Relational Approach to Program Slicing

Ivan Vankov

Abstract

Program slicing is a well known technique for program analysis with a variety of useful applications. This thesis investigates a new method for program slicing based on relational representation of program facts and algorithms. A core slicing algorithm is presented and it is shown that it can be applied to a broad range of problems. A number of generalized routines are developed to extract program facts from source code without being constrained by a particular programming language. The advantages of the proposed approach are demonstrated by applying it to slicing Pico and Java programs.

Acknowledgements

I would like to thank Prof. dr. Paul Klint for introducing me to the subject and guiding me throughout the whole project. He gave me a lot of freedom and let me enjoy my graduation work from the start to the end. Jurgen Vinju from the Interactive Software Development and Renovation Group at CWI assisted me when I was stuck with various implementation obstacles.

Also I would like to express my gratitude to the administration of the University of Amsterdam and the Faculty of Science for opening the master program in computer science to international students and supporting us in all possible ways during these two unforgettable years.

Finally I would like to thank my collaborators at Madison Touche, who tolerated me while I was engaged with my study.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents a new approach to program slicing based on relational representation of program facts. Program slicing is a method to isolate parts of a program which are responsible for certain behavior. It has various applications in software understanding, constructing, measuring, verifying and refactoring of programs. Research has shown that programmers identify slices when coding and debugging and it is a natural way of looking at program code. The motivation of this graduation project was to propose a program slicing method incorporating existing techniques, but featuring better versatility and applicability.

1.1 Program slicing

The notion of program slicing was first introduced by Mark Weiser [1]. He defined program slicing as reducing a program to a minimal form that produces a desired subset of the behavior of the original program. A *slice* is an independent executable program obtained by removing redundant statements with respect of a given slicing criterion. A *slicing criterion* is defined by a statement and a set of variables and it specifies the subset of the program which is responsible for setting the values of the variables just before executing the statement. If we identify statements by numbers and variables by name, a slicing criterion is a pair $\langle s, v \rangle$, where i is the number of statement and v is the set of variables we are interested in. Figure 1.1 illustrates slicing of a sample program[1] for a given slicing criterion.

Weiser's definition of program slicing is a kind of static program analysis. A *static slice* is constructed before the program is executed and it contains all statements that *may* affect the values of the variables included in the slicing criterion. Other authors (e.g. [2], [3]) suggested *dynamic slicing* where the slice is sensitive to a particular program input. Weiser's slicing is also considered *backward* because it traces all program statements which may be executed prior to a specified program point. In this thesis the term program slice will refer to a *static backward slice*. We will use Weiser's definition of a slicing criterion,

---

[1]All sample program in this paper are fragments of syntactically correct Java code, not necessarily executable. Line numbers are used to identify program statements.

```
#1      int n = read();              #1      int n = read();
#2      int fac = 1;                 #2      int fac = 1;
#3      int sum = 0;
#4      while(i <= n)                #4      while(i <= n)
        {                                    {
#5          fac = fac * i;           #5          fac = fac * i;
#6          sum = sum + 1;
#7          i = i + 1;               #7          i = i + 1;
        }                                    }
#8      write(fac);                  #8      write(fac);
#9      write(sum);
```

    (a) Original program   (b) Slice on the value of fac at #8

Figure 1.1: Program slicing example

but it will consist of a statement identifier and a *single* variable - slices will be constructed with respect to one variable per statement only.

Weiser proposed a simple algorithm for program slicing based on data flow equations. He considered slicing of single block programs only and did not mention how to handle more sophisticated program constructs. Ottenstein and Ottenstein pointed out that *program dependence graphs* can be efficiently used for slicing [4]; once a program is represented by its program dependence graph, the slicing problem is simply a vertex reachability problem, which is easily computable. Horwiz, Reps and Binkley extended the idea by introducing system dependence graphs capable of interprocedural procedural slicing where the slice can cross the procedure boundaries [10]. Other studies, such as [5] applied it for object oriented slicing. Slicing programs based on non-imperative programming paradigms, such as functional and logic programming, is also possible (see for instance [6]). In this graduation work we will focus on slicing pure procedural programs only, but some ideas will be presented how to extend the same method for object oriented slicing.

1.2   The relational approach

The majority of studies dealing with program slicing are based on some kind of graph representation of the program dependencies. The benefit of this approach is that control and data flow information is combined in a single data structure making analysis easier to perform. The disadvantage is that program and system dependencies graphs are too coarse and not suitable for some more sophisticated tasks. Initially designed for compiler optimization they reduce the various program dependencies to dependencies between statements only. Information about relations between certain variables is discarded. The procedural and object oriented structure of programs can not be expressed completely as well, for example it is not possible to distinguish whether a statement is dependent to a procedure definition or to a procedure call - they produce the same kind of dependence. Some of these problems may be solved by annotating the graphs with additional information and introducing special case treatment in the slicing algorithm, however thus we will lose the main advantage of the approach - its simplicity. This is why researchers started looking for alternative

representations, which are able to accommodate a broader range of program analysis problems. The relational representation of data is widely used in many computer science fields, such as databases and knowledge engineering, so it was a natural step to apply it for program facts representation.

The basic idea of the relational approach is to represent program facts as relations and the slicing algorithm as a series of relational operations. It was first proposed by Jackson [8], though he did not provide any evidence that the method was successfully applied. Later Paul Klint elaborated it in [9] by giving a precise list of algorithmic steps and showed its correctness by examples made by hand. Their work set a solid base for solving more sophisticated slicing problems and building an automatic program slicer.

Representing program facts as relations has several advantages. Firstly it is very flexible because we can add as much information as we want and not overwhelm the structure of the representation. Also we may introduce new relations and thus implement new types of dependences without breaking the existing model. Secondly, the relational model let us formulate questions about the program as relational queries which is convenient and efficient. There is a variety of programming languages specialized for querying relational data and we may use one of them. Lastly, it is relatively easier to extract program facts in the form of relations rather than constructing graphs - relation instances may be generated consequently by several passes through the source code and there is no need to maintain complex data structures. This property makes the fact extraction suitable for implementation by means of a high level abstraction tool which can save a lot of work. The disadvantage of the relational view of programs is that it is less intuitive than graphs and it is virtually not possible to comprehend program dependences just by a looking at the raw data. The relational slicing algorithm is also not as straightforward as the simple graphs traversals used in the alternative program dependence graph approach. One of the challenges of my thesis is to show that the drawbacks of the relational approach are compensated by its virtues.

1.3   Problem statement

The goal of this graduation work is to extend the existing relational algorithm to cover more complex slicing problems and to develop a prototype capable of automatic program slicing. This includes the selection of several program constructs typical of procedural programming languages and adapting them to the relation approach. It is not possible to cover all aspects of program slicing in a single project, but there must be a clear proposal how to extend the method. The developed techniques have to be verified that they are correct and complete. A formal proof can hardly be derived for a subject as program slicing, where most publications tend to use informal descriptions and there is no firm criteria about correctness and completeness. Instead various case studies that show the relevance of the proposed method have to be investigated. For this purpose we need a prototypical program slicer that implements the given ideas and that can be applied to a variety of sample programs. An important consideration is that the proposed relational approach has to be language independent as long as we deal with imperative kinds of programming. That is why neither the theoretical slicing techniques, nor the prototype should be bound to a particular

programming language. The final solution must be as generic as possible and has to highlight a clear path for its further development and application.

1.4   Outline of this paper

The rest of the paper describes in detail the ideas sketched above.

In **Chapter 2** a basic relational algorithm is described and it is justified that it can be applied to a variety of program entities without modification. A crucial moment is the observation that with this approach the complexity of slicing is transferred to the fact extraction phase.

Chapter 3 introduces the generic, language independent fact extractor and elaborates on its applicability to various imperative programming languages.

Chapter 4 presents two case studies of applications of the relational approach - slicing Pico and Java programs. It gives concrete examples what changes in the syntax definition of a language are needed to prepare it for slicing.

In the last **Chapter 5** the project is summarized and several conclusions are drawn. It also makes proposals for further development of the presented ideas.

Chapter 2

The slicing algorithm

2.1 Program analysis terminology

Program slicing is a kind of program analysis and some conceptual program analysis terms have to be introduced in order to describe formally the relational slicing algorithm. Their definition can be found in any book about compilers construction, such as [7]. The terms refer either to the control or the data flow graph of a program. It is possible to combine both graphs in a single Program Dependence Graph (PDG) [4]. Consider the program and its PDG in Figure 2.1. The solid and dot lines show the control and data flow, respectively.

#1 bool x = false;	
#2 int t = 0;	
#3 for (int i = 1; i < 10; i++)	
{	
#4 if (x)	
#5 t = t * i;	
else	
#6 t = t + i;	
#7 x = true;	
}	

(a)   (b)
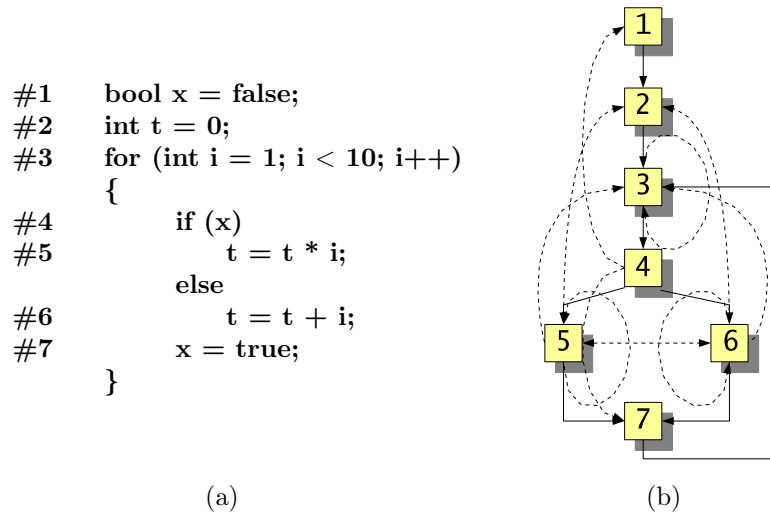
Figure 2.1: Sample program and its PDG

A *predecessor* of a program statement s is any other statement that may be executed just before s. Using the program graph notation the predecessors of a node are all nodes with control flow connections directed to it. The predecessor of statement #4 in Figure 2.1 is #3, while #7 has two predecessors - #5 and #6.

