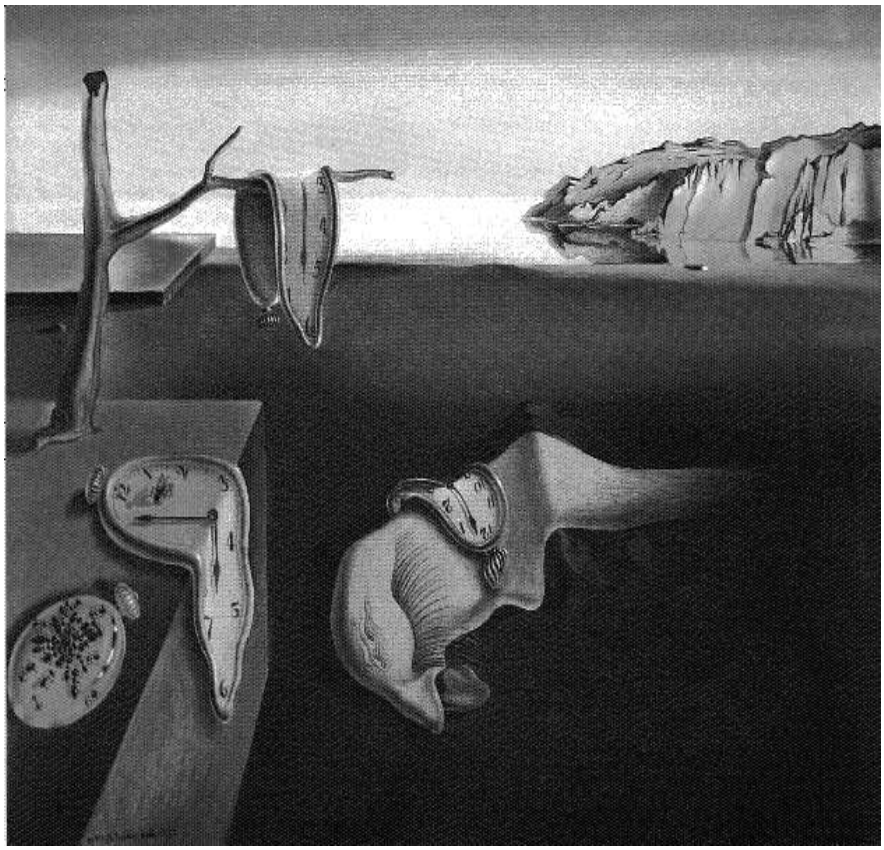
Institut für Automation
Abt. für Automatisierungssysteme

Technische Universität Wien

Projektbericht Nr. 183/1-138
June 2005

Symbolic Evaluation of Imperative Programming Languages

Bernd Burgstaller



Salvador Dali, "Die Beständigkeit der Erinnerung"

DISSERTATION

Symbolic Evaluation of Imperative Programming Languages

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

Ao.Univ.Prof. Dr. Johann Blieberger

Inst.-Nr. E183/1
Institut für Rechnergestützte Automation,
Arbeitsgruppe Automatisierungssysteme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Bernd Burgstaller

Matr.-Nr. 8925663
Mitterberg 11
8665 Langenwang

Wien, im Februar 2005 _____

In Erinnerung an meine Mutter Renate Burgstaller

Für meinen Vater Raimund Burgstaller

Zusammenfassung

Symbolische Analyse ist eine statische Programmanalysemethode, die Daten- und Kontrollflussinformation an wohldefinierten Programmpunkten beschreibt. Information dieser Art ist von großer Bedeutung für Test und Verifikation von Programmen, sowie für Laufzeitabschätzungen und Programmparallelisierung. Weiters ist symbolische Analyse für optimierende Compiler sowie für Codegeneratoren von Bedeutung.

Gegenstand dieser Dissertation ist ein neuer Ansatz in der symbolischen Analyse, der auf Pfadausdrücken beruht. Pfadausdrücke werden bei diesem Ansatz dazu verwendet, die Kontrollflusseigenschaften eines Programms zu erfassen. Einen wesentlichen Teil der Arbeit stellt jene algebraische Struktur dar, in der die symbolische Analyse stattfindet. Wir beschreiben Syntax und Semantik einer einfachen Turing-äquivalenten Fluss-Sprache, die uns als Basis für die Definition dieser Struktur dient. In ihrem Mittelpunkt steht der Superkontext, mit dessen Hilfe wir die möglichen Variablenbindungen an einem wohldefinierten Programmpunkt beschreiben.

Um den Seiteneffekt eines Eingabeprogramms zu beschreiben, bilden wir seine einzelnen Programmanweisungen auf Funktionen ab, die ihrerseits einen Superkontext auf einen Superkontext abbilden. Pfadausdrücke werden sodann im Kontext dieser Funktionen interpretiert, womit wir eine funktionale Beschreibung des Eingabeprogramms erhalten. Ein Korrektheitsbeweis sichert die Richtigkeit dieser funktionalen Beschreibung im Hinblick auf die konkrete Semantik der Fluss-Sprache ab.

Die beschriebene Methode der symbolischen Analyse ist weniger komplex als existierende Methoden, da sie im Wesentlichen aus der Anwendung der funktionalen Programmbeschreibung auf einen Superkontext besteht. Das Ergebnis dieser Funktionsanwendung ist wiederum ein Superkontext, der die Variablenbindungen nach Ausführung des jeweiligen Eingabeprogramms beschreibt. Die beschriebene Methode kann weiters Lösungen für beliebige Programmpunkte reduzibler sowie irreduzibler Kontrollflussgraphen berechnen und ist damit mächtiger als existierende Methoden.

Abstract

Symbolic analysis is a static program analysis technique that captures data and control flow information at well-defined program points. This information is useful in the areas of program verification, program testing, worst case execution time analysis, and parallelization. Optimizing compilers and code generators can also benefit from the type of information provided by symbolic analysis.

In this thesis we take a novel approach to symbolic analysis that is based on path expressions. Path expressions allow us to capture the control flow information that is inherent in a program. By reinterpreting path expressions we obtain a mapping from the path expression algebra into the symbolic analysis domain. A major topic of this thesis is therefore the symbolic analysis domain itself. We describe syntax and semantics of a simple yet Turing-equivalent flow language which serves as the basis for the definition of this domain. At the heart of it is the supercontext, an algebraic structure capable of describing all possible variable bindings valid at a well-defined program point.

To describe the computational effects of the input program, we map single statements to functions from supercontexts to supercontexts. The reinterpretation of path expressions takes place in the context of these functions, thus obtaining a functional description of the input program. We develop a proof that establishes the correctness of these functional descriptions with respect to the concrete semantics of the flow language.

Our method is less complex than existing methods in the sense that it reduces to the application of the functional description to a given supercontext, yielding a supercontext describing the variable bindings valid after execution of the corresponding input program. It is more general than existing methods because it can derive solutions for arbitrary graph nodes of reducible and irreducible flow graphs.

Acknowledgments

Vielmehr bestand das Unersetzliche, und beim E. P. besonders,
in einer dem Begriff der Freundschaft eher sogar entgegengesetzten Kompromisslosigkeit,
in einer unerbittlich dem Gegenstand (und nicht der Person) zugewandten Insistenz,
die sich von Maßstab und Forderung nichts abhandeln ließ, komme wer da wolle.

— Friedrich Torberg, "Kaffeehaus war überall", Briefwechsel mit Käuzen und Originalen.

I sincerely thank my advisor, Prof. Johann Blieberger, for his continuous guidance and support. It will forever remain a mystery to me, how one can immediately sense a problem in the work of somebody else, sometimes even at a distance. I greatly admire his deep understanding of mathematics and computer science, and his lightning-like flashes of ingenuity.

I am deeply indebted to Prof. Bernhard Scholz, who co-advised me during my work on this thesis. The discussions with him always give me deeper insights, and his broad and in-depth views on computer science are intriguing.

I thank Prof. Bernhard Gramlich for his advice and for his valuable comments and suggestions when grading this thesis. His lectures on term rewrite systems were inspiring and helped me to gain a new perspective on symbolic analysis.

I appreciate the help of Prof. Schildt, who, during my time as assistant professor at the department, granted me the time to start work on this thesis.

I thank Heinz Deinhart for solving all my system administration issues in a perfect way, and for his forgiveness towards a user that constantly demanded more resources for his experiments.

This work would not have been possible without the support of my family.

Contents

List of Symbols

Sets and Functions

$\mathbb{N}$	natural numbers	9
$\mathbb{Z}$	integers	9
$\mathbb{B}$	truth values	9
$\mathbb{V}$	set of program variables	11
$\underline{\mathbb{V}}$	set of initial value variables	40
$\mathbb{L}$	set of loop index variables	82
IV	set of induction variables	79
$\mathbb{Z}[\mathbf{x}]$	multivariate polynomials in $\mathbb{Z}$	40
$Q(\mathbb{Z}[\mathbf{x}])$	quotient field of multivariate polynomials	41
$f : R \to S$	function definition	9
$f^{(n)}$	function of arity n	42
$f[y \mapsto v]$	updated function definition	10
$\mathcal{D}om(f)$	function domain	9
$\circ$	function composition	10
f^k	iterated function composition	53
ι	identity function	53
graph(f)	graph of a function	10

Grammars and Graphs

(V_N, V_T, P, S)	context-free grammar	11
Tree_G	set of derivation trees	12
$L(G)$	language defined by grammar G	12
$[\![\ ]\!]$	emphatic brackets	16
$\langle N, E\rangle$	simple graph	17
$\langle N, E, n_e, n_x\rangle$	control flow graph	17
$\langle S, E, s_e, s_x\rangle$	Flow control flow graph	19
N	node set of a graph	17
E	edge set of a graph	17
n_e	entry node	17
n_x	exit node	17
h(e)	edge head	17
t(e)	edge tail	17
in(n)	set of incoming edges	17

$\mathrm{out}(n)$	set of outgoing edges	17
$\mathrm{succ}(x)$	set of successor nodes	17
$\mathrm{pred}(x)$	set of predecessor nodes	17
π	graph path	17

Standard Semantics

$env \in Environment$	environments	19
$s \in S$	states	19
$pred$	branch predicate	20
σ	side-effect (but cf. also p. 58)	20
δ	transition function	20
δ^*	iterated transition function	21
$\mathrm{e} : \mathrm{pred} \Rightarrow \mathrm{assign}$	association of predicate and side-effect with e	26
$(Q, \Sigma, \Gamma, \delta_{\mathrm{TM}}, q_0, B, F)$	Turing machine	30
δ_{TM}	Turing machine transition function	30
$e : s_1 \Rightarrow s_2; D$	Turing machine transition diagram edge	32
env_{TM}	emulation environment	32
c	emulation counter	32
T	emulation tape	32

Symbolic Semantics

$S_{\mathrm{con}}[\![P]\!]$	standard semantic program denotation	37
$S_{\mathrm{sym}}[\![P]\!]$	symbolic semantic program denotation	37
$\mathrm{e}(v_1, \ldots, v_n)$	symbolic expression	39
$e(\mathbf{v})$	symbolic expression, short form	39
$\_$	initial value operator	40
Rnd	rounding operation	41
SymExpr	symbolic expression domain	41
SymPred	symbolic predicate domain	46
$\mathrm{div_s}$	symbolic division operator	42
$\mathrm{rem_s}$	symbolic remainder operator	43
$s \in S$	program states	49
$c \in C$	program contexts	49
$[s, p]$	program context, verbose notation	49
$\overline{c}$	closure context	84
$[s, p, rss]$	closure context, verbose notation	84
p	path condition	49
pc	pathcondition extraction	49
st	state extraction	49
$pred_s$	symbolic branch predicate (cf. also p. 58)	53
$pred_c$	standard semantic branch predicate	58
σ_{s}	symbolic side-effect (cf. also p. 58)	53
σ_{c}	standard semantic side-effect	57
σ	substitution (but cf. also p. 20)	58
σ_{env}	initial variable substitution from env	59

Path Expressions and Symbolic Evaluation

List of Figures

List of Tables

Chapter 1

Introduction

The only mental tool by means of which a very finite
piece of reasoning can cover a myriad of cases is called "abstraction".
— Edsger W. Dijkstra, in "The Humble Programmer", 1972 Turing Award Lecture

This dissertation is about static program analysis using a novel approach to symbolic evaluation based on path expressions. Path expressions allow us to capture the control flow information inherent in a program. By reinterpreting the operators of the path expression algebra we obtain a mapping from path expressions into the symbolic analysis domain. The symbolic analysis domain is also an algebra that we will explore in the course of the thesis. The main constituent of this domain is the supercontext, which captures (i.e., abstracts from) all possible variable bindings at a given program point. Our method is simpler and more general (e.g., can derive a solution for arbitrary graph nodes of reducible and irreducible flow graphs) than existing methods.

We begin the dissertation by introducing the concept of symbolic analysis in Section 1.1. In Section 1.2 we briefly introduce path expressions and the main ingredients of our symbolic analysis domain. A more thorough presentation of these topics follows in Chapters 4 and 5. In Section 1.3 we highlight the contributions of the dissertation. In Section 1.4 we finally present the overall organization of the dissertation.

1.1 Symbolic Analysis

Symbolic analysis of programs is a static program analysis technique that captures data and control flow information at well defined program points. This information is useful in the areas of program verification, program testing, worst case execution time analysis, and parallelization. Optimizing compilers and code generators can also benefit from the type of information provided by symbolic analysis.

As an introductory example, consider the simple statement sequence of Figure 1.1, written in an arbitrary imperative programming language. In line 1 two scalar variables u and v are declared. The read statement in line 2 assigns both variables a new value. Within the statement sequence from line 3 to line 5 we change values of the variables u and v by a sequence of assignment statements.

```
1  integer::u,v;
2  read (u,v);
3  u := u + v;
4  v := u - v;
5  u := u - v;
```

Figure 1.1: Simple Statement Sequence

Although the input values for variables u and v are not available at compile time, we want to determine the effect of the computation, that is, the values of u and v at the end of line 5.

With symbolic analysis we proceed as follows: instead of computing with numbers for variables u and v, we assign them *symbolic values*. Let us assume that the read statement in line 2 yields the symbolic value $\underline{u}$ for variable u, and $\underline{v}$ for variable v. Hence our symbolic computation after the read statement assumes the following variable bindings.

$$\begin{aligned} u &= \underline{u} \\ v &= \underline{v} \end{aligned} \tag{1.1}$$

Resuming our computation at the assignment statement in line 3, we assign variable u the values of u plus v. Variable v is not affected by this assignment. The new variable binding after line 3 is as follows

$$\begin{aligned} u &= \underline{u} + \underline{v} \\ v &= \underline{v} \end{aligned} \tag{1.2}$$

Continuing our symbolic computation for the remaining assignments, we get the variable bindings depicted in Equation (1.3) at the end of line 5.

$$\begin{aligned} u &= (\underline{u} + \underline{v}) - (\underline{u} + \underline{v} - \underline{v}) \\ v &= \underline{u} + \underline{v} - \underline{v} \end{aligned} \tag{1.3}$$

Given the fact that the above symbolic expressions are built from variables u and v, which represent scalar values, we can apply the operations and identities valid in the ring of multivariate polynomials to achieve a simplification, arriving at the following variable bindings.

$$\begin{aligned} u &= \underline{v} \\ v &= \underline{u} \end{aligned} \tag{1.4}$$

The above simplification is obvious in a purely mathematical sense, due to the equivalence of expressions. However, for a computer implementation (e.g., with the help of a computer algebra system) the notion of expression equivalence is a concept that explicitly has to be taken care of. For this reason we will maintain a strict distinction between "abstract" mathematical objects and their projections on a "real" computer throughout this thesis.

Comparing Equations (1.1) and Equation (1.4) it becomes evident that the effect of the computation depicted in Figure 1.1 is to swap the values of variable u and v. Due to the symbolic nature of the analysis this property is independent

of possible concrete input values for u and v. By applying symbolic analysis, namely *forward substitution* and *simplification* techniques we have revealed the program semantics of the example. As already indicated in the introduction, this information is valuable in many areas, e.g., for program optimization. For instance, a code generator could decide from the variable bindings of Equation (1.4), that an overflow check for the expression u − v in lines 4 and 5 is redundant and can be omitted. As we have pointed out in [BB03], these facts cannot be derived by current state-of-the-art compiler technology.

```
1  integer::u;
2  read (u);
3  while u < 100 loop
4     u := 3*u + 1;
5  end loop;
```

Figure 1.2: Example Loop

A more elaborate example program containing a loop is depicted in Figure 1.2. A loop may imply an infinite number of iterations, and in general we do not know the number of iterations at compile time. Figure 1.3 lists the first

Iteration	Symbolic Value of u
0	$\underline{\mathrm{u}}$
1	$3 \cdot \underline{\mathrm{u}} + 1$
2	$9 \cdot \underline{\mathrm{u}} + 4$
3	$27 \cdot \underline{\mathrm{u}} + 13$
⋮	⋮

Figure 1.3: Sequence of Symbolic Values for Variable u

few values that variable u will assume during subsequent iterations of the loop. Therein iteration 0 denotes the value of u before entering the loop. This value is of course due to the read statement in line 2. Iteration 1 denotes the value of variable u after the first iteration of the loop, iteration 2 the value after the second iteration, and so on, ad infinitum. Hence, if we are interested in the symbolic value of variable u after the assignment in line 4, we have to consider infinitely many variable bindings*.

A necessary step in order to handle infinitely many variable bindings is to find a finite representation for the infinite sequence of symbolic values partially listed in Figure 1.3. For this purpose we employ *recurrences* [Ros95], which consist of a boundary condition and a recurrence relation. By that means the sequence from Figure 1.3 can be written as

$$\mathrm{u}(0) = \underline{\mathrm{u}} \tag{1.5}$$

$$\mathrm{u}(i+1) = 3 \cdot \mathrm{u}(i) + 1. \tag{1.6}$$

The literature contains a wealth of methods that can be applied to obtain closed forms for recurrence relations. A closed form is a formula the value of

*On a contemporary hardware platform the number of possible variable bindings will of course reduce to $\approx 2^{32}$.

which depends on the iteration number (i in the above case), and not on variable values of previous iterations. In this way a closed form is a representation that is as compact as "ordinary" symbolic expressions, which is not true for the recurrence relation itself. Nevertheless, deriving closed forms for recurrence relations is undecidable in the general case, and our analysis method must therefore provide a means of approximation in order to derive a less precise (yet computable) result in such a case. The goal in that respect is of course to stay as precise as possible, which is supported by the fact that approximations can be introduced on a per variable basis.

The following equation depicts the closed form for the recurrence relation of Equation (1.6).

$$\mathrm{u}(i) = 3^i \cdot \underline{\mathrm{u}} + \frac{(3^i - 1)}{2}, \quad \text{for } i \geq 0 \tag{1.7}$$

It describes the value of the expression $3 * \mathrm{u} + 1$ in line 4 of Figure 1.2 after $i \geq 0$ iterations of the loop body. In this way we have obtained a finite representation for the infinitely many variable bindings possible for variable u after the end of line 4.

Coming back to the notion of range checks, if we want to determine whether the expression $3 * \mathrm{u} + 1$ is within a predetermined range $[l, u]$, the righthand-side of Equation (1.7) holds the key to deciding this question.

Since computers employ integer arithmetic operations, the solutions of a symbolic equation are in effect the solutions of a Diophantine equation. Diophantine equations are also undecidable in the general case, which gives us another potential cause of approximation.

It should however by noted that undecidability of a problem cannot be attributed to symbolic analysis per se, for any other analysis method attempting a problem of a given complexity will experience the same inadequacy.

A problem that might be attributed to symbolic analysis is the combinatorial explosion resulting from complicated control flow (e.g., a huge number of `if` statements, `exits` from loops, or undisciplined use of `goto` statements). In order to gain empirical data on the problem size of actual programs we have conducted a comprehensive survey that yielded reassuring results (cf. Section 1.3).

In concluding it should be stated that symbolic analysis is a static program analysis method of impressive power and flexibility, that can be applied to a variety of problems. By its nature, symbolic analysis is however also an expensive method and should be applied with great care. It would certainly be inappropriate to apply it to problems that can already be solved by conventional techniques. However, there exist many applications, e.g., in the area of safety related systems, where conventional techniques fail, and where the potential gain due to sophisticated program analysis is already justified.

1.2 Path Expressions and the Symbolic Analysis Domain

Existing work on symbolic evaluation captures the control flow information of a program from its control flow graph (CFG). A CFG is a directed labeled graph where the nodes correspond to the basic blocks of a program, and edges represent transfer of control between them. As an example, consider Figure 1.4,

which depicts a program and its associated control flow graph. To emphasize the correspondence of program statements and flow graph nodes, we have added the CFG node names as a comment to the respective statements.

Our approach takes a different route. At the heart of our work on symbolic analysis are *program paths*. In brief, we denote by a program path a sequence of

```
1   integer::u,i,j;
2   read (u,i,j);              -- node n1
3   if u < 10 then             -- node n1
4     repeat
5       if i < j then          -- node n2
6           i := i + 1;        -- node n3
7       else
8           i := i + 3;        -- node n4
9       end if;
10    until i > 100;           -- node n5
11  end if;
```

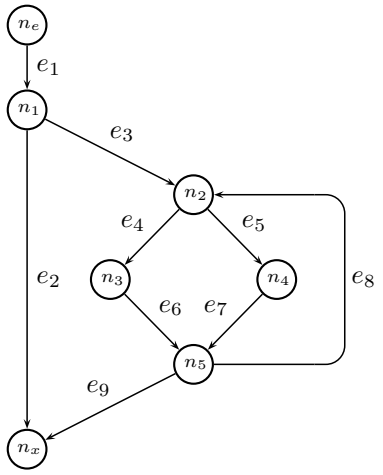


Figure 1.4: Example Program with Control Flow Graph

adjacent CFG edges, which we connect by the "$\cdot$" operator (a formal treatment is given in Chapter 5). As an example, we list a few program paths from node n_e to node n_3 for the CFG in Figure 1.4.

$$
\begin{array}{l}
e_1 \cdot e_3 \cdot e_4 \\
e_1 \cdot e_3 \cdot e_4 \cdot e_6 \cdot e_8 \cdot e_4 \\
e_1 \cdot e_3 \cdot e_5 \cdot e_7 \cdot e_8 \cdot e_4 \\
e_1 \cdot e_3 \cdot e_5 \cdot e_7 \cdot e_8 \cdot e_4 \cdot e_6 \cdot e_8 \cdot e_4 \\
\quad \vdots \qquad\qquad\qquad\quad \vdots
\end{array}
$$

We quickly run out of steam with an enumeration, given the fact that there exists an infinite number of program paths from node n_e to node n_3.

What we need is a finite representation for this infinite number of program paths. Robert E. Tarjan has shown in [Tar81] how path problems on directed graphs can be solved via *path expressions*. A path expression of type (v, w) is a regular expression the language of which consists of the set of program paths from node n_v to node n_w. Given the concept of path expressions, our work consists of the following two steps.

(1) Associate program constructs with CFG edges, as they are the "building blocks" of path expressions.

(2) Provide a mapping from path expressions into the symbolic domain.

A prerequisite to step (2) is of course the provision of an appropriate symbolic analysis domain which can capture the possibly infinite set of different variable bindings at a given program point. Moreover, for each variable binding

it must encode the condition, under which this variable binding is valid. Finally, all operations in the symbolic domain must have the same properties as their counterparts in the concrete domain (e.g., symbolic integer division has to "mimic" the division on integers). The main ingredients of our symbolic analysis domain are

- an algebra for integer-valued symbolic expressions,
- an algebra for symbolic predicates,
- a structure called supercontext to capture the variable bindings at a given program point.

Comparing our approach based on path expressions with CFG-based methods, it must be said that under certain idealized conditions (i.e. in the presence of a data-flow framework with insertion- and loopbreaking rules, driven by an elimination algorithm [Bli02]) a CFG-based approach is equivalent to our approach. However, for all other cases our approach is more general (e.g., it is equally well-suited for irreducible CFGs). Furthermore, path expressions provide a convenient abstraction from the intricacies of complex CFGs, and they constitute an algebra, a fact that grants a smooth interaction with the algebras of the symbolic analysis domain.

1.3 Contributions

In this section we outline the major contributions of this dissertation.

Contribution 1 We define syntax and semantics of a Turing-equivalent flow language that models the standard semantics of program execution. This language serves us as a vehicle to develop the symbolic analysis methodology. Due to its Turing-equivalence we can map arbitrary programming language constructs to this flow language, which provides the possibility of further extensions, without having to change the core of our methodology for symbolic analysis.

Contribution 2 We define a symbolic domain that parallels the concrete domain of integer arithmetic used by our flow language. This symbolic domain provides notions for symbolic expressions, predicates, and structures to capture and manipulate the set of variable bindings valid at a given program point.

Contribution 3 We model symbolic execution along program paths and prove its correctness with respect to the concrete semantics of our flow language.

Contribution 4 We define a mapping from path expressions into the symbolic domain and prove its correctness with respect to the already established correctness of symbolic execution along program paths. This mapping constitutes a mechanism to automatically derive the set of all possible variable bindings at a given program point, and it automatically derives all involved recurrence relations.

Contribution 5 We set up a data-flow problem that computes path expressions from arbitrary (i.e., reducible and irreducible) control flow graphs.

Contribution 6 We define metrics on path expressions in order to asses the required analysis effort. We prove the minimality of the generated path expressions (cf. Contribution (5)) with respect to a given metric.

Contribution 7 We conduct an extensive study to gain empirical data regarding the actual metric figures that can be expected for symbolic analysis of contemporary real-world applications.

Contribution 8 We maintain a strict distinction between "abstract" mathematical concepts and their projections on a "real" computer. While the theory of symbolic analysis is purely mathematical, we also outline the issues arising in the context of an actual implementation.

1.4 Organization of the Dissertation

In Chapter 2 we present background material and notations that are needed for the work described in this thesis. In Chapter 3 we define syntax and semantics of a Turing-equivalent flow language that models the standard semantics of program execution. This language serves us as a vehicle to develop the symbolic analysis methodology. In Chapter 4 we develop the symbolic analysis domain and the notion of symbolic *execution* along whole program paths. Chapter 5 is denoted to symbolic *evaluation* of programs. It introduces path expressions and the mapping from path expressions into the symbolic analysis domain, and treats loops, induction variables, and recurrence relations. Furthermore it outlines issues arising in the context of an actual implementation. Chapter 6 contains our path expression generation method, metrics on path expressions, and the results of our study conducted on contemporary real-world applications. In Chapter 7 we discuss related work, in Chapter 8 we draw our conclusions and outline future work.

Chapter 2

Background and Notation

Be patient, for the world is broad and wide.
— Edwin A. Abbott, *Flatland: A Romance of Many Dimensions* (1884)

The purpose of this section is threefold: first of all, it shall define the terms and notational conventions of methodologies that this work is based upon. This goal can hardly be achieved without introducing the concepts itself (goal two). Due to practical reasons such an introduction can neither be in-depth nor self-contained. The third goal of this section is therefore to point the interested reader towards the literature — in an attempt to make the introduction in-depth, and, by transitive closure, self-contained.

2.1 Sets and Functions

The following sets are often used throughout this work:

1. Natural Numbers in the sense of the axiomatization due to Peano (cf. e.g., [Jac74]): $\mathbb{N} = \{0, 1, 2, \ldots\}$.
2. Integers (cf. [Jac74]): $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
3. Truth values (Booleans): $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$.

Another set that we will often use is the set of program variables ($\mathbb{V}$), which we will define subsequent to the introduction of functions.

For two sets R and S, f is a *function* from R to S, written $f : R \to S$, if, to each element of R, f associates exactly one member of S. The expression $R \to S$ is called the *arity* or *functionality* of f. R is the *domain* of f, denoted by $\mathcal{D}om(f)$, and S is the *codomain* of f. Calculating a result y by presenting an argument a to f is called *application* and is denoted by $y = f(a)$.

A function f is said to be *one-to-one*, or *injective*, iff $f(x) = f(y)$ implies that $x = y$ for all x and y in the domain of f. A function f from A to B is called *onto*, or *surjective*, iff for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$.

A *partial* function f from A to B is an assignment to each element a in a subset of A, called the *domain of definition* of f, of a unique element b in B. We say that f is *undefined* for elements in A that are not in the domain of

definition of f. When the domain of definition of f equals A, we say that f is a *total* function.

It is often necessary to define new *updated* functions from existing ones. $f[y \mapsto v]$ denotes a new function f' derived from f by updating f with a new value v at y:

$$f'(x) = f[y \mapsto v](x) = \begin{cases} v, & \text{if } x = y \\ f(x), & \text{otherwise.} \end{cases}$$

Functions can be combined using the composition operation. For $f : R \to S$ and $g : S \to T$, $g \circ f$ is the function with domain R and codomain T such that $\forall x \in R : g \circ f(x) = g(f(x))$.

Functions can have arbitrarily complex domains and codomains. For example, if $R \times S$ is the domain of a function f, then f is said to take two arguments. Likewise, if $R \times S$ is the codomain, then f is said to return a pair of values. We may use the notation R^2 denoting the cartesian product $R \times R$, and its generalization R^n, denoting $R \times \cdots \times R$ (n times). *Function domains* can serve as domain or codomain of functions. The function $f : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ takes a natural number and associates a function from the function domain of functions with arity $\mathbb{N} \to \mathbb{N}$ to it. Since $\to$ associates to the right, the above example can also be written as $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. This is based on the following equality:

$$A \to (B \to C) = A \to B \to C.$$

Functions can be described via a set if we collect pairs $(x, f(x))$ in a set. For function $f : R \to S$, the set

$$\text{graph}(f) = \{(x, f(x) \mid x \in R\}$$

is called the *graph* of the function.

Example 2.1 $\text{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$\text{graph}(\text{add}) = \{((0,0),0),\ ((0,1),1),\ ((1,0),1),\ ((1,1),2),\ ((2,0),2), \cdots\}$

Although the graph representation of a function provides insight into its structure, it is inconvenient to use in practice. Example (2.2) denotes function "add" of Example (2.1) as an *equation*. The equational notation comprises name and arity of the function, together with an equational specification of "add".

Example 2.2 $\text{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$\text{add}(m, n) = m + n$

The value of a function given in equational form is determined by evaluation based on substitution and simplification. An equation $f(x) = \alpha$, for $f : A \to B$, represents a function. In order to use the equational definition to map a specific $a_o \in A$ to $f(a_0) \in B$, a_0 has to be *substituted* for all occurrences of x in α. This substitution is denoted by $[x \mapsto a_0]\alpha$. This expression evaluates then to its underlying value through *simplification*.

As an example we use the equational definition of Example (2.2) to evaluate "add$(2, 8)$": Substitution of 2 for m and 8 for n in the expression on the right-hand side of the equation of "add" gives $[m \mapsto 2][n \mapsto 8]m + n = 2 + 8$. Simplification of the expression $2 + 8$ uses knowledge of the primitive operation $+$ to obtain $2 + 8 = 10$.

Definition 2.1 We denote the finite set of program variables by $\mathbb{V}$. Given the n-bounded finite index-set $\mathrm{IS} = \{x \mid x \leq n\} \subset \mathbb{N}$ and a total function $\mathrm{Idx} : \mathrm{IS} \to \mathbb{V}$ that is one-to-one and onto, we write v_i, with $i \in \mathrm{IS}$, to denote element $\mathrm{Idx}(i)$ of $\mathbb{V}$. If the meaning is clear from the context, we also use unique named constants that are functions of arity $\to \mathbb{V}$. These are denoted by lowercase letters, e.g., i, j, k.

2.1.1 References

Sets and Functions If not otherwise stated, the definitions that have been used in this section are due to [Sch86] and [Ros95].

2.2 Syntax and Semantics of Programming Languages

There are two main aspects of a computer language — its syntax and its semantics. The syntax defines the correct form for legal programs and the semantics determines what they compute. While the syntax of a language is always formally specified in a variant of BNF, the more important part of defining its semantics is mostly left to natural language, which is ambiguous and leaves many questions open. Hence, methods were developed to describe the semantics of computer languages. *Denotational semantics* is such a methodology for giving precise meaning to a computer language.

2.2.1 Syntax

As mentioned above, the syntax of a computer language is concerned only with the structure of programs. The syntax treats a language as a set of strings over an alphabet of symbols. The syntax is usually given by a grammar that gives *productions* for generating strings of symbols using auxiliary *nonterminal* symbols.

Formally, a *context-free grammar* (or *grammar*, for short) is a quadruple (V_N, V_T, P, S). Therein V_N and V_T denote finite sets of *nonterminal* and *terminal* symbols, $V_N \cap V_T = \emptyset$. P is a finite set of *productions*. Each production is of the form $\mathrm{A} ::= \alpha_1 \alpha_2 \ldots \alpha_n$, where A is a nonterminal, and

$$\underset{1 \leq i \leq n}{\alpha_i} \in (V_N \cup V_T).$$

Subscripts for terminals and nonterminals on the righthand-side of productions may be used in order to distinguish between symbols of the same kind (e.g. $A ::= \alpha_{1_1} \alpha_{1_2}$). S is a distinct nonterminal of the set V_N known as the *start symbol*.

Example 2.3 Consider the definition of arithmetic expressions with operators $+$ and $*$ and operands represented by the symbol **id**. The corresponding context-free grammar is denoted by $(\{E\}, \{+, *, (,), \mathbf{id}\}, P, E)$, where

$$\begin{array}{lcl} E & ::= & E + E \\ E & ::= & E * E \\ E & ::= & (E) \\ E & ::= & \mathbf{id} \end{array}$$

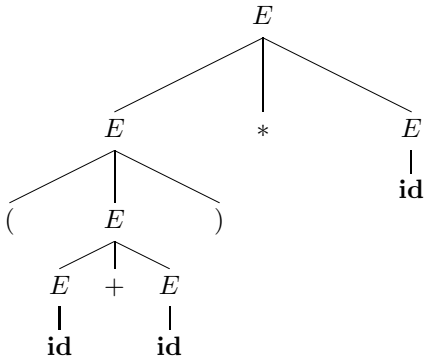


Figure 2.1: Derivation Tree

are the productions of P.

The fact that a grammar contains $A ::= \alpha_1$, $A ::= \alpha_2$,..., $A ::= \alpha_k$ as productions for a nonterminal A can be abbreviated by

$$A \ ::= \ \alpha_1 \ \mid \ \alpha_2 \ \mid \ \cdots \ \mid \ \alpha_k.$$

The nodes of a *derivation tree* are labeled with nonterminals or terminals of a given grammar. If a given node n in the tree is labeled with A and its successor nodes $n_1, n_2, \cdots, n_k$ are labeled with $X_1, X_2, \cdots, X_k$, then $A ::= X_1 X_2 \cdots X_k$ is a production in this grammar. Formally, if $G = (V_N, V_T, P, S)$ is a context-free grammar, then a tree is a derivation tree for G iff the following requirements hold:

(1) Every node is labeled with a symbol of $V_N \cup V_T$.

(2) The root-node is labeled with S.

(3) If a non-leaf node is labeled with A, then $A \in V_N$.

(4) If a given node n is labeled with A and its successor nodes $n_1, n_2, \cdots, n_k$ are labeled from left to right with $X_1, X_2, \cdots, X_k$, then $A ::= X_1 X_2 \cdots X_k$ is a production in P.

(5) If a leaf-node is labeled with A, then $A \in V_T$.

The set of all derivation trees according to G is denoted by Tree_G. Figure 2.1 contains the derivation tree of the arithmetic expression $(\mathbf{id} + \mathbf{id}) * \mathbf{id}$ for the grammar of Example (2.3). Starting with the root-node labeled with the start-symbol E, the derivation tree denotes the sequence of productions necessary to *derive* $(\mathbf{id} + \mathbf{id}) * \mathbf{id}$ from E.

The word w that is expressed by the labels of the leave nodes of a derivation tree t when read from left to right is called the *front* of a tree. The language $L(G)$ that is defined by a grammar G can then be described as the set of words

$$\{w \in V_T^* \mid \exists t \in \text{Tree}_G : \text{front}(t) = w\}.$$

Informally, this set contains those words w that have the following two properties:

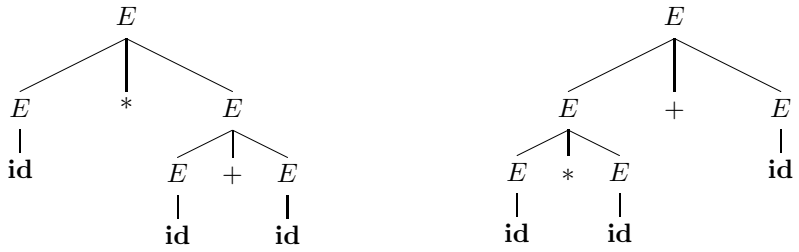


Figure 2.2: Derivation Trees for $\mathbf{id} * \mathbf{id} + \mathbf{id}$

(1) w consist of a sequence of terminal symbols of the set V_T,

(2) a derivation tree t with front w for grammar G exists.

Often there exists more than one derivation tree for a given word. In this case the underlying grammar is said to be *ambiguous*. Consider Figure 2.2 which contains two derivation trees for the same word, but with different meanings: the tree to the left of Figure 2.2 suggests an evaluation order equivalent to the parenthesized expression $\mathbf{id} * (\mathbf{id} + \mathbf{id})$, whereas its counterpart denotes an evaluation order that is equivalent to $(\mathbf{id} * \mathbf{id}) + \mathbf{id}$. The choice is important, because compilers as well as semantic definitions (cf. Section 2.2.2) assign meanings based on the structure of derivation trees.

Ambiguous grammar definitions can often be rewritten into an unambiguous form if additional nonterminals and productions are introduced. However, the price paid is that the revised definitions are more complex and that the corresponding derivation trees contain additional, artificial levels of structure with semantically irrelevant details.

We can avoid those modified, complex grammars if we shift our view and regard language as a set of derivation trees instead of strings (words) over a set of symbols (terminals). Contrary to strings which may stand for several derivation trees in an ambiguous grammar, the structure of a derivation tree itself is unambiguous.

With this change of view, strings are regarded only as abbreviations for derivation trees. As we have seen in Figure 2.2, there exist abbreviations that are ambiguous in terms of the grammar of Example (2.3). This grammar is therefore not capable of assigning a unique derivation tree to a string, but it is sufficient for specifying the structure of derivation trees for arithmetic expressions.

Due to this fact it is common practice to use two related grammars: one complex but unambiguous grammar to determine the derivation tree that a string abbreviates, and one simpler grammar to analyze the tree's structure and determine its semantics. The complex grammar represents the so-called *concrete syntax* of a language, while the latter represents so-called *abstract syntax*. There exists a formal relationship between abstract and concrete syntax: the tree generated for a string in terms of the grammar for concrete syntax identifies a derivation tree for the string in terms of the grammar for abstract syntax.

Abstract syntax can be regarded as an abstraction from concrete syntax where syntactic details such disambiguation are sacrificed for simplicity and enhanced readability. Concrete syntax mainly addresses parsing problems. These

problems are due to the string-format of programmer-delivered input programs, but they are of no concern for static program analysis. It will be shown in Section 2.2.2 that program semantics are derived from derivation trees. In this way we are only concerned with abstract syntax in the remainder of this work.

2.2.2 Denotational Semantics

Denotational Semantics defines the semantics of a programming language on the basis of its abstract syntax. It uses functions in order to associate semantic values (called *denotations*) to syntactically valid structures. A simple example of such a function maps an arithmetic expression to its value:

$$\text{val} : \text{Expression} \rightarrow \text{Integer}.$$

In this way $\text{val}(2+(5*3)) = 17$ and $\text{val}((2+4)*2) = 12$. A function of this kind is called *semantic function* or *valuation function*, and its domain is a *syntactic domain* that we consider as a set of derivation trees with a structure specified through the productions of a grammar. The codomain of a semantic function is a semantic domain. In case of function "val" the syntactic domain consists of the set of derivation trees of syntactically valid arithmetic expressions, and the semantic domain is the domain of integer numbers specified in Section 2.1. The exact interpretation of the example $\text{val}(2 + (5 * 3)) = 17$ is therefore that the derivation tree abbreviated by the string $2 + (5 * 3)$ is associated with the semantic value 17.

We are now ready to enumerate the ingredients of the *denotational definition* of a language together with the associated notational conventions that we will use henceforth. As an example we develop denotational definitions for the language of *numerals* and *digits* in parallel.

Formally a denotational definition of a language consists of three parts: the abstract syntax definition of the language (syntactic domain), the semantic algebras (semantic domains), and the valuation functions.

Syntactic Domains

A denotational definition considers the syntactic domain as a set of derivation trees. The structure of these trees is defined through the productions $p \in P$ of a grammar $G = (V_N, V_T, P, S)$. The nonterminal symbols $v_N \in V_N$ of G identify corresponding sets of derivation trees where a nonterminal symbol itself corresponds to a variable denoting an element of such a set. The first part of a syntactic domain definition lists the nonterminal symbols together with the sets of derivation trees to which they are related.

$$\text{D} \ : \ \text{Digit} \tag{2.1}$$
$$\text{N} \ : \ \text{Numeral}. \tag{2.2}$$

This definition is to be read as follows: The nonterminal symbol D is regarded as a variable that may denote any element of the set Digit of derivation trees, while the nonterminal N denotes any element of the set Numeral of derivation trees. The second part of a syntactic domain definition contains the productions $p \in P$ themselves.

$$\begin{array}{lll} \mathrm{N} & ::= & \mathrm{N\,D} \mid \mathrm{D} \\ \mathrm{D} & ::= & \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'} \end{array}$$

Semantic Domains

Semantic Domains are based on semantic algebras (cf. Section 2.1). A definition of a semantic domain lists the domain and the operations of the underlying semantic algebra. The following is an example of a simple semantic domain called "Integer".

$$\begin{array}{l} \text{Domain } z : \text{Integer} = \mathbb{Z} \\ \quad + : \text{ Integer} \times \text{Integer} \to \text{Integer} \\ \quad - : \text{ Integer} \times \text{Integer} \to \text{Integer} \\ \quad * : \text{ Integer} \times \text{Integer} \to \text{Integer} \\ \quad / : \text{ Integer} \times \text{Integer} \setminus \{0\} \to \text{Integer} \end{array}$$

The first line of this example defines that $\mathbb{Z}$ is the underlying domain of the semantic domain "Integer" and that the symbol z is to be used as a variable for elements of this semantic domain. The operations comprise the usual arithmetic operators on $\mathbb{Z}$. Since their properties are well-known, it suffices to list only name and arity of these functions.

Valuation Functions

A valuation function maps the abstract syntax structures of a language to meanings drawn from semantic domains. The domain of a valuation function is the set of derivation trees of a language. Valuation functions are defined structurally. They determine the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree.

