p_1\rangle \tag{B16}$$

$$\wedge\langle p\ ,\ p\rangle = p \tag{B17}$$

$$\wedge\langle p\ ,\ \neg\langle\ p\ \rangle\rangle = \mathbf{F} \tag{B18}$$

$$\vee\langle p_1\ ,\ p_2\rangle = \vee\langle p_2\ ,\ p_1\rangle \tag{B19}$$

$$\vee\langle p\ ,\ p\rangle = p \tag{B20}$$

$$\vee\langle p\ ,\ \neg\langle\ p\ \rangle\rangle = \mathbf{T} \tag{B21}$$

$$\wedge\langle p_1\ ,\ \vee\langle p_2\ ,\ p_3\rangle\rangle = \vee\langle\wedge\langle p_1\ ,\ p_2\rangle\ ,\ \wedge\langle p_1\ ,\ p_3\rangle\rangle \tag{B22}$$

$$\vee\langle p_1\ ,\ \wedge\langle p_2\ ,\ p_3\rangle\rangle = \wedge\langle\vee\langle p_1\ ,\ p_2\rangle\ ,\ \vee\langle p_1\ ,\ p_3\rangle\rangle \tag{B23}$$

$$[p\ \triangleright (\wedge\langle p\ ,\ q\rangle)] = [p\ \triangleright q] \tag{B24}$$

Figure A.2: Additional equations of $\text{PIM}_t^=$.

Bibliography

[1] ACETO, L., BLOOM, B., AND VAANDRAGER, F. Turning SOS rules into equations. In *Proc. IEEE Symp. on Logic in Computer Science* (Santa Cruz, CA, 1992), pp. 113–124.

[2] AGRAWAL, H. *Towards automatic debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

[3] AGRAWAL, H. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (Orlando, Florida, 1994), pp. 302–312. SIGPLAN Notices 29(6).

[4] AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60–73. Also Purdue University technical report SERC-TR-93-P.

[5] AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience 23*, 6 (1993), 589–616.

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

[7] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[8] ALPERN, B., WEGMAN, M., AND ZADECK, F. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, 1988), pp. 1–11.

[9] ALT, M., ASSMANN, U., AND VAN SOMEREN, H. Cosy compiler phase embedding with the cosy compiler model. In *Compiler Construction '94* (1994), P. A. Fritzson, Ed., vol. 786 of *LNCS*, Springer-Verlag, pp. 278–293.

[10] BAHLKE, R., MORITZ, B., AND SNELTING, G. A generator of language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques* (1987), pp. 92–101. SIGPLAN Notices 22(7).

[11] BAHLKE, R., AND SNELTING, G. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems 8*, 4 (1986), 547–576.

[12] BALL, T. *The Use of Control-Flow and Control Dependence in Software Tools*. PhD thesis, University of Wisconsin-Madison, 1993.

[13] BALL, T., AND HORWITZ, S. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 206–222.

[14] BALLANCE, R. A., MACCABE, A. B., AND OTTENSTEIN, K. J. The program dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, NY, 1990), pp. 257–271.

[15] BALZER, R. EXDAMS - Extendable Debugging And Monitoring System. In *Proceedings of the AFIPS SJCC* (1969), vol. 34, pp. 567–586.

[16] BANNING, J. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages* (1979), pp. 29–41.

[17] BARENDREGT, H., VAN EEKELEN, M., GLAUERT, J., KENNAWAY, J., PLASMEIJER, M., AND SLEEP, M. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages* (Eindhoven, The Netherlands, 1987), vol. 259 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141–158.

[18] BARNES, J. *Programming in Ada*, second ed. International Computer Science Series. Addison-Wesley, 1982.

[19] BARTH, J. A practical interprocedural data flow analysis algorithm. *Communications of the ACM 21*, 9 (1978), 724–736.

[20] BATES, S., AND HORWITZ, S. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Charleston, SC, 1993), pp. 384–396.

[21] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering* (Baltimore, 1993).

[22] BERGERETTI, J.-F., AND CARRÉ, B. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 37–61.

[23] BERGSTRA, J., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[24] BERGSTRA, J., AND KLOP, J. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences 32*, 3 (1986), 323–362.

[25] BERRY, D. *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edinburgh, 1991.

[26] BERTOT, Y. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 327–337. *SIGPLAN Notices* 26(6).