Statement d *dominates* statement n if by any execution of the program d is

executed before n. The graph meaning of domination is that every path from the root of the control flow graph to n goes through d. In Figure 2.1 statement #4 dominates #5, #6 and #7, but #5 does *not* dominate #7. .

A *variable definition* is a pair $\langle s, v \rangle$ and means that a value is assigned to the variable v at statement s. The variable definitions of the sample program are:

$$\langle 1, x \rangle, \langle 2, t \rangle, \langle 3, i \rangle, \langle 5, t \rangle, \langle 6, t \rangle, \langle 7, x \rangle$$

A *variable use* is a pair $\langle s, v \rangle$ and means that the value of variable v is used at statement s. The variable uses in our example are:

$$\langle 3, i \rangle, \langle 4, x \rangle, \langle 5, t \rangle, \langle 5, i \rangle, \langle 6, t \rangle, \langle 6, i \rangle$$

A variable definition is *killed* at statement s if the value of the variable is changed there.

The *reaching definitions* of a statement s are all variable definitions that may be still valid (not killed) just before the execution of s. The reaching definitions of the program in Figure 2.1 are shown in Figure 2.2.
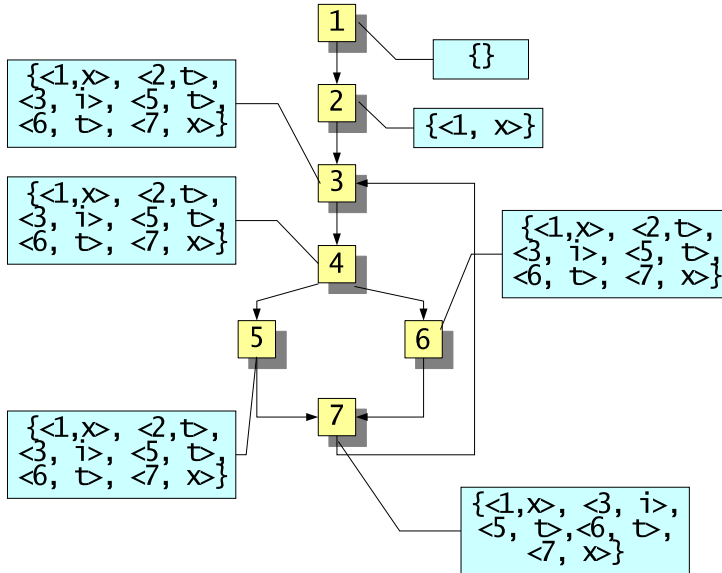
Figure 2.2: Reaching definitions

2.2 Informal description

The idea of backward program slicing is intuitively recognized by every programmer. Generally it consists of tracing back the program fragments that contribute to the values of the variables specified in the slicing criterion. If we disregard conditional operators this means that we have to find all reaching definitions of the slicing criterion that set some of its variables. Then for each of the variables used in these definition statements we run the slicing process

again recursively. Finally all dependences are summed to form the desired program slice. Conditional operator statements do not directly set any values but they may change the order of execution and thus influence the final value of the variables in question. That is why the dependeces to variables that determine the outcome of a specific test of a conditional statement have to be included in the slice if the possible execution paths may affect the slicing criterion variables in a different way. For example in Figure 2.1 the value of x at statement #4 contributes indirectly to the value of t at statement #7 because it determines whether t will be multiplied or incremented by i at statements #5 and #6, respectively.

2.3   Formal specification

The exact algorithm presented in this chapter was first proposed in [8] and is described in detail in [9]. It is able to produce backward slices given a criterion $\langle s, v \rangle$, where s is a statement identifier and v is a variable. The final slice contains all variable uses that are important with regard to the slicing criterion. This algorithm is applicable to single block procedural programs only and, for example, there are no instructions how to do interprocedural or object oriented slicing.

A relational approach to program slicing implies that the program facts have to be represented as instances of relations. The following relations are required for program slicing:

- The control flow relation (Pr) relates a statement to each of its predecessors:

$$Pr : \{\langle s_i, s_j \rangle \mid s_i \in S, s_j \in P_{s_i}\},$$

  where S is a set of all the statements used throughout the program and P_{s_i} are the predecessors of statement s_i.

- The variable definition relation (Df):

$$Df : \{\langle s_i, v \rangle \mid s_i \in S, v \in V_{s_i}^d\},$$

  where $V_{s_i}^d$ is the set of variables defined at statement s_i.

- The variable uses relation (Us):

$$Us : \{\langle s_i, v \rangle \mid s_i \in S, v \in V_{s_i}^u\},$$

  where $V_{s_i}^u$ is the set of variables used at statement s_i.

- The unary relation Cr defines which statements are control flow statements:

$$Cr : \{\langle s_i \rangle \mid s_i \in C_s\},$$

  where C_s contains all control flow statements in the program.

Two additional variables are introduced to tie together control and data flow:

- The variable θ represents the outcome of a specific test of a conditional statement. The conditional statement defines θ and all statements that are control dependent on this conditional statement will use θ.

- The variable ε represents the potential execution dependence of a statement on some conditional statement. The dependent statement defines ε and an explicit (control) dependence is made between ε and the corresponding θ.

Once program dependencies are available in the form of relations the computation of a program slice can proceed in 10 steps:

1. Compute the R_1 relation that defines which variable definitions are undone (killed) at each statement:

$$R_1 = \{\langle s_i, \langle s_j, v \rangle \rangle \mid \langle s_i, v \rangle \in Df, \langle s_j, v \rangle \in Df, s_i \neq s_j\}$$

2. Compute the reaching definitions relation R_2 by the following system of equations:

$$R_2' = \{\langle s, \langle s, v \rangle \rangle \mid \langle s, v \rangle \in Df\}$$
$$R_2'' = \{\langle s_i, d \rangle \mid \langle s_j, s_i \rangle \in Pr, \langle s_j, d \rangle \in R_2'''\}$$
$$R_2''' = \{\langle s, d \rangle \mid \langle s, d \rangle : R_2' \cup (R_2'' \setminus R_1)\}$$
$$R_2 = R_2''$$

This is a fixed point computation which recursively (because of the mutual referring of R_2'' and R_2''') accumulates all definitions which occur at (R_2') or prior to a given statement (R_2''), but which has not been killed yet (assured by R_2'''). The concise definition of the reaching definitions concept distinctly demonstrates the elegance of the relational approach.

3. Compute the relation R_3 that relates variable uses to their corresponding definitions:

$$R_3 = \{\langle \langle s_i, v \rangle, \langle s_j, v \rangle \rangle \mid \langle s_i, v \rangle \in Us, \langle s_i, \langle s_j, v \rangle \rangle \in R_2\}$$

4. Compute the relation R_4 that connects a variable definition to the variable uses in the same statement:

$$R_4 = \{\langle \langle s_i, v_1 \rangle, \langle s_i, v_2 \rangle \rangle \mid \langle s_i, v_1 \rangle \in Df, \langle s_i, v_2 \rangle \in Us\}$$
$$\cup$$
$$\{\langle \langle s_i, v \rangle, \langle s_i, \varepsilon \rangle \mid \langle s_i, v \rangle \in Df\}$$
$$\cup$$
$$\{\langle \langle s_i, \theta \rangle, \langle s_i, v \rangle \mid s_i \in Cr, \langle s_i, v \rangle \in Us\}$$

Apart from the trivial case when a variable is assigned a value determined by the value of another variable this relation also sets the dependence of the variable being defined to ϵ, which stands for the potential execution of the whole statement. Also, according to the definition of θ its value is defined by the outcome of a specific test and all variables used in statements whose execution is control dependent on the test are using it. Hence in this case we have a dependence between a variable definition (θ) and the variable uses in the same statement (the control statement) and we have to add it to the relation R_4.

5. Compute the dominators relation R_5:

$$R_5' = \{s_i \mid \langle s_i, s_j \rangle \in Pr\} \cup \{s_j \mid \langle s_i, s_j \rangle \in Pr\}$$
$$R_5'' = \{s_i \mid \langle s_i, s_j \rangle \in Pr\} \setminus \{s_j \mid \langle s_i, s_j \rangle \in Pr\}$$
$$R_5''' = \{\langle s_i, s_j \rangle \mid \langle s_i, s_j \rangle \in Pr, s_i \in R_5''\}$$
$$R_5 = \{\langle s_i, s_j \rangle \mid s_i \in R_5', s_j \in R_5' \setminus (R_5'' \cup s_i),$$
$$\{\langle s_t, s_j \rangle \in R_5''' \circ \{\langle s_k, s_l \rangle \in Pr, s_k \langle\rangle s_i, s_l \langle\rangle s_i\}+\}\},$$

where $+$ is the transitive closure operation. Here R_5' is the carrier of the predecessors relation - it contains all statements identifiers appearing in any of the relation components. R_5'' consists of only those statements that have no predecessors and R_5''' contains the Pr instances that are formed by the statements in R_5''. Finally R_5 generates the dominator tuples using the idea that a statement s_i dominates all other statements except for the ones that can be reached from the root of the program (R_5'') without passing through s_i.

6. Compute the control dominator relation R_6 containing only control-flow dominators:
$$R_6 = \{\langle s_i, s_j \rangle \mid \langle s_i, s_j \rangle \in R_5, s_i \in Cr\}$$

7. Compute the relation R_7 that links all ε variables to their corresponding θs:
$$R_7 = \{\langle \langle s_i, \varepsilon \rangle, \langle s_j, \theta \rangle \rangle \mid \langle s_j, s_i \rangle \in R_6\}$$

This relation expresses the control dependences of program statements to their control-flow dominators. The idea is that control dependent statements are executed (potential execution is indicated by ϵ) only if the condition of the corresponding control-flow statement, represented by θ, is evaluated to a certain value.