We introduce a distinct valuation function for each syntactic domain. It is a notational convention to name the valuation function after the nonterminal symbol of the corresponding syntactic domain. To distinguish between the two, the latter is printed in bold. The valuation function for the syntactic domain Digit (cf. Equation (2.1)) is therefore written as

$$\mathbf{D} : \text{Digit} \to \text{Integer}.$$

The derivation trees for the language digit, defined by the production

$$\mathrm{D} \quad ::= \quad \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'},$$

can be enumerated as follows.

$$\begin{array}{ccccccc} \mathrm{D} & & \mathrm{D} & & \mathrm{D} & \cdots & \mathrm{D} \\ | & & | & & | & & | \\ \text{'0'} & & \text{'1'} & & \text{'2'} & \cdots & \text{'9'} \end{array}$$

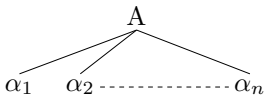
The valuation function $\mathbf{D}$ assigns a meaning to each of these derivation trees.

$$\mathbf{D}\Big(\begin{array}{c}\mathrm{D}\\|\\\text{'0'}\end{array}\Big) = 0, \quad \mathbf{D}\Big(\begin{array}{c}\mathrm{D}\\|\\\text{'1'}\end{array}\Big) = 1, \ldots, \mathbf{D}\Big(\begin{array}{c}\mathrm{D}\\|\\\text{'9'}\end{array}\Big) = 9$$

Clearly this two-dimensional notation is inadequate for complex derivation trees. The following onedimensional abbreviation uses only the right-hand side of productions and introduces *emphatic brackets* $[\![]\!]$ to enclose the syntactic arguments of valuation functions.

$$\mathbf{D}[\![\text{'0'}]\!] = 0, \mathbf{D}[\![\text{'1'}]\!] = 1, \ldots, \mathbf{D}[\![\text{'9'}]\!] = 9$$

Note that although emphatic brackets enclose only the right-hand side of a production $\mathrm{A} ::= \alpha_1\alpha_2 \ldots \alpha_n$, the actual argument to the valuation function is the derivation tree



with root node A and successors $\alpha_1, \alpha_2, \ldots, \alpha_n$.

To complete our example, we note that

$$\mathbf{N} : \text{Numeral} \rightarrow \text{Integer}$$

denotes the arity of the valuation function for the syntactic domain Numeral (cf. Equation (2.2)). It is defined by two productions

$$\begin{array}{lcl} \mathrm{N} & ::= & \mathrm{N\,D} \\ & | & \mathrm{D} \end{array}$$

which lead to the following two equations describing the valuation function **N**.

$$\mathbf{N}[\![\mathrm{ND}]\!] = 10 * \mathbf{N}[\![\mathrm{N}]\!] + \mathbf{N}[\![\mathrm{D}]\!] \tag{2.3}$$

$$\mathbf{N}[\![\mathrm{D}]\!] = \mathbf{D}[\![\mathrm{D}]\!] \tag{2.4}$$

2.2.3 References

Programming Language Syntax Languages and context-free grammars are treated in [HU79]. A comparison of concrete and abstract syntax is given in [Wat91], [Mos90], and [Sch86].

Programming Language Semantics Both [Lou93] and [Feh89] present and compare different approaches of specifying programming language semantics.

Denotational Semantics A tutorial on this subject can be found in [Ten76]. [Lou93], [Sch86], and [All86] present thorough introductions to the subject and its application to programming languages. Further in-depth sources of information include [Wat91], [Feh89], and [Mos90, GS90]. The denotational semantic notations adopted throughout this work are based on [Lou93] and [Sch86].

2.3 Control Flow Graphs

A *simple* graph $G = \langle N, E \rangle$ consists of N, a nonempty set of nodes, and E, a set of unordered pairs of distinct elements of N. A *directed* graph $G = \langle N, E \rangle$ consists of a set of nodes N and a set of edges E that are ordered pairs of elements of N. Each edge e of a directed graph has a *head* $\mathrm{h}(e) \in N$ and a *tail* $\mathrm{t}(e) \in N$. Thus the edge e leads from $\mathrm{h}(e)$ to $\mathrm{t}(e)$. The set of incoming edges for a given node $n \in N$ is defined as $\mathrm{in}(n) = \{e \in E : \mathrm{t}(e) = n\}$. Likewise we can define the set of outgoing edges for a node $n \in N$ as $\mathrm{out}(n) = \{e \in E : \mathrm{h}(e) = n\}$.

Definition 2.2 A *path* $\pi = \langle e_1, e_2, \ldots, e_k \rangle$ is a sequence of edges such that $\mathrm{t}(e_r) = \mathrm{h}(e_{r+1})$ for $1 \leq r \leq k-1$. Containment of an edge e in a path π, denoted as $e \in \pi$, is defined as

$$e \in \pi = \langle e_1, e_2, \ldots, e_k \rangle \Leftrightarrow \underset{1 \leq r \leq k}{\exists r} : \; e = e_r.$$

The length $|\,\pi\,|$ of a path π denotes the number of edges contained in π. We can also define containment of a node n in a path π:

$$n \in \pi = \langle e_1, e_2, \ldots, e_k \rangle \Leftrightarrow \underset{1 \leq r \leq k}{\exists r} : \; n = \mathrm{h}(r) \vee n = \mathrm{t}(r).$$

A directed graph is *strongly connected* if there is a path from a to b and from b to a whenever a and b are nodes in the graph. A directed graph is *weakly connected* if if there is a path between any two nodes in the underlying simple graph.

Definition 2.3 A *control flow graph (CFG)* is a directed labeled graph $G = \langle N, E, n_e, n_x \rangle$ with node set N and edge set $E \subseteq N \times N$. *Entry (n_e)* and *Exit (n_x)* are distinguished nodes used to denote the start and terminal node. The start node n_e has no incoming edges ($\mathrm{in}(n_e) = \emptyset$), whereas the terminal node n_x has no outgoing edges ($\mathrm{out}(n_x) = \emptyset$). Furthermore we require that every node n is contained in a path from n_e to n_x. The set of all successors of a node $n \in N$ is denoted by $\mathrm{succ}(x)$, while the set of all predecessors of n is denoted by $\mathrm{pred}(x)$.

An intuitive way of mapping a program to a CFG is to interpret its nodes as basic blocks b_i containing the program statements, whereas its edges represent transfer of control between basic blocks. Each edge e_i of the CFG is assigned a condition $c_i = \mathrm{cond}(e_i)$ which must evaluate to true[¶] for the program's control flow to follow this edge. An early example of this approach can be found in [Kil73]. A mapping that does not target CFG nodes can be achieved if basic blocks are also mapped to CFG edges. Among others, this approach is taken in [AC76] and [Ram96]. CFGs can be transformed from the first to the latter representation by shifting basic blocks from graph nodes onto graph edges. As shown in the example depicted in Figure 2.3(a), the result of this transformation is different depending on whether basic blocks are shifted from their respective nodes onto *incoming* edges (Figure 2.3(b)), or onto *outgoing* edges (Figure 2.3(c)).

[¶]We have not defined the notion of *evaluation* yet, but for the ongoing discussion it suffices to relate it to the concept of expression evaluation found e.g. with compiler construction.

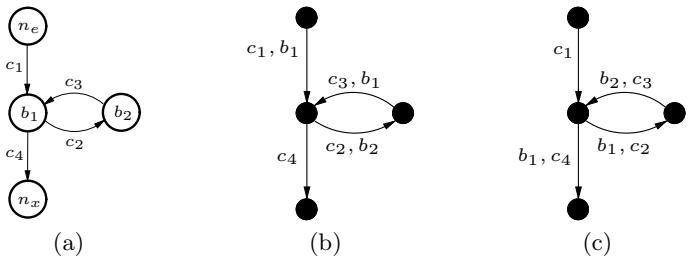


Figure 2.3: CFG Representations

The main difference between these two representations is that with the first each edge is assigned an ordered tuple $\{condition, basic\_block\}$, whereas the latter assigns an ordered tuple $\{basic\_block, condition\}$. If we consider a *forward* data-flow problem where information is propagated through the CFG along and in the direction of the CFG edges, the latter representation would require to evaluate a basic block *before* the respective condition. Should the condition evaluate to *false* we are in the unfortunate position of having to reverse or "roll-back" the effects of the preceding evaluation of the basic block. Thus forward data-flow problems are better served by the first representation. As *backward* data-flow problems propagate data-flow information in the reverse direction of the CFG edges, they are better-suited to the second representation.

2.3.1 References

Control Flow Graphs Material on control flow graphs can be found, among others, in [ASU86, CT04, BJ66].

Chapter 3

Standard Semantics of Program Execution

We define a flow language that models the standard semantic behavior of program execution similar to the approach chosen in [CR81]. Furthermore we show in Section 3.5 that our flow language can simulate a Turing machine and vice versa.

3.1 The Language **Flow**

Panta rhei.
(Everything flows).
— HERAKLIT, Greek philosopher, 540–480 BC

Informally an environment can be envisioned as a set of variable/value pairs $\{v_1 = n_1, \ldots, v_k = n_k\}$ where v_i is a program variable and $n_i \in \mathbb{Z}$ holds the value of v_i for $1 \leq i \leq k$. We require that for each variable $v_i \in \mathbb{V}$ there exists exactly one pair $v_i = n_i$ in a given environment. Due to this property such a set can also be interpreted as the graph of a function

$$env : \mathbb{V} \to \mathbb{Z}.$$

This function takes a variable identifier v as argument and associates the value of v to it. We will make extensive use of this kind of functions which makes it more convenient to reserve the term *environment* for the function instead of the graph of the function. The set of possible environments can then be represented by a class of functions

$$Environment \subseteq \{env : \mathbb{V} \to \mathbb{Z}\}. \tag{3.1}$$

Often we restrict our interest to a subset of $\mathbb{V}$. In those cases the environments are *partial* functions.

States $s \in S$ are connected by directed edges $e \in E$. The corresponding directed graph $G = \langle S, E, s_e, s_x \rangle$ has a distinguished start node s_e and a distinguished terminal node s_x. The start node s_e has no incoming edges ($\text{in}(s_e) = \emptyset$)

whereas the terminal node s_x has no outgoing edges ($\text{out}(s_x) = \emptyset$). Furthermore we require that every state s is contained in a path from s_e to s_x. We can envision the semantics of standard program execution as a forward data-flow problem as follows.

Associated with each edge $e \in E$ is a *branch predicate*

$$pred : E \to \big(Environment \to \mathbb{B}\big). \tag{3.2}$$

This predicate evaluates within a given environment $env \in Environment$ and determines how the flow of control progresses through the flow-graph G (the exact treatment of this evaluation process is deferred until Section 3.3). Control progresses from state $\text{h}(e)$ to state $\text{t}(e)$ iff $pred\big(e\big)(env) = true$, which means that the predicate associated with edge e evaluates to *true* within environment *env*. Figure 3.1 presents the general case of a state s with outgoing edges $\text{out}(s) = \{Edge_1, \ldots, e_n\}$ leading to successor states $s_1, \ldots, s_n$. In order to determine the control flow successor of state s for environment *env*, the predicates $pred\big(e_1\big)(env), \ldots, pred\big(e_n\big)(env)$ are evaluated. We can distinguish three cases with respect to the number of edges that have assigned a predicate evaluating to true.

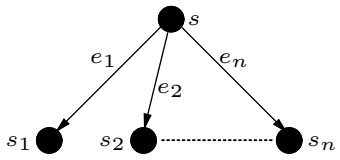


Figure 3.1: Flow States

(1) The predicate of a *single* edge $e \in \text{out}(s)$ evaluates to true. In this case the state $\text{t}(e)$ is the successor state for state s.

(2) There exists a subset of edges $\{e_1, \ldots, e_k\} \subseteq \text{out}(s), k > 1$, for which the assigned predicate evaluates to *true*. In this case we have states $\text{t}(e_1), \ldots, \text{t}(e_k)$ qualifying as possible successor states. This results in *indeterminism* or *parallelism*, depending on the number of successor states we accept.

(3) Zero predicates evaluate to true. In this case the computation stops due to the lack of a successor state. State s is called a *halting state*.

We require that for any given state $s \neq s_x$ and environment *env* the branch predicate of exactly one outgoing edge evaluates to true:

$$\exists e_i \in \text{out}(s) : \;\; pred(e_i)(env) \wedge \underset{j \neq i}{\forall} e_j : \neg pred(e_j)(env). \tag{3.3}$$

Also associated with each edge e is a *side-effect* σ representing the effect of a computational step on a given environment $env \in Environment$:

$$\sigma : E \to (Environment \to Environment). \tag{3.4}$$

The transition function δ has the set $S \times Environment$ as its domain and codomain:

$$\delta : (S, Environment) \to (S, Environment).$$

Execution of a transition $(s, env) \to (s', env')$ via an edge e is defined as

$$\begin{aligned}(s, env) \to &(s', env') : \\ &\exists e \in \text{out}(s) : \ t(e) = s' \wedge pred(e)(env) \\ &\Rightarrow env' = \sigma(e)(env),\end{aligned} \tag{3.5}$$

where $\Rightarrow$ denotes *implication*.

The iterated transition function $\delta^* : (S, Environment) \to (S, Environment)$ is defined as

$$\begin{aligned}\delta^*(s_x, env) &= (s_x, env) \\ \delta^*(s, env) &= \delta^*(\delta(s, env)).\end{aligned} \tag{3.6}$$

For any graph $G = \langle S, E, s_e, s_x \rangle$ the environment env_x of the terminal state s_x represents the result of program execution along the sequence of transitions $(s_e, env_e) \xrightarrow{*} (s_x, env_x)$. Depending on the structure of G and the initial environment env_e such a transition sequence may not exist. Deciding on its existence is in general equivalent to the halting problem.

3.2 Syntax and Semantics of Side-Effects

Until now the side-effects of a computation have only been described to the extend that they are represented by a class of functions that map environments to environments (cf. Equation (3.4)). We are now going to specify the denotational definition of side-effects of the language Flow based on the grammar

$$G = \big(\{\text{assign}, \text{ident}, \text{expr}, \text{binop}, \text{num}, \text{dig}\},\ \{+, -, *, \text{div}, \text{rem}, (,),\ \},\ P,\ \text{assign}\big).$$

Table 3.1 (I) defines the *syntactic domain* of G. It contains the productions $p \in P$ that define the structure of derivation trees for assignment-statements, e.g., $a := b * (c + d)$. The production for the nonterminal "ident" has been omitted for brevity, it identifies the set of derivation trees that correspond to elements of the set $\mathbb{V}$ of possible variable identifiers.

The nonterminal "num" denotes elements of $\mathbb{Z}$, the set of integers. This set, together with the arithmetic operations of integer addition ("+") and multiplication ("$\cdot$") forms an *integral domain* $(\mathbb{Z}; +, \cdot)$. Although divisibility plays a central role for computations, division is not possible in an integral domain in general, e.g., $\frac{5}{2} = 2.5 \notin \mathbb{Z}$. However, due to the *division property* stated in [GCL92, p. 30], the integers $\mathbb{Z}$ form a *Euclidean domain*. This means that for all elements $a, b \in \mathbb{Z}$ with $b \neq 0$, there exist elements $q, r \in \mathbb{Z}$ such that

$$a = b \cdot q + r. \tag{3.7}$$

In this equation variable q denotes the *quotient* and variable r denotes the *remainder* of the operation $\frac{a}{b}$. We write $q = a \operatorname{div} b$ to denote integer division, and $r = a \operatorname{rem} b$ to denote the remainder operation. It should be noted that quotient q and remainder r are not unique in $\mathbb{Z}$ in general, if $r \neq 0$. For example, if $a = -8$ and $b = 3$, then we have

$$-8 = 3 \cdot (-2) - 2 = 3 \cdot (-3) + 1,$$

(I) Syntactic Domain

assign	:	Assignment	num	:	Numeral
expr	:	Expression	dig	:	Digit
ident	:	Identifier	binop	:	Binary Operator

assign	::=	ident := expr
expr	::=	expr binop expr
	|	− expr
	|	ident
	|	num
	|	(expr)
binop	::=	+ | − | ∗ | div | rem
num	::=	num dig | dig
dig	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(II) Semantic Domain

Integers

Domain $z : \text{Integer} = \mathbb{Z}$

$+, -, \cdot \ : \ \text{Integer} \times \text{Integer} \to \text{Integer}$

$\text{div}, \text{rem} \ : \ \text{Integer} \times \text{Integer} \setminus \{0\} \to \text{Integer}$

Program Variables

Domain $v : \mathbb{V}$

Environments

Domain $env : \mathit{Environment} \subseteq \{f : \mathbb{V} \to \mathbb{Z}\}$

(III) Valuation Functions

assign : Assignment $\to$ *Environment* $\to$ *Environment*

$\quad \mathbf{assign}[\![\text{ident:=expr}]\!](env_1) =$

$\qquad \lambda env_2.env_2\big[\mathbf{ident}[\![\text{ident}]\!] \mapsto \mathbf{expr}[\![\text{expr}]\!](env_2)\big](env_1)$

expr : Expression $\to$ *Environment* $\to$ Integer

$\quad \mathbf{expr}[\![\text{expr}_1 \text{ binop } \text{expr}_2]\!](env) =$

$\qquad \mathbf{expr}[\![\text{expr}_1]\!](env) \ \mathbf{binop}[\![\text{binop}]\!] \ \mathbf{expr}[\![\text{expr}_2]\!](env)$

$\quad \mathbf{expr}[\![-\text{expr}]\!](env) = -\mathbf{expr}[\![\text{expr}]\!](env)$

$\quad \mathbf{expr}[\![\text{ident}]\!](env) = env\big(\mathbf{ident}[\![\text{ident}]\!]\big)$

$\quad \mathbf{expr}[\![\text{num}]\!](env) = \mathbf{num}[\![\text{num}]\!]$

$\quad \mathbf{expr}[\![(\text{expr})]\!](env) = \mathbf{expr}[\![\text{expr}]\!](env)$

num : Numeral $\to$ Integer

$\quad \mathbf{num}[\![\text{num dig}]\!] = 10 \cdot \mathbf{num}[\![\text{num}]\!] + \mathbf{dig}[\![\text{dig}]\!]$

$\quad \mathbf{num}[\![\text{dig}]\!] = \mathbf{dig}[\![\text{dig}]\!]$

dig : Digit $\to$ Integer

$\quad \mathbf{dig}[\![0]\!] = 0, \ \mathbf{dig}[\![1]\!] = 1, \ldots, \mathbf{dig}[\![9]\!] = 9$

ident : Identifier $\to \mathbb{V}$ (omitted)

binop : Binary Operator $\to \{+, -, \cdot, \text{div}, \text{rem}\}$ (omitted)

Table 3.1: Denotational Definition of Side-Effects

so that both pairs $q = -2$, $r = -2$ and $q = -3$, $r = 1$ satisfy Equation (3.7). We can adopt one of the following rounding modes to make quotient and remainder unique in $\mathbb{Z}$.

- Round towards ∞:
$$a \operatorname{div} b = \lceil a/b \rceil. \tag{3.8}$$
- Round towards $-\infty$:
$$a \operatorname{div} b = \lfloor a/b \rfloor. \tag{3.9}$$
- Round towards zero (truncate):
$$a \operatorname{div} b = \begin{cases} \lfloor a/b \rfloor, & \text{if } a/b \geq 0 \\ \lceil a/b \rceil, & \text{if } a/b < 0. \end{cases} \tag{3.10}$$

For the remainder of this work we will always assume the *round towards zero* rounding mode* for the integer division and remainder operations.

Variable identifiers, natural numbers and environments make up the *semantic domain* of side-effects of the language Flow. Each of these subdomains is listed in Part (II) of Table 3.1. We introduce the domain-name "Integer" for the Euclidean domain $\mathbb{Z}$. The class of functions named *Environment* consists of function that map values to variable identifiers and has been introduced in Equation (3.1).

Table 3.1 (III) finally describes the *valuation functions* for each syntactic subdomain of Part (I). According to Section 2.2.2 those valuation functions map derivation trees to meanings drawn from semantic domains. Valuation functions are defined structurally in a sense that they determine the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree. This structural decomposition lends itself nicely to a bottom-up description of the respective valuation functions.

binop: This valuation function maps the terminals "+", "−", "∗", "div" and "rem" to the corresponding arithmetic operations of integer addition, subtraction, multiplication, division, and remainder over Integer × Integer → Integer. Algorithms for these operations have been omitted for space considerations, the interested reader is referred to [BBS02, pp. 95–102] and [Knu97, pp. 265–273]. The terminal "−" may also serve as an unary operator defined on Integer → Integer denoting change of sign.

ident: The valuation function for identifiers maps a derivation tree representing a variable identifier to the corresponding identifier $v \in \mathbb{V}$. Since we have left out the exact syntactic specification for identifiers, we restrict ourselves to the arity of **ident** in Table 3.1.

dig, **num**: Together, these valuation functions calculate the value of a given integer numeral. They correspond to the valuation functions **D** and **N** introduced in Section 2.2.2.

*In fact this is the rounding mode adopted by many contemporary programming languages such as Ada, C, and also Java.

expr: These valuation functions take an expression as argument and return a function that maps an environment to an integer number. The syntactic structure of the expression derivation tree is specified between the emphatic brackets ($[\![\ldots]\!]$) on the left-hand side of the given equations, whereas the functions $f : \mathit{Environment} \to \text{Integer}$ represent the right-hand sides.

- $\mathbf{expr}[\![\text{num}]\!](\mathit{env}) = \mathbf{num}[\![\text{num}]\!]$: For a derivation tree $[\![\text{num}]\!]$ the argument *env* describing the environment is not needed. This follows from the arity of the valuation function **num**. This argument is therefore discarded on the right-hand side of this equation.
- $\mathbf{expr}[\![\text{ident}]\!](\mathit{env}) = \mathit{env}(\mathbf{ident}[\![\text{ident}]\!])$: For a derivation tree of structure $[\![\text{ident}]\!]$ denoting an identifier we determine the corresponding identifier $v \in \mathbb{V}$ that is then evaluated within the environment-argument *env*.
- $\mathbf{expr}[\![\text{expr}_1 \text{ binop } \text{expr}_2]\!](\mathit{env})$: For a tree $[\![\text{expr}_1 \text{ binop } \text{expr}_2]\!]$ we recursively determine the values of the sub-expressions $[\![\text{expr}_1]\!]$ and $[\![\text{expr}_2]\!]$, which are then combined using the arithmetic operation that is returned from $\mathbf{binop}[\![\text{binop}]\!]$.

assign: This valuation function takes a derivation tree $[\![\text{ident:=expr}]\!]$ corresponding to an assignment statement as argument. From this it returns a function that maps an environment env_1 supplied as an argument to an environment env_2. Environment env_2 is generated from env_1 by updating env_1 with a new value $\mathbf{expr}[\![\text{expr}]\!]$ at $\mathbf{ident}[\![\text{ident}]\!]$. Note in this context that environments are functions!

We conclude by noting that the valuation function **assign** returns a function of arity $f : \mathit{Environment} \to \mathit{Environment}$. This matches the requirements for side-effects stated in Equation (3.4) and repeated at the beginning of this section. In this way the denotational definition of Table 3.1 is a valid definition for side-effects of the language Flow.

3.3 Syntax and Semantics of Branch-Predicates

Until now we have only required that a branch predicate is a function that maps an environment to a boolean value. This requirement has been stipulated in Equation (3.2). In accordance with the definition of side-effects in Section 3.2 we will now describe the nature of Flow branch predicates by means of a denotational definition based on the grammar

$$G = \big(\{\text{pred}, \text{rel-op}, \text{expr}\},\ \{\text{and}, \text{or}, \text{not}, <, \leq, =, \geq, >, \neq,\ \},\ P_{\text{pred}},\ \text{pred}\big).$$

From the set of G's nonterminals we see that this definition depends on the definition of side-effects due to the occurrence of the nonterminal "expr". Hence a branch predicate may contain expressions, a fact that is also manifested in the *syntactic domain* of branch predicates given in Table 3.2 (I). From this table it follows that derivation trees of expressions are connected by binary function symbols "$<$", "$\leq$", "$=$", "$\geq$", "$>$", and "$\neq$". The resulting constructs as well

as the literals "*true*" and "*false*" serve as arguments for the binary function symbols "or" and "and" as well as the unary function symbol "not".

(I) Syntactic Domain

pred : Predicate
rel-op : Relational Operator

pred ::= true
| false
| pred and pred
| pred or pred
| not pred
| expr rel-op expr
| (pred)
rel-op ::= $<$ | $\leq$ | $=$ | $\geq$ | $>$ | $<>$

(II) Semantic Domain

Boolean Values
Domain b : Boolean $= \mathbb{B}$
$<, \leq, =, \geq, >, \neq$: Integer $\times$ Integer $\rightarrow$ Boolean
$\wedge, \vee$: Boolean $\times$ Boolean $\rightarrow$ Boolean
$\neg$: Boolean $\rightarrow$ Boolean

(III) Valuation Functions

pred : Predicate $\rightarrow$ *Environment* $\rightarrow$ Boolean
$\mathbf{pred}[\![\text{true}]\!](env) = true$
$\mathbf{pred}[\![\text{false}]\!](env) = false$
$\mathbf{pred}[\![\text{pred}_1 \text{ and } \text{pred}_2]\!](env) = \mathbf{pred}[\![\text{pred}_1]\!](env) \wedge \mathbf{pred}[\![\text{pred}_2]\!](env)$
$\mathbf{pred}[\![\text{pred}_1 \text{ or } \text{pred}_2]\!](env) = \mathbf{pred}[\![\text{pred}_1]\!](env) \vee \mathbf{pred}[\![\text{pred}_2]\!](env)$
$\mathbf{pred}[\![\text{not pred}]\!](env) = \neg\mathbf{pred}[\![\text{pred}]\!](env)$
$\mathbf{pred}[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!](env) = \mathbf{expr}[\![\text{expr}_1]\!](env)$
$\mathbf{rel\text{-}op}[\![\text{rel-op}]\!]\ \mathbf{expr}[\![\text{expr}_2]\!](env)$

rel-op : Relational Operator $\rightarrow \{<, \leq, =, \geq, >, \neq\}$ (omitted)

Table 3.2: Denotational Definition of Branch-Predicates

In addition to the semantic domain for side-effects we need the semantic algebra of boolean values of Table 3.2 (II) for the denotational definition of branch predicates. This algebra, henceforth named "Boolean", has the truth values $\mathbb{B} = \{true, false\}$ as its carrier set.

The operations $<, \leq, =, \geq, >$, and $\neq$ are of arity Integer$\times$Integer $\rightarrow$ Boolean and denote the usual relational connectives based on the order "$<$" and "$\leq$" on $\mathbb{Z}$. In principle these connectives can also be seen as an extension of the domain of integers, but in order to avoid the forward reference to the semantic domain of boolean values we have placed them here.

The operations "$\wedge$", "$\vee$", and "$\neg$" denote conjunction, disjunction, and negation. We complete the denotational definition of branch predicates with a description of the valuation functions listed in Table 3.2 (III).

rel-op: This valuation function maps the binary function symbols "<", "≤", "=", "≥", ">", and "<>" to the corresponding relational operations in the semantic subdomain Boolean.

pred: The valuation function **pred** takes a derivation tree of the set "Predicate" as input and returns a function of arity *Environment* → Boolean.

pred⟦true⟧(*env*), **pred**⟦false⟧(*env*): Therein literals "*true*" and "*false*" are mapped to the corresponding truth values.

pred⟦pred_1 and pred_2⟧(*env*): For a subtree ⟦pred_1 and pred_2⟧ we determine the values of the subtrees ⟦pred_1⟧ and ⟦pred_2⟧ which are then combined by conjunction.

pred⟦pred_1 or pred_2⟧(*env*): Again we determine the values of the subtrees ⟦pred_1⟧ and ⟦pred_2⟧ which are then combined by disjunction.

pred⟦not pred⟧(*env*): First we determine the value of the subtree ⟦pred⟧ which is then negated.

pred⟦expr_1 rel-op expr_2⟧(*env*): We determine the values of the subtrees ⟦expr_1⟧ and ⟦expr_2⟧ by means of the valuation function **expr** of Table 3.1. These results are then combined using the relational operator returned from **rel-op**⟦rel-op⟧.

In summing up we note that the valuation function **pred** returns a function of arity $f : \mathit{Environment} \rightarrow \text{Boolean}$ which is consistent with the requirements for branch predicates stated in Equation (3.2). Therefore the denotational definition of Table 3.2 is a valid definition for branch predicates of the language Flow.

3.4 A Flow Example Program

We have spent the three preceding sections defining a flow language with side-effects and branch predicates that allows us to model the standard semantic behavior of programs. In order to formulate the first Flow example program we need a notation to associate branch predicates and side-effects with CFG edges.

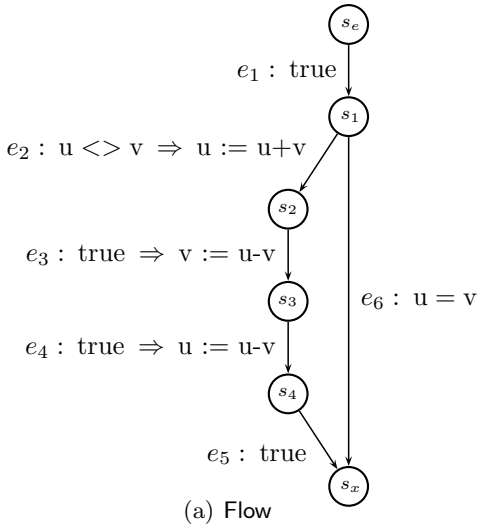
$$\text{edge} \quad ::= \quad \text{e} : \text{ pred } \Rightarrow \text{ assign} \tag{3.11}$$

Equation (3.11) specifies such a notation where **e**⟦e⟧ denotes a CFG edge $e \in E$, and **pred**⟦pred⟧ and **assign**⟦assign⟧ specify the branch predicate and side-effect associated with e. If a CFG edge has assigned the identity function ι, this notation degrades to

$$\text{edge} \quad ::= \quad \text{e} : \text{ pred.} \tag{3.12}$$

Figure 3.2 (a) contains our first Flow example program. For comparison and to facilitate reading, Figure 3.2 (b) contains this example in terms of the imperative programming language Ada (cf. [Ada95]). The purpose of this program is to swap the values of two variables in place. The comments in lines (3–6) of the Ada-version denote the corresponding components of the Flow program, e.g., $\mathit{pred}(e_2)$ denotes the branch predicate of edge e_2, and $\sigma(e_2)$ denotes the side-effect of this edge.

It is instructive to consider the execution of this simple example for concrete values of "u" and "v". The CFG in Figure 3.2 (a) contains two program paths



(a) Flow

```
1   procedure Swap (u, v : in out integer) is
2   begin
3       if u /= v then                -- pred(e2)
4           u := u+v;                 -- σ(e2)
5           v := u-v;                 -- σ(e3)
6           u := u-v;                 -- σ(e4)
7       end if;
8   end Swap;
```

(b) Ada

Figure 3.2: Example Program

from state s_e to s_x, namely $\pi_1 = \langle e_1, e_2, e_3, e_4, e_5 \rangle$ and $\pi_2 = \langle e_1, e_6 \rangle$. Path π_1 is the one chosen if the values of variables "u" and "v" are unequal and where the swapping takes place. Figure 3.3 shows the Flow program of Figure 3.2 (a) together with the graphs of the functions describing the environments env_i valid at states s_i during an execution that has as an initial environment env_e with

$$\text{graph}(env_e) = \{(u, 2), (v, 4)\}.$$

The semantic effect of program execution along path π_1 with initial environment env_e can be calculated by the iterated transition function of Equation (3.6).

For the first iteration, we have $\delta^*(s_e, env_e) = \delta^*(\delta(s_e, env_e))$. It is Equation (3.5) that tells us how to compute $\delta(s_e, env_e)$: state s_e has edge e_1 as its sole outgoing edge. The associated branch predicate of edge e_1 evaluates to *true*,

$$pred(e_1)(env_e) = \mathbf{pred}[\![\text{true}]\!](env_e) = true,$$

which implies that $env_1 = \sigma(e_1)(env_e) = \iota(env_e) = env_e$. This result is in accordance with the graph of the environment env_1 depicted in Figure 3.3.

For the next iteration of the iterated transition function we have to determine $\delta(s_1, env_1))$. From the outgoing edges e_2 and e_6 of state s_1 we evaluate

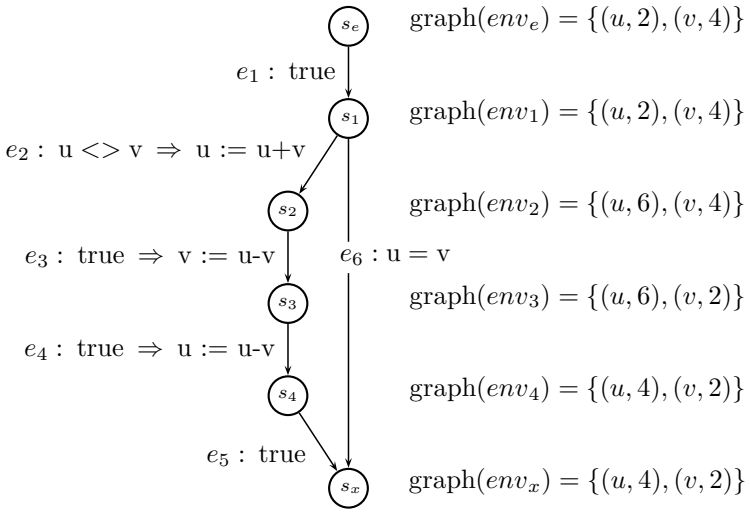


Figure 3.3: Flow Example Execution

the associated branch predicates

$$pred(e_2)(env_1) = \mathbf{pred}[\![\text{u<>v}]\!](env_1) = \ldots = 2 \neq 4 = true \tag{3.13}$$

and

$$pred(e_6)(env_1) = \mathbf{pred}[\![\text{u=v}]\!](env_1) = \ldots = 2 = 4 = false. \tag{3.14}$$

These results are in line with Equation (3.3) that requires that the branch predicate of exactly one outgoing edge evaluates to *true*. Due to Equation (3.13) control flow progresses from state s_1 via edge e_2 to state s_2. Environment env_2 is then the result of applying the result of the valuation function for the side-effect of edge e_2 to the argument env_1.

$$\begin{aligned}
env_2 &= \sigma(e_2)(env_1) = \\
&= \mathbf{assign}[\![\text{u} := \text{u} + \text{v}]\!](env_1) = \\
&= \lambda env_n.env_n\big[\mathbf{ident}[\![\text{u}]\!] \mapsto \mathbf{expr}[\![\text{u} + \text{v}]\!](env_n)\big](env_1) = \\
&= \lambda env_n.env_n\big[u \mapsto \mathbf{expr}[\![\text{u}]\!](env_n)\ \mathbf{binop}[\![+]\!]\ \mathbf{expr}[\![\text{v}]\!](env_n)\big](env_1) = \\
&= \lambda env_n.env_n\big[u \mapsto env_n(\mathbf{ident}[\![\text{u}]\!]) + env_n(\mathbf{ident}[\![\text{v}]\!])\big](env_1) = \\
&= \lambda env_n.env_n\big[u \mapsto env_n(u) + env_n(v)\big](env_1) = \\
&= \lambda env_n.env_n\big[u \mapsto 2 + 4\big] = \\
&= \lambda env_n.env_n\big[u \mapsto 6\big]
\end{aligned} \tag{3.15}$$

The three final lines of the above evaluation show that environment env_2 is derived from env_1 by updating environment env_1 with the new value 6 at "u". This is exactly the semantic effect of the assignment statement of edge e_2. The resulting graph for environment env_2 is again given in Figure 3.3.

$$
\begin{aligned}
env_3 &= \sigma(e_3)(env_2) = \\
&= \mathbf{assign}[\![\mathrm{v} := \mathrm{u} - \mathrm{v}]\!](env_2) = \\
&= \lambda env_n.env_n\big[\mathbf{ident}[\![\mathrm{v}]\!] \mapsto \mathbf{expr}[\![\mathrm{u} - \mathrm{v}]\!](env_n)\big](env_2) = \\
&= \lambda env_n.env_n\big[v \mapsto \mathbf{expr}[\![\mathrm{u}]\!](env_n)\ \mathbf{binop}[\![-]\!]\ \mathbf{expr}[\![\mathrm{v}]\!](env_n)\big](env_2) = \\
&= \lambda env_n.env_n\big[v \mapsto env_n(\mathbf{ident}[\![\mathrm{u}]\!]) - env_n(\mathbf{ident}[\![\mathrm{v}]\!])\big](env_2) = \\
&= \lambda env_n.env_n\big[v \mapsto env_n(u) - env_n(v)\big](env_2) = \\
&= \lambda env_n.env_n\big[v \mapsto 6 - 4\big] = \\
&= \lambda env_n.env_n\big[v \mapsto 2\big] \qquad (3.16)
\end{aligned}
$$

$$
\begin{aligned}
env_4 &= \sigma(e_4)(env_3) = \\
&= \mathbf{assign}[\![\mathrm{u} := \mathrm{u} - \mathrm{v}]\!](env_3) = \\
&= \lambda env_n.env_n\big[\mathbf{ident}[\![\mathrm{u}]\!] \mapsto \mathbf{expr}[\![\mathrm{u} - \mathrm{v}]\!](env_n)\big](env_3) = \\
&= \lambda env_n.env_n\big[u \mapsto \mathbf{expr}[\![\mathrm{u}]\!](env_n)\ \mathbf{binop}[\![-]\!]\ \mathbf{expr}[\![\mathrm{v}]\!](env_n)\big](env_3) = \\
&= \lambda env_n.env_n\big[u \mapsto env_n(\mathbf{ident}[\![\mathrm{u}]\!]) - env_n(\mathbf{ident}[\![\mathrm{v}]\!])\big](env_3) = \\
&= \lambda env_n.env_n\big[u \mapsto env_n(u) - env_n(v)\big](env_3) = \\
&= \lambda env_n.env_n\big[u \mapsto 6 - 2\big] = \\
&= \lambda env_n.env_n\big[u \mapsto 4\big] \qquad (3.17)
\end{aligned}
$$

It follows from the structure of the CFG of Figure 3.3 that control flow continues from state s_2 via edges e_3, e_4, and e_5. As a consequence, the branch predicates of those edges evaluate to *true*. Otherwise the computation would come to a halt at one of the states s_2, s_3, or s_4, which would contradict Equation (3.3).

For this reason we may restrict ourselves to the calculation of the corresponding environments for the remaining iterations of the iterated transition function δ^*. Evaluation (3.16) shows that environment env_3 is derived from env_2 by updating environment env_2 with the new value 2 at "v". This is the semantic effect of the assignment statement of edge e_3. Likewise Evaluation (3.17) shows the derivation of environment env_4 from env_3. Finally, edge e_5 has the identity function ι as side-effect. Hence environment env_x does not differ from env_4.

A comparison of the graphs of environments env_e and env_x of Figure 3.3 shows that for environment env_x the values of "u" and "v" have been swapped. This was the intention for our first Flow example program. It is left as an exercise to the interested reader to verify that the Ada program of Figure 3.2 (b) has the same semantic behavior when passed the parameters $u = 2$ and $v = 4$.

In concluding this section we note that the side-effects computed by the iterated transition function δ^* along program path $\pi_1 = \langle e_1, e_2, e_3, e_4, e_5 \rangle$ correspond to the function composition

$$\sigma(e_5) \circ \sigma(e_4) \circ \sigma(e_3) \circ \sigma(e_2) \circ \sigma(e_1)(env_e).$$

3.5 Turing-Equivalence of the Language Flow

Wherever you go, you go with the flow...
(In praise of the river.)
— From the movie "The Wind in the Willows", based on a novel by Kenneth Grahame.

In this section we finally show that the computational model of the language Flow is equivalent to the computational model of a Turing machine. This will be achieved in three steps.

We start with the definition of a Turing machine that essentially corresponds to the "basic model" of a Turing machine from [HU79]. In the second step we show how such a Turing machine can be represented by a so-called transition diagram. Finally we map transition diagrams to Flow control flow graphs, thereby showing that the computational capability of Flow programs is no less than that of a Turing machine. Conversely, the reduction from Flow programs to Turing machines is immediate.

3.5.1 Turing Machine Notation

The basic model of a Turing machine has finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time. The tape has a *rightmost* cell but is infinite to the *left*. Each cell of the tape may hold exactly one of a finite number of tape symbols. Initially, the n rightmost cells, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input symbols. The remaining infinity of cells each hold the blank, which is a special tape symbol that is not an input symbol. Initially the tape head is at the rightmost cell that holds the input. A Turing machine accomplishes three tasks in one move, depending on the symbol scanned by the tape head and the state of the finite control: it changes state, prints a symbol on the tape cell scanned, and moves its head left (L) or right (R) one cell.

Formally a Turing machine is denoted by

$$M = (Q, \Sigma, \Gamma, \delta_{\mathrm{TM}}, q_0, B, F)$$

where

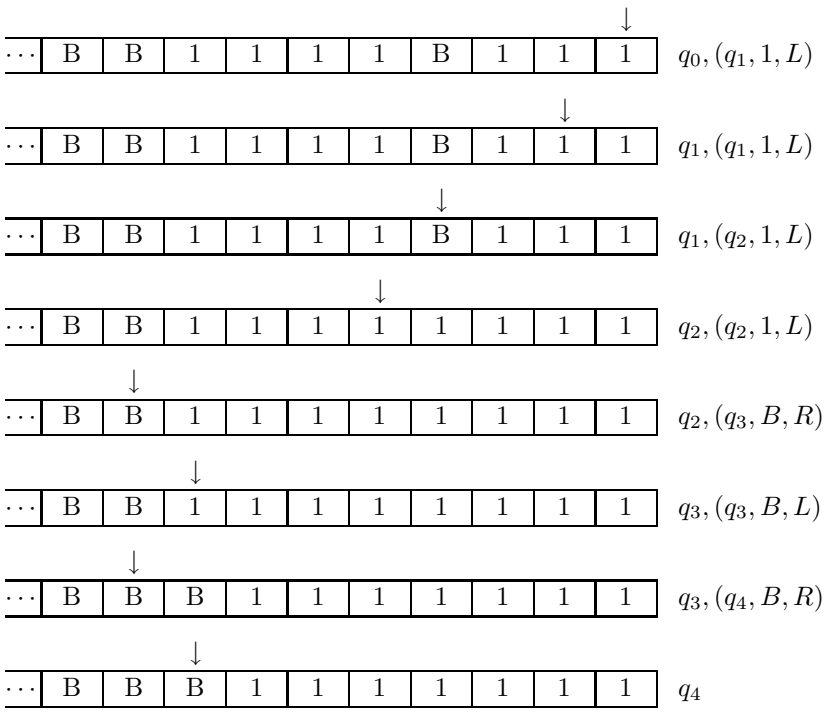
- Q is the finite set of states,
- Γ is the finite set of allowable tape symbols,
- $B \in \Gamma$ is the blank,
- $\Sigma \subseteq \Gamma \setminus \{B\}$ is the set of input symbols,
- δ_{TM} is a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ and determines the next move (although δ_{TM} may be undefined for some arguments),
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of final states.

For our purpose we restrict Γ to the set {1,B}. This does not affect the capabilities of the formalism since we can apply *unary* encoding to the input. However, in order to keep the examples used in this discussion simple and illustrative, we allow the blank symbol to separate strings of input symbols.