[27] BERTOT, Y. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle*. PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[28] BERTOT, Y. Origin functions in lambda-calculus and term rewriting systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP '92)* (1992), J.-C. Raoult, Ed., vol. 581 of *LNCS*, Springer-Verlag.

[29] BINKLEY, D. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, Florida, November 1992).

[30] BINKLEY, D. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems 2*, 1–4 (1993), 31–45.

[31] BINKLEY, D. Slicing in the presence of parameter aliasing. In *Proceedings of the Third Software Engineering Research Forum* (Orlando, Florida, November 1993), pp. 261–268.

[32] BINKLEY, D. Interprocedural constant propagation using dependence graphs and a data-flow model. In *Proceedings of the 5th International Conference on Compiler Construction—CC'94* (Edinburgh, UK, 1994), P. Fritzson, Ed., vol. 786 of *LNCS*, pp. 374–388.

[33] BINKLEY, D., HORWITZ, S., AND REPS, T. Program integration for languages with procedure calls, 1994. Submitted for publication.

[34] CARTWRIGHT, R., AND FELLEISEN, M. The semantics of program dependence. In *Proceedings of the ACM 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, 1989), pp. 13–27.

[35] CHENG, J. Slicing concurrent programs – a graph-theoretical approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 223–240.

[36] CHOI, J.-D., BURKE, M., AND CARINI, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (1993), ACM, pp. 232–245.

[37] CHOI, J.-D., AND FERRANTE, J. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1097–1113.

[38] CHOI, J.-D., MILLER, B., AND NETZER, R. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 491–530.

[39] COOPER, K., AND KENNEDY, K. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, 1988), pp. 57–66. *SIGPLAN Notices* 23(7).

[40] CYTRON, R., FERRANTE, J., ROSEN, B. K., AND WEGMAN, M. N. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 451–490.

[41] DESPEYROUX, T. Typol: a formalism to implement natural semantics. Tech. Rep. 94, INRIA, 1988.

[42] DEURSEN, A. V. An algebraic specification for the static semantics of Pascal. Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991.

[43] DEURSEN, A. V. *Executable Language Definitions—Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.

[44] DEURSEN, A. V. Origin tracking in primitive recursive schemes. Report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), 1994.

[45] DEURSEN, A. V., AND DINESH, T. Origin tracking for higher-order term rewriting systems. In *Proceedings of the International Workshop on Higher-Order Algebra, Logic and Term Rewriting* (Amsterdam, September 1993), J. Heering, K. Meinke, B. Möller, and T. Nipkow, Eds., vol. 816 of *LNCS*.

[46] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. Report CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[47] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.

[48] DINESH, T. Type checking revisited: Modular error handling. In *International Workshop on Semantics of Specification Languages* (1993).

[49] DINESH, T., AND TIP, F. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[50] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the fifth workshop on Languages and Compilers for Parallel Computing* (New Haven, Connecticut, 1992), pp. 329–337.

[51] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92* (California, 1992), pp. 131–145.

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

[53] FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (1987), 319–349.

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

[55] FIELD, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

[56] FIELD, J. A graph reduction approach to incremental rewriting. In *Proceedings of the 5th International Conference on Rewriting Techniques and Applications* (1993), C. Kirchner, Ed., vol. 690 of *LNCS*, pp. 259–273.

[57] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995). To appear.

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

[59] FRITZSON, P., SHAHMEHRI, N., KAMKAR, M., AND GYIMOTHY, T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303–322.

[60] GALLAGHER, K. *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, 1989.

[61] GALLAGHER, K., AND LYLE, J. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (1991), 751–761.

[62] GOPAL, R. Dynamic program slicing based on dependence relations. In *Proceedings of the Conference on Software Maintenance* (1991), pp. 191–200.

[63] GRISWOLD, W. G. Direct update of dataflow representations for a meaning-preserving program restructuring tool. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering* (December 1993), pp. 42–55.

[64] GRISWOLD, W. G., AND NOTKIN, D. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology 2*, 3 (July 1993), 228–269.

[65] GUPTA, R., HARROLD, M., AND SOFFA, M. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance* (1992), pp. 299–308.

[66] GUPTA, R., AND SOFFA, M. A framework for generalized slicing. Technical report TR-92-07, University of Pittsburgh, 1992.

[67] HAUSLER, P. Denotational program slicing. In *Proceedings of the 22nd Hawaii International Conference on System Sciences* (Hawaii, 1989), pp. 486–494.

[68] HEERING, J., HENDRIKS, P., KLINT, P., AND REKERS, J. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices 24*, 11 (1989), 43–75.