8. Compute the relation R_8 that combine use and definition dependencies with control dependencies:

$$R_8 = R_3 \cup R_7$$

At this step R_8 contains all possible dependences between variable uses and variables definitions, both the trivial cases and the dependences formed by control flow operators.

9. Compute R_9 that contains dependencies of uses on uses:

$$R_9 = (R_8 \circ R_4)\star,$$

where $\circ$ stands for the composition operation and $\star$ is the reflexive closure operation. The composition links the variable uses dependent on certain variable definitions with the variable uses, that the definitions, on their part, are dependent on. The reflexive closure is needed to trace chain dependences. As a result we have all dependences between variable uses throughout the program and the computation of a program slice is now trivial.

10. The backward slice for a given slicing criterion (i, v) is the projection of R_9 for the slicing criterion:

$$Sl(s_i, v_1) = \{\langle s_j, v_2 \rangle \mid \langle \langle s_i, v_1 \rangle, \langle s_j, v_2 \rangle \rangle \in R_9\}$$

The execution of the algorithm is illustrated in Figure 2.3. The resulting backward slice contains variables uses, among which is the execution variable ε, which means that execution of the corresponding statement may contribute to the final value of the variable specified in the slicing criterion. Complete implementation of the slicing algorithm in RScript is included in Appendix A. It is very close to the formal description given in this chapter due to the relational orientation of RScript.

#1	n = 2;
#2	if (c > 10)
#3	t = n;
	else
#4	n = t;
#5	z = n;

Us	Df	Pr	Cr
$\langle 2, c \rangle$	$\langle 1, n \rangle$	$\langle 1, 2 \rangle$	$\langle 2 \rangle$
$\langle 3, n \rangle$	$\langle 3, t \rangle$	$\langle 2, 3 \rangle$	
$\langle 4, t \rangle$	$\langle 4, n \rangle$	$\langle 2, 4 \rangle$	
$\langle 5, n \rangle$	$\langle 5, z \rangle$	$\langle 3, 5 \rangle$	
		$\langle 4, 5 \rangle$	

#1	n = 2;
#2	if (c > 10)
	else
#4	n = t;
#5	z = n;

(a) Sample Program   (b) Relational representation   (c) Slice

$R_1 = \{\langle 1, \langle 4, n \rangle \rangle, \langle 4, \langle 1, n \rangle \rangle\}$

$R_2 = \{\langle 4, \langle 1, n \rangle \rangle, \langle 3, \langle 1, n \rangle \rangle, \langle 2, \langle 1, n \rangle \rangle, \langle 5, \langle 4, n \rangle \rangle, \langle 5, \langle 3, t \rangle \rangle, \langle 5, \langle 1, n \rangle \rangle\}$

$R_3 = \{\langle \langle 3, n \rangle, \langle 1, n \rangle \rangle, \langle \langle 5, n \rangle, \langle 1, n \rangle \rangle, \langle \langle 5, n \rangle, \langle 4, n \rangle \rangle\}$

$R_4 = \{\langle \langle 3, t \rangle, \langle 3, n \rangle \rangle, \langle \langle 4, n \rangle, \langle 4, t \rangle \rangle, \langle \langle 5, z \rangle, \langle 5, n \rangle \rangle, \langle \langle 1, n \rangle, \langle 1, \varepsilon \rangle \rangle,$
$\quad \langle \langle 3, t \rangle, \langle 3, \varepsilon \rangle \rangle, \langle \langle 4, n \rangle, \langle 4, \varepsilon \rangle \rangle, \langle \langle 5, z \rangle, \langle 5, \varepsilon \rangle \rangle, \langle \langle 2, \theta \rangle, \langle 2, c \rangle \rangle\}$

$R_5 = \{\langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 2, 5 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 1, 5 \rangle\}$

$R_6 = \{\langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 2, 5 \rangle\}$

$R_7 = \{\langle \langle 5, \varepsilon \rangle, \langle 2, \theta \rangle \rangle, \langle \langle 4, \varepsilon \rangle, \langle 2, \theta \rangle \rangle, \langle \langle 3, \varepsilon \rangle, \langle 2, \theta \rangle \rangle\}$

$R_8 = \{\langle \langle 3, n \rangle, \langle 1, n \rangle \rangle, \langle \langle 5, n \rangle, \langle 1, n \rangle \rangle, \langle \langle 5, n \rangle, \langle 4, n \rangle \rangle, \langle \langle 5, \varepsilon \rangle, \langle 2, \theta \rangle \rangle,$
$\quad \langle \langle 4, \varepsilon \rangle, \langle 2, \theta \rangle \rangle, \langle \langle 3, \varepsilon \rangle, \langle 2, \theta \rangle \rangle\}$

$R_9 = \{\langle \langle 3, \varepsilon \rangle, \langle 2, c \rangle \rangle, \langle \langle 4, \varepsilon \rangle, \langle 2, c \rangle \rangle, \langle \langle 5, \varepsilon \rangle, \langle 2, c \rangle \rangle, \langle \langle 5, n \rangle, \langle 4, \varepsilon \rangle \rangle,$
$\quad \langle \langle 5, n \rangle, \langle 4, t \rangle \rangle, \langle \langle 5, n \rangle, \langle 1, \varepsilon \rangle \rangle, \langle \langle 3, n \rangle, \langle 1, \varepsilon \rangle \rangle, \langle \langle 5, n \rangle, \langle 2, c \rangle \rangle\}$

$Sl(5, z) = \{\langle 4, \varepsilon \rangle, \langle 4, t \rangle, \langle 2, c \rangle, \langle 1, \varepsilon \rangle\}$

(c) Algorithmic steps

Figure 2.3: Program slicing in action

2.4 Extending the algorithm

The relational algorithm described above is designed to slice single block programs only. It is not aware of complex program entities such as procedures and classes. For example it is not possible to apply it directly for interprocedural slicing. To enable such functionality and in order to keep the relational view

of the problem we can add more information in the form of new relations containing data about procedural calls, arguments and formal parameters, return statements, etc. Then we will have to extend the algorithm with more relational operations that handle the new relations. The same will have to be done to enable object oriented slicing (we need new relations for class inheritance and polymorphism). Finally we will end up with such a complicated algorithm that it will be extremely hard to verify it and make amendments to it. For the sake of better understanding and efficient implementation one would like to keep the process of slicing as simple and straightforward as possible.

The key question is whether we actually need to change the given 'core' slicing algorithm to handle complex program entities. We know that all procedural programming languages eventually are mapped to a low-level language that computers can understand. Such a machine language is fairly simple and is generally also not aware of program entities others than variable definitions, variable uses and control flow operators. In fact any procedural program can be transformed to consist of a single block - this is what compilers do. The same holds for object oriented programs. Hence it must be possible first to make a syntax transformation of a program to eliminate complex entities and then extract the relations we already know. Thus the interprocedural and object oriented slicing information will be presented without introducing new relations and the same slicing algorithm can be used.

Once we have decided that we are going to use the basic relations for slicing any kind of procedural program we have to figure out how to represent the information needed for slicing various program entities in terms of these relations. For example we can investigate the cases of variable scoping and interprocedural slicing and see what additional relation instances are required.

2.5 Variable scoping

Variable scoping is a programming concept implemented in almost every imperative language. It lets the programmers define a region of a program where a variable is only valid and makes it possible to use the same identifiers for variables ocurring in different scopes. The slicing algorithm is not able to recognize variable scopes as there is no relation describing them. The solution to this problem is to include scoping information in the variable identification process. It is sufficient to change the definition of a variable by adding the scope identifier. Thus different variables with the same lexical identifier will but distinguished by their scope identifiers. Nothing is changed in the relational slicing algorithm itself, thought in some particular implementations, such as the *RScript* one, the definition of the variable type needs to be modified.

2.6 Interprocedural slicing

There are actually two kinds of interprocedural slicing - slicing descending into procedure (function) calls and slicing ascending from function definitions into the contexts they are called [10]. Consider the following program fragment:

It is a obvious that a backward slicing process started from statement #5 for variable b must descend into the function calls at #4 and the resulting slice

```
int f(int x)                     int f(int #6 x)
{                                {
#1        return x;              #1          return x;
}                                }
void g()                         void g()
{                                {
#2        int a = 10;            #2          int a = 10;
#3        int b = a + 1;         #3          int b = a + 1;
#4        b = f(a);              #4          b = f(a);
#5        return b;              #5          return b;
}                                }
           (a)                              (b)
```

Figure 2.4: Sample procedural program

must include statement #1 which contributes to the final value of b. On the other hand a backward slicing starting from statement #1 must ascend into the calling context of the function and the slice must include statement #2.

A function call makes two contributions to the control flow relation of a program. First the end point statements of the function are added to the list of predecessors of the statements calling it. Second the predecessors of the statements calling a function have to be added to the predecessors of its first statement. For example the predecessor relation for the program fragment listed in Figure 2.4a must be:

$$Pr = \{\langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 1, 4 \rangle, \langle 4, 5 \rangle\}$$

Function calls also have impact on the data flow of a program. The return value of a function is actually a new variable which has to be included in the data flow relations. We will refer to this variable as 'function variable' and we will treat it the same way as other variables with the exception that it will not be included in the final slices. The function variable can be identified by the name of the function and its arity, for example $f/1$, $g/0$. etc. For every function call within a statement we assume that the statement is using the corresponding function variable. The return statements of a function are defining its function variable. In particular for the program listed in Figure 2.4a we have to add the tuples $\langle 4, f/1 \rangle$ and $\langle 1, f/1 \rangle$ to the Us and Df relation respectively.