	Symbol	
State	B	1
q_0	(q_0, B, L)	$(q_1, 1, L)$
q_1	$(q_2, 1, L)$	$(q_1, 1, L)$
q_2	(q_3, B, R)	$(q_2, 1, L)$
q_3	(q_4, B, R)	(q_3, B, L)
q_4	—	—

Figure 3.4: Transition Function δ_{TM}

Example 3.1 We consider a Turing machine which adds two numbers given as input in the form of two strings of "1"s, separated by a blank[†]. The algorithm works by replacing the separating blank by "1", and the leftmost "1" by a blank. Figure 3.4 depicts the transition function δ_{TM} for this example. Therein q_0 denotes the start state, and q_4 the final state. Below we depict a finite part of the input tape during several phases of a sample computation. In addition we show the position of the tape head (↓), the current state $q_i \in Q$, and the corresponding value of the transition function δ_{TM}.



[†]This example is taken from [Lei03].

3.5.2 Turing Machine Transition Diagrams

We can represent the transitions of a Turing machine as defined in the previous section as a transition diagram (cf. also [HMU01, Section 8.2.4]). The nodes of the transition diagram correspond to the states of the Turing machine, and for simplicity we use the same names. Nodes are connected by directed edges e that are labeled

$$e : s_1 \Rightarrow s_2; D, \tag{3.18}$$

where $s_1, s_2 \in \Gamma$ are tape symbols, and $D \in \{L, R\}$ denotes a direction. Nodes q_u and q_v are connected by an edge e, iff

$$\delta_{\mathrm{TM}}(q_u, s_1) = (q_v, s_2, D). \tag{3.19}$$

Figure 3.5 below depicts the transition diagram for Example 3.1.

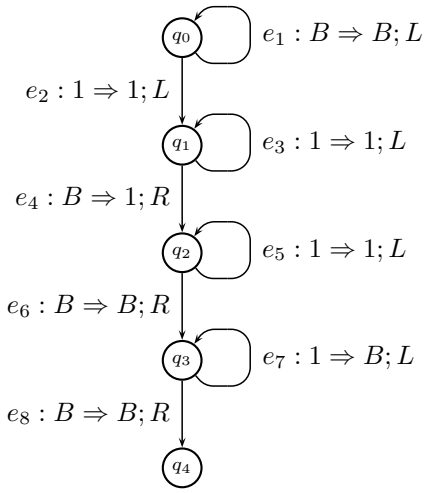


Figure 3.5: Transition Diagram for Example 3.1

3.5.3 Mapping Transition Diagrams to Flow Graphs

It remains to show how a transition diagram can be mapped to a Flow control flow graph. The position of the tape head and the tape itself constitute the state of the computation of a Turing machine, which we can encode with an environment $env_{\mathrm{TM}} \in \mathit{Environment}$ of two variables, namely c and T. We let variable c (the counter) represent the position of the tape head, and $T \in \mathbb{Z}$ represents the tape itself. Given the binary representation of $T \in \mathbb{Z}$, we define that a blank (B) is represented by 0, and that the input symbol 1 is represented by the digit 1. In this way the i^{th} cell ($i \geq 0$) of the input tape corresponds to bit i of T.

The alert reader will have noticed that there is a discrepancy between the number $T \in \mathbb{Z}$, which is per definition finite, and the infiniteness of the input tape. We will resolve this problem shortly; For the time being we *assume* that T magically grows and shrinks with the size of the problem at hand.

In the beginning, the n rightmost cells of the tape, for some finite $n \geq 0$, hold the input. Our first task is to transfer this initial configuration to the

environment env_{TM}. Since n is finite, we can build a predicate $\mathrm{B}(i)$ which is *true* iff the i^{th} rightmost cell of the tape contains the symbol 1. We request the following conditions for our initial environment env_{TM} to hold.

$$(T \operatorname{rem} 2^{i+1}) \geq 2^i \Leftrightarrow \mathrm{B}(i) \tag{3.20}$$

$$c = 0 \tag{3.21}$$

Equation (3.20) states that a bit in the binary representation of T is set iff the corresponding cell of the tape contains the symbol 1. In Equation (3.21) we define the initial position of the counter c to be on the rightmost cell of the tape.

The left side of Equation (3.20) provides us a means to test for bit i of T, which we can also use to set and clear bits. That is, given a T' where bit i is not set, we can set it by adding 2^i to T'. Conversely, given a T' where bit i is set, we can clear it by subtracting 2^i from T'. In our emulation of a Turing machine these operations correspond to a test for the symbol at the position i of the tape head, and to the printing of a symbol on the scanned tape cell. An increment of the counter c by one corresponds to a move of the tape head to the left, whereas decrementing c by one corresponds to a move of the tape head to the right.

The initial configuration of the tape of Example 3.1 together with the positions of the respective tape cells are shown below.

										↓
$\cdots$	0	0	1	1	1	1	0	1	1	1
$\cdots$	9	8	7	6	5	4	3	2	1	0

With this example the initial input is restricted to cells 0–7, and the value of $(T)_{10}$ is 247. From this example it becomes clear that all but finitely many cells of our tape-representation are virtual in the sense that they are not contained in T. To be specific, we drop infinitely many trailing blank cells. It must however be noted that T in conjunction with Equation (3.20) yields the correct result for the general case $i \geq 0$.

Suppose now that in the course of a computation the tape counter is advanced past position ω denoting the leftmost position "physically" represented by T ($\omega = 7$ in the above example). Advancing the tape head from position ω to position $\omega + 1$ is just an act of incrementing the counter c, which does not affect T. Testing for the content of the cell at position $\omega + 1$ does not affect T either, despite the fact that it will yield the correct result (a blank). Assume now that our emulation of a Turing machine is smart and writes to its tape only in case when the new symbol computed by the transition function δ_{TM} differs from the scanned symbol. In "writing" blanks a computation can then advance the tape head k steps to the left without affecting T. But as soon as the computation requires to write a 1 to tape cell $\omega + k$, the addition of $2^{\omega+k}$ to T extends T to and including position $\omega + k$.

The opposite case is also possible: suppose again that position ω denotes the leftmost position "physically" represented by T. This means that the cell at position ω is the leftmost cell containing a non-blank symbol. Suppose furthermore that the next non-blank cell from position ω to the right is at position $j < \omega$. If the cell at position ω is rewritten with a blank, the subtraction of 2^{ω} shrinks T

to and including position j. The only exception to this rule is the case where we rewrite the last non-blank symbol of the tape by a blank, in which case T becomes zero.

Drawing from the countable set $\mathbb{Z}$ of integers we have managed to represent the infinite tape of a Turing machine within the finite representation of env_{TM}. In doing so we have shown that the size of T indeed grows and shrinks with the problem size, which resolves the discrepancy raised in the introduction of this section.

It remains now to show how the transition function δ_{TM} can be emulated by a Flow graph with a Flow standard semantic iterated transition function that, when applied to the initial environment env_{TM}, computes the same result as δ_{TM}. As already announced we use the intermediate step via the transition diagrams of Section 3.5.2 to establish this mapping.

Strictly speaking, we can regard a transition diagram as a Flow control flow graph, if we can overcome the following obstacles.

(1) The edges of a transition diagram need to be mapped to Flow graph edges.

(2) The entry node n_e of a Flow graph is a distinct node without incoming edges, i.e. $\mathrm{in}(n_e) = \emptyset$. This is not required for the start state of a Turing machine (cf. e.g., Figure 3.5).

(3) The exit node n_x of a Flow graph is a distinct node without outgoing edges, i.e. $\mathrm{out}(n_x) = \emptyset$. In contrast, a Turing machine has a set $F \subseteq Q$ of final states.

In order to solve the first topic in the above list, we examine the different types of edges that may arise due to the transition function δ_{TM}. Table 3.3

(1)	$(q_u, 1)$	$\rightarrow$	$(q_v, 1, L)$	(5)	$(q_u, 1)$	$\rightarrow$	(q_v, B, L)
(2)	$(q_u, 1)$	$\rightarrow$	$(q_v, 1, R)$	(6)	$(q_u, 1)$	$\rightarrow$	(q_v, B, R)
(3)	(q_u, B)	$\rightarrow$	(q_v, B, L)	(7)	(q_u, B)	$\rightarrow$	$(q_v, 1, L)$
(4)	(q_u, B)	$\rightarrow$	(q_v, B, R)	(8)	(q_u, B)	$\rightarrow$	$(q_v, 1, R)$

Table 3.3: Enumeration of δ_{TM} Transitions from State q_u to State q_v

enumerates the different transitions possible from state q_u to state q_v. Each table-entry corresponds to an application of δ_{TM} in the sense of Equation (3.19).

Each of these transitions can occur as an edge of a transition diagram, and we will be occupied in the following to find a Flow resemblance for them. We can partition the entries from Table 3.3 in two sets with similar properties. Entries (1)–(4) constitute transitions where the scanned symbol is written back to the tape. As mentioned above, our Flow Turing machine emulation does not execute a write operation on T in such a case, which reduces these cases to a mere move of the tape head. Entries (5)–(8) constitute transitions where the scanned symbol differs from the symbol written back to the tape. Hence the entries of this group involve a write operation on T as well as a move of the tape head.

For one member of each of these groups we will show the Flow graph portion corresponding to the respective transition diagram edge. The remaining cases are solved by analogy.

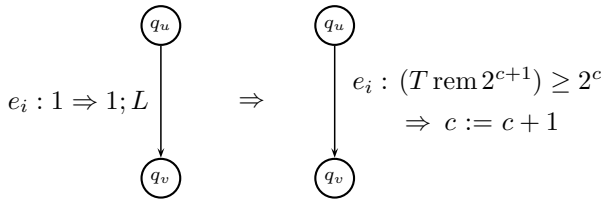


Figure 3.6: Case (1) of Table 3.3

Consider Figure 3.6 which treats Entry (1) of Table 3.3. This entry resembles the case where the tape head scans 1, writes back 1 to the tape, and moves the tape head left. The Flow graph portion to the right of Figure 3.6 does the right thing: if the tape head scans a 1 (expressed by the branch predicate $(T \operatorname{rem} 2^{c+1}) \geq 2^c$, the tape head is moved left by incrementing the tape counter c by one.

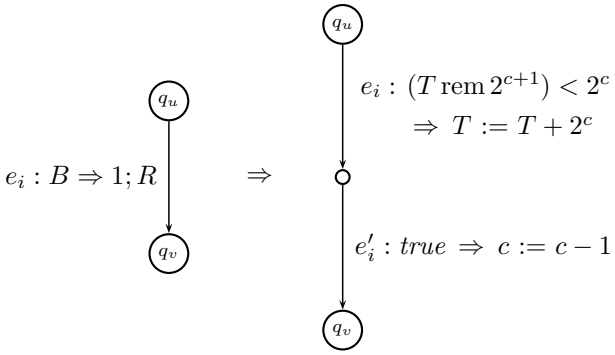


Figure 3.7: Case (8) of Table 3.3

Now consider Figure 3.7 which treats Entry (8) of Table 3.3. This entry resembles the case where the tape head scans a blank, writes back 1 to the tape, and moves the tape head right. We need two edges e_i and e'_i to support both the update of T and c. The Flow graph portion to the right of Figure 3.7 acts according to Entry (8) of Table 3.3: if the tape head scans a blank (expressed by the branch predicate $(T \operatorname{rem} 2^{c+1}) < 2^c$, we write back the symbol 1 (expressed by the side-effect of edge e_i). Thereafter the side-effect of edge e'_i advances the tape head to the right.

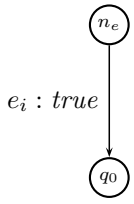


Figure 3.8: The Flow Graph Entry Node of the Translation

Coming back to the above list of items to be overcome, we can resolve the second one by introducing the Flow graph portion depicted in Figure 3.8. It does nothing more than jump unconditionally to the start state q_0, without any

other side-effect.

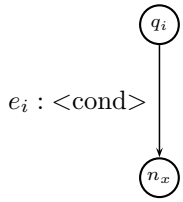


Figure 3.9: The Flow Graph Exit Node of the Translation

Finally, the third item in the above list is a mere technicality which is rooted in the fact that the transition function δ_{TM} is a partial function in the sense that it need not provide values for all input symbols in any state. Hence a Turing machine may abandon its computation in any state, including the start state q_0. For the ongoing discussion we shall call a state in which a Turing machine may abandon its computation a *halting state* (as already indicated, the set of halting states need not coincide with the set F of final states). Since we have required a distinct exit node n_x for Flow graphs, we have to provide n_x itself and insert an edge of the type depicted in Figure 3.9 from any Flow graph node that corresponds to a halting state of the Turing machine to node n_x. Determining the halting states of a Turing machine is equivalent to spotting the values $(q, x) \in Q \times \Gamma$ in the domain of the transition function δ_{TM}, for which δ_{TM} is not defined (cf. also Figure 3.4). The condition <cond> in Figure 3.9 is due to the fact that a state may be a halting state only for a subset of the possibly scanned symbols from Γ.

A final note is due to nonterminating computations on Turing machines. Since the set Q of states of a Turing machine M is finite, a nonterminating computation implies a cycle in the transition diagram of M which in turn implies a cycle in the corresponding Flow graph. Moreover, as the computation does not terminate, this cycle is iterated over and over. A computation of this kind on a Cele in a Flow control flow graph is commonly referred to as an *endless loop*. Often the properties of the cycle alone already imply a nonterminating computation (that means, nonterminating for every possible input), in which case the cycle itself is also termed an endless loop.

Summing up, we have shown how a Turing machine M given in the form of a transition diagram as introduced in Section 3.5.2 can be mapped to a Flow graph by replacing distinct transition diagram portions by the respective Flow graph counterparts. Adding entry and exit nodes concludes the case. The result of the iterated transition function δ^* of the resulting Flow program, applied to the initial environment env_{TM}, coincides with the computation of the Turing machine M. This leads us to the following theorem.

Theorem 1. The computational capability of the Flow language is equivalent to the computational capability of a Turing machine.

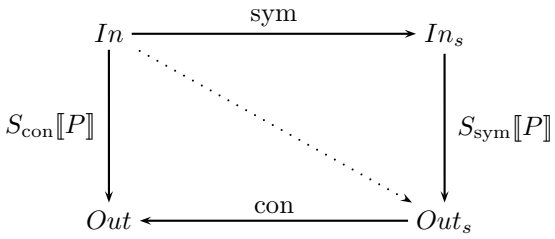
Proof. The reduction from Turing machines to Flow graphs follows immediately from structural induction on Turing machines with k states, using the mapping of Turing machines to Flow graphs established in this section. The reduction from Flow graphs to Turing machines is immediate, e.g., by translating Flow graphs to a representation in an imperative programming language. □

Chapter 4

Semantics of Symbolic Program Execution

The purpose of abstracting is not *to be vague,*
but to create a new semantic level in which one can be absolutely precise.
— Edsger W. Dijkstra, in "The Humble Programmer", 1972 Turing Award Lecture

In this chapter we develop a semantic model for symbolic program execution. The main difference to the standard semantics presented in the previous chapter is that in this model the value of a variable v is described by a symbolic expression instead of a concrete value $z \in \mathbb{Z}$. The relation between standard semantics and semantics of symbolic program execution is depicted in Diagram (4.1).



(4.1)

Therein the standard semantics of a program P is derived by the valuation function S_{con} that takes a program P as argument and returns a standard-semantic functional description of the so-called *side-effect* of P. The side-effect $S_{\text{con}}[\![P]\!]$ is a function that maps a concrete input to a concrete output, written as $S_{\text{con}}[\![P]\!](In) = Out$, where $In, Out \in Environment$. We have defined the standard semantics of Flow programs in Chapter 3, and the computation of $S_{\text{con}}[\![P]\!](In)$ is equivalent to an application of the iterated transition function δ^* to the start state s_e and environment In, denoted by $\delta^*(s_e, In)$.

Likewise we derive the semantics of symbolic program execution from program P, denoted by $S_{\text{sym}}[\![P]\!]$. It is the purpose of function S_{sym} to transform P into a representation that is based on symbolic values instead of concrete ones. The side-effect $S_{\text{sym}}[\![P]\!]$ of this representation is therefore a function that maps a symbolic input In_s to the corresponding symbolic output Out_s. Symbolic input and output belong to the class $Environment_S$ of *symbolic environments*

that replaces the concrete environments $env : \mathbb{V} \to \mathbb{Z}$ which are not able to bind identifiers to symbolic expressions.

$$In_s, Out_s \in Environment_S \tag{4.2}$$

We will develop the properties of symbolic environments in the course of this chapter. Diagram (4.1) contains two additional functions, sym and con, that we need in order to relate input and output of the functional descriptions $S_{\text{con}}[\![P]\!]$ and $S_{\text{sym}}[\![P]\!]$. Function sym transfers a concrete environment to the symbolic domain, whereas function con *instantiates* a symbolic environment with a concrete one.

We can now proclaim the commutation of concrete and symbolic execution depicted in Diagram (4.1). If a program P is conventionally executed with the standard semantics $S_{\text{con}}[\![P]\!]$ over a given input In, the result of the symbolically executed program $S_{\text{sym}}[\![P]\!]$ over input $In_s = \text{sym}(In)$ and instantiated by In is the same.

$$S_{\text{con}}[\![P]\!](In) = \text{con}\big(In, S_{\text{sym}}[\![P]\!]\big(\text{sym}(In)\big)\big) \tag{4.3}$$

We will prove this commutation property of concrete and symbolic execution in Section 4.4.

As a prerequisite we need to develop the methodology required to compute $S_{\text{sym}}[\![P]\!]$. At the heart of our analysis is the control flow graph together with the standard-semantic model of the language Flow (cf. Section 3.2). As a consequence we are investigating a forward data-flow problem, and the mapping of a program P to a control flow graph is that of Figure 2.3(b) on page 18.

We derive side-effects from program-paths along the edges of control flow graphs. We start in Section 4.2 with the side-effect along a single CFG edge e. In Section 4.3 we extend this result to whole program paths. Section 4.4 addresses the combined side-effects along several program paths and the proof of Condition (4.3).

To begin with, the following section defines the symbolic expression domain that serves as a counterpart to the arithmetic and relational expressions of the Flow standard semantics.

4.1 The Domain of Symbolic Expressions

Symbolic expressions are central to the concept of symbolic program execution, because we will use them to describe the values of program variables and computations. The introduction of this chapter already hints into this direction in stating that symbolic environments env_s are needed in order to bind symbolic expressions to program variables.

In this section we develop the semantic domain SymDom of symbolic expressions and symbolic predicates of the Flow language. As with the preceding definitions of semantic domains, this initial view will be purely mathematical in the sense that we do not consider any issues of computer representation of mathematical objects here. In Chapter 5 we will have to exchange this abstract view with a more concrete one that takes into account representational issues as well.

Since not all issues are related to the same level of detail, we will in fact use a hierarchy of abstractions, with the before-mentioned initial view at the top.

This hierarchical view is due to Geddes et al. [GCL92, Chapter 3] and comprises the following levels of abstraction.

1. The *object* level is the abstract level where the elements of a domain are considered to be primitive objects. Our semantic domains are located at this level.

2. The *form* level is the level of abstraction in which the representation of an object in terms of "basic symbols" is treated. At this level it is recognized that a given object may have different valid form-level representations in terms of the chosen symbols. In this way we distinguish on this level between the following different representations of a given bivariate polynomial $\mathrm{p}(x, y)$:

$$\mathrm{p}(x, y) = 12x^2y - 4xy + 9x - 3 \tag{4.4}$$
$$\mathrm{p}(x, y) = (3x - 1)(4xy + 3). \tag{4.5}$$

3. The *data structure* level is concerned with the representation of an object inside a computer.

As already pointed out, we will focus in this section on the object level. In Chapter 5 we will address issues relevant to the form level. We will leave out the data structure level altogether due to its commonalities with the area of computer algebra systems. An in-depth coverage of data structure level issues that can be utilized for our domain of symbolic expressions and predicates is given by Davenport et al. in [DST93].

4.1.1 The Integer-Valued Symbolic Expression Domain

It is worth noting that the arithmetic capabilities in the *standard semantic model* of our Flow language are based on the integer number system, constituting an *integer arithmetic*. This is exemplified by the definition of the standard semantic domain in Table 3.1, where variable identifiers and arithmetic operators range over the integers. Transferring this property to the symbolic domain, we have to require that symbolic expressions corresponding to the standard semantic arithmetic model of the Flow language have to be *integer-valued*.

Definition 4.1 An expression e in the variables $v_1, \ldots, v_n$, henceforth denoted by $\mathrm{e}(v_1, \ldots, v_n)$, is integer-valued in the sets $S_{v_1}, \ldots, S_{v_n}$, iff

$$\underset{1 \le i \le n}{\forall} v_i \in S_{v_i} : \ \mathrm{e}(v_1, \ldots, v_n) \in \mathbb{Z}.$$

Using this definition we may omit the explicit mention of the sets S_{v_i} if they are clear from context. For brevity we might also write $e(\mathbf{v})$, where $\mathbf{v} = (v_1, \ldots, v_n)$.

Example 4.1 The expression $\mathrm{e}(x) = \frac{x(x+1)}{2}$ is integer-valued in the set $S_x = \mathbb{Z}$, since for every integer $x \in S_x$, either $x - 1$ or x is even; hence $2 \,|(x(x + 1))$.

Furthermore, we need a means to distinguish between a variable and its *initial value* to avoid the circularity of describing the value of a variable in terms of the variable itself. As an example, consider the initial *concrete* environment env_e with

$$\mathrm{graph}(env_e) = \{(u, 2), (v, 4)\}$$

of our Flow example program presented in Section 3.4. One possibility to define an initial *symbolic* environment env_s would be to consider variables initially as *undefined*, denoted by the value "$\perp$" that is distinct from all other values of the domain of integers:

$$\text{graph}(env_s) = \{(u, \perp), (v, \perp)\}.$$

This approach is however too unspecific to be of any practical use; Consider the statement u := u+v that adds the values of variables u and v deteriorating to

$$\text{u} := \text{u+v} = \perp + \perp = \perp.$$

In losing the information that variable u amounts to the sum of the values of variables u and v, we fall back to the unpleasant state of knowing nothing regarding the value of u (except that it is undefined).

From this example it becomes clear that we have to be able to distinguish between the initial values of different program variables. Talking about values it seems sensible to represent an initial value of a program variable v as a named constant related to v. It should however be noted that in the general case we have no prior knowledge on the initial value of v (e.g., if v is a formal parameter of a procedure). In this way the initial value can be any value in the domain of v and it is therefore more appropriate to have the initial value itself provided by a variable. Given the set $\mathbb{V}$ of program variables, we can define a set $\underline{\mathbb{V}}$ that is isomorphic to $\mathbb{V}$ and contains the variables providing the initial values for the variables in $\mathbb{V}$. The initial value operator

$$\_ : \mathbb{V} \to \underline{\mathbb{V}}$$

maps a variable $v \in \mathbb{V}$ to the corresponding variable in $\underline{\mathbb{V}}$ that provides the initial value of v. As a shorthand notation we write $\underline{v}$ for the application of the initial value operator to variable v.

We can utilize multivariate polynomials over the integers $\mathbb{Z}$ as the underlying algebraic structure for symbolic expressions. Let us consider the general case of a *multivariate* polynomial in $\nu \geq 1$ indeterminates. For any commutative ring R, the notation $R[x_1, \ldots, x_\nu]$, or $R[\mathbf{x}]$ where $\mathbf{x} = (x_1, \ldots, x_\nu)$, denotes the set of all expressions of the form

$$a(\mathbf{x}) = \sum_{e \in \mathbb{N}^\nu} a_e \mathbf{x}^e$$

with $a_e \in R$, where only a finite number of coefficients a_e are nonzero. Due to $(\mathbb{Z}; +, \cdot)$ constituting not only a ring but also an Euclidean domain, we can use this Euclidean domain to define the set of multivariate polynomials $\mathbb{Z}[\mathbf{x}]$ over the integers $\mathbb{Z}$ in the indeterminates $\mathbf{x}$. To relate these polynomials to Flow programs, we require that the indeterminates $\mathbf{x} = (x_1, \ldots, x_\nu)$ are program variables expressed through their initial value variables, that is,

$$\underset{1 \leq i \leq \nu}{x_i} \in \underline{\mathbb{V}}. \tag{4.6}$$

With the operations of addition and multiplication on multivariate polynomials, $\mathbb{Z}[\mathbf{x}]$ constitutes a unique factorization domain [GCL92, Theorem 2.7].

Clearly all polynomials p($\mathbf{x}$) from the unique factorization domain $\mathbb{Z}[\mathbf{x}]$ are integer-valued expressions in the sets $\mathbb{Z}_{\mathbf{x}} = (\mathbb{Z}_{x_1}, \ldots, \mathbb{Z}_{x_\nu})$.

To support division in an integral domain, it has to be extended to a field. For a polynomial domain $D[\mathbf{x}]$ over a unique factorization domain D, the quotient field $Q(D[\mathbf{x}])$ is called the field of rational functions over D in the indeterminates $\mathbf{x}$. Elements of $Q(D[\mathbf{x}])$ are (equivalence classes of) quotients of the form

$$a(\mathbf{x}) \,/\, b(\mathbf{x}), \text{ where } a(\mathbf{x}), b(\mathbf{x}) \in Q(D[\mathbf{x}]) \text{ with } b(\mathbf{x}) \neq 0.$$

The operations of addition and multiplication in $D[\mathbf{x}]$ can be extended to the quotient field $Q(D[\mathbf{x}])$ as follows. If $a(\mathbf{x})/b(\mathbf{x})$ and $c(\mathbf{x})/d(\mathbf{x})$ in $Q(D[\mathbf{x}])$, then

$$\big(a(\mathbf{x})/b(\mathbf{x})\big) + \big(c(\mathbf{x})/d(\mathbf{x})\big) = \big(a(\mathbf{x})d(\mathbf{x}) + b(\mathbf{x})c(\mathbf{x})\big) \,/\, b(\mathbf{x})d(\mathbf{x}) \tag{4.7}$$

$$\big(a(\mathbf{x})/b(\mathbf{x})\big) \cdot \big(c(\mathbf{x})/d(\mathbf{x})\big) = a(\mathbf{x})c(\mathbf{x}) \,/\, b(\mathbf{x})d(\mathbf{x}). \tag{4.8}$$

As a result, we can extend the unique factorization domain $\mathbb{Z}[\mathbf{x}]$ to a quotient field $Q(\mathbb{Z}[\mathbf{x}])$. Unfortunately the resulting rational functions are not integer-valued expressions in the sets $\mathbb{Z}_{\mathbf{x}}$ in the general case. This means that the division operator / of the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ is not suitable to take the role of the Flow integer division operator div.

Example 4.2 The rational function x/y is not an integer-valued expression in the sets $S_x = \mathbb{Z}$, $S_y = \mathbb{Z}$. Neglecting this fact could lead to the cancellation of the variable y in the expression $y \cdot (x/y) = \not{y} \cdot (x/\not{y}) = x$, which is incorrect within the arithmetic of integers, e.g., $4 \cdot (3/4) = 4 \cdot 0 = 0 \neq 3$.

By means of a rounding operation we can "wrap" the rational function x/y from the above example to obtain an integer-valued expression. For this purpose we introduce the following rounding operation in anticipation of the semantic domain SymExpr of symbolic expressions whose introduction will follow thereafter.

Definition 4.2 Let the rounding operation Rnd of arity

$$\mathrm{Rnd} : \mathrm{SymExpr} \to \mathrm{SymExpr}$$

implement the desired rounding mode, selected from Equations (3.8), (3.9), or (3.10) on page 23. It yields an integer-valued result and is therefore idempotent, i.e.

$$\mathrm{Rnd}\big(\mathrm{Rnd}(e)\big) = \mathrm{Rnd}(e).$$

If we select rounding towards zero, the rounding mode applied with the Flow integer arithmetic, then we must be able to determine whether the argument of the rounding operation is less than zero. This decision, which cannot be made based on the symbolic expression alone, will further occupy us in Section 5.5.2. In the meantime we have to carry on keeping such expressions unevaluated and treat them like single independent variables, e.g.,

$$\mathrm{Rnd}(x/y) + \mathrm{Rnd}(x/y) = 2 \cdot \mathrm{Rnd}(x/y).$$

Given the rounding operation Rnd, the unique factorization domain $\mathbb{Z}[\mathbf{x}]$, and the quotient field $Q(\mathbb{Z}[\mathbf{x}])$, we can now define the semantic domain SymExpr of symbolic expressions. Let

$$f^{(n)} \in \{+^{(2)}, -^{(2)}, -^{(1)}, \cdot^{(2)}\}$$

denote functions $f^{(n)}$ corresponding to the Flow arithmetic operations of addition, subtraction, change of sign, and multiplication, where (n) denotes the respective arity. Contrary to their standard semantic counterparts, these functions operate on values of the domain SymExpr, e.g.,

$$+^{(2)} : \text{SymExpr} \times \text{SymExpr} \to \text{SymExpr}.$$

The same holds, as already mentioned, for the rounding operation Rnd.

Definition 4.3 The set of symbolic expressions of the domain SymExpr is inductively defined as

(i) $\mathbb{Z}[\mathbf{x}] \subset \text{SymExpr}$ (i.e. every polynomial in $\mathbb{Z}[\mathbf{x}]$ is a symbolic expression),

(ii) for all $f^{(n)}$ and all $e_1, \ldots, e_n \in \text{SymExpr}$, we have $f^{(n)}(e_1, \ldots, e_n) \in \text{SymExpr}$ (i.e., application of functions $f^{(n)}$ to symbolic expressions yields symbolic expressions),

(iii) for all $e_1, e_2 \in \text{SymExpr}$, we have $e_1/e_2 \in \text{SymExpr}$, iff e_1/e_2 is an integer-valued symbolic expression,

(iv) for all $e_1, e_2 \in \text{SymExpr}$, we have $\text{Rnd}(e_1/e_2) \in \text{SymExpr}$, and consequently, $\lceil e_1/e_2 \rceil \in \text{SymExpr}$, and $\lfloor e_1/e_2 \rfloor \in \text{SymExpr}$.

The functions $f^{(n)}$ constitute the corresponding operations on multivariate polynomials and rational functions, with the only extension that they do accept arguments "wrapped" by the rounding operator Rnd or by the *floor* and *ceiling* functions.

From Equations (4.7) and (4.8) it follows that the result of a sum or product of two rational functions that are integer-valued in $\mathbb{Z}$ is again integer-valued in $\mathbb{Z}$. As can be seen from the following example, this need not hold for the division of integer-valued rational functions:

Example 4.3

$$e(n) = \frac{n \cdot (n-1)}{2} \,/\, \frac{n \cdot (n+1)}{2} = \frac{n-1}{n+1}, \;\; e(2) = \frac{1}{3} \notin \mathbb{Z}.$$

We have already pointed out that the properties of the division operator / of the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ do not permit it as the counterpart of the integer division operator div of the Flow standard semantics. For this reason we will investigate properties of symbolic integer division and remainder operations in the following section.

The Symbolic Integer Division and Remainder Operations

In utilizing the division operator / of the quotient field $Q(\mathbb{Z}[\mathbf{x}])$, we can model the integer division of two symbolic expressions $e_1(\mathbf{x})$ and $e_2(\mathbf{x})$, $e_2(\mathbf{x}) \neq 0$, as

$$e_1(\mathbf{x}) \text{ div}_\text{s } e_2(\mathbf{x}) = \text{Rnd}\big(e_1(\mathbf{x}) \,/\, e_2(\mathbf{x})\big), \tag{4.9}$$

where the symbolic division operator div_s denotes the counterpart of the integer division operator div of the Flow standard semantics. Likewise, for $e_2(\mathbf{x}) \neq 0$, we get

$$e_1(\mathbf{x}) \text{ rem}_\text{s } e_2(\mathbf{x}) = e_1(\mathbf{x}) - e_2(\mathbf{x}) \cdot \big(e_1(\mathbf{x}) \text{ div}_\text{s } e_2(\mathbf{x})\big) \tag{4.10}$$

for the symbolic remainder operator $\mathrm{rem_s}$. Since the rounding operation Rnd involves floor and ceiling functions, the expressions resulting from the symbolic division and remainder operations are rather inflexible (we can move integer terms in and out of floor and ceiling functions, but little more (cf. [GKP94]). We are therefore happy if we can do without them, which makes the following special cases of symbolic integer division especially appealing simplifications.

Case 1: $e_1(\mathbf{x}) = a(x)$ and $e_2(\mathbf{x}) = b(x) \neq 0$ are univariate integer-valued polynomials in the same indeterminate in the polynomial domain $\mathbb{Q}(x)$ over the field $\mathbb{Q}$ of rational numbers. Since $\mathbb{Q}$ is a field, it follows from [GCL92, Theorem 2.5 (v)] that $\mathbb{Q}(x)$ is a Euclidean domain.

On the analogy of integer division (cf. Equation (3.7) on p. 21), this domain facilitates the division of $a(x)$ by $b(x)$ to obtain a quotient polynomial $q(x)$ and a remainder polynomial $r(x)$ satisfying

$$a(x) = q(x) \cdot b(x) + r(x), \qquad \deg(\mathrm{r(x)}) < \deg(\mathrm{b(x)}), \tag{4.11}$$

where $\deg(\ldots)$ denotes the *degree* of a polynomial. Contrary to the integer division, the quotient and remainder polynomials satisfying the above relation are unique. They can be determined by polynomial long division, using e.g. Algorithm D of [Knu97, Section 4.6].

We are interested in the quotient polynomial $q(x)$ to simplify the symbolic division and remainder operations of Equation (4.9) and (4.10). In order to avoid solutions in $x \in \mathbb{Z}$, where the remainder $r(x)$ gets arbitrarily large, we restrict our interest (and hence the applicability of the intended simplification) to values of the indeterminate x satisfying the *side-condition*

$$\left|\frac{r(x)}{b(x)}\right| \leq \frac{1}{2}. \tag{4.12}$$

Lemma 1. Given the side-condition stated in Equation (4.12), we have

$$\left|\mathrm{Rnd}\left(\frac{a(x)}{b(x)}\right) - \mathrm{RndN}\big(q(x)\big)\right| \leq 1,$$

with RndN denoting the *round to nearest integer* rounding function.

Proof. From Equation (4.11) we get

$$\frac{a(x)}{b(x)} = q(x) + \frac{r(x)}{b(x)}.$$

Due to the side-condition stated in Equation (4.12), the result of dividing polynomial $a(x)$ by $b(x)$ in the quotient field $Q(\mathbb{Q}[\mathbf{x}])$ is contained in the interval $[q(x) - 0.5,\ q(x) + 0.5]$. This interval includes two integer numbers u_1 and u_2, if $q(x)$ is located exactly between u_1 and u_2, and one integer number u in all the other cases. Treating the latter case first, we have $u = \mathrm{RndN}\big(q(x)\big)$. Then $\frac{a(x)}{b(x)} \in [q(x) - 0.5,\ q(x) + 0.5]$ implies that $\mathrm{Rnd}\big(\frac{a(x)}{b(x)}\big) \in \{u - 1, u, u + 1\}$, which satisfies Lemma 1.

In the case where the interval $[q(x) - 0.5,\ q(x) + 0.5]$ contains two integers u_1 and u_2, we have $[q(x) - 0.5,\ q(x) + 0.5] = [u_1,\ u_2]$. ($u_1$ and u_2 constitute the boarders of the interval.) Hence $\mathrm{RndN}\big(q(x)\big) \in \{u_1,\ u_2\}$, and due to the side-condition, $\mathrm{Rnd}\big(\frac{a(x)}{b(x)}\big) \in \{u_1,\ u_2\}$. □

The basic meaning of Lemma 1 is that, given the side-condition of Equation (4.12), the integer-valued result of $a(x) \operatorname{div}_s b(x)$ is an expression among the set

$$\{\mathrm{RndN}\big(q(x)\big) - 1,\ \mathrm{RndN}\big(q(x)\big),\ \mathrm{RndN}\big(q(x)\big) + 1\}.$$

Let $q'(x) = \mathrm{RndN}(q(x))$ and $r'(x) = a(x) - \big(b(x) \cdot q'(x)\big)$. Note that if $q(x)$ is integer-valued, then $q'(x) = q(x)$. The expressions resulting from the symbolic integer division and remainder operations are then determined from Table 4.1. Cases (1) and (2) compute the same expression for the symbolic quotient (and

	expressions		condition		
case	$a(x) \operatorname{div}_s b(x)$	$a(x) \operatorname{rem}_s b(x)$	$a(x)$	$b(x)$	$r'(x)$
1	$q'(x) - 1$	$r'(x) + b(x)$	> 0	> 0	< 0
2	$q'(x) - 1$	$r'(x) + b(x)$	< 0	< 0	> 0
3	$q'(x) + 1$	$r'(x) - b(x)$	< 0	> 0	> 0
4	$q'(x) + 1$	$r'(x) - b(x)$	> 0	< 0	< 0
5	$q'(x)$	$r'(x)$	else		

Table 4.1: Resulting Symbolic Expressions for Univariate Polynomials

also for the remainder), so we can combine these cases using the conjunction of both conditions. The same holds for cases (3) and (4). The solutions stated in Equation (4.9) and (4.10) account for the additional case of an invalid side-condition. This leaves us with a total of four cases for the symbolic integer division and remainder operations of univariate polynomials in the same indeterminate.

Example 4.4 Consider the univariate polynomials $a(x) = 3x^5 + x^2 + x + 5$ and $b(x) = 5x^2 - 3x + 1$. Performing polynomial long division results in the quotient polynomial

$$q(x) = \frac{3x^3}{5} + \frac{9x^2}{25} + \frac{12x}{125} + \frac{116}{625}$$

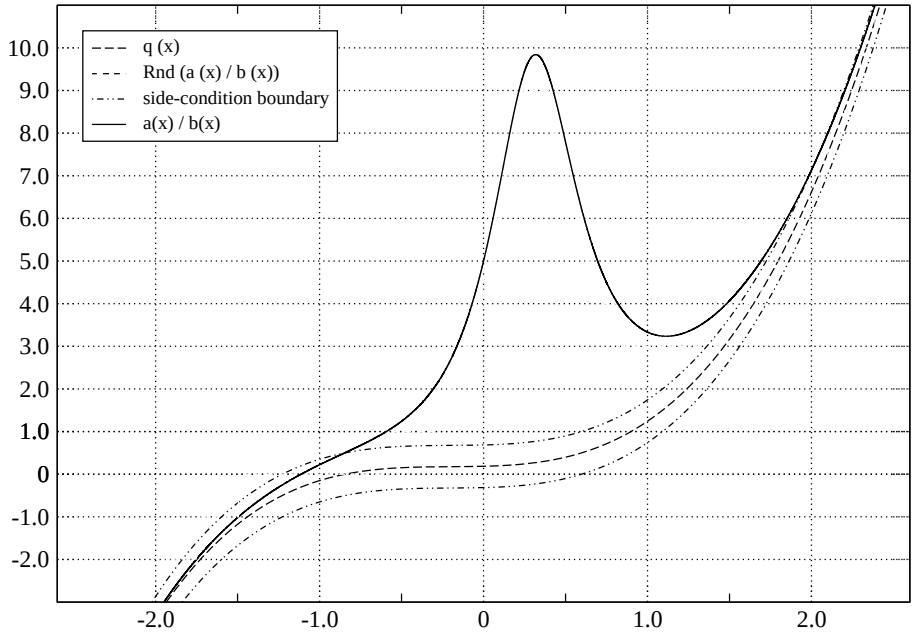
and the remainder polynomial

$$r(x) = \frac{913x}{625} + \frac{3009}{625}.$$

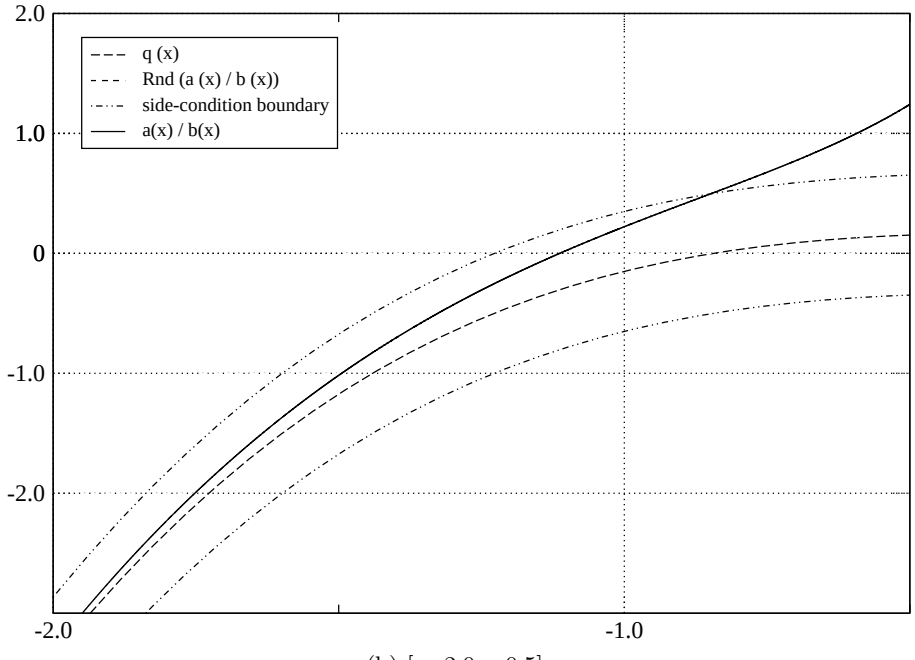
An illustration of this example is given in Figure 4.1. The side-condition is valid for $x \in (-\infty, -1] \cup [3, \infty)$, as can be seen from the graphs of the side-condition boundaries $q(x) - 0.5$ and $q(x) + 0.5$ in Figure 4.1 (a).

It is instructive to evaluate the involved polynomials in order to see how quotient and remainder correspond to the symbolic quotient and remainder expressions evaluated for a given value. Polynomial evaluation at $x = -1$ gives $a(-1) = 2$ and $b(-1) = 9$, hence we have $2 \operatorname{div} 9 = 0$ and $2 \operatorname{rem} 9 = 2$ for quotient and remainder computed from the integer arithmetic of the Flow concrete semantics (cf. also Figure 4.1 (b)).

Furthermore, $q'(-1) = \mathrm{RndN}(-0.1504) = 0$, and $r'(-1) = 2$. From the values of $a(-1)$, $b(-1)$, and $r'(-1)$ it follows that Case (5) of Table 4.1 is



(a) $[-2.6, 2.6]$



(b) $[-2.0, -0.5]$

Figure 4.1: Example of Polynomial Integer Arithmetic

applicable. Hence $q'(-1)$ and $r'(-1)$ represent the result of the evaluation of the symbolic quotient and remainder expressions, and this result corresponds to the values computed from the integer arithmetic of the Flow concrete semantics.