[69] HENDRIKS, P. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[70] HORWITZ, S. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (White Plains, New York, 1990), pp. 234–245. *SIGPLAN Notices* 25(6).

[71] HORWITZ, S., PFEIFFER, P., AND REPS, T. Dependence analysis for pointer variables. In *Proceedings of the ACM 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, 1989). *SIGPLAN Notices* 24(7).

[72] HORWITZ, S., PRINS, J., AND REPS, T. Integrating non-interfering versions of programs. In *Conference Record of the ACM SIGSOFT/SIGPLAN Symposium on Principles of Programming Languages* (1988), pp. 133–145.

[73] HORWITZ, S., PRINS, J., AND REPS, T. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), ACM, pp. 146–157.

[74] HORWITZ, S., PRINS, J., AND REPS, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (1989), 345–387.

[75] HORWITZ, S., AND REPS, T. Efficient comparison of program slices. *Acta Informatica 28* (1991), 713–732.

[76] HORWITZ, S., AND REPS, T. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, 1992), pp. 392–411.

[77] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[78] HUET, G., AND LÉVY, J.-J. Computations in orthogonal rewriting systems part I and II. In *Computational Logic; essays in honour of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, 1991, pp. 395–443.

[79] HWANG, J., DU, M., AND CHOU, C. Finding program slices for recursive procedures. In *Proceedings of the 12th Annual International Computer Software and Applications Conference* (Chicago, 1988).

[80] HWANG, J., DU, M., AND CHOU, C. The influence of language semantics on program slices. In *Proceedings of the 1988 International Conference on Computer Languages* (Miami Beach, 1988).

[81] JACKSON, D., AND ROLLINS, E. J. Abstraction mechanisms for pictorial slicing. In *Proceedings of the IEEE Workshop on Program Comprehension* (Washington, November 1994).

[82] JACKSON, D., AND ROLLINS, E. J. A new model of program dependences for reverse engineering. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering* (New Orleans, LA, December 1994).

[83] JIANG, J., ZHOU, X., AND ROBSON, D. Program slicing for C - the problems in implementation. In *Proceedings of the Conference on Software Maintenance* (1991), pp. 182–190.

[84] JOHNSON, R., PEARSON, D., AND PINGALI, K. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (Orlando, Florida, 1994), pp. 171–185. SIGPLAN Notices 29(6).

[85] KAHN, G. Natural semantics. In *Fourth Annual Symposium on Theoretical Aspects of Computer Science* (1987), F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., vol. 247 of *LNCS*, Springer-Verlag, pp. 22–39.

[86] KAMKAR, M. An overview and comparative classification of static and dynamic program slicing. Technical Report LiTH-IDA-R-91-19, Linköping University, Linköping, 1991. To appear in *Journal of Systems and Software*.

[87] KAMKAR, M. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linköping University, 1993.

[88] KAMKAR, M., FRITZSON, P., AND SHAHMEHRI, N. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance* (Montreal, Canada, 1993), pp. 386–395.

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

[90] KAMKAR, M., SHAHMEHRI, N., AND FRITZSON, P. Interprocedural dynamic slicing and its application to generalized algorithmic debugging. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '92* (1992).

[91] KAMPERMAN, J., DINESH, T., AND WALTERS, H. An extensible language for the generation of parallel data manipulation and control packages. In *Proceedings of the Poster Session of Compiler Construction '94* (1994), P. Fritzson, Ed. Appeared as technical report LiTH-IDA-R-94-11, Linköping University.

[92] KAMPERMAN, J., AND WALTERS, H. ARM – Abstract Rewriting Machine. In *Computing Science in the Netherlands* (1993), H. Wijshoff, Ed. Also appeared as CWI report CS-R9330.

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

[94] KLOP, J. Term rewriting systems. Report CS-R9073, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

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

[96] KOORN, J. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992.

[97] KOORN, J. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, University of Amsterdam, 1994.

[98] KOREL, B., AND FERGUSON, R. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science 2*, 2 (1992), 199–215.

[99] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988), 155–163.

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

[101] KUCK, D., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages* (1981), pp. 207–218.

[102] LAKHOTIA, A. Graph theoretic foundations of program slicing and integration. Report CACS TR-91-5-5, University of Southwestern Louisiana, 1991.

[103] LAKHOTIA, A. Improved interprocedural slicing algorithm. Report CACS TR-92-5-8, University of Southwestern Louisiana, 1992.