Another data flow relation issue is the mapping of arguments to formal parameters. It can be regarded as a set of Df instances, one for each parameter. However parameters first appear in function declaration part so that they do not have associated statements. This problem can be overcome by adding 'virtual' statements for each parameter. These statements are used in the slicing process but may be excluded from the final slices. The modified version of the program fragment of Figure 2.4a is shown in Figure 2.4b. With these modifications the data flow relations have the following instances:

$$Df = \{\langle 6, x \rangle, \langle 1, f/1 \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, b \rangle, \langle 5, g/1 \rangle\}$$
$$Us = \{\langle 6, a \rangle, \langle 1, x \rangle, \langle 3, a \rangle, \langle 4, a \rangle, \langle 4, f/1 \rangle, \langle 5, b \rangle\}$$

The Pr relation also has to be modified to include the virtual statements:

$$Pr = \{\langle 2, 3 \rangle, \langle 3, 6 \rangle, \langle 6, 1 \rangle, \langle 1, 4 \rangle, \langle 4, 5 \rangle\}$$

The Cr relation will have no instances for this program:

$$Cr = \{\}$$

Now we can run the slicing algorithm to see how it works for interprocedural slicing. The parameter 'virtual' statements and the function variables are excluded from the slices:

$$Sl(1, x) = \{\langle 2, \varepsilon \rangle, \langle 4, a \rangle, \langle 4, \varepsilon \rangle\}$$
$$sl(5, b) = \{\langle 4, \varepsilon \rangle, \langle 4, a \rangle, \langle 2, \varepsilon \rangle, \langle 1, \varepsilon \rangle, \langle 1, x \rangle\}$$

2.7 From program slicing to fact extraction

We saw in the previous chapter that we can use the relational program slicing algorithm for interprocedural slicing just by extracting the relevant instances of the basic relations from the program. We can make the assumption that the same approach can be applied to object oriented slicing, type-aware slicing, etc. In order to slice object oriented programs we need to add more instances to the Pr relation connecting each method call with the possible methods that could have been called, taking into account inheritance and polymorphism issues. Also we need to resolve class members references as in the case of OO slicing the member variables can be access in a variety of ways. However this will not require introducing any new relations. Type aware slicing requires to involve the type of functions and variables in their identification and again does not lead to extra relations, hence there is no need to change the core algorithm.

In turns out the core algorithm can slice any procedural program and the only difficulty is to supply the desired relations. At this point of my graduation work I realized its subject has shifted a bit from program slicing to fact extraction techniques. This is not accidentally - I deliberately tried to keep the slicing algorithm universal and to encode all the language dependent complexities by means of a few basic relations. I believe this approach makes program slicing more understandable, easier to implement and most importantly it leads to a general program slicing method which can be easily applied to any procedural programming language. However it also poses a new problem which has not been addressed quite well in the existing literature and it is still a major challenge in the field of program analysis - automated source code facts extraction. The difficulty is that all complications due to variable scoping, function identification, variations of control flow statements, now have to be worked out at this preliminary phase. Fact extraction turns into a crucial part of the relational program slicing approach.

Chapter 3

Fact extraction

3.1   The challenge of fact extraction

Extracting the desired relations from the source code of a program is not a trivial task. We first need to parse the program, build a kind of efficient representation and then operate on it. It can be a tedious and time consuming job to implement a fact extractor using conventional tools as flex/bison and C/C++. That is why I decided to use the *ASF+SDF* formalism, which is particularly suitable for code transformation tasks [12]. SDF lets you describe the grammar of the language in a flexible manner and automatically builds its parse tree. By ASF you can define term rewriting functions that operate on the parse tree, including traversal functions, which are very useful for fact extraction. I also used the *MetaEnvironment* which provides a convenient interface to the ASF+SDF back end [13].

According to Sloane and Holdsworth the process of fact extraction can be generalized to three basic steps [11]. First the program is parsed and the parse tree (Figure 3.1) is constructed. Then the tree is annotated with additional information and finally the needed information is generated by traversing the parse tree. This approach inspired me to develop a universal and versatile technique for fact extraction. I did not want to constrain my research by selecting a certain programming language to experiment with. The relational algorithm turned out to be language independent, so it was a logical consequence to let the fact extractor have the same property. Another source of motivation was that at the early stage of this project I tried to develop ad hoc solutions to extract the desired relations from Pico[1] and Java programs. Pico is a pretty simple language and with a few tricks and changes to its grammar I managed to complete the task. However when I had to deal with the much more complicated Java language I came across serious problems. After a few weeks of digging into Java 1.5 grammar and probing various techniques I got completely lost in numerous implementation details and realized that I need a more general method for fact extraction.

---

[1]Pico is a toy language that I used in my experiments, it is further described in Section 4.1

```
int i;
while(i < 10)
{
        write(i);
        i = i++;
}
```
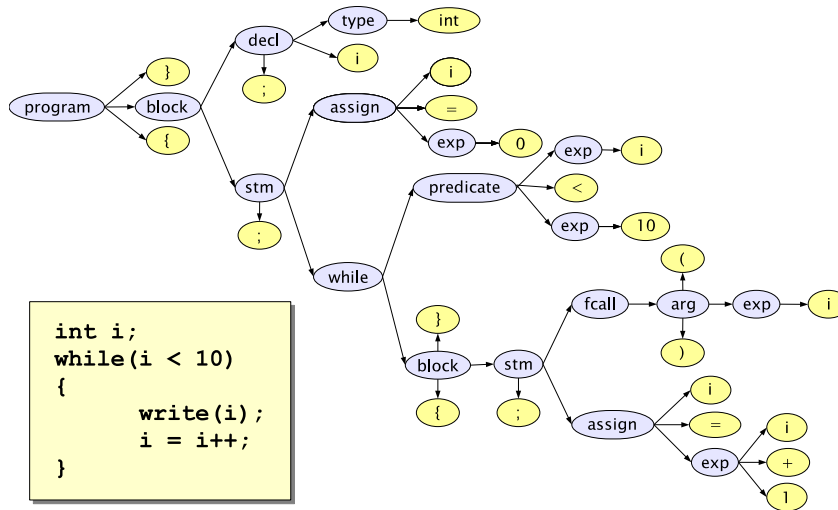
Figure 3.1: Parse tree of a program

3.2 Generalized fact extraction

The main idea of generalized, language-independent fact extraction is that it
works with abstract language constructs, rather than concrete syntax. This is
possible because if we take a look at any of the widespread procedural languages
(in this project only such languages are considered) we will find out that they
all share the same kind of program constructs - functions, variable declarations,
a limited number of control flow operators, such as an *if then else* statement, *for*
loop, a *pre* and *post* conditional loop. Although these common constructs have
syntax specific to the given language they have the same semantics, hence they
have to generate the same kind of data and control flow relations. That is why
a fact extractor is able to abstract from concrete syntax and operate just on
program entities that are common to all procedural languages. For example in
order to generate the control flow relation Pr for a *while do* construct we need
only to know which part of it is the condition, where does the executable block
start and where it ends. Of course the fact extractor also has to be aware of the
general notion of a program statement so that it can distinguish the separate
statements in the execution block and compute the order in which they may be
executed. But it doesn't need to 'know' how exactly a loop with precondition
is defined in a specific language. The same idea can be applied for extracting
variable definitions. The fact extractor has to be aware only of the notions
of a variable definition and a variable and then it can descend into variable
definitions and search for any variable occurrences - if it finds any it has to add
them to the Df relation.

However if the fact extractor doesn't take into account the syntax of a spe-
cific language, how is it going to operate on its abstract syntax tree? The
answer is - by means of traversal functions. The ASF formalism allows to define
traversal functions that can both accumulate information by traversing the tree
and transform it. A traversal function does not need to be defined to handle
all the syntax of a language, but it works just on certain 'points of interest'.

More details how to use traversal functions for source code transformation are given in [14]. What we benefit from implementing the fact extractor by means of traversal functions is that we can keep it abstract, operating just on the abstract common procedural program entities. Then, when we decide to apply it to a specific language, we just need to point out which part of its grammar corresponds to which of the abstract entities.

The major difficulty of the generalized fact extraction approach is the selection of program entities which are typical for procedural languages. Naturally not all of them have all the common entities, for example some scripting languages such as PHP lack the concept of variable declaration, instead the memory for a variable is allocated the first time it is used. So we should take the union of all procedural programming notions and then for a specific language use just a relevant subset of it. Figure 3.2 shows a table of the program entities I have selected for the fact extraction prototype. The list is by no means complete and more entities have to be added, such as class denotations, a *switch* statement, types and others.

Program	General notion of a program
Begin Scope	Beginning of a variable scope, it could be an opening curly bracket, the beginning of a loop, function definition, etc
End Scope	End of variable scope
Variable	The notion of a variable, including arrays, local variables, class member fields, etc
Variable Id	Variable identifier, usually terminal symbol in the input language grammar
Variable Use	Any context where a variable occurrence is considered to be a variable use
Variable Definition	Any context where the first variable occurrence is considered to be a variable definition
Function Id	Function identifier
Function Definition	Definition of a function, also a class method
Function Parameter	Definition of a parameter used in a function definition
Function Call	Function call, either a single statement or used within expression
Function Argument	Argument used in a function call
Statement	General notion of a statement, could be any program statement
Return Statement	Statement that sets the return value of a function
If Statement	*If then else* conditional statement
If Block	The first block of a *If then else* statement that is executed if the condition is satisfied
Else Block	The second block of a *If then else* statement that is executed if the condition is not satisfied
Do While Statement	A loop with a pre condition
Do While Block	The inner executable statements of a *do while* loop
For Statement	For loop
For Block	The inner executable statements of a *for* loop
While Do Statement	A loop with a post-condition
While Do Block	The inner executable statements of a *while do* loop

Figure 3.2: Common procedural program entities

3.3 Implementation

The implementation of the generalized fact extractor consists of two parts - a set of abstract algorithms and mappings of a target language grammar to the generalized program entities. It is illustrated in Figure 3.3. There is one mapping module for each language that we want to process - Pico and Java in this particular case. The fact extraction algorithm works just with the abstract syntax entities and the mapping serves to associate concrete syntax elements to them. All the complexity is concentrated in the abstract algorithms, which do not have to be changed for a specific language, instead we have to provide a new mapping for it.
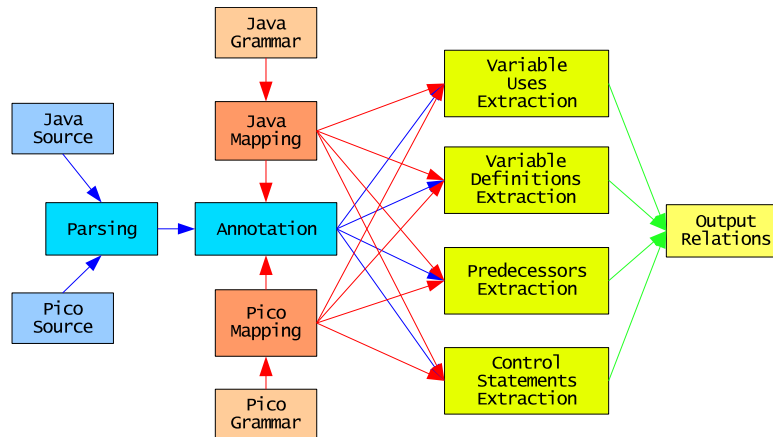

Figure 3.3: Implementation of the fact extractor

The fact extraction process is based on the parse-annotate-extract paradigm proposed by Sloane and Holdsworth in [11]. The advantage of this approach is that there is no need to maintain sophisticated data structures during the extraction process which could be overwhelming when using the ASF+SDF formalism. All the information specific to various language constructs is kept in the form of annotations attached to the parse tree nodes.