Case 2: $e_1(\mathbf{x}) = a(\mathbf{x})$ and $e_2(\mathbf{x}) = b(\mathbf{x}) \neq 0$ are multivariate integer-valued polynomials in the polynomial domain $\mathbb{Q}(\mathbf{x})$ over the field $\mathbb{Q}$ of rational numbers, and $a(\mathbf{x})$ is a multiple of $b(\mathbf{x})$, i.e. $a(\mathbf{x}) = q(\mathbf{x}) \cdot b(\mathbf{x})$. Since $\mathbb{Q}$ is a field, it follows from [GCL92, Theorem 2.7 (v)] that $\mathbb{Q}(\mathbf{x})$ is a *unique factorization domain* but not an Euclidean domain if the number of indeterminates is greater than one.

In this unique factorization domain we can factor the polynomials $a(\mathbf{x})$ and $b(\mathbf{x})$ in a unique way into *irreducible* polynomials and thus determine whether polynomial $a(\mathbf{x})$ is a multiple of $b(\mathbf{x})$. If this is the case, then the result of dividing $a(\mathbf{x})$ by $b(\mathbf{x})$ is $q(\mathbf{x})$, and the remainder is zero.

It should be noted that the quotient polynomial $q(\mathbf{x})$ need not be integer-valued; In this case we have to apply the rounding function Rnd to obtain an integer-valued result.

If $a(\mathbf{x})$ is not a multiple of $b(\mathbf{x})$, then the result of polynomial long division depends on which indeterminate of the polynomials is considered the main variable (as noted by Haghighat in [Hag95, Section 3.4]). In this case we have to fall back to the solutions of Equation (4.9) and (4.10).

A final issue on the simplification of elements of the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ is pointed out in [BL82, p. 19] and [DST93, pp. 89]. It concerns the difference of elements of $Q(\mathbb{Z}[\mathbf{x}])$ and the functions of arity $\mathbb{Z} \times \cdots \times \mathbb{Z} \to \mathbb{Z}$ which they represent. For example, the expressions $(x-1)/(x+1)$ and $(x^2-2x+1)/(x^2-1)$ represent the same element of $Q(\mathbb{Z}[\mathbf{x}])$, but the functions $f(x) = (x-1)/(x+1)$ and $g(x) = (x^2 - 2x + 1)/(x^2 - 1)$ are different, since the first is defined at the value $x = 1$ (where it takes the value 0), whereas $g(x)$ is not defined as it becomes $0/0$. If such singularities were not preserved during symbolic analysis, we would actually change the semantics of the input program. With the above example functions we must not simplify $g(x)$ to $f(x)$ without taking care of the value of $g(x)$ at $x = 1$. In Section 5.5.2 we will see how this is achieved by our symbolic analysis method.

4.1.2 The Symbolic Predicate Domain

Let $f \in \{<, \leq, =, \geq, >\}$ denote functions corresponding to the relational connectives of the Flow language. They are extensions of their standard semantic counterparts which operate on values of the symbolic expression domain SymExpr, and return values of the symbolic predicate domain SymPred, e.g.,

$$\leq: \text{SymExpr} \times \text{SymExpr} \to \text{SymPred}.$$

Moreover, let $l^{(n)} \in \{\wedge^{(2)}, \vee^{(2)}, \neg^{(1)}\}$ denote the logical connectives of conjunction, disjunction and negation. They are extensions of their standard semantic counterparts that operate on values of the symbolic predicate domain SymPred, e.g.,

$$\wedge : \text{SymPred} \times \text{SymPred} \to \text{SymPred}.$$

Definition 4.4 The set of symbolic predicates of the domain SymPred is inductively defined as

(i) $\mathbb{B} \subset \text{SymPred}$ (i.e., the boolean values *true* and *false* constitute symbolic predicates),

(ii) for all f and all $e_1, e_2 \in \text{SymExpr}$, we have $f(e_1, e_2) \in \text{SymPred}$ (i.e., application of relational connectives to symbolic expressions yields symbolic predicates),

(iii) for all l and all $e_1, \ldots e_n \in \text{SymPred}$, we have $l(e_1, \ldots, e_n) \in \text{SymPred}$ (i.e., application of logical connectives to symbolic predicates yields symbolic predicates).

We are interested in the algebraic properties of the domain of symbolic predicates, esp. if we want to transform a symbolic predicate into a simpler, yet equivalent, symbolic predicate. If we can show that the symbolic predicate domain is a *Boolean algebra*, then all results established about Boolean algebras in general also apply to the symbolic predicate domain. Using a definition from Rosen (cf. [Ros95, p. 614]) that is based on operation properties we show that the symbolic predicate domain is indeed a Boolean algebra.

Definition 4.5 A **boolean algebra** is as a set B together with two binary operations $\wedge$ and $\vee$, elements *true* and *false*, and a unary operation $\neg$ such that the following properties hold for all x, y, and z in B.

$$\left.\begin{array}{l} x \wedge \mathit{true} = x \\ x \vee \mathit{false} = x \end{array}\right\} \quad \text{Identity laws}$$

$$\left.\begin{array}{l} x \wedge \neg x = \mathit{false} \\ x \vee \neg x = \mathit{true} \end{array}\right\} \quad \text{Domination laws}$$

$$\left.\begin{array}{l} (x \wedge y) \wedge z = x \wedge (y \wedge z) \\ (x \vee y) \vee z = x \vee (y \vee z) \end{array}\right\} \quad \text{Associative laws}$$

$$\left.\begin{array}{l} x \wedge y = x \wedge y \\ x \vee y = x \vee y \end{array}\right\} \quad \text{Commutative laws}$$

$$\left.\begin{array}{l} x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \\ x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \end{array}\right\} \quad \text{Distributive laws}$$

We can view the symbolic predicate domain SymPred as an algebra and consider the subalgebra $\mathcal{B}_{\text{SymPred}}$ that is generated by the set $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ *. The carrier set of this subalgebra corresponds to the set of symbolic predicates obtained by Rules (i) and (iii) of Definition 4.4. Let this set denote set B of Definition 4.5, and let the logical connectives $l^{(n)}$ denote the required operations $\wedge$, $\vee$, and $\neg$. The fact that $\mathcal{B}_{\text{SymPred}}$ is a boolean algebra is then readily verified by truth tables for the laws listed with Definition 4.5.

It is worth mentioning that truth tables abstract from the actual argument structure and are only concerned with the two possible values of *true* and *false*

*Regarding the notions of *"subalgebra"* and *"algebra generation"* we refer the reader to Definition 3.2.2 of [BN98].

that these arguments may assume. Moreover, the symbolic predicates obtained from Rule (ii) of Definition 4.4 also assume a value of *true* or *false* which makes truth tables applicable to the whole set of symbolic predicates obtained from Definition 4.4. (In fact the truth tables used for the subalgebra $\mathcal{B}_{\mathrm{SymPred}}$ also apply to the symbolic predicate domain.) Hence we finally arrive at the fact that the domain of symbolic predicates constitutes a Boolean algebra.

(1)	Identity	$p \wedge \mathit{true} = p$ $p \vee \mathit{false} = p$
(2)	Domination	$p \wedge \neg p = \mathit{false}$ $p \vee \neg p = \mathit{true}$
(3)	Associativity	$(p \wedge q) \wedge r = p \wedge (q \wedge r)$ $(p \vee q) \vee r = p \vee (q \vee r)$
(4)	Commutativity	$p \wedge q = q \wedge p$ $p \vee q = q \vee p$
(5)	Distributivity	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
(6)	Idempotency	$p \wedge p = p$ $p \vee p = p$
(7)	Annihilation	$p \wedge \mathit{false} = \mathit{false}$ $p \vee \mathit{true} = \mathit{true}$
(8)	Double Complement	$\neg \neg p = p$
(9)	DeMorgan's	$\neg(p \wedge q) = (\neg p) \vee (\neg q)$ $\neg(p \vee q) = (\neg p) \wedge (\neg q)$
(10)	Absorption	$p \wedge (p \vee q) = p$ $p \vee (p \wedge q) = p$

Table 4.2: Algebraic Properties of the Symbolic Predicate Domain

Table 4.2 lists several identities from Boolean algebra that we are now able to use in order to simplify symbolic predicates (they hold for all p, q, and r in SymPred). Concerning simplifications we are always interested in those leading to a result in the subalgebra $\mathcal{B}_{\mathrm{SymPred}}$, as this is the most precise result that we can achieve.

There are simplifications that require knowledge on the values of variables contained in symbolic predicates. As an example, consider the symbolic predicate $a < 5$. Without prior knowledge on the actual value of a, we cannot determine whether this predicate yields *true* or *false*. Hence it is in this case not possible to achieve the above-mentioned simplification and we have to leave such a predicate as is.

The process of *simplification* of symbolic predicates, treated rigidly at the form level, will occupy us in Section 5.5. In the meantime we stay on the abstract level and are therefore free to perform such simplifications ad-lib.

4.2 Single-Edge Symbolic Execution

It is not enough for a substance to be simple, indivisible,
or at least undecomposed for us to call it an element.
It is also necessary for it to be abundantly distributed in nature
and to enter as an essential and constituent principle
in the composition of a great number of bodies.
— Antoine Laurent Lavoisier, French chemist, (1743-1794)

4.2.1 Program States and Contexts

Informally a *program state*[†] $s \in S$ can be envisioned as a set of ordered tuples $\{(v_1, e_1), (v_2, e_2), \cdots, (v_k, e_k)\}$ where v_i is a program variable and e_i is an associated symbolic expression describing the value of v_i for $1 \leq i \leq k$. We require that for each variable v_i there exists exactly one tuple in a given state s. Hence the set of possible states can also be represented by a class of functions

$$S \subseteq \{f : \mathbb{V} \to \text{SymExpr}\} \tag{4.13}$$

mapping symbolic expressions to program variables. A *clean slate* program state s maps all variables in its domain to the corresponding initial value variables:

$$\forall v \in \mathcal{D}om(s) : s(v) = \underline{v}. \tag{4.14}$$

Note that if we restrict our interest to a subset of $\mathbb{V}$ the program states are *partial* functions.

A *program context*[‡] $c \in C$ is defined by an ordered tuple $[s, p]$ where s denotes a program state, and pathcondition $p \in \text{SymPred}$ describes the condition for which the variable bindings specified through s hold (cf. [Bli02, FS03]). A *clean slate* program context consists of a clean slate state and a *true* pathcondition.

4.2.2 Symbolic Side-Effects and Branch Predicates

Table 4.3 contains the denotational definition of side-effects in the symbolic domain of the Flow language. We have omitted the definition of the syntactic domain which is of course identical to the one for standard-semantic side-effects given in Table 3.1 (I) on page 22.

The semantic domain of symbolic side-effects contains symbolic expressions, initial value variables, and symbolic predicates introduced in Section 4.1. States, pathconditions, and program contexts have been introduced in the previous section. With program contexts we introduce the operations st and pc in order to access state and pathcondition of a program context.

To complete this denotational definition of side-effects in the symbolic domain we give a bottom-up description of the valuation functions listed in Table 4.3 (II).

[†]Or *state*, if the context of use prevents a potential ambiguity due to the flow language states of Chapter 3.

[‡]Or *context*, for short.

(I) Semantic Domain

Symbolic Expressions

Domain $e(\mathbf{x})$: SymExpr

$+, -, \cdot \quad : \text{SymExpr} \times \text{SymExpr} \to \text{SymExpr}$

$\text{div}_\text{s}, \text{rem}_\text{s} \; : \text{SymExpr} \times \text{SymExpr} \setminus \{e(\mathbf{x}) \mid e(\mathbf{x}) = 0\} \to \text{SymExpr}$

Program Variables

Domain $v : \mathbb{V}$

Initial Value Variables

Domain $\underline{v} : \underline{\mathbb{V}}$

Path Conditions

Domain p : SymPred

States

Domain $s : S \subseteq \{f : \mathbb{V} \to \text{SymExpr}\}$

Contexts

Domain $c : C \subseteq [S \times \text{SymPred}]$

$\text{pc} : C \to \text{SymPred} \qquad \text{st} : C \to S$

$\text{pc}\,([s, p]) = p \qquad \text{st}\,([s, p]) = s$

(II) Valuation Functions

assign : Assignment $\to C \to C$

$\mathbf{assign}[\![\text{ident:=expr}]\!]\big([s_1, p_1]\big) =$

$\quad \lambda\big[s_2, p_2\big].\big[\lambda s_3.s_3\big[\mathbf{ident}[\![\text{ident}]\!] \mapsto \mathbf{expr}[\![\text{expr}]\!](s_3)\big](s_2),\, p_2\big]\big([s_1, p_1]\big)$

expr : Expression $\to S \to$ SymExpr

$\mathbf{expr}[\![\text{expr}_1\ \text{binop}\ \text{expr}_2]\!](s) = \mathbf{expr}[\![\text{expr}_1]\!](s)\ \ \mathbf{binop}[\![\text{binop}]\!]\ \ \mathbf{expr}[\![\text{expr}_2]\!](s)$

$\mathbf{expr}[\![-\text{expr}]\!](s) = -\mathbf{expr}[\![\text{expr}]\!](s)$

$\mathbf{expr}[\![\text{ident}]\!](s) = s\big(\mathbf{ident}[\![\text{ident}]\!]\big)$

$\mathbf{expr}[\![\text{num}]\!](s) = \mathbf{num}[\![\text{num}]\!]$

$\mathbf{expr}[\![(\text{expr})]\!](s) = \mathbf{expr}[\![\text{expr}]\!](s)$

num : Numeral $\to$ Integer

$\mathbf{num}[\![\text{num dig}]\!] = 10 \cdot \mathbf{num}[\![\text{num}]\!] + \mathbf{dig}[\![\text{dig}]\!]$

$\mathbf{num}[\![\text{dig}]\!] = \mathbf{dig}[\![\text{dig}]\!]$

dig : Digit $\to$ Integer

$\mathbf{dig}[\![0]\!] = 0,\ \mathbf{dig}[\![1]\!] = 1, \ldots, \mathbf{dig}[\![9]\!] = 9$

ident : Identifier $\to \mathbb{V}$ (omitted)

binop : Binary Operator $\to \{+, -, \cdot, \text{div}_\text{s}, \text{rem}_\text{s}\}$ (omitted)

Table 4.3: Symbolic Domain: Denotational Definition of Side-Effects

binop: This valuation function maps the terminals "+", "−", "∗", "div" and "rem" to the corresponding operations of addition, subtraction, multiplication, division (div_s), and remainder (rem_s) in the domain SymExpr of symbolic expressions. The terminal "−" may also serve as an unary operator defined on SymExpr → SymExpr denoting change of sign.

ident: The valuation function for identifiers maps a derivation tree representing a variable identifier to the corresponding identifier $v \in \mathbb{V}$. In this way it is exactly the same function as the one for the standard-semantics, given in Table 3.1 on page 22.

dig, **num**: Together, these valuation functions calculate the value of a given integer numeral. They are the same functions as those for the standard-semantics given in Table 3.1 on page 22.

expr: These valuation functions take an expression as argument and return a function that maps a state to a symbolic expression. The syntactic structure of the expression derivation tree is specified between the emphatic brackets ($[\![\ldots]\!]$) on the left-hand side of the given equations, whereas the functions $f : S \to \text{SymExpr}$ represent the right-hand sides.

expr$[\![\text{num}]\!](s) =$ **num**$[\![\text{num}]\!]$: For a derivation tree $[\![\text{num}]\!]$ the argument s describing the program state is not needed. This follows from the arity of the valuation function **num**. This argument is therefore discarded on the right-hand side of this equation.

expr$[\![\text{ident}]\!](s) = s\big($**ident**$[\![\text{ident}]\!]\big)$: For a derivation tree of structure $[\![\text{ident}]\!]$ denoting an identifier we determine the corresponding program variable $v \in \mathbb{V}$ that is then evaluated within the argument s representing a program state.

expr$[\![\text{expr}_1\,\text{binop}\,\text{expr}_2]\!](s)$: For a tree $[\![\text{expr}_1\,\text{binop}\,\text{expr}_2]\!]$ we recursively determine the values of the sub-expressions $[\![\text{expr}_1]\!]$ and $[\![\text{expr}_2]\!]$ which are then combined using the operation that is returned from evaluating the valuation function **binop**$[\![\text{binop}]\!]$.

assign: This valuation function takes a derivation tree $[\![\text{ident:=expr}]\!]$ corresponding to an assignment statement as argument. From this it returns a function that maps a program context $c_1 = [s_1, p_1]$ supplied as an argument to a program context $c_2 = [s_2, p_2]$. Context c_2 is generated from c_1 by updating the state of c_1 with a new value **expr**$[\![\text{expr}]\!]$ at **ident**$[\![\text{ident}]\!]$.

Table 4.4 contains the denotational definition of branch predicates in the symbolic domain of the Flow language. Again we have omitted the definition of the syntactic domain since it is identical to the one for standard-semantic branch predicates given in Table 3.2 (I) on page 25. The semantic domain of symbolic branch predicates depends on symbolic predicates and on program contexts. We complete the denotational definition of branch predicates with a description of the valuation functions listed in Table 4.4 (II).

rel-op: The valuation function **rel-op** maps the binary function symbols "<", "≤", "=", "≥", ">", and "<>" to the corresponding relational connectives in the domain SymPred of symbolic predicates.

(I) Semantic Domain

Symbolic Predicates

Domain $p : \text{SymPred}$

$<, \leq, =, \geq, >, \neq \ : \ \text{SymPred} \times \text{SymPred} \to \text{SymPred}$

$\wedge\,, \vee \ : \ \text{SymPred} \times \text{SymPred} \to \text{SymPred}$

$\neg \ : \ \text{SymPred} \to \text{SymPred}$

(II) Valuation Functions

pred : Predicate $\to C \to C$

$\mathbf{pred}[\![\text{true}]\!]\big([s_1, p_1]\big) = [s_1, p_1]$

$\mathbf{pred}[\![\text{false}]\!]\big([s_1, p_1]\big) = \lambda\big[s_2, p_2\big].\big[s_2,\, \mathit{false}\big]\big([s_1, p_1]\big)$

$\mathbf{pred}[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]\big([s_1, p_1]\big) =$
$\quad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\ \text{pc}\ \big(\mathbf{pred}[\![\text{pred}_1]\!]([s_2, \mathit{true}])\big)$
$\qquad\qquad \wedge\ \text{pc}\ \big(\mathbf{pred}[\![\text{pred}_2]\!]([s_2, \mathit{true}])\big)\big)\big]\ \big([s_1, p_1]\big)$

$\mathbf{pred}[\![\text{pred}_1 \text{ or } \text{pred}_2]\!]\big([s_1, p_1]\big) =$
$\quad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\ \text{pc}\ \big(\mathbf{pred}[\![\text{pred}_1]\!]([s_2, \mathit{true}])\big)$
$\qquad\qquad \vee\ \text{pc}\ \big(\mathbf{pred}[\![\text{pred}_2]\!]([s_2, \mathit{true}])\big)\big)\big]\ \big([s_1, p_1]\big)$

$\mathbf{pred}[\![\text{not pred}]\!]\big([s_1, p_1]\big) =$
$\quad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \neg\big(\text{pc}\ \big(\mathbf{pred}[\![\text{pred}]\!]([s_2, \mathit{true}])\big)\big)\big]\ \big([s_1, p_1]\big)$

$\mathbf{pred}[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!]\big([s_1, p_1]\big) =$
$\quad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\ \mathbf{expr}[\![\text{expr}_1]\!]\big(s_2\big)$
$\qquad\qquad \mathbf{rel\text{-}op}[\![\text{rel-op}]\!]\ \mathbf{expr}[\![\text{expr}_2]\!]\big(s_2\big)\big)\big]\ \big([s_1, p_1]\big)$

rel-op : Relational Operator $\to \{<, \leq, =, \geq, >, \neq\}$ (omitted)

Table 4.4: Symbolic Domain: Denotational Definition of Branch Predicates

pred: This valuation function takes a derivation tree from the set "Predicate" as input and returns a function f of arity $C \to C$ according to the structure of its syntactic argument. This structure is specified between the emphatic brackets ($[\![\ldots]\!]$) on the left-hand side of the given equations, whereas the functions $f : C \to C$ represent the right-hand sides.

It is the purpose of these functions to evaluate the branch predicate in the program state of the context $c = [s_1, p_1]$ passed as argument. The result is a symbolic branch predicate p'. The pathcondition of the resulting context is the conjunction of p' and the pathcondition p_1 of the argument context. The application of f to the argument context c can therefore be written as

$$f\big([s_1, p_1]\big) = [s_1, p_1 \wedge p'\,]. \tag{4.15}$$

$\mathbf{pred}[\![\text{true}]\!]\big([s_1, p_1]\big)$: We have $p' = \mathit{true}$, and by Identity (1) of Table 4.2 we get $f\big([s_1, p_1]\big) = [s_1, p_1 \wedge \mathit{true}] = [s_1, p_1]$.

$\mathbf{pred}[\![\text{false}]\!]\big([s_1, p_1]\big)$: We have $p' = \mathit{false}$, and by Identity (7) of Table 4.2

we get $f\big([s_1, p_1]\big) = [s_1, p_1 \wedge \mathit{false}\,] = [s_1, \mathit{false}]$.

pred$[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]\big([s_1, p_1]\big)$: For a subtree $[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]$ we determine the symbolic branch predicate p' from the values of the subtrees $[\![\text{pred}_1]\!]$ and $[\![\text{pred}_2]\!]$. These values are in fact program contexts from which we derive p' as the conjunction of the pathconditions that have been extracted via operation pc.

pred$[\![\text{pred}_1 \text{ or } \text{pred}_2]\!]\big([s_1, p_1]\big)$: Again we determine the symbolic branch predicate p' from the values of the subtrees $[\![\text{pred}_1]\!]$ and $[\![\text{pred}_2]\!]$. These values are program contexts from which we derive p' as the disjunction of the the extracted pathconditions.

pred$[\![\text{not pred}]\!]\big([s_1, p_1]\big)$: We determine the value of the symbolic branch predicate p' from the negated pathcondition of the context that is the value of the subtree $[\![\text{pred}]\!]$.

pred$[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!]\big([s_1, p_1]\big)$: We determine the values of the subtrees $[\![\text{expr}_1]\!]$ and $[\![\text{expr}_2]\!]$ by means of the valuation function **expr** of Table 4.3. These results are then combined using the relational operator returned from **rel-op**$[\![\text{rel-op}]\!]$ to form the symbolic branch predicate p'.

4.2.3 The Symbolic Single-Edge Solution

In this section we describe the effect of a computational step associated with a single edge e of a Flow program on the state of the computation, expressed by a program context. We can express the effect of such a computational step on a program context by a member of the class of functions

$$F \subseteq \{f : C \to C\} \tag{4.16}$$

with the following properties.

P1) F contains the identity function ι.

P2) F is closed under composition: $\forall f, g \in F : f \circ g \in F$.
f^k denotes iterated composition such that $f^0 = \iota$ and $f^k = f^{k-1} \circ f$.

The identity function ι of Property (P1) can be envisioned as a `null`-statement that has no affect on the state of computation. Property (P2) ensures that we can compose the computational steps of several edges, which is a property that we will need in Section 4.3 to express the computational effects of entire program paths.

An edge transition function $M_s : E \to F$ assigns a function $f \in F$ to each edge $e \in E$ of the CFG. From Chapter 3 it follows that every edge e has associated a branch predicate $\mathit{pred}_s(e)$ and a side-effect $\sigma_s(e)$. In the symbolic domain these functions are of arity $C \to C$, which contrasts their standard semantic counterparts stated in Equations (3.2) and (3.4)[§].

Our notational convention defined in Equation (3.11) allows us to formulate branch predicate and side-effect for a given CFG edge e according to the Flow

[§]To distinguish between the two, we subscript the functions from the symbolic domain with the letter s.

syntax. The valuation function $\mathbf{edge}[\![\ldots]\!]$ maps such a construct to the respective valuation functions for branch predicates and side-effects. In making use of the denotational definitions for branch predicates and side-effects we can define the functions $f \in F$ in the following way.

$$\begin{aligned} f &= M_s(e)\big([s,p]\big) \\ &= \sigma_s(e) \circ pred_s(e)\big([s,p]\big) && \text{by Equation (3.5)} \\ &= \mathbf{edge}[\![\text{e : pred} \Rightarrow \text{assign}]\!]\big([s,p]\big) && \text{by Equation (3.11)} \\ &= \mathbf{assign}[\![\text{assign}]\!]\big(\mathbf{pred}[\![\text{pred}]\!]\big([s,p]\big)\big) && \text{from Tables 4.3, 4.4} \end{aligned} \tag{4.17}$$

It follows immediately from the denotational definitions of side-effects (cf. Table 4.3) and branch predicates (cf. Table 4.4) that the function class of Equation (4.17) fulfills Properties (P1) and (P2) above.

Example 4.5 Figure 4.2 shows the Flow example program introduced in Section 3.4. We determine the transition function f for edge e_2, which has associated the Flow construct u <> v ⇒ u := u+v. Applying Equation (4.17) and the valuation functions for branch predicates and side-effects, we get

$$\begin{aligned} f &= M_s(e_2)\big([s,p]\big) \\ &= \mathbf{edge}[\![\text{e}_2 \text{ : u <> v } \Rightarrow \text{ u := u+v}]\!]\big([s,p]\big) \\ &= \mathbf{assign}[\![\text{u := u+v}]\!]\big(\mathbf{pred}[\![\text{u <> v}]\!]\big([s,p]\big)\big) \\ &= \mathbf{assign}[\![\text{u := u+v}]\!]\big(\lambda\big[s_1,p_1\big].\big[s_1, p_1 \wedge s_1(u) \neq s_1(v)\big]\big([s,p]\big)\big) \\ &= \lambda\big[s_2,p_2\big].\big[s_2\big[u \mapsto s_2(u) + s_2(v)\big], p_2\big] \\ &\qquad\qquad \big(\lambda\big[s_1,p_1\big].\big[s_1, p_1 \wedge s_1(u) \neq s_1(v)\big]\big([s,p]\big)\big) \\ &= \lambda\big[s_3,p_3\big].\big[s_3\big[u \mapsto s_3(u) + s_3(v)\big], p_3 \wedge s_3(u) \neq s_3(v)\big]\big([s,p]\big). \end{aligned}$$

Note that in the last step of the above simplifications we have combined branch predicate and side-effect to obtain a single function f as the solution of symbolic execution along edge e_2.

4.3 Single-Path Symbolic Execution

We can extend the transition function M_s from edges e to program paths π if we distinguish between forward (M_{fw}) and backward (M_{bw}) problems:

$$M_{\text{fw}}(\pi) = \begin{cases} \iota, \text{ if } \pi \text{ is the empty path} \\ M_s(e_k) \circ M_s(e_{k-1}) \circ \cdots \circ M_s(e_1), \text{ if } \pi = \langle e_1, \ldots, e_k \rangle \end{cases} \tag{4.18}$$

$$M_{\text{bw}}(\pi) = \begin{cases} \iota, \text{ if } \pi \text{ is the empty path} \\ M_s(e_1) \circ \cdots \circ M_s(e_{k-1}) \circ M_s(e_k), \text{ if } \pi = \langle e_1, \ldots, e_k \rangle. \end{cases} \tag{4.19}$$

In this way the transition function for a program path π is the composition of the edge transition functions contained in π. As a shorthand notation we may also use f_e for $M_s(e)$ and f_π for $M_s(\pi)$.

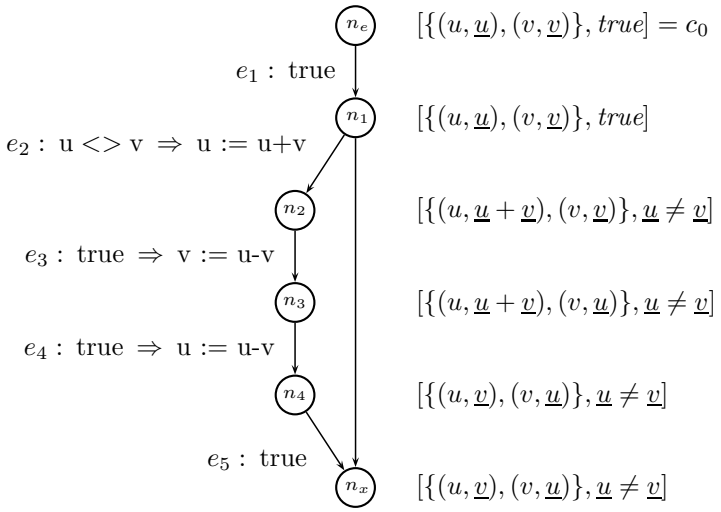


Figure 4.2: Symbolic Execution Along Path $\pi = \langle e_1, e_2, e_3, e_4, e_5 \rangle$

4.3.1 The Single-Path Solution

Clearly if the computational effect of a single statement of a Flow program is described by a function $f \in F$, the computational effect of program execution along a path π is defined by

$$M_{\text{fw}}(\pi)(c_0), \tag{4.20}$$

where c_0 denotes the initial context on entry to π. *Proof by induction on the length of π omitted.*

Example 4.6 In Example 4.5 we have determined transition function $M_s(e_2)$ which represents the effect of symbolic program execution along edge e_2 of the Flow program given in Figure 4.2. After determination of all transition functions along the program path $\pi = \langle e_1, e_2, e_3, e_4, e_5 \rangle$, the effect of symbolic execution along path π can be calculated according to Equation (4.20).

We assume that the initial context c_0 passed as argument to $M_{\text{fw}}(\pi)$ contains two program variables u and v holding their initial values $\underline{u}$ and $\underline{v}$. Then the program contexts depicted in Figure 4.2 illustrate the transformation of the initial context c_0 during symbolic execution along π. The program context shown for node n_x represents the resulting context for $M_{\text{fw}}(\pi)(c_0)$.

Note that in the preceding example we determined the computational effect of execution along one path, although the control flow graph contained in fact two program paths from node n_e to node n_x. Symbolic execution along multiple program paths will occupy us in the next section.

4.4 Multi-Path Symbolic Execution

Figure 4.3 shows three distinctive structural examples of CFGs. The CFG in Figure 4.3 (a) contains a single program path $\pi = \langle e_1, e_2 \rangle$ from n_e to n_x. The program context valid in n_x is thus defined by $f_{e_2} \circ f_{e_1}(c_0)$, where c_0 denotes the initial context on entry to π. The CFG in Figure 4.3 (b) contains two

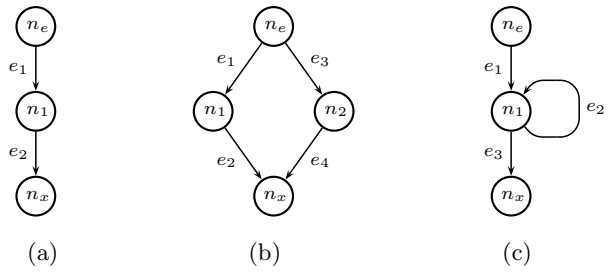


Figure 4.3: Distinctive Structural Examples of Control Flow Graph Portions

paths from n_e to n_x. Each path contributes a partial symbolic solution stated as a program context valid in n_x: $c_{\pi_1} = f_{e_2} \circ f_{e_1}(c_0)$ along path $\pi_1 = \langle e_1, e_2 \rangle$, and $c_{\pi_2} = f_{e_4} \circ f_{e_3}(c_0)$ along path $\pi_2 = \langle e_3, e_4 \rangle$. Hence the description of the symbolic solution for Node n_x of Figure 4.3 (b) has to take into account c_{π_1} and c_{π_2}. Finally the CFG depicted in Figure 4.3 (c) contains a cycle introduced by edge e_2. Due to this edge the number of program paths from n_e to n_x is *infinite*. As a consequence the number of program contexts valid at n_x is also infinite.

From the above examples we conclude that the description of the symbolic solution in terms of program contexts increases with the number of program paths through a CFG. To be specific, each program path from the entry node n_e to a given node n contributes one program context to the symbolic solution at node n. As long as CFGs are acyclic, the number of program contexts of this symbolic solution is finite. With the introduction of cycles the number of program paths from the entry node to a given node n, and hence the number of program contexts of the symbolic solution at node n, becomes infinite (cf. also [MR90, p. 134]).

4.4.1 Supercontexts

Sections 4.2 and 4.3 showed that a program context is a valid description of the computational effect of symbolic execution along a single edge or program path. In order to describe the joint effects of execution along several program paths, we need a structure that allows us to aggregate program contexts.

Due to their illustrative nature the approach chosen here is based on *sets*. Taking into account the fact that these sets can become infinite in case of CFG cycles, we will replace this set-based structure by a finite structure in Section 5.5.1.

A *supercontext* $sc \in SC$ is a collection of program contexts $c \in C$ and can be envisioned as a (possibly) infinite set as follows:

$$sc = \{c_1, \ldots, c_k, \ldots\} = \{[s_1, p_1], \ldots, [s_k, p_k], \ldots\}.$$

We write $c \in sc$ to denote that context c is an element of the supercontext sc. For supercontexts $sc_1, sc_2 \in SC$ the supercontext union operation $sc_1 \cup sc_2$ contains those contexts that are either in sc_1, or in sc_2, or in both. If we regard single contexts as one-element supercontexts, we can use the supercontext union operation to denote a supercontext sc through union over its context elements.

For this purpose we use the following notation for supercontexts:

$$sc \in SC = \left[\bigcup_{k=0}^{\infty} [s_k, p_k] \right]. \tag{4.21}$$

Let C be the set of all possible program contexts, and $SC = P(C)$ the power set of C. By $\cup$ we denote the supercontext union operation. Then the algebraic structure $\langle SC, \cup \rangle$ has the following properties.

P1) $\langle SC, \cup \rangle$ is closed under $\cup$: $A, B \in SC \Rightarrow A \cup B \in SC$.

P2) $\cup$ is associative: $(A \cup B) \cup C = A \cup (B \cup C), \quad \forall A, B, C \in SC$.

P3) Let $\emptyset \in SC$ be the *empty* collection of contexts. Then $\langle SC, \cup \rangle$ contains the identity element $E = \emptyset$: $A \cup E = E \cup A = A, \quad \forall A \in SC$.

P4) $\cup$ is commutative: $A \cup B = B \cup A, \quad \forall A, B \in SC$.

P5) No element of SC except E has an inverse in SC such that $A \cup A^{-1} = A^{-1} \cup A = E$.

It is worth mentioning that supercontexts correspond to the notion of *symbolic environments* mentioned in the introduction of this chapter.

4.4.2 The Meet Over All Paths Solution

An intuitive definition of the *meet over all paths (MOP) solution* for monotone data-flow frameworks is given in [Hec77, p. 169]. Therein this solution is the maximum information, relevant to the problem at hand, which can be derived from every possible execution path from the initial node to that node.

Let $\mathrm{Path}(a, b)$ denote the set of program paths from node a to node b. All possible execution paths from the initial node n_e to a given node n are then contained in $\mathrm{Path}(n_e, n)$. For each such path π its computational effect f_π on the initial context c_0 is calculated. The meet over all paths solution for symbolic execution is then the union over the resulting contexts, written as

$$\mathrm{mop}(n) = \bigcup_{\pi \in \mathrm{Path}(n_e, n)} f_\pi(c_0). \tag{4.22}$$

Union denotes the supercontext union operation, and the MOP solution for symbolic execution is therefore represented by a supercontext.

4.4.3 A Correctness Proof for Symbolic Execution

In this section we give a proof of correctness for the MOP solution stated in Equation (4.22). At the same time we also proof Condition (4.3) stated on page 38, since the symbolic side-effect $S_{\mathrm{sym}}[\![P]\!]$ of program P on the initial context c_0 corresponds to the MOP solution at the exit-node n_x of P.

At the heart of the proof we will relate the standard semantics with the semantics of symbolic program execution and show that the two commute. To distinguish between standard semantic and symbolic program execution, we will henceforth use σ_{c} to denote the standard-semantic side-effect associated

with an edge e, and σ_s to denote its symbolic counterpart. Likewise we will use $pred_c$ and $pred_s$ to distinguish between standard-semantic and symbolic branch predicates. This frees the letter σ which from now on will be used to denote *substitutions* that will be introduced in the course of the proof.

There is one distinction needed regarding the semantics of symbolic program execution: with the introduction of the symbolic rounding operation Rnd in Definition 4.2 on page 41 we noted that in order to implement rounding towards zero we must be able to determine whether the argument of the rounding operation is less than zero. This decision has been postponed to Section 5.5.2, leaving such expressions unevaluated in the meantime. As a consequence, the symbolic integer division operation considered here is restricted to that of Definition 4.9 on page 42 and defers application of division-related simplifications (Case 1 and Case 2 subsequent to Definition 4.9) to Section 5.5.2. It is Section 5.5.2 where we will also prove the correctness of these simplifications.

As already announced in this introduction we start with a few definitions that we need in order to accomplish the proof.

Definition 4.6 Given the set $\underline{\mathbb{V}}$ of initial value variables and SymExpr, the domain of integer-valued symbolic expressions. A SymExpr-**substitution** — or simply substitution, if the domain SymExpr is irrelevant or clear from the context of use — is a function $\sigma : \underline{\mathbb{V}} \to \text{SymExpr}$ such that $\sigma(x) \neq x$ for only finitely many xs. The finite set of variables that σ does not map to themselves is called the **domain** of σ: $\mathcal{D}om(\sigma) ::= \{x \in \underline{\mathbb{V}} \mid \sigma(x) \neq x\}$. If $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$, then we may write σ as

$$\sigma = \{x_1 \to \sigma(x_1), \ldots, x_n \to \sigma(x_n)\}.$$

The application of a substitution σ to an expression e simultaneously replaces all occurrences of variables by their respective σ-images. Any substitution σ can be **extended** to a mapping $\widehat{\sigma}$: SymDom $\to$ SymDom over the domain of integer-valued symbolic expressions and symbolic predicates in the following way.

$$\widehat{\sigma}(x) = \sigma(x), \text{ for } x \in \underline{\mathbb{V}} \tag{4.23}$$

$$\widehat{\sigma}(f(e_1, \ldots, e_n)) = f(\widehat{\sigma}(e_1), \ldots, \widehat{\sigma}(e_n)), \text{ for a non-variable expression} \tag{4.24}$$

From Equation (4.24) it follows that the extension $\widehat{\sigma}$ is an **endomorphism** on the domain SymDom that coincides with the identity mapping on almost all variables. To simplify notation we distinguish between a substitution σ and its extension $\widehat{\sigma}$ only at places where this distinction is crucial. It should be noted that the concept of SymExpr-substitutions is closely related to substitutions on terms presented in [BN98].

Definition 4.7 An **extended environment** $\overline{env} \in \mathit{Environment} \times \mathbb{B}$ is an ordered tuple $[env, b]$ where env denotes an environment from the Flow standard semantics (cf. Equation (3.1) on p. 19), and $b \in \mathbb{B}$ is a boolean value denoting the standard semantic pathcondition. It is worth noting that such a pathcondition was not needed for the transition function δ of the Flow standard semantics being only defined for edges e for which $pred_c(e)(env) = \mathit{true}$ (cf. Equation (3.5)) and therefore assuming an implicit pathcondition that is always *true*.

Definition 4.8 In order to describe the standard semantic computational effect of an *arbitrary* edge e, we use the function class

$$F_c \subseteq \{f_c : \mathit{Environment} \times \mathbb{B} \to \mathit{Environment} \times \mathbb{B}\}$$

with its member functions f_c defined as

$$f_c = M_c(e)\big([\mathit{env}, b]\big) = \big([\sigma_c(e)(\mathit{env}), b \wedge \mathit{pred}_c(e)(\mathit{env})]\big), \tag{4.25}$$

where $M_c : E \to F_c$ represents the standard semantic edge transition function. It is easy to see that for the domain of the transition function δ the function class of Equation (4.25) is equivalent to δ in terms of the computed environment $\sigma_c(e)(\mathit{env})$ as well as the computed pathcondition $b \wedge \mathit{pred}_c(e)(\mathit{env})$. (The latter due to the fact that with the transition function δ we have the implicit pathcondition which is always *true*, since δ is only defined on edges e such that $\mathit{pred}_c(e)(\mathit{env}) = \mathit{true}$). On the other hand, for values outside the domain of δ we note that the function class of Equation (4.25) computes an extended environment with a standard semantic pathcondition of false which is due to the fact that those cases comprise edges e for which $\mathit{pred}_c(e)(\mathit{env}) = \mathit{false}$.

On the analogy of symbolic program execution we can extend the standard semantic edge transition function M_c from edges e to program paths π. Resembling Equation (4.18), we get

$$M_c(\pi) = \begin{cases} \iota, \text{ if } \pi \text{ is the empty path} \\ M_c(e_k) \circ M_c(e_{k-1}) \circ \cdots \circ M_c(e_1), \text{ if } \pi = \langle e_1, \ldots, e_k \rangle \end{cases}$$

for a forward problem. Therefore the standard semantic transition function for a program path π is the composition of the standard semantic edge transition functions contained in π.

Definition 4.9 Function **sym** transfers an extended environment $[\mathit{env}, b]$ to a program context in the symbolic domain.

$$\begin{aligned} &\mathrm{sym} : \mathit{Environment} \times \mathbb{B} \to C \\ &\mathrm{sym}\big([\mathit{env}, b]\big) = \lambda[\mathit{env}', b'].[\lambda \mathit{env}''.s[\forall v \in \mathcal{D}om(\mathit{env}'') : v \mapsto \underline{v}](\mathit{env}'), b'](\mathit{env}, b) \end{aligned}$$

The extended environment $[\mathit{env}, b]$ and the state of the resulting program context $[s, b]$ coincide on the contained program variables, that is, $\mathcal{D}om(\mathit{env}) = \mathcal{D}om(s)$. They differ however in the provided values, since state s maps program variables v_i to the corresponding initial value variables $\underline{v_i}$. The extended environment and the resulting program context furthermore agree on the pathcondition b.

Definition 4.10 Function **con** takes an environment *env* and a program context c as input and returns an extended environment $\overline{\mathit{env}}$ which is the result of evaluation of c with the values provided by *env*.

$$\begin{aligned} &\mathrm{con} : \mathit{Environment} \times C \to \mathit{Environment} \times \mathbb{B} \\ &\mathrm{con}(\mathit{env}, c) = \lambda[\mathit{env}', [s, p]].[\mathit{env}'[\forall v \in \mathcal{D}om(s) : v \mapsto \sigma_{\mathit{env}}(s(v))], \sigma_{\mathit{env}}(p)](\mathit{env}, c) \end{aligned}$$

Therein the SymExpr-substitution σ_{env} can be written as

$$\sigma_{\mathit{env}} = \{\underline{v_1} \to \mathit{env}(v_1), \ldots, \underline{v_n} \to \mathit{env}(v_n)\}, \tag{4.26}$$

which means that occurrences of the initial value variables $\underline{v_i}$ are replaced by the concrete value of the corresponding variable v_i in *env* (recall that SymDom contains expressions in terms of the initial value variables from $\underline{\mathbb{V}}$ (cf. Equation (4.6) on page 40). For this to work we need the side-condition that the domains of *env* and s coincide, that is, $\mathcal{D}om(env) = \mathcal{D}om(s)$.