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

[105] LANDI, W., AND RYDER, B. G. Pointer-induced aliasing: A problem classification. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages* (January 1991), pp. 93–103.

[106] LARUS, J., AND CHANDRA, S. Using tracing and dynamic slicing to tune compilers. Computer sciences technical report #1174, University of Wisconsin-Madison, 1993.

[107] LEUNG, H., AND REGHBATI, H. Comments on program slicing. *IEEE Transactions on Software Engineering SE-13*, 12 (1987), 1370–1371.

[108] LYLE, J. *Evaluating Variations on Program Slicing for Debugging*. PhD thesis, University of Maryland, 1984.

[109] LYLE, J., AND WEISER, M. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications* (Beijing (Peking), China, 1987), pp. 877–883.

[110] LYLE, J. R., AND BINKLEY, D. Program slicing in the presence of pointers. In *Proceedings of the Third Software Engineering Research Forum* (Orlando, Florida, November 1993), pp. 255–260.

[111] MARANGET, L. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages* (1991), pp. 225–269.

[112] MARANGET, L. *La Stratégie paresseuse*. PhD thesis, l'Université Paris VII, Paris, 1992. In French.

[113] MAYDAN, D., HENNESSY, J., AND LAM, M. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 1–14. *SIGPLAN Notices* 26(6).

[114] MEULEN, E. V. D. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

[115] MEULEN, E. V. D. Deriving incremental implementations from algebraic specifications. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology* (1992), Workshops in Computing, Springer-Verlag, pp. 277–286.

[116] MILLER, B., AND CHOI, J.-D. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, 1988), pp. 135–144. *SIGPLAN Notices* 23(7).

[117] NING, J., ENGBERTS, A., AND KOZACZYNSKI, W. Automated support for legacy code understanding. *Communications of the ACM 37*, 3 (1994), 50–57.

[118] O'DONNELL, M. *Computing in Systems Described by Equations*, vol. 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[119] OTT, L. M., AND THUSS, J. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering* (1989), pp. 198–204.

[120] OTTENSTEIN, K., AND OTTENSTEIN, L. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (1984), pp. 177–184. *SIGPLAN Notices* 19(5).

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

[122] PAN, H., AND SPAFFORD, E. H. Fault localization methods for software debugging. *Journal of Computer and Software Engineering* (1994). To appear.

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

[124] REPS, T. Algebraic properties of program integration. *Science of Computer Programming 17* (1991), 139–215.

[125] REPS, T. On the sequential nature of interprocedural program-analysis problems. Unpublished report, University of Copenhagen, 1994.

[126] REPS, T. Private communication, 1994.

[127] REPS, T., AND BRICKER, T. Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton, 1989), pp. 46–55. *ACM SIGSOFT Software Engineering Notes* Vol.17 No.7.

[128] REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering* (New Orleans, LA, December 1994). To appear.

[129] REPS, T., SAGIV, M., AND HORWITZ, S. Interprocedural dataflow analysis via graph reachability. Report DIKU TR 94-14, University of Copenhagen, Copenhagen, 1994.

[130] REPS, T., AND YANG, W. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages* (1989), vol. 352 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 60–74.

[131] SHAHMEHRI, N. *Generalized Algorithmic Debugging*. PhD thesis, Linköping University, 1991.

[132] SHAPIRO, E. *Algorithmic Program Debugging*. MIT Press, 1982.

[133] STALLMAN, R., AND PESCH, R. *Using GDB, A guide to the GNU Source-Level Debugger*. Free Software Foundation/Cygnus Support, 1991. Version 4.0.

[134] TIP, F. Animators for generated programming environments. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *LNCS*, Springer-Verlag, pp. 241–254.

[135] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. Report CS-R9453, Centrum voor Wiskunde en Informatica (CWI), 1994. Accepted for TAPSOFT'95.

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

[137] VENKATESH, G. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 107–119. *SIGPLAN Notices* 26(6).

[138] WADLER, P., AND HUGHES, R. Projections for strictness analysis. In *Proc. Conf. on Functional Programming and Computer Architecture* (Portland, OR, 1987), vol. 274 of *LNCS*, pp. 385–406.

[139] WAHBE, R., LUCCO, S., AND GRAHAM, S. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation* (Albuquerque, NM, 1993), pp. 1–12. *SIGPLAN Notices* 28(6).

[140] WALTERS, H. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[141] WEGMAN, M., AND ZADECK, F. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (1991), 181–210.