The first step of the fact extraction is to parse the input language. We assume the grammar of the language being processed is encoded in SDF. The choice of SDF is important as it allows to make modifications to a grammar without breaking its lookahead constraints and thus introducing ambiguous grammar sections[2]. SDF also provides a way to manually resolve ambiguities, when it is not possible to be done automatically.

Once the input program is parsed we have its parse tree representation. The next step is to annotate the tree. Annotation is needed primary to resolve ambiguities and cross references in the program, such as variable scopes and function calls. The following annotations are currently implemented:

- A variable occurrence is annotated with an unique identifier - a pair of its lexical identifier and the numeric id of the scope in which it is defined.

---

[2]Some parser generators require all ambiguities to be resolved with a certain number of looks (*lookaheads*) in the incoming token string. SDF doesn't have such a limitation.

Variables are also annotated with the line numbers of the statements they occur in.

- A statement is annotated with its line number - an integer value. Function parameters also get a statement identifier, because they are treated as 'hidden' assignment statements, where parameters are given the values of arguments.

- A function definition is annotated with a unique function symbol - a pair of the function lexical identifier and its arity.

- A function parameter is annotated with the list of possible arguments. For example if there is a function:

```
void f(int a, int b)
```

and a set of calls to it:

```
 f(10, x), f(y, x), f(x + y, z)
```

then a will be annotated with $\{y, x\}$ and b with $\{x, z\}$.

An example of all annotations is shown in Figure 3.4
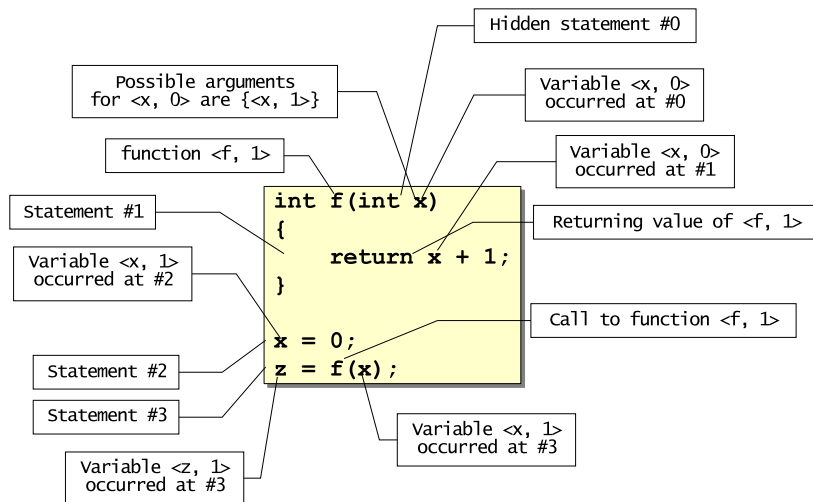
Figure 3.4: Program annotation

The final step of the fact extraction algorithm is to traverse the annotated syntax tree and generate the desired facts, or relations in our case. It consists of four modules:

- **Extracting** Us. This module generates relation instances for all variable uses. Functions calls are also added - they use the return value of the function, denoted by the function symbol. The variable uses in the list of possible arguments attached to each function parameters are also included. For example:

```
void f(int #1 a)
{
#2    return a + 1;
}


#3 z = x;
#4 y = f(x + 1) * x;
```

generates the following relation:

$$Us = \{\langle 1, x \rangle, \langle 2, a \rangle, \langle 3, x \rangle, \langle 4, x \rangle, \langle 4, f/1 \rangle\}$$

- **Extracting** Df. Except for the ordinary variable definitions (assignments) Df relation are also generated by return statements (they define the return value of the corresponding function) and function parameters(they are define the value of the parameter variable). For example:

```
void f(int #1 a)
{
#2    return a + 1;
}


#3 z = x;
#4 y = f(x + 1) * x;
```

generates:

$$Df = \{\langle 1, a \rangle, \langle 2, f/1 \rangle, \langle 3, z \rangle, \langle 4, y \rangle\}$$

- **Extracting** Pr. The Pr relation is generated by a set of predefined control flow statements. All statements that are not explicitly described as control flow are considered to be simple single line ones, such as assignments or assertions. The control flow statements are treated according to their specific behavior. The general strategy to handle a control flow statement is first to 'catch' the whole statement and then to process the block which is being executed conditionally or in a loop. For each composite statement there are a number of statements that are possible 'terminations' of this statement, for example in the following case:

```
#1 if (x < 10)
{
#2    z = 10;
}
else
#3    z = 20;
}
```

the terminations are #1 and #3, because these are the last statements that can be executed within this composite statement. The termination

23

of a single line statement is the line number of the statement itself. The set of termination statements are passed as a parameter to the top-down traversal function that generates the predecessor relations. Such a traversing must visit the statement in their correct execution order, except for the control flow statements, which are processed in a special way. As already explained the parameters of a function are treated as hidden assignment statements, so they also have to be added to the desired set of relations as predecessors of any function call. Also function calls are preceded by the return statements of the corresponding function:

```
void int f(int #1 a, int #2 b)
{
#3    if (a)
#4        return b;
      else
#5        return a;
}

#6# z = 10;
#7# x = z * f(10, z);
```

generates:

$$Pr = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 3, 5 \rangle, \langle 6, 1 \rangle, \langle 4, 7 \rangle, \langle 5, 7 \rangle\}$$

- **Extracting** Cr. The relation is populated with the identifiers of all control flow statements.

More details about the implementation of the generalized fact extraction algorithms are given in appendix B.

3.4   Evaluation

The biggest advantage of the fact extraction method proposed in this chapter is its generality. It is not constrained to a particular programming language and it is easy to understand and modify it due to its high level of abstraction. However the tendency to abstract from details often leads to troubles when dealing with real world problems. Some potential obstacles are:

Completeness. We have assumed that all procedural programming languages are based on the same concepts. However it may turn out that there is variety of languages that do not fit in this model. For example Perl and Python are thought to be procedural languages but they have some features typical of functional programming. We can extend the model to cover them but there will always be other examples that fall out of it. So we can not claim the fact extractor and, as a result, the slicing algorithm are totally language independent and universal.

Extensibility. Ideally implementing a new type of language construct that the fact extractor can handle will require just adding a new module. In reality

it is not so simple because the various fact extractor modules are tightly connected to each other. For example the generation of the variable uses relation is dependent on the annotation of control flow elements such as return statements. That is why it can take significant amount of time and efforts to extend the algorithm.

Correctness. The relevance of the generated relations is crucial for correct program slicing. We assume that similar procedural language constructs have similar semantics, hence they will produce the same relations. This is not necessarily true. For example in some languages the variables declared in the body of a loop are still available after its execution and in other languages - they are not.

Some of these problems can be overcome by tweaking the input language grammar so it fits better into the model. Changing grammars is not fatal as long as it doesn't change their properties from the point of view of program slicing. It is not necessary to keep modified grammars equivalent to the their original versions as only syntactically correct programs are going to be sliced. In the next chapter we will see how we can make a language 'sliceable' just by making slight modifications to its grammar.

Chapter 4

A case study - Pico and Java slicing

In order to show the advantages of the presented relational program slicing approach we need to apply it to at least two languages. Ideally a case study will include more languages that feature a variety of programming constructs, but such a sophisticated experiment will require unacceptable amount of time and effort. That is why I choose for my test a pair of languages that have little in common and must be representative enough for the family of procedural programming languages.

4.1 Pico

Pico is a toy language which was developed as a part of the ASF+SDF distribution. Its syntax and semantics are very simple - it has just a few standard control flow operators, three data types and the notion of variables. This functionality is not sufficient for advanced slicing experiments and I had to extend it by adding procedures and variable scopes. There is no wide accepted standard for Pico so I was free to modify its syntax and semantics. Initially I developed an ad hoc fact extractor for Pico but later I replaced it with the generalized solution and a mapping of its grammar to the abstract syntax elements. I didn't need to do significant restructuring of Pico grammar to enable slicing it - this is partly because it is very simple, but also because when I was extending it I had in mind that I am going to slice this language later. However there are a few examples that illustrate how I had to adapt Pico to the generalized fact extractor and thereafter apply the relation slicing algorithm to it:

- **Variable Scopes**. Original Pico lacks variables scopes - variables can be defined only in the beginning of the program and they are available everywhere. Later I extended its grammar to support 'blocks' that start with a variable definition section and limit the variable scope. Blocks can be nested within the program, they may also form the body of loops or procedures. Below is a fragment from the Pico grammar that describes their syntax:

    . . .

```
"begin" -> BEGIN
"end" -> END
BEGIN DECLS SERIES END -> BLOCK
STATEMENT? -> SERIES
STATEMENT ";" SERIES -> SERIES
IFSTATEMENT -> STATEMENT
WHILEDOSTATEMENT -> STATEMENT
BLOCK  -> STATEMENT
"function" FUNC-ID "(" FPARAMS ")" BLOCK -> FUNCTION
...
```

It is easy to distinguish the *begin scope* and *end scope* defined in Figure 3.2 - *BEGIN* end *END*, respectively. These two syntax elements denote the beginning and end of a variable scope. However there is a problem with the function declaration - function parameters must be available throughout the function body and if we just map *BEGIN* and *END* the parameters will fall out of the function body variable scope. So in the case of a function declaration the scope must begin with the beginning of the declaration itself. The following modification to the syntax of a Pico function solves the problem:

```
"function" -> FSTART
FSTART -> BEGIN
BEGIN FUNC-ID "(" FPARAMS ")" "begin" DECLS SERIES END -> FUNCTION {prefer}
```

- **Variable Definitions**. We need to identify the syntax element that represents a context where a variable can be defined. Usually this is the left hand side of an assignment statement. Pico defines it as follows:

  ```
  VARIABLE ":=" EXP -> STATEMENT
  ```

  It is obvious that we can not map *VARIABLE* to *variable definition* - not any occurrence of a variable changes its value. We have to modify the assignment statement:

  ```
  LHS ":=" EXP -> STATEMENT
  VARIABLE -> LHS
  ```

  Now we can map *LHS* to *variable definition*.