We have now everything in place to set up the condition to be met for the concrete and symbolic execution to commute on a single edge e. As illustrated in the commutative diagram of Figure 4.4, we start with an extended environment $[env, b]$ passed as argument to function $M_c(e)$ representing the standard semantic computational effect associated with edge e. This results in a new extended environment $[env_1, b \wedge b_1]$ which we consider the result of standard semantic program execution along edge e (for reasons mentioned with Definition 4.8 this coincides with the transition function δ iff $b = \mathit{true}$ and $b_1 = \mathit{pred}_c(e)(env) = \mathit{true}$).

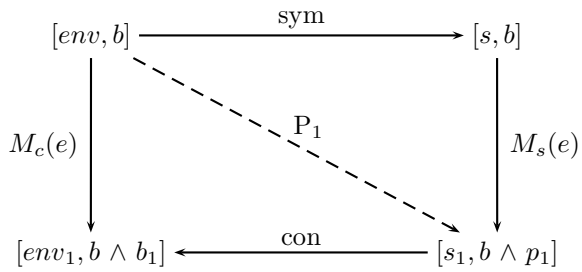


Figure 4.4: Commutation of Single-Edge Concrete and Symbolic Execution

On the other hand, transferring the initial extended environment $[env, b]$ via function sym to the symbolic domain gives us a program context $[s, b]$. We can apply function $M_s(e)$ that represents the computational effect in the symbolic domain associated with edge e. (Note that we use M_s to distinguish the edge transition function into the symbolic domain from its standard semantic counterpart M_c.) The result is a program context $[s_1, b \wedge p_1]$ that differs from the input-context $[s, b]$ in two ways.

- Program state s_1 represents program state s updated according to the side-effect $\sigma_s(e)$ of edge e (cf. Equation (4.17) and Table 4.3).
- For the pathcondition $b \wedge p_1$ we have $b \in \mathbb{B}$ and $p_1 \in \mathrm{SymPred}$. p_1 represents the branch predicate $\mathit{pred}_s(e)$ of the symbolic domain associated with edge e (cf. Equation (4.17) and Table 4.4). The conjunction $b \wedge p_1$ is due to Equation (4.15).

The transformation of the program context $[s_1, b \wedge p_1]$ from the symbolic to the standard semantic domain via function con uses environment *env* of the initial extended environment $[env, b]$ which provides the actual values to be substituted for the initial value variables occurring in s_1 and p_1. This is denoted by the dashed line in Figure 4.4, where P_1 is the projection function from extended environments to the first element, the contained environment.

For the concrete and symbolic execution to commute on edge e, the result of the above transformation must agree with the extended environment obtained

as a result from standard semantic program execution. This process that is illustrated in Figure 4.4 is generalized in the following lemma.

Lemma 2. Given a Flow program with its underlying control flow graph $G = \langle N, E, n_e, n_x \rangle$. For any edge $e \in E$ of G, the condition

$$M_c(e)\big([env, b]\big) = \mathrm{con}\big(env, M_s(e)\big(\mathrm{sym}\big([env, b]\big)\big)\big)$$

holds which means that concrete and symbolic Flow program execution commute for single edges.

In the course of the proof of Lemma 2 we will compare the valuation functions of concrete and symbolic side-effects stated in Table 3.1 and Table 4.3. To facilitate this comparison we have summarized the relevant parts of both definitions, amended with line-numbers, in Table 4.5. To distinguish valuation functions into the concrete domain from valuation functions into the symbolic domain, we use the subscripting-schema already applied with side-effects. Valuation functions common to both definitions have been left without subscript. Likewise we will compare the valuation functions of concrete and symbolic branch predicates from Table 3.2 and Table 4.4; The relevant parts of those definitions are summarized in Table 4.6.

Proof. It follows from the definitions of functions sym and con that the transformation of an extended environment $[env, b]$ into the symbolic domain is invertible using function con and the contained environment env, that is

$$\mathrm{con}\big(env, \mathrm{sym}\big([env, b]\big)\big) = [env, b]. \tag{4.27}$$

From the valuation functions for assignment statements (cf. lines 1–3 of Table 4.5) we know that a single side-effect updates exactly one program variable of a given environment or state (at most one if we would allow the identity function ι as side-effect). Line 11 of both definitions show us that concrete and symbolic semantics employ the same valuation function for identifiers. Hence it follows from line 3 of both definitions (from "**ident**⟦ident⟧", to be specific), that for a given side-effect the concrete and symbolic valuation functions update the *same* program variable.

With this argument and Equation (4.27) we establish that in the diagram of Figure 4.4 the environment env_1 coincides with environment env in all but one program variable, and that this is consistent with commutation. Without loss of generality we may assume v_i as the updated program variable.

In order to prove that concrete and symbolic execution commute on environment env_1, we must show that they update program variable v_i consistently. The notion of consistent update can be formalized based on the initial environment env, the corresponding state s, the concrete and symbolic valuation functions computing the new value for v_i (cf. line 3 of the corresponding definitions of Table 4.5), and the substitution σ_{env} (cf. Equation (4.26)) that is part of function con. Due to the involved concrete and symbolic valuation functions $\mathbf{expr}_c$ and $\mathbf{expr}_s$, the notion of consistent update is named $\mathbf{expr}_s^c$-consistency.

$$\mathbf{expr}_s^c\text{-consistency} \;\Leftrightarrow\; \mathbf{expr}_c[\![\mathrm{expr}]\!](env) = \sigma_{env}\big(\mathbf{expr}_s[\![\mathrm{expr}]\!](s)\big) \tag{4.28}$$

Recall that in this equation "expr" denotes a derivation tree corresponding to a Flow expression. We prove $\mathbf{expr}_s^c$-consistency by structural induction on the

1 $\mathbf{assign}_c$: Assignment $\to$ *Environment* $\to$ *Environment*
2 $\mathbf{assign}_c[\![\text{ident:=expr}]\!](env_1) =$
3 $\lambda env_2.env_2\big[\mathbf{ident}[\![\text{ident}]\!] \mapsto \mathbf{expr}_c[\![\text{expr}]\!](env_2)\big](env_1)$
4 $\mathbf{expr}_c$: Expression $\to$ *Environment* $\to$ Integer
5 $\mathbf{expr}_c[\![\text{expr}_1\ \text{binop}\ \text{expr}_2]\!](env) =$
6 $\mathbf{expr}_c[\![\text{expr}_1]\!](env)\ \mathbf{binop}_c[\![\text{binop}]\!]\ \mathbf{expr}_c[\![\text{expr}_2]\!](env)$
7 $\mathbf{expr}_c[\![-\text{expr}]\!](env) = -\mathbf{expr}_c[\![\text{expr}]\!](env)$
8 $\mathbf{expr}_c[\![\text{ident}]\!](env) = env\big(\mathbf{ident}[\![\text{ident}]\!]\big)$
9 $\mathbf{expr}_c[\![\text{num}]\!](env) = \mathbf{num}[\![\text{num}]\!]$
10 $\mathbf{expr}_c[\![(\text{expr})]\!](env) = \mathbf{expr}_c[\![\text{expr}]\!](env)$
11 **ident** : Identifier $\to \mathbb{V}$ (omitted)
12 $\mathbf{binop}_c$: Binary Operator $\to \{+, -, \cdot, \text{div}, \text{rem}\}$ (omitted)

(a) Concrete Domain

1 $\mathbf{assign}_s$: Assignment $\to C \to C$
2 $\mathbf{assign}_s[\![\text{ident:=expr}]\!]\big([s_1, p_1]\big) =$
3 $\lambda\big[s_2, p_2\big].\big[\lambda s_3.s_3\big[\mathbf{ident}[\![\text{ident}]\!] \mapsto \mathbf{expr}_s[\![\text{expr}]\!](s_3)\big](s_2),\ p_2\big]\big([s_1, p_1]\big)$
4 $\mathbf{expr}_s$: Expression $\to S \to$ SymExpr
5 $\mathbf{expr}_s[\![\text{expr}_1\ \text{binop}\ \text{expr}_2]\!](s) =$
6 $\mathbf{expr}_s[\![\text{expr}_1]\!](s)\ \mathbf{binop}_s[\![\text{binop}]\!]\ \mathbf{expr}_s[\![\text{expr}_2]\!](s)$
7 $\mathbf{expr}_s[\![-\text{expr}]\!](s) = -\mathbf{expr}_s[\![\text{expr}]\!](s)$
8 $\mathbf{expr}_s[\![\text{ident}]\!](s) = s\big(\mathbf{ident}[\![\text{ident}]\!]\big)$
9 $\mathbf{expr}_s[\![\text{num}]\!](s) = \mathbf{num}[\![\text{num}]\!]$
10 $\mathbf{expr}_s[\![(\text{expr})]\!](s) = \mathbf{expr}_s[\![\text{expr}]\!](s)$
11 **ident** : Identifier $\to \mathbb{V}$ (omitted)
12 $\mathbf{binop}_s$: Binary Operator $\to \{+, -, \cdot, \text{div}_s, \text{rem}_s\}$ (omitted)

(b) Symbolic Domain

Table 4.5: Comparison: Concrete vs. Symbolic Side-Effects

Flow expression derivation trees according to lines 5–10 of the $\mathbf{expr}_c$ and $\mathbf{expr}_s$ valuation functions of Table 4.5. Lines 8 and 9 constitute the cases of the induction basis, as they represent the leaves of expression derivation trees (cf. also the specification of Flow expression derivation trees given in Part (I) of Table 3.1). The inductive step of the definition of expression derivation trees consists of the case expressed through lines 5 and 6, which also subsumes the cases denoted by lines 7 and 10.

Basis: First we must show that $\mathbf{expr}_c[\![\text{ident}]\!](env) = \sigma_{env}\big(\mathbf{expr}_s[\![\text{ident}]\!](s)\big)$. This holds if $env\big(\mathbf{ident}[\![\text{ident}]\!]\big) = \sigma_{env}\big(s\big(\mathbf{ident}[\![\text{ident}]\!]\big)\big)$ which follows from Equation (4.27).

Furthermore we must show that $\mathbf{expr}_c[\![\text{num}]\!](env) = \mathbf{expr}_s[\![\text{num}]\!](s)$ which is trivially true since the valuation function $\mathbf{num}[\![\text{num}]\!]$ for numbers is common to the concrete and symbolic domain (cf. Table 4.5).

Induction: In the inductive step we must show that

$$\mathbf{expr}_c[\![\text{expr}_1 \text{ binop } \text{expr}_2]\!](env) = \sigma_{env}\big(\mathbf{expr}_s[\![\text{expr}_1 \text{ binop } \text{expr}_2]\!](s)\big).$$

Substituting the right-hand sides of lines 6 from Table 4.5 and using the shorthand notations

$$e_{c1} = \mathbf{expr}_c[\![\text{expr}_1]\!](env), \qquad e_{s1} = \mathbf{expr}_s[\![\text{expr}_1]\!](s),$$
$$e_{c2} = \mathbf{expr}_c[\![\text{expr}_2]\!](env), \qquad e_{s2} = \mathbf{expr}_s[\![\text{expr}_2]\!](s),$$

we must show that

$$e_{c1}\ \mathbf{binop}_c[\![\text{binop}]\!]\ e_{c2} = \sigma_{env}\big(e_{s1}\ \mathbf{binop}_s[\![\text{binop}]\!]\ e_{s2}\big). \tag{4.29}$$

For the inductive step we may assume that $e_{c1} = \sigma_{env}(e_{s1})$, and that $e_{c2} = \sigma_{env}(e_{s2})$. Based on the binary operator $\mathbf{binop}[\![\text{binop}]\!]$ we can then distinguish the following three cases.

Case 1: Let $\mathbf{binop}[\![\text{binop}]\!] \in \{+, -, \cdot\}$. From Definition 4.6 of substitution σ_{env} we may write

$$\sigma_{env}\big(e_{s1}\ \mathbf{binop}_s[\![\text{binop}]\!]\ e_{s2}\big) = \sigma_{env}(e_{s1})\ \mathbf{binop}_s[\![\text{binop}]\!]\ \sigma_{env}(e_{s2}),$$

but only since this rewriting step is backed by the algebras of multivariate polynomials and rational functions. And this is in fact a crucial step, as it marks the transition from algebra to arithmetic, from the domain of symbolic expressions to the integer numbers. It is this identity that allows us to perform symbolic program execution in the domain SymExpr of symbolic expressions and to substitute integer values for the initial value variables in the resulting expressions in order to generate the result corresponding to a given standard-semantic program execution. Or, stated in brief: if concrete and symbolic calculations commute, then it is due to this identity!

Having said that, there is now a note due to the binary operation $\mathbf{binop}_s[\![\text{binop}]\!]$ on the right-hand side of the above identity: strictly speaking, this operation operates on values of the concrete domain, so it should read $\mathbf{binop}_c[\![\text{binop}]\!]$. But since the concrete domain constitutes a subalgebra of the symbolic domain, we may also use the operations from the symbolic domain here.

Returning to our proof, we insert the above identity into Equation (4.29), giving us

$$e_{c1}\ \mathbf{binop}_c[\![\text{binop}]\!]\ e_{c2} = \sigma_{env}(e_{s1})\ \mathbf{binop}_s[\![\text{binop}]\!]\ \sigma_{env}(e_{s2}).$$

This equation holds due to the assumption of the inductive step and the before-mentioned relation of concrete and symbolic operations, which concludes the proof for Case 1.

Case 2: Let $\mathbf{binop}_c[\![\text{binop}]\!] = \text{div}$ and let $\mathbf{binop}_s[\![\text{binop}]\!] = \text{div}_\text{s}$. Definition 4.9 of integer division and further simplifications yield

$$\begin{aligned}
&\sigma_{env}\big(e_{s1}\ \mathbf{binop}_s[\![\text{binop}]\!]\ e_{s2}\big) = \\
&\quad = \sigma_{env}\big(\text{Rnd}\big(e_{s1}\ /\ e_{s2}\big)\big) = \text{Rnd}\big(\sigma_{env}(e_{s1})\ /\ \sigma_{env}(e_{s2})\big) = \\
&\quad = \begin{cases} \big\lfloor \sigma_{env}(e_{s1})\ /\ \sigma_{env}(e_{s2}) \big\rfloor, & \text{if } \sigma_{env}(e_{s1})\,/\,\sigma_{env}(e_{s2}) \geq 0 \\ \big\lceil \sigma_{env}(e_{s1})\ /\ \sigma_{env}(e_{s2}) \big\rceil, & \text{else.} \end{cases}
\end{aligned}$$

Again we have used the round towards zero rounding mode. Inserting the above right-hand side into Equation (4.29) concludes the proof for Case 2 due to the assumption of the inductive step.

Case 3: Let $\mathbf{binop}_c[\![\text{binop}]\!] = \text{rem}$ and let $\mathbf{binop}_s[\![\text{binop}]\!] = \text{rem}_\text{s}$. The proof of Case 3 then follows from Cases 1 and 2 and from the assumption of the inductive step.

With the proof of Case 3 we have finally established $\mathbf{expr}_s^c$-consistency of Flow expression derivation trees as postulated in Equation (4.28). This means that concrete and symbolic execution commute on environment env_1 as depicted in Figure 4.4, and it remains to show that they also commute regarding the pathcondition $b \wedge b_1$.

From Figure 4.4 we observe that with the pathcondition $b \wedge b_1$ of the extended environment $[env_1, b \wedge b_1]$ the boolean variable b is due to the extended input-environment $[env, b]$. Moreover, the boolean variable b_1 is due to the branch predicate associated with edge e. Hence in order to show that concrete and symbolic execution commute regarding the pathcondition $b \wedge b_1$, we must show that they commute on b_1 (Strictly speaking, we only need to show that they commute on b_1 if $b = \mathit{true}$, for $\mathit{false} \wedge b_1 = \mathit{false}$, from Table 4.2.)

On the analogy of $\mathbf{expr}_s^c$-consistency we proceed with the definition of $\mathbf{pred}_s^c$-consistency as the condition that must be met for concrete and symbolic branch predicates to commute on the boolean variable b_1 as depicted in Figure 4.4.

$$\begin{aligned} \mathbf{pred}_s^c\text{-consistency} \;\Leftrightarrow\; & \overline{b} \wedge \mathbf{pred}_c[\![\text{pred}]\!](env) = \\ & = \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\text{pred}]\!]\big([s, \overline{b}]\big)\big)\big) \end{aligned} \tag{4.30}$$

We will establish $\mathbf{pred}_s^c$-consistency of Flow predicates by structural induction on the predicate derivation trees according to the corresponding valuation functions listed in Table 4.6 (the specification of the Flow predicate derivation trees themselves has been given in Part (I) of Table 3.2). In the by-case definition of the valuation function $\mathbf{pred}_s$ in Table 4.6 (b) the induction basis consists of Case 1 (line 2), Case 2 (line 3), and Case 6 (lines 12–14) which represent the leaves of the Flow predicate derivation trees. The inductive step then consists of Case 3 (lines 4–6), Case 4 (lines 7–9), and Case 5 (lines 10–11) of Table 4.6 (b).

What follows is the inductive proof of $\mathbf{pred}_s^c$-consistency of Flow predicates as outlined above.

Basis: We establish Case 1 by the following sequence of transformations.

$$\begin{aligned} \overline{b} \wedge \mathbf{pred}_c[\![\mathit{true}]\!](env) &= \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\mathit{true}]\!]\big([s, \overline{b}]\big)\big)\big) \\ \overline{b} \wedge \mathit{true} &= \text{pc}\big(\sigma_{env}\big([s, \overline{b}]\big)\big) \\ \overline{b} &= \overline{b} \end{aligned}$$

Likewise for Case 2:

$$\begin{aligned} \overline{b} \wedge \mathbf{pred}_c[\![\mathit{false}]\!](env) &= \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\mathit{false}]\!]\big([s, \overline{b}]\big)\big)\big) \\ \overline{b} \wedge \mathit{false} &= \text{pc}\big(\sigma_{env}\big([s, \mathit{false}]\big)\big) \\ \mathit{false} &= \mathit{false}. \end{aligned}$$

$$
\begin{array}{l}
1 \quad \mathbf{pred}_c : \text{Predicate} \to \mathit{Environment} \to \text{Boolean} \\
2 \quad \mathbf{pred}_c[\![\text{true}]\!](\mathit{env}) = \mathit{true} \\
3 \quad \mathbf{pred}_c[\![\text{false}]\!](\mathit{env}) = \mathit{false} \\
4 \quad \mathbf{pred}_c[\![\text{pred}_1 \text{ and } \text{pred}_2]\!](\mathit{env}) = \mathbf{pred}_c[\![\text{pred}_1]\!](\mathit{env}) \wedge \mathbf{pred}_c[\![\text{pred}_2]\!](\mathit{env}) \\
5 \quad \mathbf{pred}_c[\![\text{pred}_1 \text{ or } \text{pred}_2]\!](\mathit{env}) = \mathbf{pred}_c[\![\text{pred}_1]\!](\mathit{env}) \vee \mathbf{pred}_c[\![\text{pred}_2]\!](\mathit{env}) \\
6 \quad \mathbf{pred}_c[\![\text{not pred}]\!](\mathit{env}) = \neg \mathbf{pred}_c[\![\text{pred}]\!](\mathit{env}) \\
7 \quad \mathbf{pred}_c[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!](\mathit{env}) = \mathbf{expr}_c[\![\text{expr}_1]\!](\mathit{env}) \\
8 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{rel\text{-}op}_c[\![\text{rel-op}]\!]\ \mathbf{expr}_c[\![\text{expr}_2]\!](\mathit{env})
\end{array}
$$

(a) Concrete Domain

$$
\begin{array}{l}
1 \quad \mathbf{pred}_s : \text{Predicate} \to C \to C \\
2 \quad \mathbf{pred}_s[\![\text{true}]\!]\big([s_1, p_1]\big) = [s_1, p_1] \\
3 \quad \mathbf{pred}_s[\![\text{false}]\!]\big([s_1, p_1]\big) = \lambda\big[s_2, p_2\big].\big[s_2,\ \mathit{false}\big]\big([s_1, p_1]\big) \\
4 \quad \mathbf{pred}_s[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]\big([s_1, p_1]\big) = \\
5 \qquad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\text{pc}\ \big(\mathbf{pred}_s[\![\text{pred}_1]\!]\big([s_2, \mathit{true}]\big)\big) \\
6 \qquad\qquad\qquad\qquad \wedge\ \text{pc}\ \big(\mathbf{pred}_s[\![\text{pred}_2]\!]\big([s_2, \mathit{true}]\big)\big)\big)\big]\ \big([s_1, p_1]\big) \\
7 \quad \mathbf{pred}_s[\![\text{pred}_1 \text{ or } \text{pred}_2]\!]\big([s_1, p_1]\big) = \\
8 \qquad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\text{pc}\ \big(\mathbf{pred}_s[\![\text{pred}_1]\!]\big([s_2, \mathit{true}]\big)\big) \\
9 \qquad\qquad\qquad\qquad \vee\ \text{pc}\ \big(\mathbf{pred}_s[\![\text{pred}_2]\!]\big([s_2, \mathit{true}]\big)\big)\big)\big]\ \big([s_1, p_1]\big) \\
10 \quad \mathbf{pred}_s[\![\text{not pred}]\!]\big([s_1, p_1]\big) = \\
11 \qquad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \neg\big(\text{pc}\ \big(\mathbf{pred}_s[\![\text{pred}]\!]\big([s_2, \mathit{true}]\big)\big)\big)\big]\ \big([s_1, p_1]\big) \\
12 \quad \mathbf{pred}_s[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!]\big([s_1, p_1]\big) = \\
13 \qquad \lambda\big[s_2, p_2\big].\big[s_2, p_2 \wedge \big(\ \mathbf{expr}_s[\![\text{expr}_1]\!]\big(s_2\big) \\
14 \qquad\qquad\qquad\qquad \mathbf{rel\text{-}op}_s[\![\text{rel-op}]\!]\ \mathbf{expr}_s[\![\text{expr}_2]\!]\big(s_2\big)\big)\big]\ \big([s_1, p_1]\big)
\end{array}
$$

(b) Symbolic Domain

Table 4.6: Comparison: Concrete vs. Symbolic Branch-Predicates

Case 6 requires us to show that

$$\overline{b} \wedge \big(\mathbf{pred}_c[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!](env)\big) = \\ = \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\text{expr}_1 \text{ rel-op } \text{expr}_2]\!]\big([s, \overline{b}\,]\big)\big)\big).$$

Substituting the right-hand sides of Case 6 from Table 4.6 and using the following shorthand notations

$$e_{c1} = \mathbf{expr}_c[\![\text{expr}_1]\!](env), \qquad e_{s1} = \mathbf{expr}_s[\![\text{expr}_1]\!](s),$$
$$e_{c2} = \mathbf{expr}_c[\![\text{expr}_2]\!](env), \qquad e_{s2} = \mathbf{expr}_s[\![\text{expr}_2]\!](s),$$

we get

$$\begin{aligned}\overline{b} \wedge \big(e_{c1} \,\mathbf{rel\text{-}op}_c[\![\text{rel-op}]\!]\, e_{c2}\big) &= \text{pc}\big(\sigma_{env}\big([s, \overline{b} \wedge (e_{s1} \,\mathbf{rel\text{-}op}_s[\![\text{rel-op}]\!]\, e_{s2})]\big)\big) \\ &= \sigma_{env}\big(\overline{b} \wedge (e_{s1} \,\mathbf{rel\text{-}op}_s[\![\text{rel-op}]\!]\, e_{s2})\big) \\ &= \overline{b} \wedge \big(\sigma_{env}(e_{s1}) \,\mathbf{rel\text{-}op}_s[\![\text{rel-op}]\!]\ \sigma_{env}(e_{s2})\big).\end{aligned}$$

It is the $\mathbf{expr}_s^c$-consistency that we have stated in Equation (4.28) which provides that $e_{c1} = \sigma_{env}(e_{s1})$, and that $e_{c2} = \sigma_{env}(e_{s2})$. Knowing that the relational connective $\mathbf{rel\text{-}op}_s[\![\text{rel-op}]\!]$ on the right-hand side of the above identity operates in this particular case on the concrete domain where it is identical to its standard semantic counterpart concludes Case 6.

Induction: In Case 3 we must show that

$$\overline{b} \wedge \big(\mathbf{pred}_c[\![\text{pred}_1 \text{ and } \text{pred}_2]\!](env)\big) = \\ = \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]\big([s, \overline{b}\,]\big)\big)\big).$$

Substituting the right-hand sides of Case 3 from Table 4.6 and using the following shorthand notations

$$p_{c1} = \mathbf{pred}_c[\![\text{pred}_1]\!](env), \qquad p_{s1} = \mathbf{pred}_s[\![\text{pred}_1]\!]\big([s, \mathit{true}]\big),$$
$$p_{c2} = \mathbf{pred}_c[\![\text{pred}_2]\!](env), \qquad p_{s2} = \mathbf{pred}_s[\![\text{pred}_2]\!]\big([s, \mathit{true}]\big),$$

we get

$$\begin{aligned}\overline{b} \wedge \big(p_{c1} \wedge p_{c2}\big) &= \text{pc}\big(\sigma_{env}\big([s, \overline{b} \wedge (\text{pc}(p_{s1}) \wedge \text{pc}(p_{s2}))]\big)\big) \\ &= \overline{b} \wedge \big(\sigma_{env}(\text{pc}(p_{s1})) \wedge \sigma_{env}(\text{pc}(p_{s2}))\big).\end{aligned}$$

The inductive hypothesis implies that $\mathit{true} \wedge p_{c1} = \sigma_{env}(\text{pc}(p_{s1}))$, and that $\mathit{true} \wedge p_{c2} = \sigma_{env}(\text{pc}(p_{s2}))$, which concludes Case 3.

Case 4 is dual to Case 3 in the sense that the syntactic token "and" is to be replaced by "or", and that the pathconditions of p_{s1} and p_{s2} are joined by disjunction.

Case 5 finally adds no new aspects to this routine:

$$\overline{b} \wedge \mathbf{pred}_c[\![\text{not pred}]\!](env) = \text{pc}\big(\sigma_{env}\big(\mathbf{pred}_s[\![\text{not pred}]\!]\big([s, \mathit{true}]\big)\big)\big)$$
$$\overline{b} \wedge \neg\, \mathbf{pred}_c[\![\text{pred}]\!](env) = \overline{b} \wedge \neg\, \sigma_{env}\big(\text{pc}\big(\mathbf{pred}_s[\![\text{pred}]\!]\big([s, \mathit{true}]\big)\big)\big).$$

Considering the fact that from the inductive hypothesis we have

$$true \wedge \mathbf{pred}_c[\![\text{pred}]\!](env) = \sigma_{env}\big(\text{pc}\big(\mathbf{pred}_s[\![\text{pred}]\!]\big([s, true]\big)\big)\big)$$

concludes the case.

With the proof of Case 5 we have established $\mathbf{pred}_s^c$-consistency of Flow predicate derivation trees as postulated in Equation (4.30). Combining this result with $\mathbf{expr}_s^c$-consistency proves the single-edge commutation property of concrete and symbolic execution as depicted in Figure 4.4. □

Having proved the commutation of single-edge concrete and symbolic execution, we will now extend this result to whole program paths. For this we need the following definition that extends substitutions from the domain of symbolic expressions and predicates to program states.

Definition 4.11 Any extended substitution $\widehat{\sigma}$ can be further extended to a mapping $\widehat{\widehat{\sigma}} : S \to S$ in the following way.

$$\widehat{\widehat{\sigma}}\,(s) = \lambda s.s\big[\forall v \in \mathcal{D}om(s) : v \mapsto \widehat{\sigma}\big(s(v)\big)\big]$$

As with Definition 4.6 we only distinguish between a substitution and its extensions at places where this distinction is crucial.

Lemma 3. Given a Flow program with its underlying control flow graph $G = \langle N, E, n_e, n_x\rangle$. For any program path π through G, the condition

$$M_c(\pi)\big([env, b]\big) = \text{con}\big(env, M_s(\pi)\big(\text{sym}\big([env, b]\big)\big)\big)$$

holds, which means that concrete and symbolic Flow program execution commute for program paths.

Proof. By induction on the length of program path π. The basis step $|\,\pi\,| = 1$ denotes the case of single-edge commutation already proved with Lemma 2.

Inductive step: Suppose the lemma is true for k edges, and let path π contain $k+1$ edges, that is, $|\,\pi\,| = k+1$. The commutative diagram of Figure 4.5 then illustrates the inductive step.

Starting with program context $[s, b]$, symbolic execution proceeds along edges $e_1, \ldots, e_k$ resulting in program context $[s_k, b \wedge p_1 \wedge \cdots \wedge p_k]$. Due to the assumption of the inductive step we know that concrete and symbolic execution along edges $e_1, \ldots, e_k$ commute, hence

$$[env_k, b \wedge b_1 \wedge \cdots \wedge b_k] = \text{con}(env, [s_k, b \wedge p_1 \wedge \cdots \wedge p_k]).$$

(In Figure 4.5 the environment env of the above equation is provided across projection P_{1_1}.) Moreover it follows from Lemma 2 that concrete and symbolic execution along edge $e_k + 1$ commute. This is depicted in the lower half of Figure 4.5: if we transform the extended environment $[env_k, b \wedge b_1 \wedge \cdots \wedge b_k]$ to the symbolic domain, we get the program context $c'_k = [s'_k, b \wedge b_1 \wedge \cdots \wedge b_k]$. Symbolic execution along edge e_{k+1} then gives us

$$M_s(e_{k+1})\big([s'_k, b \wedge b_1 \wedge \cdots \wedge b_k]\big) = [s'_{k+1}, b \wedge b_1 \wedge \cdots \wedge b_k \wedge p'_{k+1}] = c'_{k+1}.$$

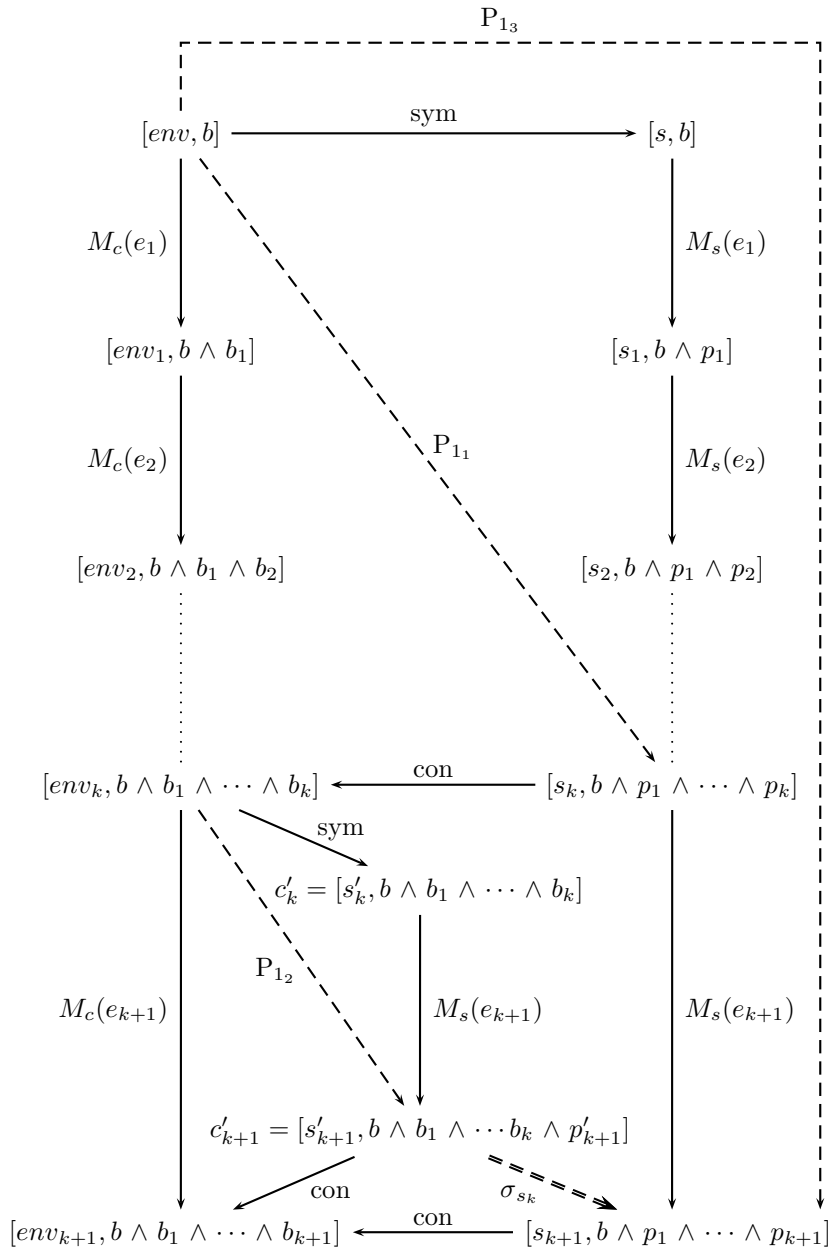


Figure 4.5: Commutation of Single-Path Concrete and Symbolic Execution

The resulting program context c'_{k+1} can be transformed back to the concrete domain, using the environment env_k provided through projection P_{1_2}. Due to Lemma 2 the result coincides with the result we get from concrete execution along edge e_{k+1}.

Now let σ_{s_k} be a substitution that transforms state s into state s_k, such that $\sigma_{s_k}(s) = s_k$. The substitution σ_{s_k} can then be written as

$$\sigma_{s_k} = \{\underline{v_1} \mapsto s_k(v_1), \ldots, \underline{v_j} \mapsto s_k(v_j)\}, \quad \text{with } \underset{1\leq i\leq j}{v_i} \in \mathcal{D}om(s_k). \tag{4.31}$$

Observe from Figure 4.5 that because σ_{s_k} transforms state s to state s_k, it corresponds to the accumulated symbolic side-effects along edges $e_1 \cdots e_k$. Furthermore it follows from the definition of function sym in conjunction with the fact that $\mathcal{D}om(env) = \mathcal{D}om(env_k)$ that $s = s'_k$, and therefore $\sigma_{s_k}(s'_k) = s_k$.

We will now use the already established single-edge commutation along edge e_{k+1} together with the inductive hypothesis to show that the program context $c_{k+1} = [s_{k+1}, b \wedge p_1 \wedge \cdots \wedge p_{k+1}]$ is indeed the result of symbolic execution for the input-context $[s, b]$ along edges $e_1 \cdots e_{k+1}$. We will achieve this by relating the program contexts c'_{k+1} and c_{k+1} via substitution σ_{s_k} as indicated in Figure 4.5.

To begin with, we note that the endomorphism-property of σ_{s_k} that is due to Equation (4.24) together with the fact that $\sigma_{s_k}(s'_k) = s_k$ ensures that

$$\sigma_{s_k}(\sigma_{\mathrm{s}}(s'_k)) = \sigma_{\mathrm{s}}(\sigma_{s_k}(s)) \Rightarrow \sigma_{s_k}(s'_{k+1}) = s_{k+1}.$$

In other words, starting with the clean slate program state s'_k to which we apply the side-effect σ_{s} of edge e_{k+1} results in state s'_{k+1}. Applying substitution σ_{s_k} to this state yields the same result as if we would start with the clean slate program state s, apply substitution σ_{s_k}, and subject the result to the side-effect associated with edge e_{k+1}. As a consequence, the endomorphism-property of σ_{s_k} also ensures that

$$\sigma_{s_k}(p'_{k+1}) = p_{k+1}.$$

This identity is already sufficient to determine the correctness of the pathcondition $b \wedge p_1 \wedge \cdots \wedge p_k \wedge p_{k+1}$ of program context c_{k+1}, since the correctness of the subcondition $b \wedge p_1 \wedge \cdots \wedge p_k$ is already due to the inductive hypothesis.

To conclude the proof we have to show that the transformation of program context c_{k+1} into the concrete domain, using the environment env provided via the projection function P_{1_3}, coincides with the extended environment $\overline{env}_{k+1} = [env_{k+1}, b \wedge b_1 \wedge \cdots \wedge b_{k+1}]$. In that case we have

$$\mathrm{con}\big(env, [s_{k+1}, b \wedge p_1 \wedge \cdots \wedge p_{k+1}]\big) = \overline{env}_{k+1}. \tag{4.32}$$

To show this we will use the already established single-edge commutation along edge e_{k+1} from which it follows that

$$\mathrm{con}\big(env_k, [s'_{k+1}, b \wedge b_1 \wedge \cdots \wedge b_k \wedge p'_{k+1}]\big) = \overline{env}_{k+1}. \tag{4.33}$$

To make our point we will relate the program contexts c'_{k+1} and c_{k+1} and use the commutation of context c'_{k+1} and environment env_k as stated in Equation (4.33) to prove the commutation of context c_{k+1} and environment env stated in Equation (4.32).

Relating contexts again boils down to relating the contained program states and pathconditions. From the inductive hypothesis of this lemma we may assume that states s'_k and s_k are related such that

$$\sigma_{env_k}(s'_k) = \sigma_{env}(s_k) \tag{4.34}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}\big(s'_k(v)\big) = \sigma_{env}\big(s_k(v)\big) \tag{4.35}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}\big(s_k(v)\big) \tag{4.36}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}\big(\sigma_{s_k}\big(s(v)\big)\big) \tag{4.37}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}\big(\sigma_{s_k}(\underline{v})\big). \tag{4.38}$$

In the above sequence of transformations the step from Equation (4.35) to Equation (4.36) is due to the fact that state s'_k maps a variable v to the corresponding initial value variable $\underline{v}$. The step from Equation (4.36) to Equation (4.37) is due to the properties of substitution σ_{s_k}. Finally, the step from Equation (4.37) to Equation (4.38) is due to the fact that state s, like state s'_k, maps a variable v to the corresponding initial value variable $\underline{v}$.

A similar relation between state s_{k+1} and state s'_{k+1} is required for the commutation stated in Equation (4.32):

$$\sigma_{env_k}(s'_{k+1}) = \sigma_{env}(s_{k+1}) \tag{4.39}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}\big(s'_{k+1}(v)\big) = \sigma_{env}\big(s_{k+1}(v)\big). \tag{4.40}$$

We have already noted in the proof of Lemma 2 that Flow program execution along a single edge e changes the value of exactly one program variable v_ω of a given environment or state. If it was not for that single program variable, Equation (4.35) would already imply Equation (4.40). Take now a given derivation tree expr $\in$ Expression constituting the syntactic representation of the update-value of the before-mentioned program variable v_ω. A final look at Figure 4.5 confirms that for our proof the side-effect of edge e_{k+1} is evaluated twice: once within state s'_k, yielding $e'_k(\mathbf{x}) = \mathbf{expr}_s[\![\text{expr}]\!](s'_k)$ as the corresponding symbolic expression, and once within state s_k, yielding $e_k(\mathbf{x}) = \mathbf{expr}_s[\![\text{expr}]\!](s_k)$.

As with derivation trees, we can represent those expressions as trees, with the initial value variables and integers constituting the leaves of the tree. As a consequence of this "leaf-ness" of initial value variables, substitution σ_{s_k} applies only to the leaves of expression trees where it replaces the initial value variables $\underline{v_j}$ with the expression (sub)trees corresponding to the respective symbolic expression $\sigma_{s_k}(\underline{v_j})$. An example of such a transformation is depicted in Figure 4.6.

Returning to our proof the final leap addresses the condition that must be met by the symbolic expressions $e'_k(\mathbf{x})$ and $e_k(\mathbf{x})$ to satisfy Equation (4.40) regarding program variable v_ω.

$$\sigma_{env_k}(e'_k(\mathbf{x})) = \sigma_{env}(e_k(\mathbf{x})) \tag{4.41}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(\underline{v})) \tag{4.42}$$

Equation (4.42) is suggested by the already mentioned "leaf-ness" of substitution σ_{s_k}, but the fact that Equation (4.41) holds when Equation (4.42) holds follows immediately by induction on the structure of symbolic expressions (cf. also

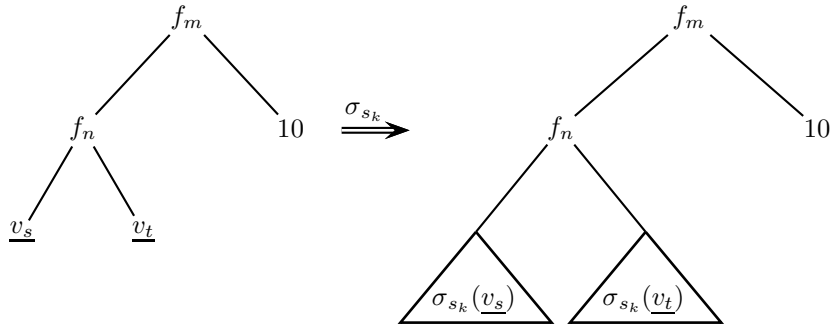


Figure 4.6: Application of Substitution σ_{s_k} to Expression $f_m(f_n(\underline{v_s}, \underline{v_t}), 10)$

Figure 4.6). The condition $\sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(\underline{v}))$ required by Equation (4.42) already follows from the inductive hypothesis of this lemma, which closes the case for program variable v_ω.

Because symbolic predicates are composed of symbolic expressions, we can close the case of the relation of predicates p'_{k+1} and p_{k+1} by the same argument. The fact that the correctness of the remaining subcondition $b \wedge p_1 \wedge \cdots \wedge p_k$ is already due to the inductive hypothesis finally establishes the commutation-property postulated in Equation (4.32), which also ends the proof of Lemma 3. □

With the definition of substitution σ_{s_k} in Equation (4.31) we noted that due to the transformation of state s to state s_k, σ_{s_k} corresponds to the accumulated symbolic side-effects along edges $e_1 \cdots e_k$. This idea can be generalized to edge transition functions, giving way to the following corollary of Lemma 3.

Corollary. Given a Flow program with its underlying control flow graph $G = \langle N, E, n_e, n_x \rangle$. For any program path $\pi = \langle e_1, e_2, \ldots, e_k \rangle$ through G, the program context $[s_k, p_k] = M_s(\pi)([s, p])$ that results from symbolic execution along path π is a partial function of arity $\mathit{Environment} \times \mathbb{B} \to \mathit{Environment} \times \mathbb{B}$ that represents the function composition $M_s(e_k) \circ M_s(e_{k-1}) \circ \cdots \circ M_s(e_1)$. This partial function is defined for those environments env for which the standard-semantic iterated transition function δ^* is defined (e.g., not in case of division by zero or other program anomalies). It coincides with the standard-semantic iterated transition function δ^* for those environments env for which $\sigma_{env}(p_k) = \mathit{true}$.

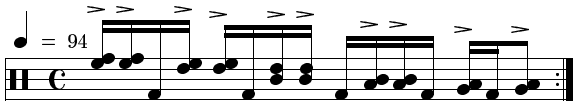
Theorem 2. Correctness of the MOP solution.

$$\bigcup_{\pi \in \mathrm{Path}(n_e, n)} \mathrm{con}(env, M_s(\pi)(\mathrm{sym}[env, b])) = \bigcup_{\pi \in \mathrm{Path}(n_e, n)} M_c(\pi)([env, b]).$$

Proof. Immediate from Lemma 3. □

Chapter 5

Symbolic Evaluation



— staccato drum-break from Phil Collins' "In the Air Tonight".