[142] WEIHL, W. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages* (1980), pp. 83–94.

[143] WEISE, D., CREW, R., ERNST, M., AND STEENSGAARD, B. Value dependence graphs: Representation without taxation. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Portland, OR, 1994), pp. 297–310.

[144] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[145] WEISER, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (1982), 446–452.

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

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

[148] WEISER, M. Private communication, 1994.

[149] YANG, W., HORWITZ, S., AND REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (Irvine, CA, December 1990), pp. 133–143. *ACM SIGSOFT Software Engineering Notes* Vol.15 No.6.

[150] ZIMA, H., AND CHAPMAN, B. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. ACM Press, New York, 1991.

Samenvatting

Over het onderwerp van dit proefschrift

Het is een bekend fenomeen dat een programma moeilijker te begrijpen wordt naarmate het "ouder" wordt, en er meerdere malen wijzigingen in zijn aangebracht. Redenen voor dergelijke wijzigingen zijn voor de hand liggend:

- Het kan nodig zijn de functionaliteit van een programma uit te breiden.
- Programma's moeten soms worden aangepast wegens overschakeling op een ander type computer of bedrijfssysteem.
- Verbetering van het mens/machine interface kan gewenst zijn.

Helaas wordt dit soort software-onderhoud niet altijd als een volwaardige activiteit beschouwd, en vindt het plaats onder grote tijdsdruk. Dergelijk programmeerwerk wordt dan ook veelal op een ad-hoc manier uitgevoerd, waarbij de "structuur" die oorspronkelijk in het programma aanwezig was grotendeels verloren gaat. Ook worden het ontwerp en de documentatie die bij het programma horen vaak niet op adequate wijze aangepast. Een programmeur die met een dergelijk programma wordt geconfronteerd, en de opdracht heeft daarin nogmaals een wijziging aan te brengen, kan hierdoor grote moeite hebben met het begrijpen ervan. Dit geldt in het bijzonder als de programmeurs die betrokken zijn geweest bij eerdere versies van het programma niet meer aanwezig zijn voor tekst en uitleg, en de source-tekst de enige beschikbare informatie omtrent het programma is.

Dit proefschrift gaat over technieken en hulpmiddelen om programma's automatisch te analyzeren, met andere woorden, over hulpmiddelen ("tools") die het begrijpen van programma's vergemakkelijken. Hierbij ligt de nadruk op het *genereren* van programma-analyse tools uit algebraïsche specificaties, in plaats van deze direct te implementeren in een specifieke programmeertaal. Een belangrijk voordeel van deze aanpak is dat deze grotendeels taal-onafhankelijk is. De nadruk ligt op twee specifieke technieken voor programma-analyse: *source-level debugging* en *program slicing*.

Source-level debugging

Onder het begrip “debugging” (letterlijk: ontluizing) worden technieken verstaan voor het opsporen van fouten in programma’s. Een “debugger” is een tool dat op deze technieken is gebaseerd. Als de communicatie tussen de debugger en de gebruiker plaatsvindt in termen van de oorspronkelijke tekst van het programma (de zgn. “source-code”), spreekt men van een “source-level debugger”. Functionaliteit die typisch door een debugger wordt geboden omvat:

Single stepping: het stapsgewijs uitvoeren van de instructies van het programma, of het simuleren hiervan.

State inspection: het inspecteren van de waarden die door het programma worden berekend op een door de gebruiker aangewezen punt.

Breakpoints: hierbij wordt de uitvoering van de instructies van het programma voortgezet totdat een vooraf gespecificeerde conditie geldt. Als voorbeeld van condities valt te denken aan het bereiken van een bepaalde locatie in de programma-tekst (“control breakpoints”), of het moment waarop een door de gebruiker gespecificeerde expressie een bepaalde waarde aanneemt (“data breakpoints”).

Program slicing

Programmasegmentatie (“program slicing”) is een techniek voor het opsplitsen van een programma in een aantal deelprogramma’s die ieder corresponderen met een specifieke berekening. Dit is het best te begrijpen door een eenvoudig voorbeeld te bestuderen. Figuur S.1 **(a)** toont een programma dat de gebruiker om een getal `n` vraagt, vervolgens de som en het produkt van de eerste `n` getallen berekent, en deze waarden afdrukt. Figuur S.1 **(b)** toont een *program slice* ten opzichte van de instructie `write(produkt)`, de laatste regel van het programma. Het idee is nu om alle instructies die *niet van belang zijn* voor het berekenen van de waarde van variabele `produkt` op dit punt uit het programma te *verwijderen*. Aangezien de berekeningen van de som en het produkt hier “onafhankelijk” zijn, zijn alle instructies die te maken hebben met variabele `som` niet meer aanwezig in de program slice van Figuur S.1 **(b)**. Programmasegmenten die berekend worden zonder aannames omtrent de invoer van een programma worden ook wel *statische programmasegmenten genoemd* (“static program slices”).