Appendix C contains the complete grammar of Pico and directions how it is mapped to the abstract syntax entities described in Figure 3.2. A major observation is that the modifications described above do not change Pico semantics in any way. Only syntax is changed and it is done in a way that all programs recognizable by the original grammar are recognized by the modified version as well. A side effect is that the new grammar may also accept some illegal Pico code but it is a not problem since we are slicing syntactically correct programs only. A practical program slicer will have a syntax checker which is ran before slicing begins.

The experiments with Pico showed that the program slice works in practice. I managed to apply it to Pico without changing the core relational and fact

extraction algorithms. I had to modify not more than 10 lines of code in the Pico SDF grammar to adapt it for slicing - an incredibly lower amount of work compared to developing a standalone Pico slicer. Figure 4.1 shows the result of applying the relation approach to program slicing to a sample Pico program.

```
      function f(i : natural)          ...
      begin                       #3       return i + 1;
          declare;                     ...
#3        return i + 1;           #6       i := 0;
      end                         #7       while(i) do
      begin                            ...
          declare              #10              i := i + 1;
              i : natural,           ...
              k : natural;
#5        k := 0;                        (b) Sl(9,i)
#6        i := 0;                   ...
#7        while(i) do          #3       return i + 1;
          begin                    ...
              declare;         #5       k := 0;
#9            k := f(i);       #6       i := 0;
#10               i := i + 1;  #7       while(i) do
          end;                     ...
#11       write(k);            #9               k := f(i);
#12       write(i);            #10              i := i + 1;
      end                          ...
        (a) Sample program              (c) Sl(11,k)
```

$$Sl(3,i) = \{\langle 10,i\rangle, \langle 10,\epsilon\rangle, \langle 9,i\rangle, \langle 9,\varepsilon\rangle, \langle 6,\epsilon\rangle, \langle 7,i\rangle\}$$
$$Sl(5,k) = \{\}$$
$$Sl(6,i) = \{\}$$
$$Sl(7,i) = \{\langle 6,\epsilon\rangle, \langle 10,\epsilon\rangle, \langle 10,i\rangle\}$$
$$Sl(9,i) = \{\langle 6,\epsilon\rangle, \langle 10,\epsilon\rangle, \langle 10,i\rangle, \langle 7,i\rangle, \langle 3,\epsilon\rangle, \langle 3,i\rangle\}$$
$$Sl(10,i) = \{\langle 10,\epsilon\rangle, \langle 6,\epsilon\rangle, \langle 7,i\rangle\}$$
$$Sl(11,k) = \{\langle 5,\epsilon\rangle, \langle 9,\epsilon\rangle, \langle 9,i\rangle, \langle 3,i\rangle, \langle 3,\epsilon\rangle, \langle 10,i\rangle, \langle 10,\epsilon\rangle, \langle 6,\epsilon\rangle, \langle 7,i\rangle\}$$
$$Sl(12,i) = \{\langle 6,\epsilon\rangle, \langle 10,\epsilon\rangle, \langle 10,i\rangle, \langle 7,i\rangle\}$$

(d) All slices

Figure 4.1: Pico program slicing

4.2 Java

The case study of Pico program slicing showed that the method is correct, but it is not convincing enough that it can be used in real world applications. That is why I decided to do a second experiment with a widely used programming language. The choice of Java was reasonable as it has may of the features of modern imperative programming and at the same time it has a well defined standard and its syntax and semantics are relatively simple compared to C++

and Perl, for example. Also I was able to use an existing Java 1.5 SDF grammar [15], defined in a clean and understandable way.

Java is a much more sophisticated programming language than Pico and naturally its grammar requires more modifications to be adapted to the generalized fact extractor. Figure 4.2 shows some quantitative information about the amount of work I had to do to apply the relation program slicing to Java. It is observable that the number of modifications is small compared to the total size of the grammar and the changes are concentrated in specific parts of the grammar. We may confidently assume that implementing a complete Java program slicer will require an amount of effort consistent with the presented results.

Number of lines	
Original grammar	1863
Modified grammar	1890
Percentage differnece relative to the original version	1.45%
Number of non-terminal symbols	
Original grammar	165
Modified grammar	179
Percentage differnece relative to the original version	8.48%
Number of rules	
Original grammar	525
Modified grammar	541
Percentage differnece relative to the original version	3.05%
Number of common rules in the original	
and the modified version	489
Relative to the original version	93.14%

Figure 4.2: Java program slicing modifications summary

The modifications to the Java grammar are similar to the Pico ones. For example I had to modify the block statement to have a distinct beginning and ending used to denote variable scopes:

```
"{" BlockStm* "}" -> Block
```

was changed to:

```
"{" -> BlockStart
"}" -> BlockEnd
 BlockStart BlockStm* BlockEnd -> Block
```

Many of the modifications were just renaming certain parts of the grammar so that they can be mapped to the abstract syntax entities. For example the syntax of method invocations should have an explicit argument symbol:

```
MethodId "(" {Expr ","}* ")" -> MethodInvocation
```

was changed to

```
MethodId "(" {Argument ","}* ")" -> MethodInvocation
Expr -> Argument
```

There were also other kinds of changes but none of them required more than a few lines of additional SDF code. Once the fact extractor was adapted to Java I developed a variety of test cases to see how the relational program slicing algorithm works with this language. All experiments confirmed the correctness of the method. Appendix D contains the source code and results of one of the experiments.

Chapter 5

Summary

The goal of my graduation work was to thoroughly study the basic relational approach proposed in [8] and [9] and give ideas how it could be extended to cover more aspects of program slicing. I also had to develop a prototype which shows that the proposed methods are viable and can be applied in practice. In the course of my work several unexpected obstacles appeared, some of them due to the intrinsic properties of the subject and some due to implementation problems. I tried to find answers to the most challenging questions and to suggest solutions for the rest. In this chapter there is a brief summary of the achievements of this project, its deficiencies and failures and finally there is a judgment whether how successful it was.

5.1   The relational approach in general

The relational approach to program slicing turned to have several positive properties. In my opinion the most valuable result is the observation that the core slicing algorithm is powerful enough to solve complex slicing problems. That let us keep it relatively simple and concise, it is easy to understand and analyze and, due to its generality, it is not constrained by a particular programming language. The advantages of such a solution are clear - it allows building an efficient and versatile program slicer. However we have to take into account that the method is designed for slicing imperative programming languages only. In fact not even any kind of imperative programming slicing is supported, but just slicing in the sense of Weiser's definition. I believe this limitation is acceptable as the general idea of slicing is too broad and any reasonable project in this area has to put some constraints on its applicability.

The second important result is that it turned out that most of the complexity of the relational approach to program slicing is concentrated in the fact extraction phase. In order to keep the fact extractor consistent with the generality of the core algorithms I had to find a way to abstract from any concrete programming language details. This ambitious goal was significantly simplified by the decision to focus on the imperative programming paradigm only. The basic idea of generalized fact extraction was that all imperative languages share a set of common programming concepts and, in general, have similar syntax and semantics. This assumption is intuitively supported by the observation that once

programmers have mastered some of the popular imperative languages, they can quickly learn others just by associating the new language constructs to the ones they already know. Defining the fact extraction algorithm by means of abstract program entities makes it very easy to map it to any existing programming language that fits in the model. Given the indisputable language independence of the core slicing algorithm, this flexibility of the fact extractor leads to one of the goals of the whole program slicer - to be as general and universal as possible. Two case studies were investigated to demonstrate empirically the desired properties and they succeeded. However a deeper research of the existing imperative programming languages shows that not all of their constructs have corresponding counterparts in the suggested abstract syntax. We can extend the fact extractor by supporting more constructs but this requires changing the abstract algorithms and questions its generality. Hence we have to admit that the fact extraction, and as a result the relational approach to program slicing, is universal as long as it is applied to a limited range of programming languages sharing certain common features.

5.2 The prototype

An integral part of this graduation work is the development of an application that is capable of demonstrating the ideas discussed in the theoretical part. The prototype is of utmost importance as it is not possible to derive a formal proof of the correctness of the proposed algorithms and the only way to show their correctness is by example. It is also important because some of the challenges of the relational approach remained hidden or underestimated until it had to be put in practice.

The prototype consists of two parts - an implementation of the core slicing algorithm and a fact extractor. Both of them fully implement the ideas described in this paper. Numerous experiments showed that the implementations are correct. There is just one major flaw in the design of the fact extractor prototype - it is not straightforward to add support of new program constructs which are currently not implemented. The reason is that the abstract syntax entities are not implemented in a modular fashion, but they are tightly connected to each other. I did not manage to come out with a working solution to this problem and it remained as a major item in the future developments list described in the next section.

5.3 Future work

The subject of program slicing is quite vast and a single project is not able to give answers to all the questions that may be raised. On the other hand the relational approach attacks the problem from a new perspective and in most cases it can not make use of the results already obtained in other studies. That is why I had to limit the scope of my graduation work to some of the most important aspects of program slicing and just indicate what else has to be done.