The symbolic execution approach laid out in the previous chapter is capable of computing the MOP-solution for arbitrary program points. It is however not constructive in the sense that we have not specified a method to obtain the set of program paths leading from the entry node to a give program point needed by this approach. Furthermore, the delivered MOP-solution is infinite.

In this chapter we will lay the foundations for a MOP-solution that is both constructive and finite. As a first step we set up a method to describe program paths by means of a regular expression algebra.

5.1 Program Paths and Regular Expression Algebras

It is shown in [Tar81] how program paths π can be represented as regular expressions: Let Σ be a finite alphabet disjoint from $\{\Lambda, \emptyset, (,)\}$. A *regular expression* is any expression built by applying the following rules:

(1a) "Λ" and "$\emptyset$" are *atomic* regular expressions; for any $a \in \Sigma$, "a" is an atomic regular expression.

(1b) If R_1 and R_2 are regular expressions, then $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and $(R_1)^*$ are *compound* regular expressions.

In a regular expression, Λ denotes the empty string, $\emptyset$ denotes the empty set, $+$ denotes union, $\cdot$ denotes concatenation, and * denotes reflexive, transitive closure under concatenation. Thus each regular expression R over Σ defines a set $L(R)$ of strings over Σ as follows:

(2a) $L(\Lambda) = \{\Lambda\}$; $L(\emptyset) = \emptyset$; $L(a) = \{a\}$ for $a \in \Sigma$.

(2b) $L(R_1 + R_2) = L(R_1) \cup L(R_2) = \{w \mid w \in L(R_1) \text{ or } w \in L(R_2)\}$;

(2c) $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2) = \{w_1 w_2 \mid w_1 \in L(R_1) \text{ and } w_2 \in L(R_2)\}$;

(2d) $L(R^*) = \bigcup_{k=0}^{\infty} L(R)^k$, where $L(R)^0 = \{\Lambda\}$, and $L(R)^i = L(R)^{i-1} \cdot L(R)$.

Two regular expressions R_1 and R_2 are said to be *equivalent*, denoted by $R_1 \sim R_2$, if $L(R_1) = L(R_2)$. A regular expression R is *simple* if $R = \emptyset$ or R does not contain $\emptyset$ as a subexpression. According to the definition given in [Sal66, p. 159] two regular expressions are *identical*, denoted by $R_1 \equiv R_2$, if they contain the same symbols in the same order.

Given a CFG $G = \langle N, E, n_e, n_x \rangle$, we can regard any path π in G as a string over E, but not all strings over E are paths in G. A *path expression* P of *type* (v, w) is a simple regular expression over E such that every string in $L(P)$ is a program path from node v to node w.

5.2 Interpretation of Path Expressions

Courtingdisastus: Ave Praetor! This man wants to spin you a yarn.
Pirate: No I don't! I'm an honest sailor working the Massilia–Corsica crossing...
— From "Asterix in Corsica", by R. Goscinny and A. Uderzo.

Due to the introduction of supercontexts in Equation (4.21) we have to extend domain and codomain of the function class F introduced in Equation (4.16) from contexts to supercontexts, yielding a new function class F_{sc}:

$$F_{\text{sc}} \subseteq \{f_{\text{sc}} : SC \to SC\}. \tag{5.1}$$

We achieve this extension with the help of the wrapping operator wrap which constructs a function $f_{\text{sc}} \in F_{\text{sc}}$ of arity $SC \to SC$ from a function $f_{\text{c}} \in F$ of arity $C \to C$ in passing each context of the supercontext-argument of f_{sc} through f_{c}:

$$\begin{aligned} &\text{wrap} : (C \to C) \to (SC \to SC) \\ &\text{wrap}\big(f_{\text{c}}\big)(sc) :: = f_{\text{sc}}\left(\left[\bigcup_{i=0}^{\infty} [s_i, p_i]\right]\right) = \left[\bigcup_{i=0}^{\infty} f_{\text{c}}\big([s_i, p_i]\big)\right]. \end{aligned} \tag{5.2}$$

The function class F_{sc} has the following properties, which are easily verified from the definition of the wrapping operator, the properties of supercontexts (cf. Section 4.4.1), and the properties of the function class F on which F_{sc} is based upon.

F1) F_{sc} contains the identity function ι.

F2) F_{sc} is closed under $\cup$: $\forall f, g \in F_{\text{sc}} : (f \cup g)(x) = f(x) \cup g(x)$.

F3) F_{sc} is closed under composition: $\forall f, g \in F_{\text{sc}} : f \circ g \in F_{\text{sc}}$.

F4) F_{sc} is closed under iterated composition:

$$f^*(x) = \Big[\bigcup_{i \geq 0} f^i(x)\Big],$$

where $f^0 = \iota$ and $f^i = f^{i-1} \circ f$. Since $\langle SC, \cup \rangle$ is closed under the supercontext union operation $\cup$ (cf. Section 4.4.1), the union over $f^i(x)$ is again a supercontext.

F5) Continuity of $f \in F_{\text{sc}}$ across supercontext union $\cup$:

$$\forall f \in F_{\text{sc}} \text{ and } X \subseteq SC : f(\cup X) = \left[\bigcup_{x \in sc} f(x) \right].$$

Based on the edge transition function M (cf. Section 4.2.3) for the function class F we define a new edge transition function M_{sc} that encapsulates the wrapping operator inside:

$$\begin{aligned} &M_{\text{sc}} : E \to F_{\text{sc}} \\ &M_{\text{sc}}(e) ::= \text{wrap}\big(M(e)\big). \end{aligned} \tag{5.3}$$

We can compose edge transition functions from function class F_{sc} along program paths in the same way as already shown for function class F in Equation (4.18). In a similar way we use the shorthand notation f_e for $M_{\text{sc}}(e)$, and f_π for $M_{\text{sc}}(\pi)$.

Let $P \neq \emptyset$ be a path expression of type (v, w). For all $x \in SC$, we define a mapping ϕ as follows.

M1) $\phi(\Lambda) = \iota$,

M2) $\phi(e) = M_{\text{sc}}(e) = f_e$,

M3) $\phi(P_1 + P_2) = \phi(P_1) \cup \phi(P_2)$,

M4) $\phi(P_1 \cdot P_2) = \phi(P_2) \circ \phi(P_1)$,

M5) $\phi(P_1^*) = \phi(P_1)^*$.

Lemma 4. Let $P \neq \emptyset$ be a path expression of type (v, w). Then for all $x \in SC$,

$$\phi(P)(x) = \Big[\bigcup_{\pi \in L(P)} f_\pi(x) \Big].$$

Proof. By induction on the number of operation symbols in P. The lemma is immediate if P is atomic:

$$\phi(P)(x) = \phi(e)(x) = f_e(x).$$

Inductive hypothesis $\mathcal{H}$: Suppose the lemma is true for path expressions containing fewer than k operation symbols, and let P contain k operation symbols. We distinguish three cases.

Case 1: $P = P_1 + P_2$. Then

$$\begin{aligned}
\phi(P)(x) &= \phi(P_1)(x) \cup \phi(P_2)(x) && \text{by (F2) and (M3)}\\
&= \Big[\bigcup_{\pi\in L(P_1)} f_\pi(x)\Big] \cup \Big[\bigcup_{\pi\in L(P_2)} f_\pi(x)\Big] && \text{by } \mathcal{H}\\
&= \Big[\bigcup_{\pi\in L(P_1)\cup L(P_2)} f_\pi(x)\Big] && \text{by (P1) on p. 57}\\
&= \Big[\bigcup_{\pi\in L(P_1+P_2)} f_\pi(x)\Big] && \text{by (2b)}\\
&= \Big[\bigcup_{\pi\in L(P)} f_\pi(x)\Big] && \text{by } P = P_1 + P_2
\end{aligned}$$

Case 2: $P = P_1 \cdot P_2$. Then

$$\begin{aligned}
\phi(P)(x) &= \phi(P_2) \circ \phi(P_1)(x) && \text{by (M4)}\\
&= \phi(P_2)\big(\phi(P_1)(x)\big)\\
&= \phi(P_2)\Big(\Big[\bigcup_{\pi_1\in L(P_1)} f_{\pi_1}(x)\Big]\Big) && \text{by } \mathcal{H}\\
&= \Big[\bigcup_{\pi_1\in L(P_1)} \phi(P_2)\big(f_{\pi_1}(x)\big)\Big] && \text{by continuity (F5)}\\
&= \Big[\bigcup_{\pi_1\in L(P_1)} \Big[\bigcup_{\pi_2\in L(P_2)} f_{\pi_1\pi_2}(x)\Big]\Big] && \text{(cartesian product)}\\
&= \Big[\bigcup_{\pi_1\in L(P_1)\wedge\pi_2\in L(P_2)} f_{\pi_1\pi_2}(x)\Big]\\
&= \Big[\bigcup_{\pi\in L(P_1\cdot P_2)} f_\pi(x)\Big] && \text{by (2c)}\\
&= \Big[\bigcup_{\pi\in L(P)} f_\pi(x)\Big] && \text{by } P = P_1 \cdot P_2
\end{aligned}$$

Case 3: As in case 2 we can show that if P_1 has fewer than k operation symbols, then

$$\phi(P_1)^i(x) = \Big[\bigcup_{\pi\in L(P_1)^i} f_\pi(x)\Big] \tag{5.4}$$

for any $i \geq 0$. Suppose $P = P_1^*$. Then

$$\begin{aligned}
\phi(P)(x) &= \phi(P_1^*)(x) \\
&= \phi(P_1)^*(x) && \text{by (M5)} \\
&= \Big[\bigcup_{\substack{\pi \in L(P_1)^i, \\ i \geq 0}} f_\pi(x) \Big] && \text{by Equation (5.4)} \\
&= \Big[\bigcup_{\pi \in L(P_1^*)} f_\pi(x) \Big] && \text{by (2d)}
\end{aligned}$$

□

5.3 The Meet Over All Paths Solution Revised

We have already introduced the meet over all paths solution for symbolic execution in Section 4.4.2. It is based on the set of program paths $\pi \in \text{Path}(n_e, n)$ from the entry node n_e to a given node n of a control flow graph. In Section 4.4.2 we have left the origin of those program paths open, but with the introduction of path expressions we have now a formalism at hand that allows us to describe the set of program paths between two nodes of a control flow graph.

Moreover, as we will see in Section 6.2, there even exist data flow algorithms that are capable of calculating those path expressions.

Hence we are ready to introduce the *supercontext union over all paths solution (MOP)* in the following theorem.

Theorem 3. For any node n let $P(n_e, n)$ be a path expression representing all paths from n_e to n. Then

$$\text{mop}(n) = \phi\big(P(n_e, n)\big)(sc_0), \tag{5.5}$$

where sc_0 denotes the initial supercontext valid at entry node n_e.

Proof. Let $P \neq \emptyset$ be a path expression of type (n_e, n). The proof follows then immediately from Lemma 4. □

5.4 Loops, Induction Variables, and Systems of Recurrences

In [ASU86, p. 643] a program variable $v \in \mathbb{V}$ is called an *induction variable* of a loop L if every time the variable v changes values, it is incremented or decremented by some constant. The fact that the sequence of values a variable receives during execution of a loop is usually more complex resulted in more general definitions of classes of variables (cf. e.g. [Hag95], with [GSW95] containing a comprehensive classification and further references).

For the purpose of the following sections a general view is sufficient where we speak of an *induction variable* $v \in \mathbb{V}$ of a loop L if v changes values within L.

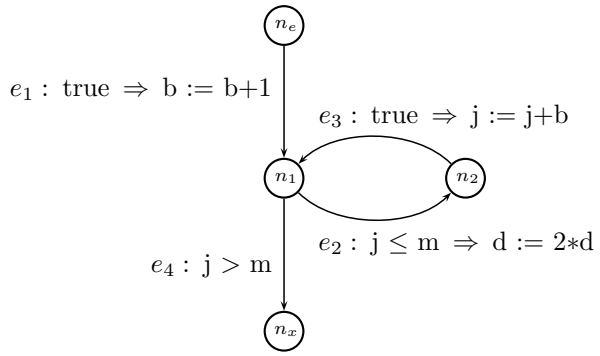


Figure 5.1: Example Loop

As an example we consider the control flow graph* depicted in Figure 5.1. It contains the loop L which consists of the edges e_2 and e_3. From the side-effects associated with those edges it follows that program variables f and i are the induction variables of L.

Assume that we want to compute the MOP solution for node n_1 which is due to the path expression $e_1 \cdot (e_2 \cdot e_3)^*$ of type (n_e, n_1). From Theorem 3 it follows that

$$\mathrm{mop}(n_1) = \phi\big(e_1 \cdot (e_2 \cdot e_3)^*\big)(sc_0),$$

where sc_0 denotes the initial supercontext valid at entry node n_e. Working the mapping ϕ further down the line involves the computation of $\phi(e_2 \cdot e_3)^*$. Let $f = M_{\mathrm{sc}}(e_3) \circ M_{\mathrm{sc}}(e_2)$ be the accumulated effect of one iteration of the loop body, then, due to the closure property regarding iterated composition (cf. Property (F4) on p. 74),

$$f^*(sc_0) = \Big[\bigcup_{i\geq 0} f^i(sc_0)\Big] = \Big[\bigcup_{i\geq 0} \underbrace{f \circ \cdots \circ f}_{i \text{ times}}(sc_0)\Big].$$

The interpretation of the above equation with respect to the control flow graph of Figure 5.1 is such that, provided we start symbolic execution at node n_1 with supercontext sc_0, $f^i(sc_0)$ represents the result after i iterations of loop L. We illustrate this with the one-element supercontext

$$sc_0 = c = \big[\{(b, \underline{b}), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\},\ true\big],$$

where, as a notational convenience, we depict the graph of the state of c rather than the state itself. It is instructive to consider $f^i(c)$ for a few values of i.

$$c_0 = f^0(c) = \big[\{(b, \underline{b}), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\},\ true\big] = \iota(c) = c \tag{5.6}$$

$$c_1 = f^1(c) = \big[\{(b, \underline{b}), (d, 2 \cdot \underline{d}), (j, \underline{j} + \underline{b}), (m, \underline{m})\},\ \underline{j} \leq \underline{m}\big] \tag{5.7}$$

$$c_2 = f^2(c) = \big[\{(b, \underline{b}), (d, 4 \cdot \underline{d}), (j, \underline{j} + 2 \cdot \underline{b}), (m, \underline{m})\},\ \underline{j} \leq \underline{m} \wedge \underline{j} + \underline{b} \leq \underline{m}\big] \tag{5.8}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$c_i = f^i(c) = \big[\{(b, \underline{b}), (d, 2^i \cdot \underline{d}), (j, \underline{j} + i \cdot \underline{b}), (m, \underline{m})\},\ true \bigwedge_{1\leq j\leq i} p(j-1)\big] \tag{5.9}$$

*This example originally appeared in [Sch01].

In this way program context c_0 of Equation (5.6) represents the result after zero iterations of loop L. Likewise, c_1 of Equation (5.7) and c_2 of Equation (5.8) denote the program contexts after the first and the second iteration respectively. Given program context c_1, the branch predicate $\mathrm{j} \leq \mathrm{m}$ of edge e_2 evaluates to $\underline{j} + \underline{b} \leq \underline{m}$, which is manifested in the pathcondition of program context c_2.

Investigation into this series of loop iterations allows us to set up a "generic" program context that represents the result after i iterations of loop L. It is based on the observation that the values a given induction variable assumes during subsequent iterations constitutes a sequence of terms that can be expressed as a *recurrence relation*. Moreover, since the variable values of the current iteration i are the sole basis for the values of the subsequent iteration $i + 1$, the definition of a term $a(i + 1)$ of such a sequence is restricted to induction variable sequence terms of index i and precludes terms of index $k < i$. The only exception to this rule is the initial term which corresponds to the value of the respective induction variable upon entry of the loop. Since the initial term satisfies the sequence before the recurrence relation takes effect, it is called the initial condition or *boundary condition* of the recurrence relation.

Due to the before-mentioned restriction one iteration of the loop body of L, represented as a function $f_{\mathrm{sc}} \in F_{\mathrm{sc}}$, completely defines the recurrence relations of the induction variables contained in L. In order to derive this system of recurrence relations from L, we construct an initial program context

$$c_0 = [s_0, \mathit{true}], \tag{5.10}$$

where s_0 is a clean slate program state, and compute

$$sc = f_{\mathrm{sc}}^1(c_0). \tag{5.11}$$

The set IV of induction variables is then characterized as

$$IV = \bigcup_{c \in sc} \{v_i \in \mathbb{V} \mid \mathrm{st}(c)(v_i) \neq \underline{v_i}\}. \tag{5.12}$$

For each induction variable $v \in IV$ the supercontext sc defines the corresponding recurrence relation under the boundary condition

$$v(0) = \underline{v}, \tag{5.13}$$

which is due to s_0 of Equation (5.10) being a clean slate program state. Because of the endomorphism property stated in Equation (4.24) this boundary condition is general in the sense that we can substitute arbitrary symbolic expressions for $\underline{v}$.

Returning to our example we note that the initial program context of Equation (5.10) corresponds to program context c_0 of Equation (5.6), and that the supercontext sc derived from Equation (5.11) is represented by the program context c_1 of Equation (5.7). The fact that there is only one program path through the loop body of our example is the reason for sc being just a one-element supercontext. From Equation (5.12) and context c_1 of Equation (5.7) we derive the set of induction variables for loop L, namely $\{d, j\}$. From the symbolic expressions computed for context c_1 and the boundary condition stated in Equation (5.13) we can set up the recurrence relations for the variables in IV.

$$d(i+1) = 2 \cdot d(i), \text{ for } i \geq 0 \qquad j(i+1) = j(i) + \underline{b}, \text{ for } i \geq 0$$
$$d(0) = \underline{d} \qquad j(0) = \underline{j}$$

The closed forms for the i^{th} terms of the above recurrence relations are

$$\begin{aligned} d(i) &= 2^i \cdot \underline{d}, \quad \text{for } i \geq 0, \\ j(i) &= \underline{j} + i \cdot \underline{b}, \quad \text{for } i \geq 0. \end{aligned} \tag{5.14}$$

Substituting these closed forms for $\underline{d}$ and $\underline{j}$ in the clean slate state of Equation (5.6) gives us the state of the "generic" program context c_i depicted in Equation (5.9).

A note is due to the pathcondition of c_i: after i iterations the pathcondition of the program path $e_2 \cdot e_3$ has been evaluated i times, and the pathcondition

$$\mathit{true} \bigwedge_{1 \leq j \leq i} p(j-1) \tag{5.15}$$

of the resulting program context c_i is the conjunction of the initial pathcondition (*true*) and the i branch-predicates resulting from the i iterations along program path $e_2 \cdot e_3$ (cf. Table 4.4). In order to have a compact notation for a pathcondition consisting of i evaluations of a single branch-predicate, we write $p(j-1)$ to denote the j^{th} evaluation of branch-predicate p, which happens in terms of the induction variable values of iteration $j-1$ of program context c_{j-1} (cf. Equations (5.6), (5.7), and (5.8)).

Together with the recurrence relations and their boundary conditions the *recurrence condition* symbolically determines the number of iterations of a loop L. In our example the recurrence condition is given by

$$j(i) \leq \underline{m},$$

and we can determine the number of iterations of the loop which requires us to find a z such that

$$\begin{aligned} z &= \min\big(\{i \mid j(i) > \underline{m}\}\big) \\ &= \min\big(\{i \mid \underline{j} + i \cdot \underline{b} > \underline{m}\}\big). \end{aligned}$$

Due to the integer arithmetic of the domain SymExpr of symbolic expressions we are only interested in integer solutions of the above problem. Hence the determination of the upper bound of a loop involves the solution of a Diophantine equation (cf. [Hil00]). Although this problem is unsolvable in the general case (cf. [Dav82]), there exist several solution methods for distinct classes of Diophantine equations (cf. e.g., [Sma96]).

A final note is due to the determination of closed forms for recurrence relations. The literature contains several classifications for recurrence relations as well as solution procedures, cf. e.g., [Lue80, GKP94, GK82, CLRS01, BZZ03].

Recent research in the area of program optimization and parallelization focused, among other topics, on the recognition and classification of induction variables and their associated closed forms (cf. e.g. [GSW95, Hag95]). Another recent approach, which is also capable of handling conditional recurrence relations, is described in [vE00, vE01, vEBS$^+$04]. Section 7.5 in the related work gives an overview of recent developments in this area.

As it is pointed out in [Bli02, p. 30], the determination of closed forms for recurrence relations is in the general case undecidable. If we cannot find a closed

form for a recurrence relation, we introduce a new value symbol for the unresolved variable (the variable for which no closed form can be determined). Value symbols of unresolved variables are linked to the associated recurrence. Evaluation of unresolved variables has to proceed based on the new value symbols instead of the recurrences. This method is due to [FS03].

Alternatively analysis precision can be degraded by resorting to a simpler semantics[†] that will derive a weaker (yet computable) result for the unresolved variable. For instance, [WCHP01] and [Fah98] present algorithms to determine the monotonic behavior of unresolved variables. Another method operating on imprecise semantics is outlined in the related work in Section 7.2.

The ultimate goal with such a degradation of analysis precision is of course to stay as precise as possible, which is supported by the fact that with symbolic analysis approximations can be introduced on a per variable basis. It is a topic for further investigations to what extend several approximations can be combined to achieve a stronger result (e.g., the monotonic behavior of a variable together with another abstraction of its value towards < 0, 0, or > 0.

5.5 Symbolic Evaluation on the Form Level

Said a young man named A. Grothendieck[‡]:
In Geometry I'm rather weak.
I'm no Altshiller-Court[§];
It's just not my forte,
So I'd best make it more Algébrique.

— Limerick from [American Mathematical Monthly 73], commenting on the increasing importance of algebra for the mathematical sciences.

5.5.1 Closure Contexts

In Section 4.4.1 we have introduced the supercontext as a means to express the MOP solution for symbolic analysis. We have chosen a set-based approach where a supercontext is a set of program contexts. Given a path expression $P(n_e, n)$ of type (n_e, n), we know from Theorem 3 (p. 77) that

$$\mathrm{mop}(n) = \phi\big(P(n_e, n)\big)(c_0) = \left[\bigcup_{k=0}^{\infty} [s_k, p_k]\right], \tag{5.16}$$

where c_0 denotes the initial program context valid at entry node n_e. Every program path of the path expression $P(n_e, n)$ contributes one program context $[s_k, p_k]$ to the solution of Equation (5.16). Strictly speaking, this solution becomes infinite only if the Flow graph corresponding to path expression $P(n_e, n)$ contains cycles (this has been pointed out in Section 4.4). In this case $P(n_e, n)$ contains at least one * operator.

While Equation (5.16) is an *exact* characterization of the MOP solution at node n, it is undecidable due to its infiniteness-property which in turn prohibits

[†]Simpler semantics are not considered in this thesis.
[‡]Alexander Grothendieck, born 1928 in Berlin, working in the area of algebraic geometry.
[§]Nathan Altshiller Court (1881–1968), geometer.

its use on the form level. To be able to use supercontexts on the form level, we have to find a finite characterization of a control flow graph cycle, that is, the argument of the * operator of path expressions, in terms of the infinitely many program contexts it adds to the solution of Equation (5.16).

We will now introduce a finite representation of the infinitely many program contexts that are due to a cycle. This representation is an extension of the concept of program contexts, and we call this finite representation a *closure context*. With the help of closure contexts the solution of Equation (5.16) becomes finite, expressible by a *finite* supercontext.

For the following discussion we reserve the term *loop body* for the control flow graph portion corresponding to the argument of the * operator of path expressions.

Definition 5.1 In analogy to the set $\mathbb{V}$ of program variables (cf. Definition 2.1), we define the set $\mathbb{L}$, $\mathbb{V} \cap \mathbb{L} = \emptyset$, of loop index variables. Given the n-bounded finite index-set $\text{IS} = \{x \mid x \leq n\} \subset \mathbb{N}$ and a total function $\text{Idl} : \text{IS} \to \mathbb{L}$ that is one-to-one and onto, we write l_i, with $i \in \text{IS}$, to denote element $\text{Idl}(i)$ of $\mathbb{L}$. If the meaning is clear from context, we also use unique named constants that are functions of arity $\to \mathbb{L}$. These are denoted by lowercase letters, e.g., $\mathsf{l}, \mathsf{m}, \mathsf{n}$. Conceptionally a loop index variable can be envisioned as an artificial program variable that is assigned the value 0 upon entry to the loop body. After each iteration of the loop body, its value is increased by one. As an

```
1  b := b + 1;
2  while j <= m loop
3     d := 2*d;
4     j := j+b;
5  end loop;
```

```
1  b := b + 1;
2  l := 0;
3  while j <= m loop
4     d := 2*d;
5     j := j+b;
6     l := l+1;
7  end loop;
```

Figure 5.2: Implicit and Explicit Loop Index Variable

example, Figure 5.2 contains the textual representation of our running example from Section 5.4, together with a "conceptional" version with an explicit loop index variable (cf. line 2 and 6). Associated with a loop index variable l_i is a (symbolic) upper bound, denoted by $\mathsf{l}_{i,\omega}$ (or just l_ω for a named constant l). This upper bound corresponds to the number of loop iterations. In Section 5.4 we have shown how the number of loop iterations can be computed from the recurrence condition of the loop.

We have also shown in Section 5.4 how the sequence of values that an induction variable assumes during subsequent loop iterations can be described by recurrence relations. An extension of the set of symbolic expressions of the domain SymExpr allows us to describe recurrence relations, closed forms, and recurrence conditions within SymExpr.

Definition 5.2 The inductive definition of the set of symbolic expressions of the domain SymExpr (cf. Definition 4.3) is extended by

(v) $\mathbb{L} \subset \text{SymExpr}$ (i.e., loop index variables are symbolic expressions),

(vi) for all $v_i \in \mathbb{V}$, and $\mathsf{l}_j \in \mathbb{L}$, $v_i(0) \in \text{SymExpr}$, $v_i(\mathsf{l}_j) \in \text{SymExpr}$, $v_i(\mathsf{l}_j + 1) \in \text{SymExpr}$, and $v_i(\mathsf{l}_j - 1) \in \text{SymExpr}$ (i.e., dereferencing the value of a program variable to specify a recurrence relation yields a symbolic expression).

Definition 5.3 A *range expression* is a symbolic expression of the form

$$0 \leq \mathsf{l}_j \leq \mathsf{l}_{j,\omega},$$

with loop index variable $\mathsf{l}_j \in \mathbb{L}$, and $\mathsf{l}_{j,\omega}$ being the upper bound of l_j. Specifically, an upper bound of $\mathsf{l}_{j,\omega} = 0$ implies zero loop iterations, as can be inferred from Figure 5.2¶. We extend the set of symbolic predicates of the domain SymPred (cf. Definition 4.4) by the following rule to include range expressions:

(iv) for all $\mathsf{l}_j \in \mathbb{L}$, $0 \leq \mathsf{l}_j \leq \mathsf{l}_{j,\omega} \subset \text{SymPred}$ (i.e., range expressions constitute symbolic predicates).

As already described in Section 5.4, a recurrence describing an induction variable consists of the boundary condition, the recurrence relation, and the recurrence condition. Alternatively, a recurrence can be characterized by a closed form expression and a recurrence condition. For the present discussion this level of formalization of a recurrence is sufficient.

Definition 5.4 We denote a system of recurrences‖ over loop index variable l by $rs(\mathsf{l})$. It is an element of RS, the set of possible recurrence systems. We can construct a set rss of k recurrence systems by

$$\bigcup_{s=0}^{k} rs\big(\text{Idl}(s)\big).$$

The loop index variables of the above set are unique, which is a crucial property in order to avoid interferences across recurrence systems. We base the remainder of this work on the assumption that loop index variables of recurrence system sets are always unique in the above way. This is in fact no restriction since we can always replace a loop index variable l_i in a recurrence system $rs(\mathsf{l}_i)$ by another loop index variable l_j, thereby converting $rs(\mathsf{l}_i)$ into $rs(\mathsf{l}_j)$ without changing the underlying algebraic properties of the recurrence system at hand.

Recurrence system sets can be nested, and the set of all recurrence system sets is denoted by RSS. For our purpose it is beneficial to impose a total order $\leq$ on the elements of a recurrence system set in order to obtain the semantics of a list. In particular, we make use of the common Lisp-like functions

- Append : $RSS_1 \times RSS_2 \rightarrow RSS$ to append an element (from RSS_1) to a list (from RSS_2),
- First : $RSS \rightarrow RSS$, to retrieve the first element of a list (our intended use prevents us from applying this operation to a non-nested recurrence system set), and

¶This contrasts the notion of range expressions in contemporary programming languages, where `range L..U` denotes the interval [L, U] (cf. also [Ada95, Section 3.5(4)]).

‖Or recurrence system, for short.

- Rest : $RSS \to RSS$ to retrieve the rest (everything but the first element) from a list.

We have now everything in place to extend the notion of program contexts to closure contexts.

Definition 5.5 A closure context $\overline{c}$ is an element of the product $\overline{C} = S \times \text{SymPred} \times RSS$, denoted by $[s, p, rss]$. For a clean slate closure context the state s is a clean slate program state, p is a *true* pathcondition, and rss is the empty set. A program context $c = [s, p]$ can be considered a special case of a closure context $\overline{c} = [s, p, rss]$ with $rss = \emptyset$. A supercontext consisting of a finite number of closure contexts is denoted by $\overline{sc}$, for the set of all such finite supercontexts we write $\overline{SC}$.

Definition 5.6 Let $P \neq \emptyset$ be a path expression of type (v, w). For all $\overline{sc} \in \overline{SC}$, we define a mapping θ by

A1) $\theta(\Lambda) = \iota$,

A2) $\theta(e) = M_{\text{sc}}(e) = f_e$,

A3) $\theta(P_1 + P_2) = \theta(P_1) \cup \theta(P_2)$,

A4) $\theta(P_1 \cdot P_2) = \theta(P_2) \odot \theta(P_1)$,

A5) $\theta(P_1^*) = \theta(P_1)^{\circledast}$.

This mapping differs from the mapping used with Lemma 4 from Section 5.2 in two operators, namely $\odot$ and $\circledast$. We proceed with the description of these operators.

Definition 5.7 Let $f = \theta(P)$ be a functional description of the accumulated side-effect of one iteration of a given loop body represented by the path expression P. For a given closure context $\overline{c_{in}} = [s_{in}, p_{in}, rss_{in}]$ we define the properties of the closure context $\overline{c_{out}} = [s_{out}, p_{out}, rss_{out}]$ resulting from the application of $f^{\circledast}$ to $\overline{c_{in}}$, that is,

$$\overline{c_{out}} = f^{\circledast}(\overline{c_{in}}). \tag{5.17}$$

The alert reader will have noticed that the intended definition of the operator $\circledast$ and the mapping θ have a mutual dependency that we can however overcome in participating loop nests and starting with the innermost loop first (the innermost loop body lacks the occurrence of a $\circledast$ operator). In this way the mutual dependency with this definition is transformed into a dependency of $\circledast$ on θ.

We have already pointed out in Section 5.4 that one iteration of the loop body determines the system of recurrences that is due to the induction variables of the loop body. In analogy to Equations (5.10) and (5.11) we start with a clean slate closure context $\overline{c_0} = [s_0, p_0, rss_0]$ and compute the result of symbolic evaluation of one iteration of the loop body, denoted by $\overline{c_1}$.

$$\overline{c_1} = [s_1, p_1, rss_1] = f(\overline{c_0}). \tag{5.18}$$

A substitution $\sigma_{s,e}$ for a given state s and an expression $e \in \text{SymExpr}$ is defined such that

$$\sigma_{s,\mathrm{l}} = \{\underline{v_1} \mapsto v_1(e), \ldots, \underline{v_j} \mapsto v_j(e)\}, \quad \text{with } \underset{1 \leq i \leq j}{v_i} \in \mathcal{D}om(s). \tag{5.19}$$

What follows is the description of $\overline{c_{out}}$ in terms of its state s_{out}, its pathcondition p_{out}, and its recurrence system set rss_{out}. Nested loops deserve treatment in their own respect.

State: The state s_{out} is computed from s_{in} by replacing the symbolic expressions that describe the values of the variables v_i by the value of the recurrence relation for v_i over loop index variable l.

$$\forall v_i \in \mathcal{D}om(s_{in}) : \; s_{out} ::= s_{in}[v_i \mapsto v_i(\mathsf{l})] \tag{5.20}$$

The graph of state s_{out} is therefore of the form $\{(v_1, v_1(\mathsf{l}), \ldots, (v_n, v_n(\mathsf{l})\}$.

Pathcondition: The pathcondition p_{out} of closure context $\overline{c_{out}}$ has the form

$$p_{in} \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge_{\mathsf{l}'=1}^{\mathsf{l}} p(\mathsf{l}' - 1). \tag{5.21}$$

Therein the first term constitutes the pathcondition of closure context $\overline{c_{in}}$. The second term is a range expression according to Definition 5.3. It defines the value of the loop index variable l to be in the interval $[0, \mathsf{l}_\omega]$. The third term denotes the pathcondition accumulated during l iterations of the loop (cf. also Equation (5.15)). It is actually a conjunction of l instances of the pathcondition p_1 from Equation (5.18), where the l'^{th} instance corresponds to $\sigma_{s_{in},(\mathsf{l}'-1)}(p_1)$.

An example will illustrate this. Assume the pathcondition $p_1 = j \leq m$ from Figure 5.2. After $\mathsf{l} > 0$ iterations the third term in the above equation will read

$$\begin{aligned} & j(0) \leq m(0) \wedge j(1) \leq m(1) \wedge \cdots \wedge j(\mathsf{l}-1) \leq m(\mathsf{l}-1) \\ = & \bigwedge_{\mathsf{l}'=1}^{\mathsf{l}} \big(j(\mathsf{l}'-1) \leq m(\mathsf{l}'-1)\big). \end{aligned}$$

The reader is invited to assure him that the third term in Equation (5.21) really corresponds to the pathcondition after $\mathsf{l} > 0$ loop iterations. The case of zero iterations is covered by the fact that the minimum value of l' is 1, which cancels the third term in this case.

Recurrence System: For the induction variables of the set IV (cf. Equation 5.12) we set up a recurrence system over the loop index variable l, from which we construct a one-element recurrence system set rss as follows.

$$rss = \left\{ \begin{bmatrix} \forall v_i \in IV : \begin{cases} v_i(0) ::= s_{in}(v_i) & \\ v_i(\mathsf{l}+1) ::= \sigma_{s_{in},\mathsf{l}}\big(s_1(v_i)\big) & (1) \end{cases} \\ rc ::= \sigma_{s_{in},\mathsf{l}}(p_1) & (2) \\ (\mathsf{l}_{i,\omega}, e) & (3) \end{bmatrix} \right\} \tag{5.22}$$

Part (1) denotes the recurrence for induction variable v_i. The boundary value of a variable upon entry of the loop body is the variable's value from the "incoming" context ($\overline{c_{in}}$ in our case). We derive the recurrence relation for variable v_i as follows. State s_1 basically contains the variable bindings after the first iteration of the loop body. In replacing all occurrences of the initial value variables $\underline{v_i} \in \underline{\mathbb{V}}$

by their recursive counterpart $v_i(\mathsf{l})$, we obtain the bindings after iteration $\mathsf{l}+1$, denoted by $v_i(\mathsf{l}+1)$. If we can derive a closed form for the recurrence relation of variable v_i, part (1) consists only of a symbolic closed form expression over loop index variable l.

Part (2) holds the recurrence condition rc for this recurrence system. The condition is basically a symbolic predicate obtained by replacing the initial value variables in the pathcondition p_1 (cf. Equation 5.18) by their recursive counterparts.

If we can derive the upper bound for the number of iterations of the loop, we store a symbolic expression describing this bound (l_ω), as indicated in part (3).

Having set up the recurrence system set rss according to Equation (5.22), the recurrence system set rss_{out} of closure context $\overline{c_{out}}$ is derived from rss_{in} by appending rss to it.

$$rss_{out} ::= \text{Append}(rss, rss_{in}) \tag{5.23}$$

Nested Loops: Nested loops deserve treatment in their own respect. The requirement to nest recurrence system sets is in fact due to the possibility of nested loops. For a given path expression P the contained loops and their nesting relation are uniquely determined.

For the following discussion we assume a loop L_1 with a loop L_2 nested within L_1. Furthermore we assign the loop index variable l_1 to loop L_1, and l_2 to loop L_2. To distinguish between loops, we subscript terms applying to both loops with the proper loop index variable.

During the computation of the result of symbolic evaluation of one iteration of the loop body of L_1, the nested loop L_2 is analyzed as well. From the definition of the $\circledast$ operator it follows that the set of recurrence systems $rss_{1\mathsf{l}_1}$ computed in this step (cf. Equation (5.18)) will contain the one-element set of recurrence systems rss_{l_2} (in the sense of Equation (5.22)) for loop L_2. This recurrence system set is an integral part of the description of L_2 that has to be kept with L_1.

However, the closure context $\overline{c_{1\mathsf{l}_1}}$ has been computed from the clean slate closure context $\overline{c_{0\mathsf{l}_1}}$, which means that the boundary conditions of the recurrences in rss_{l_2} are in terms of the initial value variables $\underline{v_i} \in \underline{\mathbb{V}}$. In order to "install" L_2 in the context of L_1, these initial value variables have to be rewritten. To be specific, each occurrence of $\underline{v_i} \in \underline{\mathbb{V}}$ in the boundary conditions of the recurrences in rss_{l_2} has to be rewritten by $v_i(\mathsf{l}_1)$. We can safely pass this task to substitution $\sigma_{s,e}$ from Equation (5.19), since the boundary conditions are the only parts in rss_{l_2} that contain initial value variables (this is in fact a direct consequence of Equation (5.22)).

Equation (5.24) describes the rewriting step in detail.

$$rss_{tmp} ::= \text{Append}\big(\sigma_{s_{in},\mathsf{l}_1}\big(\text{First}(rss_{1\mathsf{l}_1})\big), \text{Rest}(rss_{1\mathsf{l}_1})\big) \tag{5.24}$$

We extract the first element from $rss_{1\mathsf{l}_1}$ and apply the substitution $\sigma_{s_{in},\mathsf{l}_1}$. The result of the substitution is then appended to the tail of $rss_{1\mathsf{l}_1}$. The resulting set of recurrence systems rss_{tmp} has to be appended to the set of recurrence systems for loop L_1, which causes us to extend Equation (5.23) to

$$rss_{out} ::= \text{Append}\big(\text{Append}\big(rss_{tmp}, rss\big), rss_{in}\big). \tag{5.25}$$

This concludes Definition 5.7. Before we proceed with the description of the $\odot$ operator, we intertwine a short example. Lacking the $\odot$ operator yet, we will resort to ordinary function composition ($\circ$) instead.

Example 5.1 We have familiarized with the example from Figure 5.1 in the course of Section 5.4. The time has finally come to introduce this example to the form level. Still we want to compute the MOP solution for node n_1 which is due to the path expression $e_1 \cdot (e_2 \cdot e_3)^*$ of type (n_e, n_1). Using the mapping θ on this path expression, we obtain

$$\theta\big(e_1 \cdot (e_2 \cdot e_3)^*\big) = \theta(e_1) \circ \theta(e_2 \cdot e_3)^* \tag{5.26}$$

$$= \theta(e_1) \circ \theta(e_2 \cdot e_3)^{\circledast} \tag{5.27}$$

$$= \theta(e_1) \circ \big(\theta(e_2) \circ \theta(e_3)\big)^{\circledast} \tag{5.28}$$

$$= f_{e_1} \circ (f_{e_2} \circ f_{e_3})^{\circledast} \tag{5.29}$$

$$= (f_{e_2} \circ f_{e_3})^{\circledast}(f_{e_1}) \tag{5.30}$$

Our initial clean slate closure context is

$$\overline{c} = [s, p, rss] = \big[\{(b, \underline{b}), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\},\ true, \emptyset\big],$$

and we are going to compute

$$(f_{e_2} \circ f_{e_3})^{\circledast}(f_{e_1})(\overline{c}).$$

From the first step we obtain the closure context $\overline{c_{in}}$.

$$\overline{c_{in}} = f_{e_1}(\overline{c}) = \big[\{(b, \underline{b}+1), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\},\ true, \emptyset\big]$$

Therefore our computation reduces to the calculation of

$$(f_{e_2} \circ f_{e_3})^{\circledast}(\overline{c_{in}}),$$

which is precisely the situation described in Equation (5.17). We will now proceed along the lines of Definition 5.7. Equation (5.18) tells us to use a clean slate closure context $\overline{c_0}$ and compute the result of symbolic evaluation of one iteration of the loop body. We can reuse the clean slate closure context $\overline{c}$ by defining $\overline{c_0} ::= \overline{c}$ and proceed with the calculation of $\overline{c_1}$ (cf. also Equation (5.7)).

$$\overline{c_1} = (f_{e_2} \circ f_{e_3})(\overline{c_0}) = \big[\{(b, \underline{b}), (d, 2 \cdot \underline{d}), (j, \underline{j} + \underline{b}), (m, \underline{m})\},\ \underline{j} \leq \underline{m}, \emptyset\big]$$

The closure context $\overline{c_{out}}$ resulting from the computation of

$$\overline{c_{out}} = (f_{e_2} \circ f_{e_3})^{\circledast}(\overline{c_{in}})$$

can then be described in terms of its state s_{out}, its pathcondition p_{out}, and its recurrence system set rss_{out}. The loop index variable for this loop is l.

State: The state of $\overline{c_{out}}$ is obtained from the state of $\overline{c_{in}}$ by replacing the symbolic expressions that describe the values of the variables v_i by the value of the recurrence relation for v_i over loop index variable l. Hence we get

$$\{(b, \underline{b}), (d, d(\mathsf{l})), (j, j(\mathsf{l})), (m, \underline{m})\} \tag{5.31}$$

for s_{out}.