Bij *dynamische programmasegmentatie* (“dynamic program slicing”) wordt er gebruik gemaakt van het feit dat bepaalde afhankelijkheden in een programma niet worden geactiveerd voor specifieke invoerwaarden. Zo toont Figuur S.1 **(c)** een dynamic program slice ten opzichte van de eindwaarde van variabele `produkt`, voor de specifieke invoerwaarde `n = 0`. Aangezien de instructies in de **while**-lus nooit worden uitgevoerd voor invoerwaarde `n = 0` hoeven deze niet in het dynamische programma segment te worden opgenomen. Dergelijke dynamische programma segmenten zijn meestal veel kleiner dan de invoeronafhankelijke statische programma segmenten.

Het nut van programmasegmentatie voor het begrijpen van een programma is evident:

```
read(n);
i := 1;
som := 0;
produkt := 1;
while i <= n do
begin
  som := som + i;
  produkt := produkt * i;
  i := i + 1
end;
write(som);
write(produkt)
```

(a)

```
read(n);
i := 1;

produkt := 1;
while i <= n do
begin

  produkt := produkt * i;
  i := i + 1
end;

write(produkt)
```

(b)

```
read(n);
i := 1;

produkt := 1;
while i <= n do
begin



end;

write(produkt)
```

(c)

Figuur S.1: **(a)** Voorbeeldprogramma. **(b)** Statisch programma segment van het programma ten opzichte van de eindwaarde van variabele `produkt`. **(c)** Dynamisch programma segment van het programma ten opzichte van de eindwaarde van variabele `produkt` voor de specifieke invoerwaarde `n = 0`.

het stelt de programmeur in staat de aandacht te concentreren op een bepaalde berekening, en instructies die daar niet mee te maken hebben te negeren. Dynamische programmasegmentatie is nuttig als debugging techniek. Als immers een onjuiste waarde wordt berekend op een bepaald punt in het programma, kunnen alleen de instructies in de dynamische slice ten opzichte van dat programma-punt van belang zijn geweest voor de berekening van de onjuiste waarde[1]. In het algemeen kan worden gesteld dat program slicing nuttig is voor het begrijpen van programma's omdat het het overzicht helpt te vergroten.

Hoofdstuk 3 bevat een uitgebreid overzicht van de huidige vakliteratuur op het gebied van program slicing, en de toepassingen daarvan.

Algebraïsche specificaties

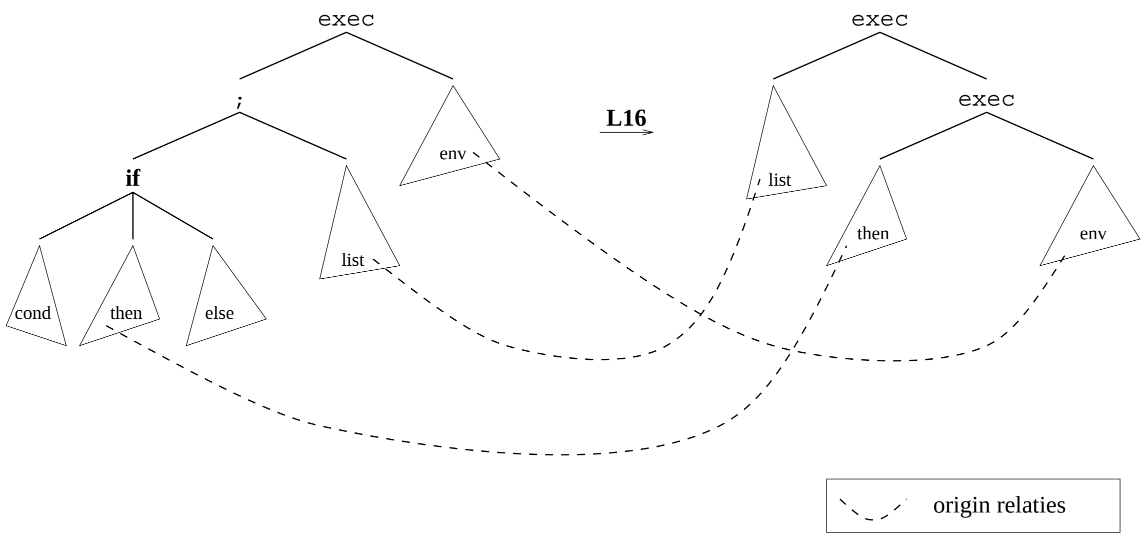
Zoals eerder genoemd bestaat de aanpak van dit proefschrift uit het *genereren* van programma-analyse tools uit algebraïsche specificaties. Een algebraïsche specificatie bestaat uit een verzameling van (conditionele) vergelijkingen die gezamenlijk de "betekenis" van instructies en programma's formeel definiëren. Als voorbeeld wordt in Figuur S.2 de executie van een **if**-instructie gespecificeerd (dit voorbeeld komt uitgebreid aan de orde in Hoofdstuk 6). Gezamenlijk definiëren deze twee vergelijkingen hoe de executie van een **if**-instructie kan worden uitgedrukt in de executie van ofwel de instructies in de **then**-tak, ofwel