A complete program slicing solution should be able to process constructs used by modern imperative programming languages. One of the major tasks is to add support for object oriented programming. The definition of object

oriented slices has already been given in various resources, such as [5]. In chapter 2 we argued that there will be no need to modify the core relational algorithm to enable object oriented slicing, nor to introduce new types of relations. Only the fact extractor needs to be modified by extending the list of abstract program entities and updating the corresponding algorithms.

Another important amendment is the support of type aware slicing. Currently the fact extraction algorithms neglects variable types which makes it impossible to slice correctly programs making use of advanced programming concept as function overloading [1]. Such a modification will not require fundamental changes of the presented methods and can be easily accommodated by including the type information in the process of variable identification. However there are also programming concepts like variable aliases and pointers which can hardly fit into the existing model. Their implementation will probably require fundamental restructuring of the relational approach to program slicing and has not been considered in this project.

Apart from extending the functionality the project can also be advanced by enhancing the architecture of its implementation. One of the most significant problems that needs to be solved is to loosen the connections between the fact extraction modules and make it possible to modify each of them independently. Thus the project can be easily facilitated to cover a broader range of existing imperative programming languages.

One of the shortcomings of the relational approach to program slicing (and to program analysis in general) is that the relational representation of data is not intuitive and it is hard to comprehend and analyze the intermediate results of the slicing process, such as program facts and the relations generated at various step of the algorithm. It is therefore necessary to develop a number of tools for visualization of relational data in a more understandable format. A visualization tool is also needed to associate the final slice with the source code where it has been extracted from. These supplementary utilities are essential if large programs are going to be sliced because in this case the complexity of information presented in terms of raw relations will be overwhelming.

5.4   Final conclusions

I think this graduation project was successful because it managed to fulfill the main goals that were set initially. The basic idea of the relational approach to program slicing was extended and it was shown it is applicable to a broad range of slicing problems. An efficient solution was developed for the fact extraction challenge without significantly sacrificing the generality of the proposed method. Finally two realistic case studies were conducted which clearly demonstrated how the theoretical ideas work in practice. As a result the relational approach proved to be viable solution to the problem slicing dilemma. I believe the project has accumulated enough knowledge and experience to set the foundation of a further, more sophisticated study of the subject.

---

[1]Function overloading is a programming concept that allows programmers to define two or more functions that differ only by the types of their arguments.

Bibliography

[1] M. Weiser. Program slicing. In Proceeding of the Fifth International Conference on Software Engineering, pages 439-449, May 1981.

[2] H. Agrawal and J. Horgan. Dynamic program slicing. Technical Report SERC-TR-56-P, Purdue University, 1989.

[3] R. Gupta, M.L. Soffa, and J. Horward. Hybrid Slicing: Integrating dynamic information with static analysis. ACM Transactions of Software Engineering Methodology, pages 370-397, 1997.

[4] K. Ottenstein and L. Ottenstein. The program dependence graph in software development environments. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 177-184, May 1984.

[5] D. Liang and M.J. Harrold. Slicing Object Using System Dependence Graph. Proceedings of the International Conference on Software Maintenance, pages 358-367, November 1998.

[6] G. Szilagy, T. Gyimothy, and J. Maluszynski. Slicing of Constraint Logic Programs. Proceedings of the Fourth International Workshop on Automated Debugging, August 2000.

[7] A.V. Aho, R. Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1986.

[8] D. Jackson and E.J. Rollins. A New Model of Program Dependences for Reverse Engineering. Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineer, pages 2-10, 1994.

[9] P. Klint. A Tutorial Introduction to RScript. Centrum voor Wiskunde en Informatica, draft 2005.

[10] S. Horwiz, Th. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 1, pages 26-60, January 1990.

[11] A.M. Sloane, J. Holdsworth. Beyond Traditional Program Slicing. In Proceedings of the International Symposium on Software Testing and Analysis, ACM Press. pages 180-186, January 1996.

[12] M.G.J. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24, 4, pages 334-368, April 2002

[13] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. Computational Complexity, pages 365-370, 2001.

[14] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term Rewriting with Traversal Functions, ACM Transactions on Software Engineering Methodology 12, 2, pages 152-190, April 2003.

[15] Java-front 0.5, Stratego – Strategies for Program Transformation. Released October 2004. `http://www.program-transformation.org/Stratego/JavaFrontRelease05`

Appendix A

Implementation of the relational algorithm in RScript

```
type stm = int
type var = <str, str>
type def = <stm, var>
type use = <stm, var>
type slice_el = <stm, str>
type slice = bag[slice_el]

%%Computer unique elements of a bag, that is a - set
bag[&T] U(bag[&T] V) = {X | &T X : V}
%%Compute transitive closure
rel[&T, &T] closure(rel[&T, &T] set) = {<X, Y> |
    <&T X, &T Y> : set*, X != Y}
%%Compute statement predecessors set
bag[stm] predecessor(rel[stm, stm] P, stm S) = P[-, S]
%%Compute statement successors set
bag[stm] successor(rel[stm, stm] P, stm S) = P[S, -]

rel[stm, def] KILL = {}

%%Compute the variable definitions that are reachable by
%%each statement and may affect its execution
rel[stm, def] reaching-definitions(rel[stm, var] DEFS,
    rel[stm, stm] PRED) = IN
where
    bag[stm] STATEMENT = carrier(PRED)
    rel[stm, def] DEF = {<S, <S, V>> | <stm S, var V> : DEFS}
    equations
        initial
            rel[stm, def] IN init {}
            rel[stm, def] OUT init DEF
        satisfy
            IN = {<S, D> |
                    int S : STATEMENT,
                    stm P : predecessor(PRED, S),
                    def D : OUT[P]}
            OUT = {<S, D> |
```

```
                        int S : STATEMENT,
                        def D : DEF[S] union (IN[S] \ KILL[S])}
    end equations
end where

%%Compute all pairs of a statement dominating another one
rel[stm, stm] dominators(rel[stm, stm] PRED) = DOMINATES
where
    bag[stm] STATEMENTS = U(carrier(PRED))
    bag[stm] ROOT = U(domain(PRED)) \ U(range(PRED))
    rel[stm, stm] DOMINATES = {<S, S1> |
                        stm S : STATEMENTS,
                        stm S1 : STATEMENTS \ U(ROOT union {S}) \
                            range(reachX(ROOT, {S}, PRED))}
end where

%%Backward slicing, the slicing criterion is
%%(statement, variable)
slice BackwardSlice(
            bag[stm] CTRL,
            rel[stm, stm] PRED,
            rel[stm, var] USES,
            rel[stm, var] DEFS
            stm Stat,
            var Var
            ) = OUTPUT
where
    %%Step 1, Compute R1
    KILL = {<S1, <S2, V>> | <stm S1, var V> : DEFS,
                        <stm S2, V> : DEFS, S1 != S2}
    %%Step 2, Compute R2
    rel[stm, def] REACH = reaching-definitions(DEFS, PRED)
    %%Step 3, Compute R3
    rel[use, def] use-def = {<<S1, V>, < S2, V>> |
        <stm S1, var V> : USES, <stm S2, V> : REACH[S1]}
    %%Step 4, Compute R4
    rel[def, use] def-use-per-stm =
                    {<<S, V1>, <S, V2>> | <stm S, var V1> : DEFS,
                                <S, var V2> : USES}
                    union
                    {<<S, V>, <S, <"EXEC", "EXEC">>> |
                            <stm S, var V> : DEFS}
                    union
                    {<<S, <"TEST", "TEST">>, <S, V>> |
                            stm S : CTRL,
                        <S, var V> : domainR(USES, {S})
                    }
    %%Step 5, Compute R5
    rel[stm, stm] dom = dominators(PRED)
    %%Step 6, Compute R6
```

```
    rel[stm, stm] CONTROL-DOMINATOR = domainR(dominators(PRED),
            CTRL)
    %%Step 7, Compute R7
    rel[def, use] control-dependence = { <<S2, <"EXEC", "EXEC">>,
        <S1, <"TEST", "TEST">>> | <stm S1, stm S2> :
        CONTROL-DOMINATOR}
    %%Step 8, Compute R8
    rel[use, def] use-control-def = use-def union
            control-dependence
    %%Step 9, Compute R9
    rel[use, use] USE-USE =
        closure(use-control-def o def-use-per-stm)
    bag[stm] stms = U(carrier(PRED))
    %%Final Step, Compute Sl(s, v)
    slice OUTPUT = {<S, VSTR> | <Stat, Var> : USES,
        use V : USE-USE[<Stat, Var>],
        stm S : stms, S == first(V),
    str VSTR <- second(second(V))}
end where


%%Compute backward slice starting from any variable in
%%a given statement
slice BackwardSliceStm(
            bag[stm] CTRL,
            rel[stm, stm] PRED,
            rel[stm, var] USES,
            rel[stm, var] DEFS,
            stm Stat
            ) = OUTPUT
where
    slice OUTPUT = { <S, VSTR> | <Stat, var V> : USES,
    <stm S, str VSTR> :
    BackwardSlice(CTRL, PRED, USES, DEFS, Stat, V)}
end where


%%Computer all backward slices in a program
bag[<stm, slice>] BackwardSliceAll(
            bag[stm] CTRL,
            rel[stm, stm] PRED,
            rel[stm, var] USES,
            rel[stm, var] DEFS
            ) = OUTPUT
where
    bag[<stm, slice>] OUTPUT = {<S, Slice> |
        stm S : U(carrier(PRED)), slice Slice <-
            BackwardSliceStm(CTRL, PRED, USES, DEFS, S)}

end where

%%Statement predecessors
```

```
rel[stm, stm] PREDS
%%Variable defitions
rel[stm, var] DEFS
%%Variable uses
rel[stm, var] USES
%%List of control statments
bag[stm] CTRLS

%%test
int Stat = 8
slice SLICE = BackwardSliceStm(CTRLS, PREDS, USES,
    DEFS, Stat)

%%Get all slices
bag[<stm, slice>] SLICES = BackwardSliceAll(CTRLS, PREDS, USES,
    DEFS)
```
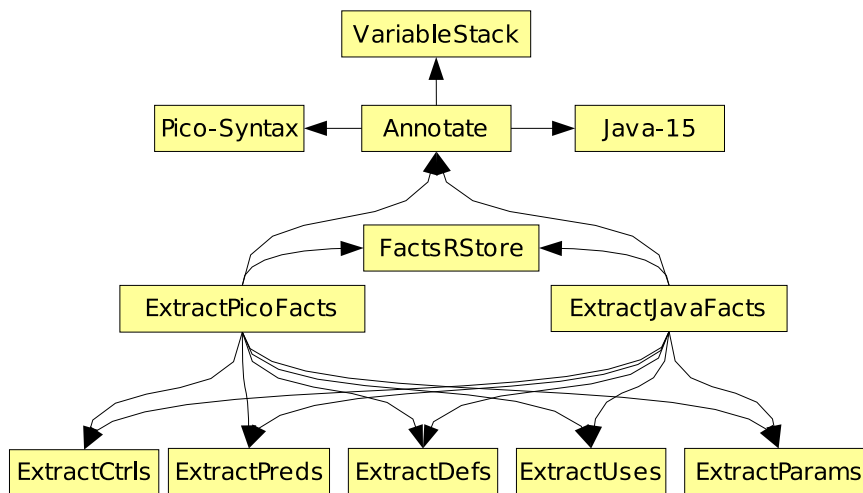
Appendix B