Pathcondition: According to Equation (5.21) we get the following pathcondition for $\overline{c_{out}}$.

$$true \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge_{\mathsf{l}'=1}^{\mathsf{l}} \big(j(\mathsf{l}'-1) \leq \underline{m}\big) \tag{5.32}$$

Recurrence System: From closure context $\overline{c_1}$ we derive the set of induction variables, namely $IV = \{d, j\}$. According to Equation (5.22) we arrive at the one-element recurrence system set rss.

$$rss = \left\{ \left[\begin{array}{ll} \begin{cases} d(0) ::= s_{in}(d) \\ d(\mathsf{l}+1) ::= \sigma_{s_{in},\mathsf{l}}\big(s_1(d)\big) \end{cases} & (1a) \\ \begin{cases} j(0) ::= s_{in}(j) \\ j(\mathsf{l}+1) ::= \sigma_{s_{in},\mathsf{l}}\big(s_1(j)\big) \end{cases} & (1b) \\ rc ::= \sigma_{s_{in},\mathsf{l}}(p_1) & (2) \\ (\mathsf{l}_\omega, e) & (3) \end{array} \right] \right\}$$

Since s_{in} denotes the state of $\overline{c_{in}}$, and s_1 the state of $\overline{c_1}$, we can apply the insertions due to the σ-substitutions to get

$$rss = \left\{ \left[\begin{array}{ll} \begin{cases} d(0) ::= \underline{d} \\ d(\mathsf{l}+1) ::= 2 \cdot d(\mathsf{l}) \end{cases} & (1a) \\ \begin{cases} j(0) ::= \underline{j} \\ j(\mathsf{l}+1) ::= j(\mathsf{l}) + \underline{b} + 1 \end{cases} & (1b) \\ rc ::= j(\mathsf{l}) \leq \underline{m} & (2) \\ (\mathsf{l}_\omega, e) & (3) \end{array} \right] \right\}.$$

Inserting the closed forms for the induction variables from Equation (5.14), we finally arrive at the recurrence system set

$$rss = \left\{ \left[\begin{array}{ll} d(\mathsf{l}) ::= 2^{\mathsf{l}} \cdot \underline{d} & (1a) \\ j(\mathsf{l}) ::= \underline{j} + \mathsf{l} \cdot (\underline{b}+1) & (1b) \\ rc ::= j(\mathsf{l}) \leq \underline{m} & (2) \\ (\mathsf{l}_\omega, e) & (3) \end{array} \right] \right\}. \tag{5.33}$$

It is now time to determine the upper bound of loop index variable l. We are interested in an integer-valued expression e for l_ω, for which the following holds.

$$\begin{aligned} e &= \min\{\mathsf{l} \mid \neg\, rc\} \\ &= \min\{\mathsf{l} \mid j(\mathsf{l}) > \underline{m}\} \\ &= \min\{\mathsf{l} \mid \underline{j} + \mathsf{l} \cdot (\underline{b}+1) > \underline{m}\} \\ &= \min\{\mathsf{l} \mid \mathsf{l} > \frac{\underline{m} - \underline{j}}{\underline{b}+1}\} \\ &= \left\lceil \frac{\underline{m} - \underline{j} + 1}{\underline{b}+1} \right\rceil \end{aligned} \tag{5.34}$$

Combining the results from Equations (5.31)–(5.34) gives us

$$\big[\{(b,\underline{b}),(d,d(\mathsf{l})),(j,j(\mathsf{l})),(m,\underline{m})\},\\ (0\le \mathsf{l}\le \mathsf{l}_\omega)\wedge\bigwedge_{\mathsf{l}'=1}^{\mathsf{l}}\big(j(\mathsf{l}'-1)\le\underline{m}\big),\{rss\}\big] \tag{5.35}$$

as the solution for $\overline{c_{out}}$. Due to the absence of nested loops the recurrence system set $\{rss\}$ has been derived from rss using Equation (5.23).

The closure context depicted in Equation (5.35) represents the result of symbolic analysis for node n_1 (cf. Figure 5.1) as specified by the mapping θ from Definition 5.6. It is a finite representation of symbolic execution along the infinitely many program paths in $L(e_1\cdot(e_2\cdot e_3)^*)$. This claim is justified by the fact that the loop index variable l of $\overline{c_{out}}$ assumes all values in the interval $[0,\mathsf{l}_\omega]$.

There is a subtle yet important difference between the original solution from the object level (the object level solution has been restated in Equation (5.16)), and the form level solution developed in this section. In the presence of a cycle the first is an *infinite* set of program contexts, while the latter, due to the upper bound l_ω, describes a *finite* set of program contexts (except for endless loops**, where $\mathsf{l}_\omega=\infty$). This discrepancy is rooted in the fact that the MOP-solution on the object level corresponds to an extensive enumeration of the program paths in $L(P)$, while our solution based on a closure context exploits the boundedness of a loop to discard program paths that induce more than l_ω loop iterations.

Semantically the two solutions are of course equivalent; Discarding a program path that induces more than l_ω loop iterations means to discard a program path that generates a *false* pathcondition (it becomes *false* with iteration $\mathsf{l}_\omega+1$). Such a program path is artificial in the sense that it cannot result from a standard-semantic program execution, because such an execution never takes an edge with a branch predicate evaluating to *false* (this attribute of the Flow language has been stated in Equation (3.3)).

Finally, if we set $\mathsf{l}_\omega=\infty$ (as with endless loops), then the solution described by the finite structure $\overline{c_{out}}$ is identical to the MOP-solution on the object level.

A note is now due to the relation of program contexts, closure contexts, and supercontexts. As we have seen, a closure context is an extended program context that represents the solution of symbolic execution along the infinitely many program paths due to a loop cycle. However, a closure context, like his elderly brother (the program context), cannot describe the result of symbolic execution along several program paths that are not part of the same loop cycle (e.g., a closure context is sufficient for the path expression e^*, but not for (e_1+e_2)). To handle the latter, we still need supercontexts. However, due to the properties of closure contexts, supercontexts consist of a *finite* number of closure contexts.

In this way we have already reached the goal set for this section. We conclude this section with a few simplifications that can be applied to closure contexts.

Closure Context Simplifications

Assume that in the above example we want to compute the MOP-solution at node n_x. At first sight this seems to be almost a trivial step, once we have the

**Endless loops have been described towards the end of Section 3.5.3.

solution for node n_1 at hand, as it only adds the evaluated branch predicate $j > m$ to the pathcondition of $\overline{c_{out}}$. The resulting pathconditon is depicted in the following equation.

$$(0 \leq \mathsf{I} \leq \mathsf{I}_\omega) \wedge \bigwedge_{\mathsf{I}'=1}^{\mathsf{I}} \big(j(\mathsf{I}'-1) \leq \underline{m}\big) \wedge j(\mathsf{I}) > \underline{m}$$

The nontrivial fact about this pathcondition is that there is only one value for the loop index variable, namely $\mathsf{I} = \mathsf{I}_\omega$, that fulfills it! Conversely, once execution leaves the loop $(e_2 \cdot e_3)^*$, the MOP-solution of this example collapses to a single program context. Although there are infinitely many program paths from node n_e to node n_x in terms of the associated path expression, the program semantics require to drop out of the loop iff I_ω is reached, not before, and not later on.

Using the fact that $\mathsf{I} = \mathsf{I}_\omega$, the above pathcondition can be rewritten to

$$\bigwedge_{\mathsf{I}'=1}^{\mathsf{I}_\omega} \big(j(\mathsf{I}'-1) \leq \underline{m}\big) \wedge j(\mathsf{I}_\omega) > \underline{m},$$

and the closure context for node n_x, based on the recurrence system set rss from Equation (5.33), is

$$\Big[\{(b,\underline{b}),(d,d(\mathsf{I}_\omega)),(j,j(\mathsf{I}_\omega)),(m,\underline{m})\}, \bigwedge_{\mathsf{I}'=1}^{\mathsf{I}_\omega} \big(j(\mathsf{I}'-1) \leq \underline{m}\big) \wedge j(\mathsf{I}_\omega) > \underline{m}, \{rss\}\Big] \tag{5.36}$$

From Equation (5.34) we know the symbolic expression for I_ω, which we can use to calculate $d(\mathsf{I}_\omega)$ and $j(\mathsf{I}_\omega)$. Inserting the results in the above closure context finally gives us

$$\Big[\{(b,\underline{b}),\Big(d,\underline{d}\cdot 2^{\left\lceil \frac{\underline{m}-\underline{j}+1}{\underline{b}+1}\right\rceil}\Big),\Big(j,\underline{j}+(\underline{b}+1)\cdot\left\lceil\frac{\underline{m}-\underline{j}+1}{\underline{b}+1}\right\rceil\Big),(m,\underline{m})\}, \bigwedge_{\mathsf{I}'=1}^{\mathsf{I}_\omega} \big(j(\mathsf{I}'-1) \leq \underline{m}\big) \wedge \Big(\underline{j}+(\underline{b}+1)\cdot\left\lceil\frac{\underline{m}-\underline{j}+1}{\underline{b}+1}\right\rceil\Big) > \underline{m}, \{rss\}\Big]. \tag{5.37}$$

If we are only interested in the accumulated side-effect arising from symbolic execution of the loop, and not in the loop internals, we might even discard the recurrence system set rss and the associated part of the pathcondition.

5.5.2 Edge-Splitting

Introduction of the symbolic rounding operation Rnd in Section 4.1.1, Definition 4.2 caused a problem with the round towards zero rounding mode. As an example, assume the Flow assignment statement

$$e : \text{<cond>} \Rightarrow \text{x} := \text{x div y}. \tag{5.38}$$

According to Equation (4.9), x div y becomes $\mathrm{Rnd}(x/y)$. Let us assume that we cannot establish the integer-valuedness of x/y, which means that further simplifications of this expression depend on whether $x/y < 0$ (cf. Equation (3.10)).

Unless we have further knowledge on the values of x and y, e.g., due to information from the pathcondition, we have to introduce a by-case distinction of the division operation. Figure 5.3 illustrates the original control flow graph portion

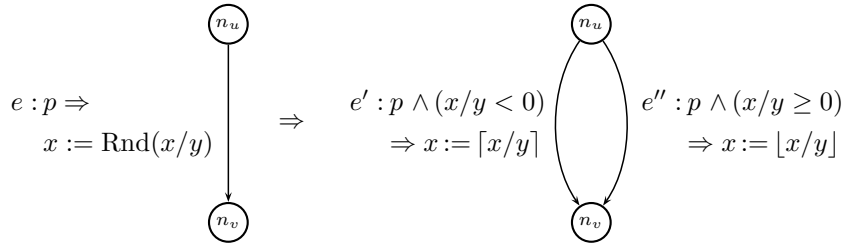


Figure 5.3: Edge-Splitting in Case of Integer Divison

with edge e and the transformation into a graph with two edges e' and e'' corresponding to the cases $x/y < 0$ and $x/y \geq 0$. Note that in this figure we have replaced the abstract syntax notation of Equation (5.38) by the corresponding symbolic expressions, e.g., <cond> has become p.

The correctness of the above transformation follows from the definition of the round towards zero rounding mode (stated in Equation (3.10)) and the fact that the pathconditions of edge e' and e'' are disjunct (a requirement of the Flow standard semantics stated in Equation (3.3)). Given a path expression P containing edge e, the transformation depicted in Figure 5.3 corresponds to the replacement of each occurrence of e in P by $(e' + e'')$.

We can apply the same transformation principle to cover the cases resulting from the integer division and remainder operations of univariate integer-valued polynomials in the same indeterminate (cf. Section 4.1.1, Table 4.1). A total of four edges is needed per division and per remainder operation. The conditions for these edges follow from the conditions of the respective cases from Table 4.1, and the fact whether the side-condition is valid or not. Again, these conditions are disjunct, thus satisfying Equation (3.3). The correctness of this transformation is then a consequence of the correctness of the division mechanism for univariate integer-valued polynomials.

Edge-splitting is also applicable to expressions with several rounding or division/remainder operations. However, split edges induce additional program paths, and excessive splitting can under certain conditions lead to a combinatorial explosion in the number of program paths. The investigation of an on-demand splitting mechanism can be regarded as a topic of further research.

Edge-splitting also occurs with expressions from the quotient field $Q(\mathbb{Z}[\mathbf{x}])$. As an example, consider the expression x/y from Equation (5.38). The function represented by this expression is undefined at $y = 0$. What we have not depicted in Figure 5.3 is an edge e_x originating at node n_u representing the asynchronous transfer of control due to a division by zero (we leave the target of this edge open, usually it will be an exception handler or the exit node n_x of the program). The branch predicate of edge e_x is the condition that the denominator equals zero ($y = 0$ in our case). The existence of this edge, and the (presumed) fact that the pathcondition p of edge e in Figure 5.3 contains the condition that the denominator is unequal to zero, makes simplifications of expressions from the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ as presented in Section 4.1.1 safe.

5.5.3 Term Representations and Normal Forms

In Section 4.1 we have introduced a hierarchy of abstractions for symbolic expressions. Until now we have treated symbolic expressions only on the topmost level, where they exist as pure mathematical objects. For an implementation of symbolic evaluation it is however necessary to discuss issues arising from the "projection" of those mathematical objects onto a real computer. This section is therefore devoted to the representation of the domain of symbolic expressions on the form level.

An important problem that has to be solved on the form level is the necessity for unique normal forms of symbolic expressions. Early results of research in this area can be found in [Mos71]. Based on [LN73], Buchberger and Loos present in [BL82] the construction of unique normal forms for multivariate polynomials, rational functions, and radical expressions.

Hence we can represent integers, multivariate polynomials, and rational functions as terms over a suitable signature Σ and specify the operational semantics of a convergent term rewrite system (cf. [BN98]) that transforms them into a unique normal form.

From [Gog80, Theorem 8] respectively [MG86, Theorem 11(2)] it follows that these normal forms constitute an initial algebra (cf. [Wec92, Grä68]). This implies that there exists a unique homomorphism into a Σ-algebra (such as $\mathbb{Z}$, $\mathbb{Z}[\mathbf{x}]$, and $Q(\mathbb{Z}[\mathbf{x}]$ from Section 4.1.1).

Our case constitutes an *order-sorted* framework, which is due to the fact that

$$\mathbb{Z} \subseteq \mathbb{Z}[\mathbf{x}] \subseteq Q(\mathbb{Z}[\mathbf{x}]). \tag{5.39}$$

The generalization of many-sorted algebra to order-sorted algebra is shown in [GM02], other sources of information include [GM96] and [GD94].

Due to Birkhoff's Theorem (cf. [BN98]) semantic equivalence coincides with syntactic equality and we can decide the first based on the latter.

Finally this notion of a convergent term rewrite system for symbolic expressions is extensible to states, program contexts, closure contexts, finite supercontexts, and path expressions. However, the next two sections show simplifications of branch-predicates with inequalities, where we also rely on semantic information.

5.5.4 Validity

$$(\forall n > 2)(\forall x)(\forall y)(\forall z)[x^n + y^n \neq z^n] \tag{5.40}$$

$$a > a + 1 = \mathit{false}. \tag{5.41}$$

Every symbolic predicate for which we can establish validity can be replaced by *true*, which is a valuable simplification for the representation of pathconditions. Unfortunately there exists no algorithm capable of determining the validity of a formula such as Equation (5.40) stated in elementary arithmetic built up from $+$, $*$, $=$, constants, variables for nonnegative integers, quantifiers over nonnegative integers, and the sentential connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ subject to the requirement that every variable in such a formula be acted on by some quantifier. This follows from a conclusion from Gödel's incompleteness theorem, [Rog87, p. 38] contains the corresponding proof.

In the following section we will deal with a subclass of elementary arithmetic for which satisfiability and validity are decidable.

A Decidable Subclass of Flow Predicates

I never can decide anything very important in any length of time, at all.
— Richard P. Feynman, theoretical physicist, in "The Pleasure of Finding Things Out".

For a subclass of elementary arithmetic called *Presburger arithmetic*, validity is decidable [Sho79]. Presburger formulas are those formulas that can be constructed by combining first degree polynomial (*affine*) constraints on integer variables with the connectives $\neg$, $\wedge$, $\vee$, and the quantifiers $\forall$ and $\exists$. Constraints are affine due to the fact that Presburger arithmetic permits addition and the usual arithmetical relations ($<, \leq, >, \geq, =$), but no arbitrary multiplication of variables[††].

The Omega test [Pug92] is a widely used algorithm for testing the satisfiability of arbitrary Presburger formulas. We can use it as a decision procedure for Flow predicates that are within this subclass of elementary arithmetic.

A method capable of transforming certain classes of general polynomial constraints into a conjunction of affine constraints has been presented by Maslov and Pugh [MP94].

5.5.5 Satisfiability

There exist predicates from the domain SymPred of symbolic predicates which are *unsatisfiable*, e.g.,

$$a < b \ \wedge \ a \geq b \tag{5.42}$$

Predicates that are unsatisfiable can be replaced by *false*. A program context with a *false* pathcondition can be discarded, which makes the recognition of unsatisfiable predicates an important issue for the representation of supercontexts.

It is a well-known result that the *satisfiability* problem for logical formulae over boolean variables is NP-complete (cf. e.g., [Sed88]). As a matter of fact, the Flow language contains no boolean variables but symbolic predicates as specified in Table 4.4 on page 52. Symbolic predicates are more general than the formulae of the before-mentioned satisfiability problem over boolean variables, which already suggests that satisfiability of symbolic predicates could be even "worse" than NP-complete.

Indeed we can establish the undecidability of the satisfiability problem for Flow predicates with an argument from [HR04]: a given predicate p is unsatisfiable if, and only if, $\neg p$ is valid. From Section 5.5.4 it follows that we cannot compute validity, hence we cannot compute satisfiability either.

Again this accounts for the general case. As already pointed out in Section 5.5.4, there exist subclasses for which we can decide satisfiability.

[††]Although it is convenient to use multiplication by *constants* as an abbreviation for repeated addition.

Chapter 6

Experimental Results

It's a contest in purposeless suffering.
— Lance Armstrong, six-times* Tour de France winner,
in "It's Not About the Bike — My Journey Back to Life".

In this chapter we dissect the entire SPEC95 benchmark suite [CPU95] in order to derive the corresponding resource requirements for symbolic evaluation. Our symbolic evaluation approach is based on path expressions. Correspondingly, our measurements are based on a series of metrics on path expressions that capture the control flow information contained in the underlying control flow graphs. Path expressions as well as the metrics data itself are computed by a data-flow framework. Our measurements show that symbolic evaluation is a methodology capable of coping with the considerable problem sizes that arise from contemporary real-world applications such as those from the SPEC95 benchmark suite.

6.1 Preliminaries

Definition 6.1 A *regular expression* R is *unambiguous* if each string in $L(R)$ is represented uniquely in R. A precise definition of the term "unique representation" follows.

Let $a_1, a_2, \ldots, a_m$ be the symbols appearing in the regular expression in their natural order, and let f be the function relating the a_i to the symbols in R. The expression R' then denotes the corresponding regular expression over the a_i. For example, if $R = (0+11)^*0$, then $m = 4$, $R' = (a_1 + a_2 \cdot a_3)^* a_4$, and $\mathrm{f}(a_1) = \mathrm{f}(a_4) = 0$, and $\mathrm{f}(a_2) = \mathrm{f}(a_3) = 1$. If $t \in L(R)$, $t = s_1 s_2 \ldots s_l$, then there are legitimate ways of assigning to each s_i an a_j. In our example, if $t = 01100$, then a legitimate assignment is $t' = a_1 a_2 a_3 a_1 a_4$ where $\mathrm{f}(t') = t$. We say that t is denoted in exactly one way if there exists a unique legitimate assignment. Formally, $t \in L(R)$ is denoted by R in exactly k ways if there are exactly k distinct strings $t_1, \ldots, t_k$ in $L(R')$ with $\mathrm{f}(t_i) = t$. It is clear that this discussion applies only to nonempty strings.

*At the time of writing.

Definition 6.2 An *automaton* $A = (Q, \Sigma, \delta, q_0, F)$ is *unambiguous* if for each $t \in L(A)$, $t \neq \Lambda$, there exists a unique path in the state transition diagram[†] of A from q_0 to a state in F.

Definition 6.3 An *automaton* $A = (Q, \Sigma, \delta, q_0, F)$ is *primitive* if no input symbol appears more than once in the state transition diagram. That is

$$(\forall s \in \Sigma)(\forall q \in Q)\big[\,|\delta(q,s)| \leq 1\big], \text{ and} \tag{6.1}$$

$$(\forall s \in \Sigma)(\forall q, q' \in Q)\big[\,(q = q') \vee (\delta(q,s) = \emptyset) \vee (\delta(q',s) = \emptyset)\big]. \tag{6.2}$$

Therefore, if A is primitive, $|\Sigma|$ is equal to the number of edges in the state transition diagram. Clearly, a primitive automaton is unambiguous.

Definition 6.4 Given an automaton $A = (Q, \Sigma, \delta, q_0, F)$, we can rename the states of A in such a way that the resulting automaton A' has the set of states $Q' = \{1, \ldots, n\}$ for some integer n. Let us use $R^k_{i,j}$ as the name of a regular expression whose language is the set of strings such that each string r denotes a path from state i to state j in A', with the additional constraint that r contains no *intermediate* state whose number is greater than k. Note that the endpoints i and j are not considered "intermediate" since we require that an intermediate state must be entered and *then* left. The following inductive definition constructs the expressions $R^k_{i,j}$ for A'.

Basis: The basis is $k = 0$. Since states are numbered such that $Q' = \{1, \ldots, n\}$, the restriction on paths is that they must not contain intermediate states at all. We distinguish two cases:

$i \neq j$**:**

$$R^0_{i,j} = \begin{cases} s_1 + \ldots + s_k & \text{if } s_1, \ldots, s_k \text{ are labels from state } i \text{ to state } j, \\ \emptyset & \text{if there are no labels from } i \text{ to } j \text{ in } A'. \end{cases}$$

$i = j$**:**

$$R^0_{i,j} = \begin{cases} \Lambda + s_1 + \ldots + s_k & \text{if } s_1, \ldots, s_k \text{ are self-labels of state } i, \\ \Lambda & \text{if there are no self-labels of state } i \text{ in } A'. \end{cases}$$

The empty string Λ of the latter case accounts for the path of length zero from state i to i itself.

Induction: For $k > 0$ we have

$$R^k_{i,j} = R^{k-1}_{i,j} \; + \; R^{k-1}_{i,k} \, (R^{k-1}_{k,k})^* \, R^{k-1}_{k,j}.$$

Informally, the definition of $R^k_{i,j}$ above means that a path from state i to state j in A' that does not pass through an intermediate state higher than k is either

1) in $R^{k-1}_{i,j}$ (in which case it never passes through an intermediate state as high as k); or

[†]Confer [HU79, p.16].

2) composed of a path in $R_{i,k}^{k-1}$ (which takes A' from state i to state k for the first time), followed by a path in $(R_{k,k}^{k-1})^*$ (which takes A' zero or more times from state k to k itself without passing through a state higher than k), followed by a path in $R_{k,j}^{k-1}$ (which takes A' from state k to state j).

Definition 6.5 Given an automaton $A = (Q, \Sigma, \delta, q_0, F)$ and a set $M \subseteq Q$, we let $R_{i,j}^M$ denote the regular expression whose language is the set of strings r such that every path from state i to state j in A is represented by a string r (and vice versa), with the additional constraint that r must not contain *intermediate* states $q \notin M$. In particular, the regular expression with the empty set, $R_{i,j}^{\emptyset}$, denotes the language of strings without intermediate states and hence amounts to the set of direct edges from state i to state j (the empty set is a subset of every set, that is, $\emptyset \subseteq S$, whenever S is a set (cf. [Ros95, pp.40])). In this way we have $L(R_{i,j}^{\emptyset}) \subseteq L(R_{i,j}^M)$, since every direct edge is considered a path from from state i to state j with zero intermediate states $q \in M$. Note that the endpoints i and j are not considered "intermediate" since we require that an intermediate state must be entered and *then* left.

Lemma 5. We can view a control flow graph $G = \langle N, E, n_e, n_x \rangle$ as the state transition diagram of an automaton $A_G = (N, E, \delta, n_e, \{n_x\})$, where the nodes of G correspond to the states of the automaton. Start node n_e and terminal node n_x denote the start- and accepting state of A_G. The edge set E corresponds to the alphabet of A_G, with the transition function

$$\delta(n, e) = m \Leftrightarrow (\mathrm{h}(e) = n \wedge \mathrm{t}(e) = m).$$

There are two direct consequences of the fact stated in Definition (2.3), that the set E of edges of a CFG consists of ordered pairs of elements of N.

(1) The transition function δ is a partial function.

(2) The automaton A_G is a primitive (and hence deterministic) automaton.

Thus for every CFG in the sense of Definition (2.3) there exists a corresponding primitive automation. The converse is not true since we require that a CFG has a single terminal node.

Definition 6.6 In a CFG, a node x *dominates* another node y iff all paths from the entry node n_e to y always pass through x. We write $x \,\mathrm{dom}\, y$ to indicate that x dominates y. If $x \,\mathrm{dom}\, y$ and $x \neq y$, then x strictly dominates y, denoted by $x \,\mathrm{stdom}\, y$. A node x is said to *immediately dominate* another node y, denoted as $x = \mathrm{idom}(y)$, if $x \,\mathrm{stdom}\, y$ and there is no other node $z \neq x$ and $z \neq y$ such that $x \,\mathrm{stdom}\, z \,\mathrm{stdom}\, y$. The dominance relation is reflexive and transitive, and can be represented by the so-called *dominator tree*. An edge $e = (x, y)$ is an edge in the dominator tree of a CFG iff $x = \mathrm{idom}(y)$. Given a node x in the dominator tree, we define $\mathrm{sub}(x)$ to be the set of nodes of the dominator subtree rooted at x.

6.2 Path Expression Generation

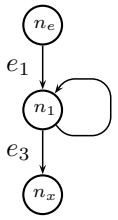
In this section we define a forward data-flow problem along with a solution procedure that allows us to compute path expressions for a given CFG. The

data-flow problem is based on the following equations.

$$R(n_e) = \Lambda \tag{6.3}$$

$$R(n) = \underset{n' \in \mathrm{Preds}(n)}{+} \left[R(n') \cdot R^{\emptyset}_{n',n} \right] \tag{6.4}$$

Recall that in Definition (2.3) on page 17 we required that $\mathrm{in}(n_e) = \emptyset$, so the empty string Λ is indeed the only regular expression that takes us from node n_e to n_e itself.



$$R(n_e) = \Lambda$$

$$R(n_1) = R(n_e) \cdot e_1 + R(n_1) \cdot e_2$$

$$R(n_e) = R(n_1) \cdot e_3$$

Figure 6.1: CFG with Infinite Number of Program Paths

Due to Equation(6.4) this data-flow problem cannot be solved by an iterative algorithm in the presence of loops. As an example consider the CFG and its associated data-flow equations depicted in Figure 6.1. The regular expression $e_1 \cdot e_2^* \cdot e_3$ is a valid path expression of type (n_e, n_x). Iterative algorithms fail to compute this expression once they attempt to determine the solution for the data-flow equation at node n_1. This is due the fact that the underlying lattice for this data-flow problem is infinite.

Application of an elimination-based algorithm can however solve this data-flow problem. Blieberger [Bli02, Section 3] and Ryder et al. [RP86] mention several elimination algorithms that exploit the structure of flowgraphs for improved time complexity. For our purposes we use Sreedhar's algorithm presented in [Sre95]. As pointed out in [Pau88], we define a normal form for our equations and set up a loopbreaking rule that allows us to handle circularities of the before-mentioned kind.

A data-flow equation E for our path expression data-flow problem is in *normal form*, if it has the form

$$E : R(n) = \begin{cases} \underset{m \in M \subseteq \mathrm{sub}(\mathrm{idom}(n))}{+} \left[R(m) \cdot R^{S_m \subseteq \mathrm{sub}(m) \setminus \{m\}}_{m,n} \right] & \text{if } n \neq n_e, \\ \Lambda & \text{else.} \end{cases} \tag{6.5}$$

Equation (6.5) subsumes data-flow equations (6.3) and (6.4), which is not obvious for the latter equation. To explore this normal form in full detail we need to take into account the properties of our elimination algorithm (in fact the normal form of Equation (6.5) is tailored to the needs of this algorithm). Since the elimination algorithm is subject of Section 6.4.1 (pp. 109), full treatment of Equation (6.5) is postponed until then. In the meantime we only note that, contrary to Equation (6.4), it is possible for a data-flow equation in normal form, to depend on other nodes than its control flow predecessors. This is due to substitutions that occur during elimination (cf. [Pau88]). It can furthermore happen that due to substitutions the left-hand side of an equation E_i occurs

several times on the right-hand side of an equation E_j, e.g.,

$$E_i : R(n_i) = R(n_k) \cdot e_1$$
$$E_j : R(n_j) = R(n_i)R^{S_1}_{n_i,n_j} + \cdots + R(n_i)R^{S_l}_{n_i,n_j}.$$

In such a case we employ the following rewrite-rule based on the distributive law of regular expression algebra (cf. e.g. [Sal66]) in order to transform E_j to normal form.

$$R_1 \cdot R_2 + R_1 \cdot R_3 \longrightarrow R_1 \cdot (R_2 + R_3) \tag{6.6}$$

Resuming the above example, we get $R(n_j) = R(n_i) \cdot (R^{S_1}_{n_i,n_j} + \cdots + R^{S_l}_{n_i,n_j})$ for the data-flow equation E_j.

It is the purpose of the *loopbreaking rule* to replace a data-flow equation E which depends on itself by an equivalent equation e for which the left-hand side of E does not appear on the right-hand side. Assume a data-flow equation for a given node n, for which the set S_n is defined as

$$S_n \subseteq \mathrm{sub}\big(\mathrm{idom}(n)\big) \setminus \big(\{\mathrm{idom}(n)\} \cup \mathrm{sub}(n)\big).$$

Then the data-flow equation under consideration is

$$E : R(n) = \underbrace{R(\mathrm{idom}(n)) \cdot R^{S_n}_{\mathrm{idom}(n),n}}_{t_1} + \underbrace{R(n) \cdot R^{S'_n \subseteq \mathrm{sub}(n)\setminus\{n\}}_{n,n}}_{t_2}. \tag{6.7}$$

Clearly this data-flow equation for node n contains a dependency on the immediate dominator of n, constituted by the term t_1. Moreover, it contains a dependency on n itself, constituted by the term t_2. By further dissecting terms t_1 and t_2, we arrive at

$$t_1 = R(\mathrm{idom}(n)) \cdot \underbrace{R^{S_n}_{\mathrm{idom}(n),n}}_{t_{1_1}}, \text{ and } t_2 = R(n) \cdot \underbrace{R^{S'_n \subseteq \mathrm{sub}(n)\setminus\{n\}}_{n,n}}_{t_{2_1}}.$$

Therein term t_{1_1} constitutes all program paths from $\mathrm{idom}(n)$ to n itself, possibly via intermediate nodes $n' \in S_n$. According to the definition of set S_n, no program path of term t_{1_1} is allowed to contain $\mathrm{idom}(n)$, n, or a node dominated by n as an intermediate node. Term t_{2_1} constitutes all program paths from node n back to n itself, possibly via nodes strictly dominated by n.

We define our loopbreaking rule to replace Equation (6.7) by the equation

$$e : R(n) = \underbrace{R(\mathrm{idom}(n)) \cdot R^{S_n}_{\mathrm{idom}(n),n}}_{t_1} \cdot \underbrace{\big(R^{S'_n \subseteq \mathrm{sub}(n)\setminus\{n\}}_{n,n}\big)^*}_{t_3}. \tag{6.8}$$

6.3 Program Path Metrics

In this section we define a series of metrics aiming at the determination of the space-requirements that can be expected for symbolic evaluation of programs. Programs are given as control flow graphs from which path-expressions are constructed. A path-expression of type (n_i, n_j) represents all program paths from

node n_i to node n_j in the underlying control flow graph (CFG). Metrics are calculated by reinterpreting the operations $+$, $\cdot$, and * that are used to construct path expressions (cf. [Tar81]).

The number of program paths through a CFG advances from finite to infinite with the introduction of cycles. Our symbolic evaluation method provides an abstraction mechanism that keeps the space requirements that are due to cycles finite. This is due to the fact that symbolic evaluation is based on *acyclic* program paths.

The term "acyclic program path" has already been coined in the literature, e.g., for path profiling ([BL96]), but we use a different approach of partitioning a given flowgraph into acyclic subgraphs. Moreover, our approach is based on path-expressions rather than on the flowgraphs themselves.

The metrics that we develop in the course of this section calculate the number of acyclic program paths through a given flow graph $G = \langle N, E, n_e, n_x \rangle$ from a path expression $R^N_{n_e,n_x}$ representing all program paths in G from the entry node n_e to its exit node n_x. Strictly speaking, these metrics determine the natural loops (cf. [ASU86, Section 10.4]) in $R^N_{n_e,n_x}$. The acyclic program paths then follow from the acyclic condensation (cf. [ZC91]) of these loops across all nesting levels.

Irreducible loops do not have the property of natural loops that they possess a unique header. In this way irreducibility introduces a kind of indeterminism in the path-expression generation data-flow problem of Section 6.2, because one of the possible loop entry nodes has then to be promoted as loop header. For the algorithms that calculate our metrics this indeterminism is of no concern since an operand of the * operator has a unique header per definition. As we will see in Section 6.4, this indeterminism will however be of concern with respect to the minimality of these metrics.

6.3.1 Loops as Black-Boxes (npp)

For the number of acyclic program paths that have to be considered by symbolic evaluation, the following metric calculates a lower bound. It treats subexpressions that correspond to cycles (loops) as black boxes with a resource-requirement of 1. Apart from that, cycles (loops) are not considered further by this metric.

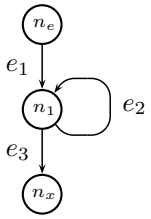


Figure 6.2: CFG with Infinite Number of Program Paths

For an example of a CFG with a cycle and hence an infinite number of program paths consider the graph depicted in Figure 6.2. The program paths from node n_e to node n_x are defined by the path expression $e_1 \cdot e_2^* \cdot e_3$ of type (n_e, n_x). The subexpression e_2^* corresponding to the cycle in Figure 6.2 accounts for the infinite number of program paths.

We can now define a mapping that allows us to compute the number of acyclic program paths (npp) from regular expressions representing sets of program paths. With the inductive definition below, P denotes a path (sub)expression, and e denotes an arbitrary CFG-edge.

$$\mathrm{npp}(\Lambda) = 0, \tag{6.9}$$

$$\mathrm{npp}(\emptyset) = 0, \tag{6.10}$$

$$\mathrm{npp}(e) = 1, \tag{6.11}$$

$$\mathrm{npp}(P_1 + P_2) = \mathrm{npp}(P_1) + \mathrm{npp}(P_2), \tag{6.12}$$

$$\mathrm{npp}(P_1 \cdot P_2) = \mathrm{npp}(P_1) * \mathrm{npp}(P_2), \tag{6.13}$$

$$\mathrm{npp}(P_1^*) = 1. \tag{6.14}$$

Equation (6.14) of this mapping assigns cycles a resource-requirement of 1 and is for this reason responsible for the black-box view of this metric with respect to loops.

Applying this metric to the example of Figure 6.2, we get

$$\begin{aligned} \mathrm{npp}(e_1 \cdot e_2^* \cdot e_3) &= \mathrm{npp}(e_1) * \mathrm{npp}(e_2^*) * \mathrm{npp}(e_3) \\ &= 1 * 1 * 1 \\ &= 1. \end{aligned} \tag{6.15}$$

A more elaborate example of a kite-shaped CFG is depicted in Figure 6.3. Equation (6.16) presents the path expression P of type (n_e, n_x). To facilitate reading we have used curly and square brackets in addition to the round brackets normally used for grouping regular expressions and path expressions.

$$\begin{aligned} P = \overbrace{\big((e_1 \cdot (e_2 \cdot e_3 + e_4)^* \cdot e_6) + (e_5 \cdot e_7)\big) \cdot e_{16}^*}^{R_1} \cdot \\ \Big\{ [\overbrace{e_9 \cdot (((e_{10} \cdot e_{12}) + (e_{11} \cdot e_{17}^* \cdot e_{13})) \cdot e_{14})^*}^{R_2} \cdot \overbrace{(((e_{10} \cdot e_{12}) + (e_{11} \cdot e_{17}^* \cdot e_{13})) \cdot e_{15})}^{R_3}] \\ + e_8 \Big\} \end{aligned} \tag{6.16}$$

Applying the npp metric to the path expression of Equation (6.16), we get

$$\begin{aligned} \mathrm{npp}(P) &= \mathrm{npp}(R_1) * \big(\mathrm{npp}(R_2) * \mathrm{npp}(R_3) + \mathrm{npp}(e_8)\big) \\ &= 2 * (1 * 2 + 1) \\ &= 6. \end{aligned} \tag{6.17}$$

In relating this calculation to the CFG of Figure (6.3), it becomes clear that the npp-metric is only a lower bound for the resource-requirements needed for symbolic evaluation: the loops corresponding to path expressions e_{16}^*, e_{17}^*, $(e_2 \cdot e_3 + e_4)^*$ and $((e_{10} \cdot e_{12}) + (e_{11} \cdot e_{17}^* \cdot e_{13}) \cdot e_{14})^*$ are all estimated to have a resource requirement of 1. While this is true for loops e_{16}^* and e_{17}^* which both have only one path through its loop body, it underestimates the number of acyclic program paths for the latter two loop bodies. We will define metrics that compensate the under-estimate of the npp-metric in the next section.

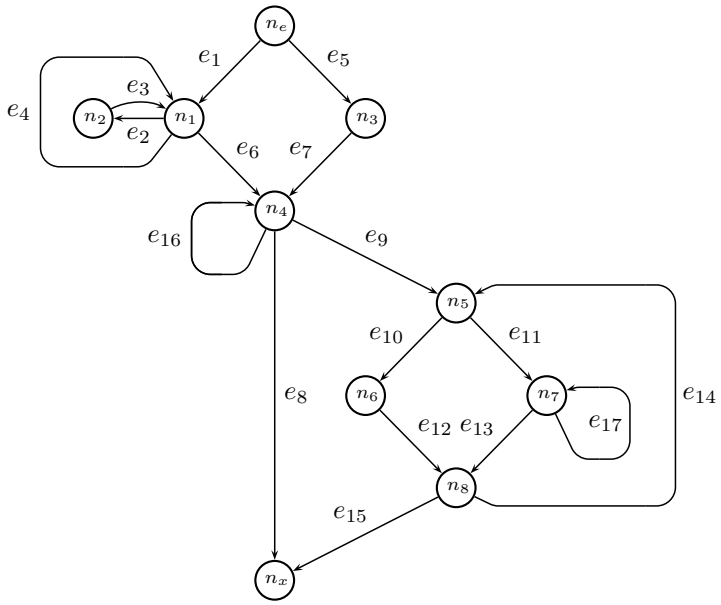


Figure 6.3: Elaborate Example: Kite-Shaped CFG

6.3.2 Loop-Aware Path Metrics

$$R_1 = e_1 \cdot (e_2^* \cdot e_3 \cdot e_4)^* e_2^* \cdot e_3 \cdot e_5$$

$$R_2 = e_1 \cdot (\underbrace{e_2}_{R_3})^* \cdot e_3 \cdot (\underbrace{e_4 \cdot e_2^* \cdot e_3}_{R_4})^* \cdot e_5$$

$$R_0 = e_1 \cdot (e_2 + e_3 \cdot e_4)^* \cdot e_3 \cdot e_5$$

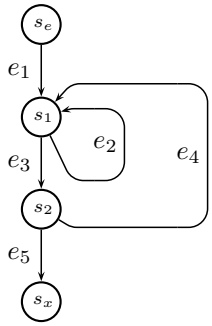


Figure 6.4: Example: Common Subexpression e_2^* in Expressions R_1 and R_2

Figure 6.4 contains a flowgraph with *nested* loops. The outer loop l_1 is along the program path $\pi_1 = \langle e_3, e_4 \rangle$, whereas the inner loop l_2 is along the path $\pi_2 = \langle e_2 \rangle$. Furthermore Figure 6.4 depicts two equivalent regular expressions R_1 and R_2 of type (s_e, s_x) for the respective flowgraph. From the npp-metric introduced in Section 6.3.1 we get

$$\text{npp}(R_1) = \text{npp}(R_2) = 1.$$

This result is an under-estimate in the sense that it does not take into account the subexpressions R_3 and R_4 (cf. Figure 6.4). This omission is due to the npp-rule

$$\text{npp}(P_1^*) = 1$$

of Equation (6.14). Informally, we can compensate this under-estimate if we add the cost for expressions R_3 and R_4 in terms of the npp-metric. Using path

expression R_2 of Figure 6.4 we can calculate the precise result r as shown in the following equation.

$$\begin{aligned} r &= \mathrm{npp}(R_2) + \mathrm{npp}(e_2^*) + \mathrm{npp}(e_4 \cdot e_2^* \cdot e_3) \\ &= 1 + 1 + 2 \\ &= 4 \end{aligned} \tag{6.18}$$

Therein the first term corresponds to the original npp-metric, whereas the second and third term account for the calculation of the npp-values of subexpressions R_3 and R_4. On the other hand, if we take path expression R_0 of Figure 6.4, we get the following result r' in terms of our metric (observe that R_0 and R_2 are equivalent regular expressions).

$$\begin{aligned} r' &= \mathrm{npp}(R_0) + \mathrm{npp}(e_2 + e_3 \cdot e_4) \\ &= 1 + 2 \\ &= 3 \end{aligned} \tag{6.19}$$

From this example it becomes immediately clear that equivalent regular expressions need not yield the same result with respect to our metric. In this particular case it is the double occurrence of subexpression e_2^* in expression R_2 that imposes the extra cost of 1 compared to expression R_0. Regarding symbolic evaluation of a program corresponding to the CFG of Figure 6.4, the double occurrence of subexpression e_2^* in R_2 would mean that the loop represented by this subexpression is actually evaluated twice: once due to regular expression R_3, and once in the course of the evaluation of expression R_4.

We cannot expect that for each double occurrence there exists an equivalent regular expression without it. Moreover, the computational complexity of the required rewrite operations might be prohibitive. Double occurrences are not even restricted to loops — the edge e_3 occurs twice in expression R_2, and e_3 could in fact be replaced by an arbitrarily complex CFG.

In this way it is desirable to detect *common subexpressions* within path expressions in order to spend the effort for symbolic evaluation only once. Conceptually this corresponds to the introduction of so-called *meta-variables* to the algebra of path expressions. For example, if we define that meta-variable $M_1 = e_2^*$, then we get

$$R_2 = e_1 \cdot M_{1_a} \cdot e_3 \cdot (e_4 \cdot M_{1_b} \cdot e_3)^* \cdot e_5$$

for expression R_2 (note that we have used an additional level of subscripts to distinguish between the two occurrences of M_1 in the subsequent discussion). If, during symbolic evaluation of expression R_2, the meta-variable M_1 is encountered for the first time at M_{1_a}, the symbolic evaluation of the expression associated with M_1 will be carried out. For the second occurrence of meta-variable M_1 at M_{1_b} we can already use the result derived at M_{1_a}.

As already pointed out, the utilization of common regular subexpressions to reduce the effort associated with symbolic evaluation is not restricted to loops. In this way we might as well introduce an additional meta-variable for the edge e_3 in R_2. In the following discussion we will however focus on the use of common subexpressions representing loops.

Given the fact that regular expressions are not unique in the sense that *equivalent* regular expressions may describe the same set of program paths

(cf. expressions R_1 and R_2 of Figure 6.4), we are faced with the problem of determining equivalence of regular expressions in order to spot common subexpressions. There exist algorithms that decide on the equivalence of regular sets (e.g., [Gin67, HU79, HMU01]), but the following two properties of the elimination algorithm of [Sre95] (cf. also Section 6.4.1) ensure that in order to show the equivalence of regular expressions representing loops it is sufficient to show their *identity*.

(1) Data-flow equations containing loops are inserted into other equations only after the path expression corresponding to the loop has been determined (this follows from the fact that the only possible remaining dependency for an equation at the time of insertion into other equations is one on its immediate dominator which clearly excludes dependencies on the loop body).