[1]Zelfs in gevallen waar de fout bestaat uit de *afwezigheid* van een bepaalde instructie kunnen programma-segmenten van nut zijn. In dergelijke gevallen is het waarschijnlijk dat het segment andere instructies bevat dan men zou verwachten.

[L16] $\texttt{exec}(\textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{StatSeq}\ \textbf{else}\ \mathit{StatSeq}'\ \textbf{end};\mathit{StatSeq}'', \mathit{Env}) = \texttt{exec}(\mathit{StatSeq}'', \texttt{exec}(\mathit{StatSeq}, \mathit{Env}))$
when $\texttt{eval}(\mathit{Exp}, \mathit{Env}) \neq 0$

[L17] $\texttt{exec}(\textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{StatSeq}\ \textbf{else}\ \mathit{StatSeq}'\ \textbf{end};\mathit{StatSeq}'', \mathit{Env}) = \texttt{exec}(\mathit{StatSeq}'', \texttt{exec}(\mathit{StatSeq}', \mathit{Env}))$
when $\texttt{eval}(\mathit{Exp}, \mathit{Env}) = 0$

Figuur S.2: Algebraïsche specificatie van de executie van een **if**-instructie.



Figuur S.3: Illustratie van termherschrijving en de origin relatie.

de instructies in de **else**-tak, al naar gelang het resultaat van de evaluatie van de controle-expressie van de **if**. De hulpfunctie `exec` die in beide vergelijkingen voorkomt beschrijft het effect van de executie van een lijst van instructies op de “status” van een programma.

Algebraïsche specificaties kunnen worden uitgevoerd (“geëxecuteerd”) door middel van *termherschrijving*: het van links naar rechts toepassen van vergelijkingen[2]. Figuur S.3 laat zien hoe een **if**-term kan worden herschreven door vergelijking **[L16]** uit Figuur S.2 toe te passen.

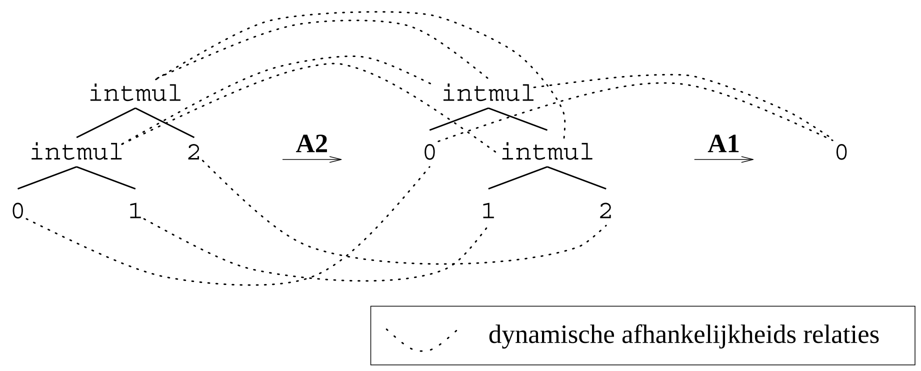
In Figuur S.3 zijn met stippellijnen een aantal *origin* relaties aangegeven. Intuitief gezien relateert de origin relatie “gelijke” termen aan elkaar. Hoofdstuk 2 bevat een formele definitie van *origin tracking*, een techniek om dergelijke origin relaties *automatisch* uit een specificatie af te leiden. In Hoofdstuk 6 wordt origin tracking gebruikt om het huidige punt van de programma-executie te kunnen traceren, en source-level debugging tools te genereren.