Fact extractor ASF+SDF speficiation modules

Imported ASD+SDF graph

Standard modules which are distributed with MetaEnvironment have been excluded from the graph.

Generalized fact extraction modules

Module	Description
VariableStack.sdf/asf	Implements a variable stack used to resolve naming conflicts when annotating variables
Annotate.sdf/asf	Implements the annotations if statements, variables and functions
ExtractDefs.sdf/asf	Extracts variable definitions
ExtractUses.sdf/asf	Extracts variable uses
ExtractPreds.sdf/asf	Extracts predecessor relations
ExtractCtrls.sdf/asf	Extract a list of control flow statements
FactsRStore.sdf/asf	Auxiliary module that is used to store extracted relations as RScript data.

Language specific modules

Module	Description
Pico-Syntax.sdf	Pico syntax
Java-15.sdf	Java syntax
ExtractPicoFacts.sdf	Mapping of Pico syntax to the abstract syntax entities
ExtractJavaFacts.sdf	Mapping of Java syntax to the abstract syntax entities

Appendix C

Pico grammar and its mapping to the abstract syntax entities

Pico grammar

```
module languages/pico/syntax/Pico-Syntax

imports languages/pico/syntax/Pico-Identifiers
imports languages/pico/syntax/Pico-Types
imports basic/Integers
imports basic/Strings

hiddens
context-free start-symbols PROGRAM

exports

sorts PROGRAM DECLS ID-TYPE STATEMENT EXP BEGIN END VARIABLE
      BLOCK LHS FUNCTION FPARAMS FPARAM FCALL FARGS FARG FUNC-ID
      FSTART RETURN SERIES IFSTM IFBLOCK ELSEBLOCK FORSTATEMENT
      FORBLOCK DOWHILESTATEMENT DOWHILEBLOCK WHILEDOSTATEMENT
      WHILEDOBLOCK BREAK
      PICO-ID STMDEL

lexical syntax
[a-z][a-z0-9]* -> PICO-ID
";" -> STMDEL

context-free syntax

"begin" -> BEGIN
"end" -> END
{FUNCTION ";"}* BLOCK -> PROGRAM
BEGIN DECLS SERIES END -> BLOCK
STATEMENT? -> SERIES
STATEMENT STMDEL SERIES -> SERIES
"declare" {ID-TYPE "," }*";" -> DECLS
VARIABLE ":" TYPE -> ID-TYPE
LHS ":=" EXP  -> STATEMENT
```

```
VARIABLE -> LHS
"if" EXP "then" IFBLOCK ELSEBLOCK? -> IFSTM
IFSTM -> STATEMENT
BLOCK -> IFBLOCK
"else" BLOCK -> ELSEBLOCK
"for" PICO-ID ":=" EXP "to" EXP "do" FORBLOCK -> FORSTATEMENT
BLOCK -> FORBLOCK
FORSTATEMENT -> STATEMENT
"do" DOWHILEBLOCK "while" EXP -> DOWHILESTATEMENT
DOWHILESTATEMENT -> STATEMENT
BLOCK -> DOWHILEBLOCK
"while" EXP "do" WHILEDOBLOCK -> WHILEDOSTATEMENT
WHILEDOSTATEMENT -> STATEMENT
BLOCK -> WHILEDOBLOCK
BLOCK  -> STATEMENT
FCALL -> STATEMENT
RETURN -> STATEMENT
BREAK -> STATEMENT
"return" EXP? -> RETURN
"break" NatCon? -> BREAK
"function" FUNC-ID "(" FPARAMS ")" "begin" -> FSTART
FSTART -> BEGIN
BEGIN DECLS SERIES END -> FUNCTION {prefer}
PICO-ID -> FUNC-ID
{FPARAM ","}* -> FPARAMS
ID-TYPE -> FPARAM
FUNC-ID "(" FARGS ")" -> FCALL
{FARG ","}* -> FARGS
EXP -> FARG
"" PICO-ID -> VARIABLE
FCALL     -> EXP
VARIABLE      -> EXP
NatCon        -> EXP
StrCon        -> EXP
EXP "+" EXP   -> EXP {left}
EXP "-" EXP   -> EXP {left}
EXP "||" EXP  -> EXP {left}
"(" EXP ")"   -> EXP {bracket}
context-free priorities
EXP "||" EXP -> EXP >
EXP "-" EXP -> EXP >
EXP "+" EXP -> EXP
```

Mapping of Pico grammar to the abstract syntax

Abstract Syntax	Pico Syntax
Program	PROGRAM
Begin Scope	BEGIN
End Scope	END
Variable	VARIABLE
Variable Id	PICO-ID
Variable Use	EXP
Variable Definition	LHS
Function Id	FUNC-ID
Function Definition	FUNCTION
Function Parameter	FPARAM
Function Call	FCALL
Function Argument	FARG
Statement	STATEMENT
Return Statement	RETURN
If Statement	IFSTM
If Block	IFBLOCK
Else Block	ELSEBLOCK
Do While Statement	DOWHILESTATEMENT
Do While Block	DOWHILEBLOCK
For Statement	FORSTATEMENT
For Block	FORBLOCK
While Do Statement	WHILEDOSTATEMENT
While Do Block	WHILEDOBLOCK

Appendix D

Java slicing example

Java source code

```
class BubbleSort
{

static private boolean compare(int x, int y)//#3, #4
{
        return x > y;#5
}

static public void main(int[] a, int n)//#7, #8
{
        int i;
        int j;
        int t;

        i = 0;#12

        while(i < n)//#13
        {
                j = i;#15
                while (j < n)//#16
                {
                        if (compare(a[i], a[j]))//#18
                        {
                                t = a[j];//#20
                                a[j] = a[i];//#21
                                a[i] = t;//#22
                        }
                        j = j + 1;//#23
                }
                System.out.println(a[i]);//#24
                i = i + 1;//#25
        }
}
```

```
}
```

Extracted relations

```
USES = {<25, <"6", "i"> >, <24, <"6", "i"> >, <24, <"6", "a"> >,
<24, <"System.out.println", "1">>,<23, <"6", "j"> >,
<22, <"6", "t"> >,<22, <"6", "i"> >,<21, <"6", "i"> >,
<21, <"6", "a"> >,<21, <"6", "j"> >,<20, <"6", "j"> >,
<20, <"6", "a"> >,<18, <"6", "j"> >, <18, <"6", "a"> >,
<18, <"6", "i"> >, <18, <"compare", "2">>, <16, <"6", "n"> >,
<16, <"6", "j"> >, <15, <"6", "i"> >,<13, <"6", "n"> >,
<13, <"6", "i"> >, <5, <"2", "y"> >, <5, <"2", "x"> >,
<4, <"6", "a"> >,<4, <"6", "j"> >,<3, <"6", "a"> >,
<3, <"6", "i"> >}


DEFS = {<25, <"6", "i"> >,<23, <"6", "j"> >,<22, <"6", "a"> >,
<21, <"6", "a"> >,<20, <"6", "t"> >,<15, <"6", "j"> >,
<12, <"6", "i"> >,<8, <"6", "n">>,<7, <"6", "a">>,
<5, <"compare", "2"> >,<4, <"2", "y">>,<3, <"2", "x">>}


PREDS = {<7, 8>,<3, 4>,<4, 5>,<8, 9>,<9, 10>,<10, 11>,
<11, 12>,<12, 13>,<13, 15>,<15, 16>,<16, 18>,<18, 20>,
<20, 21>,<21, 22>,<5, 22>,<5, 18>,<18, 3>,<22, 23>,
<18, 23>,<23, 16>,<16, 24>, <23, 24>,<24, 25>, <25, 13>,
<13, 13>, <13, 25>, <25, 25>}


CTRLS = {18,16,13}
```

Slices

Sl(25, i)

```
static public void main(int[] a, int n)//#7, #8
{
        i = 0;#12

        while(i < n)//#13
        {
                i = i + 1;//#25
        }
}
```

Sl(5, x)

```
class BubbleSort
{

static private boolean compare(int x, int y)//#3, #4
{
        return x > y;#5
```

46

```
}

static public void main(int[] a, int n)//#7, #8
{


        i = 0;#12

        while(i < n)//#13
        {
                j = i;#15
                while (j < n)//#16
                {
                        if (compare(a[i], a[j]))//#18
                        {
                                t = a[j];//#20

                                a[i] = t;//#22
                        }
                        j = j + 1;//#23
                }

                i = i + 1;//#25
        }
}
```

An interesting effect in the last example is that statement #21 is excluded from the slice Sl(5, x). This is not correct as setting any value of the array can contribute to the value of x at statement #5. The reason is that in the current fact extraction prototype arrays are treated as single variables and the assignment #22 'kills' #21.