Een andere relatie die in dit proefschrift veelvuldig aan de orde komt is de *dynamische afhankelijkheidsrelatie* (“dynamic dependence relation”). Figuur S.4 toont twee eenvoudige vergelijkingen voor berekeningen met natuurlijke getallen. Vergelijking **[A1]** zegt dat de vermenigvuldiging van het getal `0` met een willekeurig ander getal het resultaat `0` oplevert,

[2]Dit concept wordt iets ingewikkelder als vergelijkingen conditioneel zijn. Conditioneel termherschrijven komt uitgebreid aan de orde in Hoofdstuk 2.

[A1]	`intmul(0,`X`)`	=	`0`
[A2]	`intmul(intmul(`X`, `Y`),`Z`)`	=	`intmul(`X`,intmul(`Y`, `Z`))`

Figuur S.4: Vergelijkingen voor berekeningen met natuurlijke getallen.



Figuur S.5: Dynamische afhankelijkheidsrelaties die optreden bij toepassing van vergelijkingen **[A1]** en **[A2]**.

en vergelijking **[A2]** beschrijft het feit dat vermenigvuldiging een associatieve operatie is. Figuur S.5 laat zien hoe de term `intmul(intmul(0, 1), 2)` kan worden herschreven door toepassing van vergelijkingen **[A1]** en **[A2]**. De gestippelde lijnen in Figuur S.5 geven dynamische afhankelijkheden weer. Intuïtief gezien beschrijven deze relaties welke functiesymbolen in de term hebben geleid tot de "creatie" van een bepaald functiesymbool. Dit kan in het voorbeeld van Figuur S.5 worden ingezien door de afhankelijkheidsrelaties vanaf het eindresultaat `0` terug te traceren: hieruit blijkt dat het optreden van deze waarde *niet* afhankelijk is van de waarden `1` en `2` in de beginterm.

Hoofdstuk 4 bevat een formele definitie van "dynamic dependence tracking", een techniek om dynamische afhankelijksrelaties automatisch uit een algebraïsche specificatie af te leiden. In Hoofdstuk 5 wordt dynamic dependence tracking gecombineerd met het formalisme PIM [55], een equationele logica voor het beschrijven van de betekenis van programma's ontwikkeld aan het IBM T.J. Watson Research Center. In Hoofdstuk 5 wordt uitvoerig beschreven hoe het traceren van de dynamische afhankelijkheden vanuit een *waarde* die berekend is door een programma, zeer accurate program slices oplevert. In Hoofdstuk 6 wordt de dynamische afhankelijkheidsrelatie toegepast op een algebraïsche specificatie van een interpreter om dynamische programmasegmenten te berekenen.

Resultaten van dit onderzoek

De belangrijkste resultaten van het in dit proefschrift beschreven onderzoek kunnen als volgt worden samengevat:

- Hoofdstukken 2 en 4 bevatten formele definities van respectievelijk de origin relatie en de dynamische afhankelijksheidsrelatie. Verder worden hier een aantal nuttige eigenschappen van deze relaties bewezen, en wordt aangegeven hoe ze op efficiënte wijze geïmplementeerd kunnen worden.
- In Hoofdstuk 5 wordt de dynamische afhankelijkheidsrelatie “toegepast op” het PIM formalisme. Dit blijkt overeen te komen met een algemene, flexibele, en zeer precieze vorm van program slicing.
- Hoofdstuk 3 bevat een uitgebreid literatuuronderzoek over het onderwerp program slicing.
- In Hoofdstuk 6 worden de origin relatie en de dynamische afhankelijkheidsrelatie toegepast op algebraïsche specificaties van interpreters. Hierbij wordt beschreven hoe op basis van de origin relatie een aantal zeer krachtige debugging concepten op eenvoudige wijze kan worden gedefiniëerd, en hoe de dynamische afhankelijkheidsrelatie het mogelijk maakt dynamische programmasegmenten te berekenen.

De technieken die zijn beschreven in dit proefschrift zijn geïmplementeerd met behulp van de ASF+SDF Meta-Environment [93], een generator voor programmeeromgevingen die ontwikkeld is door onderzoeksgroepen bij het Centrum voor Wiskunde en Informatica en bij de Universiteit van Amsterdam.