(2) A path expression corresponding to a loop body is not altered once it has been determined.

In other words, we can be sure that we will not encounter common subexpressions representing a loop body that are equivalent but not identical.

```
1   procedure Slice (e : PathExpr; s : in out Set) is
2   begin
3    if Match (e, RDot (a, b)) or Match (e, ROr (a, b))
4    then
5        Slice (a, s);
6        Slice (b, s);
7    elsif Match (e, RStar (a)) then
8        s := s ∪ a;
9        Slice (a, s);
10   end if;
11  end Slice;
```

Figure 6.5: Slicing Procedure

The algorithm that calculates our metric for a given path-expression P consists of two steps. In the first step we extract natural loops P_1^* from P (P_1 denotes an arbitrary regular expression), and collect them in the set S. In the second step we calculate the npp-metric for the path-expression P and for all elements in the set S and sum up the results. This two-step algorithm is depicted in pseudo-code in Figure 6.5 (Step 1) and Figure 6.6 (Step 2).

The slicing procedure of Figure 6.5 takes as input a path-expression P and a set S and recursively descends the syntax tree of P. It uses pattern matching (expressed through the call to procedure "Match" in lines 3 and 7) in order to determine the structure of P. In this particular case pattern matching is very simple since the matching decision is solely based on the operation symbol of the root node of the syntax (sub)tree of P. As a notational convention we append an *underscore* to a variable to denote the fact that this variable can match arbitrary regular expressions. On the contrary, the variable name without underscore denotes the corresponding matched regular expression. If during the recursive descent of expression P a natural loop is encountered (line 7), then the corresponding loop body is added to the set S (line 8).

Input:

a path-expression P

Output:

Metric (P)

(the loop-aware metric of acyclic program paths of P)

Algorithm:

$S := \{P\}$;

Slice (P, S);

Metric $(P) = \sum_{p \in S} \text{npp}(p)$;

Figure 6.6: Metric Algorithm

Note that the fact that S is a *set* ensures that common subexpressions denoting loop bodies are counted only once. If we declare S to be a multiset, with operator $\cup$ in line 8 of Figure 6.5 denoting the multiset union operation, than our metric counts each occurrence of a loop body. This leads us to the definitions of the following two loop-aware path metrics.

Definition 6.7 Given a path-expression P. With S being a multiset and $\cup$ the multiset union operation, the metric algorithm given in Figures 6.5 and 6.6 calculates a loop-aware npp-metric (cf. Section 6.3.1) that compensates the effort associated with a subexpression b that denotes a loop body for each occurrence of b in expression P. We call this metric **ncp**, the *number of considered acyclic program paths*.

Definition 6.8 With S being a set and $\cup$ the set union operation, the metric algorithm given in Figures 6.5 and 6.6 calculates a loop-aware npp-metric (cf. Section 6.3.1) that compensates the effort associated with a loop body only once. We call this metric **loncp** which stands for *loop-optimized* **ncp**.

Coming back to the kite-shaped CFG of Figure 6.3 on page 102, we can now calculate the ncp-metric for the corresponding path-expression P of type (s_e, s_x) stated in Equation (6.16). Applying the algorithm of Figure 6.6, we get

$$\begin{aligned} \text{ncp}(P) = {} & \text{npp}(P) \\ & + \text{npp}(e_2 \cdot e_3 + e_4) \\ & + \text{npp}(e_{16}) \\ & + \text{npp}((e_{10} \cdot e_{12} + e_{11} \cdot e_{17}^* \cdot e_{13}) \cdot e_{14}) \\ & + \text{npp}(e_{17}) \\ = {} & 13. \end{aligned} \tag{6.20}$$

Due to the double occurrence of the subexpression denoting the loop body e_{17}^*, the value that we get for the ncp-metric of expression P is one higher than the value calculated by the loncp-metric.

As already pointed out for the npp metric calculations of equations (6.18) and (6.19), equivalent regular expressions need not yield the same result with respect to the npp metric. Since the metrics ncp and loncp are based on the npp

metric, they are also affected by this fact. Given a regular expression R, we often face the question whether there exists an equivalent regular expression R' that yields a lower value than R with respect to one of our metrics. Naturally this leads to the question whether a given regular expression R is *minimal* with respect to one of our metrics. In the following definition we specify the requirements for a regular expression to be npp-minimal. The definitions for the other two metrics are similar.

Definition 6.9 Let $\mathbb{A}$ denote the set of regular expressions. The equivalence relation $\sim \subseteq \mathbb{A} \times \mathbb{A}$ over regular expressions induces an equivalence class $[a]_\sim := \{a' \in \mathbb{A} \mid a \sim a'\}$ for each $a \in \mathbb{A}$. A given element a of an equivalence class $[a]_\sim$ is **npp-minimal**, if there exists no other element $b \in [a]_\sim$, $b \neq a$, such that $\mathrm{npp}(b) < \mathrm{npp}(a)$. Note that an npp-minimal element need not be *unique* in its equivalence class.

6.4 Loncp-Minimal Path-Expressions

In this section we show that the path-expressions generated by the data-flow framework defined in Section 6.2 are loncp-minimal for *reducible* flowgraphs. Finally we present an adversary argument showing that for *irreducible* flowgraphs these path expressions are not loncp-minimal in the general case.

We start out with a few lemmas that are needed to make our point. In Subsection 6.4.1 we show that the generated path expressions are *unambiguous*. This result is utilized in Subsection 6.4.2 to show that unambiguous path expressions are loncp-minimal for *reducible* flowgraphs. Subsection 6.4.3 contains the corresponding adversary argument for irreducible flowgraphs.

Lemma 6. Given two regular expressions $R_a = R_{x,y}^{M \cup \{y\}}$ and $R_b = R_{y,z}^{N \setminus \{y\}}$ over a primitive, Λ-free automaton $A = (Q, \Sigma, \delta, q_0, F)$. If the regular expressions R_a and R_b are unambiguous, then their concatenation, denoted by $R_a \cdot R_b$, is also unambiguous.

Proof. We utilize the concept of the proof of [BEGO71, Theorem 1]. According to Definition (6.3), a primitive automaton is unambiguous. Hence we know from Definition (6.2) that no two different paths through automation A correspond to the same string $t \in L(A)$, given that A is Λ-free.

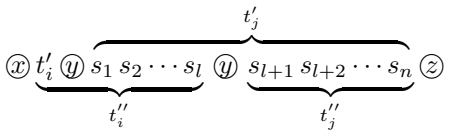


Figure 6.7: Decomposition of string $t = t'_i t'_j = t''_i t''_j$.

Assume, now, that string $t \in L(R_a \cdot R_b)$ and that t can be decomposed in two ways: $t = t'_i t'_j = t''_i t''_j$, where $t'_i, t''_i \in L(R_a)$, and $t'_j, t''_j \in L(R_b)$. If $t'_i < t''_i$, then $t''_i = t'_i s_1 s_2 \cdots s_l$, where $l \geq 1$ and $s_k \in \Sigma$, $1 \leq k \leq l < n$. Figure 6.7 depicts string t and its corresponding decompositions. Note that the positions where t passes states x, y, and z have been marked with ⓧ,ⓨ, and ⓩ.

It follows from $t_i^{''} \in L(R_a)$, that in the state transition diagram of A, s_l is an edge entering state y. However, s_l is also a symbol in t'_j, and $t'_j \in L(R_b)$ does not contain paths with intermediate state y. Thus, $t'_i = t_i^{''}$, and consequently, $t'_j = t_j^{''}$. According to the initial assumption that R_a and R_b are unambiguous, t'_i is denoted in exactly one way in R_a, and t'_j is denoted in exactly one way in R_b. As a consequence, $t = t'_i t'_j$ is denoted in exactly one way in $R_a \cdot R_b$. □

Corollary 1. Given two regular expressions $R_a = R_{x,y}^{M\setminus\{y\}}$ and $R_b = R_{y,z}^{N\setminus\{y\}}$ over a primitive, Λ-free automaton $A = (Q, \Sigma, \delta, q_0, F)$. If the regular expressions R_a and R_b are unambiguous, then their concatenation, denoted by $R_a \cdot R_b$, is also unambiguous.

The Corollary is a special case of Lemma (6) where the regular expression R_a must not contain an intermediate state y. Hence the decomposition $t = t_i^{''} t_j^{''}$ depicted in Figure 6.7 is illegal due to $t_i^{''}$ containing the intermediate state y. In fact there exists only one possible decomposition for $t \in L(R_a \cdot R_b)$, namely $t = t_i^{'''} t_j^{'''}$, where $t_i^{'''} \in L(R_a)$, and $t_j^{'''} \in L(R_b)$. □

Lemma 7. Given a regular expression $R = R_{y,y}^{M\setminus\{y\}}$ over a primitive, Λ-free automaton $A = (Q, \Sigma, \delta, q_0, F)$. If the regular expression R is unambiguous, then the regular expression R^*, denoting the closure of $L(R)$, is also unambiguous.

Proof. From the inductive definition of languages described by regular expressions (cf. Case (4), on p. 74), we have

$$L(R^*) = (L(R))^* = \bigcup_{k=0}^{\infty} L(R)^k.$$

Therein $L(R)^0 = \{\Lambda\}$, and $L(R)^k = L(R)^{k-1} \cdot L(R)$. But it also holds that

$$\bigcup_{k=0}^{\infty} L(R)^k = L\Big(\mathop{+}_{k=0}^{\infty} R^k\Big),$$

where $R^0 = \Lambda$, and $R^k = R^{k-1} \cdot R$ (proof by induction on the number of operation symbols omitted). We can then prove the lemma in two steps.

(1) First we show that the regular expressions $R^k = \big(R_{y,y}^{M\setminus\{y\}}\big)^k$, for $k \geq 0$, are unambiguous.

(2) Based on (1) we show that the regular expression $R^* = \mathop{+}_{k=0}^{\infty} R^k$ is unambiguous.

AD (1): By structural induction on the unambiguity of R^k.

Basis: We use $k = 0$, $k = 1$, and $k = 2$ as base cases. $R^0 = \Lambda$ is unambiguous since it represents the empty string in exactly one way. Furthermore, $R^1 = R^0 \cdot R = \Lambda \cdot R = R$ is unambiguous due to the initial assumption of this lemma. $R^2 = R^1 \cdot R = R \cdot R$ is unambiguous due to Corollary (1) if we substitute state y for state x and state z.

Induction: Let $R^{k+1} = R^k \cdot R$, for $k \geq 2$, be a regular expression built by the inductive step of the definition. The regular expression R is unambiguous due to the initial assumption of this lemma. For the

inductive step of the proof we may assume that R^k is unambiguous, too. It remains to show that $R^k \cdot R$ is an unambiguous regular expression.

The essential observation therein is that, although $R = R_{y,y}^{M\setminus\{y\}}$ is a regular expression without intermediate state y, the concatenation of $R_{y,y}^{M\setminus\{y\}}$ with itself yields a regular expression that contains intermediate state y. In this way R^k, for $k \geq 2$, qualifies as expression R_a in Lemma (6). Note that, strictly speaking, Definition (6.5) requires $R_a = R_{x,y}^{M\cup\{y\}}$ to contain *all* paths from state x to state y. This is not the case for R^k, since

$$L(R^k) = L(\big(R_{y,y}^{M\setminus\{y\}}\big)^k) \subseteq L(\big(R_{y,y}^{M\setminus\{y\}}\big)^*) = L(R_{y,y}^{M\cup\{y\}}).$$

The proof of Lemma (6) however includes also this sub-case. Substituting R for R_b in Lemma (6) finally shows that $R^k \cdot R$ is indeed unambiguous.

AD (2): From step (1) we already know that every term R^k in

$$R^* = \overset{\infty}{\underset{k=0}{+}} R^k$$

is unambiguous. In order to establish that R^* is unambiguous it remains to show that $L(R^i) \cap L(R^j) = \emptyset$, for $0 \leq i, j \leq \infty$ and $i \neq j$. Observe that $L(R^0) \cap L(R^j) = \emptyset$, for $0 \leq j \leq \infty$, since $L(R^0) = \{\Lambda\}$, and R (and hence R^j) are Λ-free due to A being Λ-free. Moreover, $L(R^k) \cap L(R^{k+j}) = \emptyset$, for $k \geq 1$ and $j > 1$, since $L(R^k)$ contains only paths that pass intermediate state y exactly $k-1$ times, whereas $L(R^{k+j})$ contains only paths that pass intermediate state y exactly $k+j-1$ times.

□

Lemma 8. Given two regular expressions $R_a = R_{x,y}^{M\setminus\{y\}}$ and $R_b = R_{y,y}^{N\setminus\{y\}}$ over a primitive, Λ-free automaton $A = (Q, \Sigma, \delta, q_0, F)$. If the regular expressions R_a and R_b are unambiguous, then the regular expression denoted by $R_a \cdot R_b^*$ is also unambiguous.

Proof. According to Definition (6.3), a primitive automaton is unambiguous. Hence we know from Definition (6.2) that no two different paths through automation A correspond to the same string $t \in L(A)$, given that A is Λ-free.

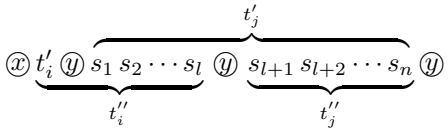


Figure 6.8: Decomposition of string $t = t'_i t'_j = t''_i t''_j$.

Assume, now, that string $t \in L(R_a \cdot R_b)$ and that t can be decomposed in two ways: $t = t'_i t'_j = t''_i t''_j$, where $t'_i, t''_i \in L(R_a)$, and $t'_j, t''_j \in L(R_b^*)$. If $t'_i < t''_i$,

then $t_i^{''} = t_i' s_1 s_2 \cdots s_l$, where $l \geq 1$ and $s_k \in \Sigma$, $1 \leq k \leq l < n$. Figure 6.8 depicts string t and its corresponding decompositions. Note that the positions where t passes states x and y have been marked with ⓧ and ⓨ.

It follows from $t_i^{'} \in L(R_a)$, that in the state transition diagram of A, s_1 is an edge emanating from state y. However, s_1 is also a symbol in $t_i^{''}$, and $t_i^{''} \in L(R_a)$ does not contain paths with intermediate state y. Thus, $t_i' = t_i^{''}$, and consequently, $t_j' = t_j^{''}$. According to the initial assumption that R_a is unambiguous, t_i' is denoted in exactly one way in R_a. From the initial assumption that R_b is unambiguous and from Lemma (7) we know that t_j' is denoted in exactly one way in R_b^*. As a consequence, $t = t_i' t_j'$ is denoted in exactly one way in $R_a \cdot R_b$. □

6.4.1 Unambiguity of Path Expressions

Lemma 9. The path-expressions for a given reducible CFG as generated by the data-flow framework described in Section 6.2 are unambiguous in general.

Proof. The elimination algorithm described in [Sre95] operates on the DJ graph of a flowgraph rather than the flowgraph itself. The first step of this proof is therefore to show how the data-flow equations (6.3) and (6.4) can be applied to a DJ graph without affecting the underlying CFG-based data-flow problem. The elimination algorithm itself comprises two phases. The *elimination phase* performs DJ graph reduction and variable substitution of the equation system until the DJ graph is reduced to its dominator tree. After the first phase the equation at every node is expressed only in terms of its immediate dominator, with the root node n_e being the only exception (cf. Equation (6.3)). The second phase of the algorithm is concerned with the *propagation* of the solution at the root-node in a top-down fashion along the dominator tree to compute the solutions at the other nodes. In this way the latter two parts of the proof establish that both phases, *elimination* and *propagation*, leave us with unambiguous path expressions.

DJ Graph-Based Path Expression Generation. As pointed out in [Sre95, Corollary 3.1], we can construct a DJ graph from a flowgraph G by adding every missing *immediate dominance edge* (x, y) if this edge is not already present in G. As a consequence a DJ graph can contain more edges than the corresponding CFG.

Data-flow equations (6.3) and (6.4) are applicable to a DJ graph if we require that for an edge $e = (x, y)$ that is part of the CFG but not of the DJ graph,

$$R_{x,y}^{\emptyset} = \emptyset. \tag{6.21}$$

This ensures that edge e will not affect our CFG-based data-flow problem since the set of paths from node x to node y considered for path expression generation is now the empty set. Note that this is in line with Definition (6.5).

It is also worth mentioning that a DJ graph can be viewed as a primitive automation, which is a result of the following two facts.

1) According to Lemma (5), the underlying CFG can be viewed as a primitive automaton.

2) The immediate dominance edges that are added during transformation of the CFG to the DJ graph are unique, hence they occur only once as an input symbol in the state transition diagram derived from the DJ graph, as required by Definition (6.3).

Elimination. In this part of the proof we establish that the elimination phase of the elimination algorithm described in [Sre95] results in data-flow equations

$$E : R(n) = \begin{cases} R(\text{idom}(n)) \cdot R^{S_n \subseteq \text{sub}(\text{idom}(n)) \setminus \{\text{idom}(n)\}}_{\text{idom}(n),n} & \text{if } n \neq n_e, \\ \Lambda & \text{else,} \end{cases} \tag{6.22}$$

where

$$R^{S_n \subseteq \text{sub}(\text{idom}(n)) \setminus \{\text{idom}(n)\}}_{\text{idom}(n),n} \tag{6.23}$$

is an *unambiguous* path expression describing all program paths from node idom(n) to node n that do not contain *strict* dominators of node n as intermediate nodes. We will achieve this result by structural induction on Property (6.4.1) below.

It is worth mentioning that Equation (6.22) is of the normal form defined in Equation (6.5). It is however more specific in the sense that it depends only on its immediate dominator (elimination has *reduced* it to a dependency on the immediate dominator). Taking this reduction into account, we will refer to the resulting normal form as the *reduced* normal form. Consequently, the form of Equation (6.5) is then referred to as the *general* normal form. Reduced normal form implies general normal form.

Property 6.4.1
Data-flow equations generated during the elimination phase are in general normal form, with the additional constraint, that the regular expressions

$$R^{S_m \subseteq \text{sub}(m) \setminus \{m\}}_{m,n}$$

contained in those normal-formed equations are unambiguous.

Note that, at first sight, Property (6.4.1) seems too weak since it requires only general normal form, whereas Equation (6.22) is in reduced normal form. It is however a property of the elimination algorithm that after elimination each equation is reduced to a dependency on its immediate dominator. In this way reduced normal form is already a consequence of elimination and need not be taken care of by our proof!

Basis: Our base case is is the system of data-flow equations we derive once we transform the CFG to the DJ graph and apply Equations (6.3) and (6.4) under the constraint specified in Equation (6.21). The resulting data-flow equations clearly fulfill Property (6.4.1), since Equations (6.3) and (6.4) are in normal form, and the regular expressions

$$R^{\emptyset}_{n',n}, \text{ with } n' \in \text{Preds}(n)$$

contained in those normal-formed equations are unambiguous due to the automaton corresponding to a DJ graph being a primitive automaton.

Induction: For the inductive step, we may assume that our system of data-flow equations fulfills Property (6.4.1). We have now to consider the effects of the *ϵ-rules* E1 and E2 (cf. [Sre95, Section 4.1]) that carry out DJ graph reduction and variable substitution in order to show that the resulting equations again fulfill Property (6.4.1). This lengthy part of the proof is contained in Sections 6.4.1 (Lemma (11)) and Section 6.4.1 (Lemma (12)).

Propagation. Elimination leaves us with data-flow equations in reduced normal form as stated in Equation (6.22). Propagation is based on the principle that the concatenation of two path expressions R_1 and R_2, where R_1 denotes all program paths from node n_e to a given node $\text{idom}(n)$, and R_2 denotes all program paths from node $\text{idom}(n)$ to node n, results in a path expression $R_1 \cdot R_2$ that denotes all program paths from node n_e to n. Equation (6.23) states the side-condition that R_2 must not contain *strict* dominators of node n as intermediate nodes.

Lemma 10. Propagation of the solution $R(n_e)$ along the dominator tree results in unambiguous path expressions.

Proof. By structural induction on the unambiguity of the involved regular expressions.

Basis: For the base case we note that $R(n_e) = \Lambda$ denotes the empty string that corresponds to all program paths from node n_e to node n_e itself unambiguously. We can then propagate the solution at the root node to nodes $n_i \in \{n \mid \text{idom}(n) = n_e\}$ that are immediately dominated by it. Thus we get the following result

$$\begin{aligned} R(n_i) &= R(n_e) \cdot R_{n_e,n_i}^{S_{n_i} \subseteq \text{sub}(n_e) \setminus \{n_e\}} \\ &= \Lambda \cdot R_{n_e,n_i}^{S_{n_i} \subseteq \text{sub}(n_e) \setminus \{n_e\}} \\ &= R_{n_e,n_i}^{S_{n_i} \subseteq \text{sub}(n_e) \setminus \{n_e\}}, \end{aligned}$$

which is unambiguous due to Property (6.4.1).

Induction: For the inductive step, we may assume that in the equation

$$R(n) = \underbrace{R_{n_e,\text{idom}(n)}^{S'_n \subseteq \text{sub}(n_e) \setminus \{n_e\}}}_{t_1} \cdot \underbrace{R_{\text{idom}(n),n}^{S_n \subseteq \text{sub}(\text{idom}(n)) \setminus \{\text{idom}(n)\}}}_{t_2}$$

term t_1 is unambiguous as it is the result of earlier propagations. Term t_2 is unambiguous due to Property (6.4.1). The unambiguity of $t_1 \cdot t_2$ follows then directly from Lemma (6) on page 106. □

The result that the propagation phase of our elimination algorithm leaves us with unambiguous path-expressions concludes also the proof of Lemma (9). □

Elimination Rule E1

Figure 6.9 illustrates the elimination of edge (y, y) due to an application of ϵ-rule E1. Dashed lines represent dominator edges, and solid lines represent join edges.

Edge (y, y) corresponds to a circular dependency and is therefore exactly the case for which the loopbreaking rule of our data-flow problem (cf. Section 6.2) has been designed. Equation (6.7) on page 99 holds for node y before the loopbreaking rule is applied, and Equation (6.8) holds afterwards.

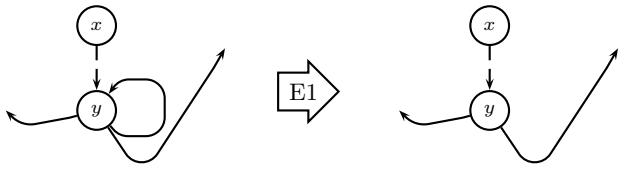


Figure 6.9: Graphical Illustration of Rule E1 for the Substitution "$y \to y$".

When setting up Property (6.4.1) on page 110, we concluded that it was sufficient to require general normal form, since the utilized elimination algorithm would, during its elimination phase, eventually reduce equations to reduced normal form. Figure 6.9 bears witness to this fact: before the application of the loopbreaking rule, the equation at note y is in general normal form, since it depends on its immediate dominator (node x), and on y itself. The loopbreaking rule removes the dependency on node y which reduces the equation to reduced normal form. Without the circular dependency at node y this would already have happened during an application of rule E2 that we will discuss in the next section. We conclude this section with the proof of the following lemma.

Lemma 11. The data-flow equation from Equation (6.8) fulfills Property (6.4.1).

Proof. As already observed, Equation (6.8) is in reduced normal form which implies general normal form. What remains to be checked for Property (6.4.1) is that the contained regular expressions have to be unambiguous. Unambiguity follows directly from Lemma (8) on page 108. □

Elimination Rule E2

The elimination algorithm described in [Sre95] actually contains two rules named E2a and E2b subsumed to rule E2. As pointed out in [Sre95, Section 10.4.1], the distinction between these two E2 rules is only minor. Our proof therefore considers only rule E2b, with the provision that the proof for rule E2a follows along the same line.

Figure 6.10 illustrates the elimination of edge (y, z) due to an application of ϵ-rule E2b. This corresponds to a substitution of the data-flow equation at node y into the equation at node z. The dotted line in Figure 6.10 from w to x represents the dominator tree path from w to x, dashed lines represent dominator edges, and solid lines represent join edges.

Before application of ϵ-rule E2b we have the following equation at node z,

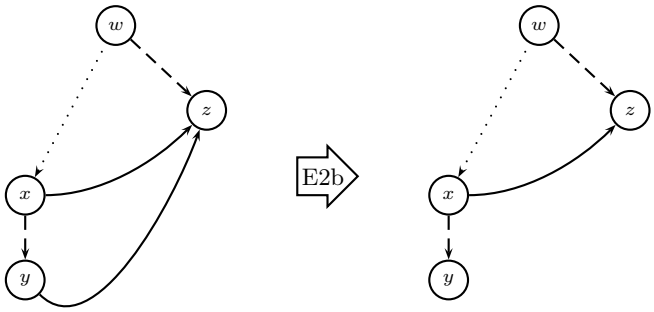


Figure 6.10: Graphical Illustration Rule E2b for the Substitution "$y \to z$".

which is also in *general* normal form.

$$\begin{aligned} R(z) = & \mathop{+}_{\substack{n \in S \subseteq \mathrm{sub}(\mathrm{idom}(z)) \\ \wedge n \notin \{x\} \cup \mathrm{sub}(y)}} R(n) \cdot R_{n,z}^{S_n \subseteq \mathrm{sub}(n) \setminus \{n\}} \\ & + R(x) \cdot R_{x,z}^{S_x \subseteq \mathrm{sub}(x) \setminus \left(\{x\} \cup \mathrm{sub}(y)\right)} \\ & + R(y) \cdot R_{y,z}^{S_y \subseteq \mathrm{sub}(y) \setminus \{y\}} \end{aligned} \tag{6.24}$$

Therein the third term (last line) represents a dependency on the equation $R(y)$ at node y. This dependency is represented by the edge (y, z) in Figure 6.10. To be specific, the regular expression

$$R_{y,z}^{S_y \subseteq \mathrm{sub}(y) \setminus \{y\}}$$

denotes paths from node y to node z that may contain only intermediate nodes that are *strictly* dominated by y. Moreover, the second term in Equation (6.24) represents a dependency on the equation $R(x)$ at node x. This dependency is represented by the edge (x, z) in Figure 6.10. The regular expression

$$R_{x,z}^{S_x \subseteq \mathrm{sub}(x) \setminus \left(\{x\} \cup \mathrm{sub}(y)\right)}$$

denotes paths from node x to node z that may contain only intermediate nodes that are *strictly* dominated by x and *not* dominated by y. Hence we have

$$S_x \cap \left\{ S_y \cup \{y\} \right\} = \emptyset. \tag{6.25}$$

The property stipulated in Equation (6.25) holds due to the facts that the equation at node y has not yet been substituted into node z, and that no equation into which node y has been substituted before this substitution has already been substituted into node z.

The data-flow equation at node z may furthermore contain dependencies on nodes as specified through the first term of Equation (6.24), but these dependencies have been omitted from Figure 6.10.

Given the data-flow equation $R(y)$ at node y,

$$R(y) = R(x) \cdot \underbrace{R_{x,y}^{S'_x \subseteq \mathrm{sub}(x) \setminus \{x\}}}_{t_1},$$

we may now perform the substitution $R(y) \to R(z)$, which yields

$$\begin{aligned} R(z) = & \underset{\substack{n \in S \subseteq \mathrm{sub}(\mathrm{idom}(z)) \\ \wedge n \notin \{x\} \cup \mathrm{sub}(y)}}{+} R(n) \cdot \underbrace{R_{n,z}^{S_n \subseteq \mathrm{sub}(n) \setminus \{n\}}}_{t_n} \\ & + R(x) \cdot \underbrace{R_{x,z}^{S_x \subseteq \mathrm{sub}(x) \setminus \left(\{x\} \cup \mathrm{sub}(y)\right)}}_{t_2} \\ & + R(x) \cdot \underbrace{R_{x,y}^{S'_x \subseteq \mathrm{sub}(x) \setminus \{x\}}}_{t_1} \cdot \underbrace{R_{y,z}^{S_y \subseteq \mathrm{sub}(y) \setminus \{y\}}}_{t_3}. \end{aligned} \tag{6.26}$$

Applying the rewrite-rule stated in Equation (6.6) on page 99, this can be brought to the general normal form

$$\begin{aligned} R(z) = & \underset{\substack{n \in S \subseteq \mathrm{sub}(\mathrm{idom}(z)) \\ \wedge n \notin \{x\} \cup \mathrm{sub}(y)}}{+} R(n) \cdot \underbrace{R_{n,z}^{S_n \subseteq \mathrm{sub}(n) \setminus \{n\}}}_{t_n} \\ & + R(x) \cdot \Big(\underbrace{R_{x,z}^{S_x \subseteq \mathrm{sub}(x) \setminus \left(\{x\} \cup \mathrm{sub}(y)\right)}}_{t_2} + \underbrace{R_{x,y}^{S'_x \subseteq \mathrm{sub}(x) \setminus \{x\}}}_{t_1} \cdot \underbrace{R_{y,z}^{S_y \subseteq \mathrm{sub}(y) \setminus \{y\}}}_{t_3} \Big). \end{aligned} \tag{6.27}$$

Lemma 12. The data-flow equation stated in Equation (6.27) fulfills Property (6.4.1).

Proof. As already observed, Equation (6.27) is in general normal form. What remains to be checked for Property (6.4.1) is that the contained regular expressions have to be unambiguous. It follows from the induction hypothesis that the regular expressions denoted by t_n and by $t_1 \cdots t_3$ are unambiguous. Moreover, the regular expression $t_1 \cdot t_3$ is unambiguous due to Lemma (6) on page 106. Finally, $t_2 + (t_1 \cdot t_3)$ is unambiguous since t_2 contains no path via y, and $t_1 \cdot t_3$ denotes only paths via y. Hence $L(t_2) \cap L(t_1 \cdot t_3) = \emptyset$. □

6.4.2 Reducible CFGs and Loncp-Minimality

Theorem 4. An unambiguous path-expression P of type (n_e, n_x) from a reducible CFG $G = \langle N, E, n_e, n_x \rangle$ is loncp-minimal in general.

Proof. If path-expression P is unambiguous, then each string in $L(P)$ is represented uniquely in P (according to Definition 6.1 on page 95). Since each such string corresponds to a program path through the flow graph G, each program path through G is in this case uniquely represented in P. We can now distinguish two cases.

(1) CFG G is acyclic. If G is acyclic, then the number of acyclic program paths as computed by the npp-metric corresponds to the actual number of program paths through G (proof by structural induction on the number of operation-symbols in P omitted). As the number of acyclic program paths cannot be lower than the number of program paths, we have established npp-minimality. Then loncp-minimality follows from the fact that for acyclic flowgraphs the npp- and the loncp-metric compute the same result.

(2) CFG G contains cycles. Due to G being irreducible, every cycle corresponds to a natural loop in G. It is the purpose of the slicing procedure depicted in Figure 6.5 on p. 104 to extract every natural loop, across all nesting levels. It is worth mentioning that a natural loop may contain other natural loops nested inside, and that these nested natural loops are not only extracted as part of the enclosing loop, but also as natural loops in their own respect (see also [Ram99]). When applying the npp-metric to a path-expression P corresponding to a natural loop nest, Equation (6.14) (p. 101) has the affect of condensing a nested loop to a single node. In this way the condensed version of P is again an acyclic graph to which Case (1) above applies. □

6.4.3 Irreducible CFGs and Loncp-Minimality

Theorem 5. Unambiguous path-expressions from irreducible CFGs are not loncp-minimal in general.

Proof. Indirect. Assume that unambiguous path-expressions from irreducible CFGs are loncp-minimal. Figure 6.11 depicts an irreducible CFG. For the unambiguous path-expression

$$R_1 = e_1 + (e_2 + e_1 \cdot e_4) \cdot (e_3 \cdot e_4)^* \cdot e_3$$

of type (n_e, n_x) we have $\text{loncp}(R_1) = 4$. On the other hand, for the *equivalent* unambiguous path-expression

$$R_2 = (e_1 + e_2 \cdot e_3) \cdot (e_4 \cdot e_3)^*$$

we get $\text{loncp}(R_2) = 3$, which concludes the case. □

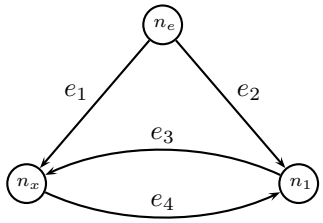


Figure 6.11: Potentially Loncp-Minimal Irreducible CFG.

As already pointed out, irreducible CFG-portions constitute loops that do not have a unique header. This not only separates them from reducible CFGs, but it also holds responsible for the above observation — depending on the node that we regard as the loop header (n_1 for R_1 vs. n_x for R_2), we get a different result from our loncp-metric.

6.5 Experiment and Evaluation

Finally I put to you for consideration by your discrete judgment that you should not wonder (but rather excuse me) if what I propound shall be not exactly comparable with the experiences and observations to be made by you and others, because there are may ways to err. One, almost inescapable, is some mistake in calculation; besides that, the smallness of these planets and their being observed with the telescope, which so greatly enlarges every object seen, means that an error even one second in the meetings and separations of these stars is made more apparent and noticeable than one thousand times as great an error in the aspects of other [naked eye] stars.
— Galileo Galilei, *Sunspot Letters*, Rome, 1613.

We shall now come to the description and evaluation of the experiments conducted on the SPEC95 benchmark suite. The following section describes the extraction of input-data and the basic setup of the experiment. Section 6.5.2 describes the validation of the data computed by our data-flow framework. In Section 6.5.3 we apply several data analysis methods from [Cle93] to evaluate the data collected in our experiment. The reader interested in the raw data collected in the experiment is referred to [BSB04, pp. 35–284].

6.5.1 Basic Setup

Diagram (6.28) depicts the basic setup of our experiment. C- and Fortran source files (SRC) are input to the GCC compiler. The command line option "-dw" causes GCC to generate debug output (DBG) that contains, among other information, the control flow information of the input program. This option is only valid with optimization enabled (at least "-dw"), which in turn also affects the structure of the control flow information. We have used GCC 3.2 for the IA-32 architecture under RedHat 9.0 for this step.

$$\text{SRC} \xrightarrow{\text{GCC}} \text{DBG} \xrightarrow{\text{SCR}} \text{CFG} \xrightarrow{\text{FW}} R^N_{n_e,n_x} \xrightarrow{\text{M}} \text{Data} \quad (6.28)$$

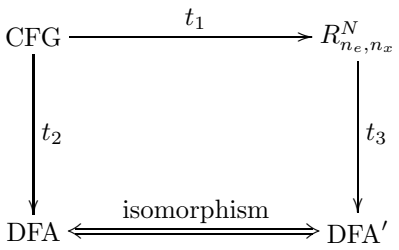
A combination of sed-, awk-, and perl scripts (SCR) is used to extract the control flow information from GCC's debug output in order to generate control flow graphs (CFG) that serve as input to our data-flow framework (FW).

It is the purpose of the data-flow framework to compute path expressions from control flow graphs according to the data-flow problem defined in Section 6.2. The metrics (M) defined in Section 6.3 have then been applied to these path expressions.

6.5.2 Validation

We have validated the implementation of our data-flow framework described in Section 6.2 in comparing the state transition diagrams corresponding to control flow graphs with the state transition diagrams corresponding to the generated path expressions. This approach is depicted in the following diagram which is

an extension of part of Diagram (6.28).



(6.29)

We take a given control flow graph CFG $= \langle N, E, n_e, n_x \rangle$ as input. According to Lemma 5 on page 97 we can view this graph as a deterministic finite automaton DFA (denoted by transformation t_2). Transformation t_1 denotes the conversion of the control flow graph to a path expression of type (n_e, n_x) by our data-flow framework (cf. also Diagram (6.28)). The generated path expression is denoted by $R^N_{n_e,n_x}$. Transformation t_3 is a compound transformation that comprises the following steps: the conversion of the path expression to a nondeterministic finite automaton, the subsequent conversion to a deterministic finite automaton, and the minimization of the result. The resulting deterministic finite automaton DFA′ is then checked for isomorphism against automaton DFA. This approach worked sufficiently well even for large flowgraphs. In fact we could validate 5048 out of 5053 graphs using this approach.

However, for the remaining 5 graphs validation turned out to be intractable. We investigated a different approach, based on [HU79, Theorem 3.8]. From there it follows that the languages $L(A_1)$ and $L(A_2)$ accepted by two finite automatons A_1 and A_2 are the same if

$$\big(L(A_1) \cap \overline{L(A_2)}\big) \cup \big(\overline{L(A_1)} \cap L(A_2)\big) = \emptyset.$$

In the above equation overlining denotes the complement of a language. Complementing a language required to compute the complement of a *deterministic* finite automaton. Making the automaton computed from the path expression $R^N_{n_e,n_x}$ deterministic turned out to be intractable for the remaining 5 graphs.

For the conversions and tests on automata and regular expressions described in this section we have used the Grail tool [RW94], a symbolic computation environment for finite-state automata and regular expressions. We have ported this tool to Linux, using the GNU toolchain with g++ 3.3. In addition we needed to add an *integer* alphabet to Grail in order to support automata with several thousand states and input symbols.

6.5.3 Data Evaluation

The SPEC CPU95 benchmark contains 18 benchmark programs divided into 8 integer (CINT95) and 10 floating point (CFP95) benchmarks. Tables 6.1 and 6.2 enumerate these benchmark programs along with the corresponding benchmark number and a short program description.

no.	name	description
099	go	an internationally ranked go-playing program
124	m88ksim	a chip simulator for the Motorola 88100 microprocessor
126	gcc	GNU C compiler (version 2.5.2)
129	compress	a in-memory version of the common Unix utility
130	li	Xlisp interpreter
132	ijpeg	image compression/decompression on in-memory images
134	perl	an interpreter for the perl language
147	vortex	an object oriented database

Table 6.1: SPEC CINT95 Benchmarks

no.	name	description
101	tomcatv	vectorized mesh generation
102	swim	shallow water equations
103	su2cor	Monte-Carlo method
104	hydro2d	Navier Stokes equations
107	mgrid	3d potential field
110	applu	partial differential equations
125	turb3d	turbulence modeling
141	apsi	weather prediction
145	fpppp	from Gaussian series of quantum chemistry benchmarks
146	wave5	Maxwell's equations

Table 6.2: SPEC CFP95 Benchmarks

All 18 benchmark programs together account for a total of 5053 procedures. Figure 6.12 depicts the distribution of these 5053 procedures among the benchmark programs.

We further differentiated between procedures with reducible and irreducible flowgraphs. From 5053 procedures, we found 5005 to contain reducible flowgraphs, while only 48 procedures turned out to be irreducible (the procedures with irreducible flowgraphs are listed in Table 6.3).

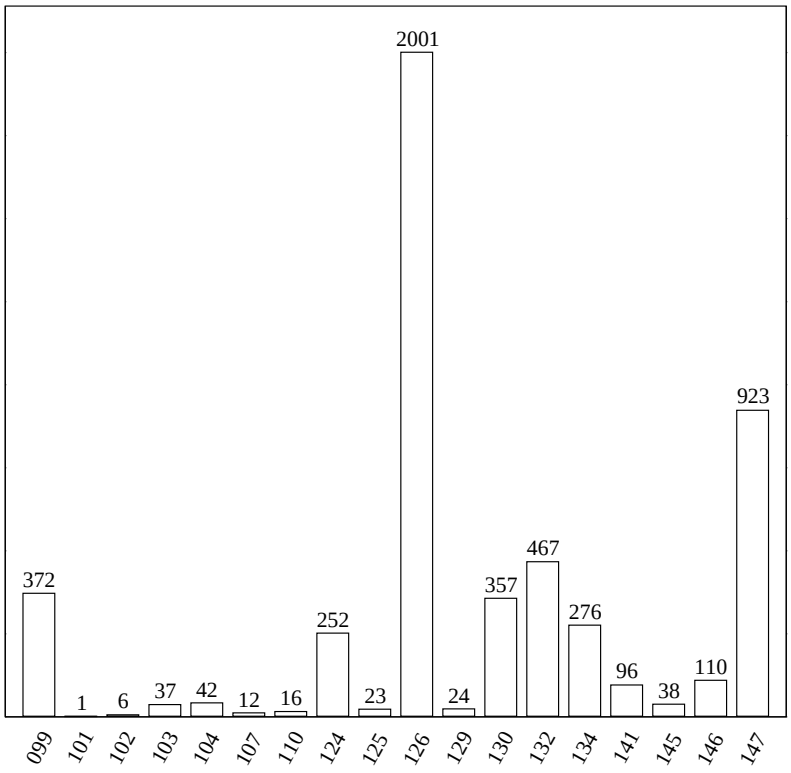


Figure 6.12: Number of Procedures Per Benchmark

#	benchmark #	procedure name
1	099	g2_c__readfile
2	124	asm_c__a_pname
3	124	cmdparser_c__pname
4	124	rm_c__rm
5	126	c-lex_c__yylex
6	126	caller-save_c__save_call_clobbered_regs
7	126	c-common_c__check_format_info
8	126	c-decl_c__store_parm_decls
9	126	combine_c__expand_field_assignment
10	126	c-parse_c__yyparse
11	126	cse_c__fold_rtx
12	126	cse_c__cse_insn
13	126	expr_c__get_pointer_alignment
14	126	flow_c__mark_used_regs
15	126	fold-const_c__div_and_round_double
16	126	loop_c__find_and_verify_loops

Table 6.3: SPEC95 Procedures with Irreducible Flow Graphs

#	benchmark #	procedure name
17	126	rtl_c__read_rtx
18	126	reorg_c__mark_target_live_regs
19	126	reorg_c__fill_simple_delay_slots
20	126	rtlanal_c__note_stores
21	126	rtl_c__read_skip_spaces
22	126	sched_c__schedule_block
23	126	stmt_c__check_for_full_enumeration_handling
24	126	stmt_c__bc_check_for_full_enumeration_handling
25	126	stmt_c__group_case_nodes
26	126	ucbqsort_c__qst
27	130	xllist_c__map
28	132	libpbm1_c__pm_init
29	132	libpbm5_c__pbm_dumpfont
30	132	rdgif_c__start_input_gif
31	134	cmd_c__cmd_exec
32	134	doarg_c__do_pack
33	134	eval_c__eval
34	134	doio_c__do_exec
35	134	dolist_c__do_match
36	134	dolist_c__do_split
37	134	dolist_c__do_unpack
38	134	dolist_c__do_kv
39	134	form_c__format
40	134	perly_c__yyparse
41	134	regcomp_c__regcomp
42	134	regcomp_c__regclass
43	134	str_c__intrpcompile
44	134	toke_c__yylex
45	134	toke_c__load_format
46	145	fmtgen_f__fmtgen_
47	145	fmtset_f__fmtset_
48	147	query_c__Query_AssertOnObject

Table 6.3: SPEC95 Procedures with Irreducible Flow Graphs

Observation 1: For the SPEC95 benchmark suite the share of irreducible control flow graphs is less than 1% and hence almost negligible.

Another topic of interest has been the edge/node ratio of SPEC95 control flow graphs. Figure 6.13 lists the ratio edges/nodes over nodes on a logarithmic scale.

Observation 2: It can be seen in Figure 6.13 that the edge/node ratio of SPEC95 control flow graphs is constant. This means that the number of edges

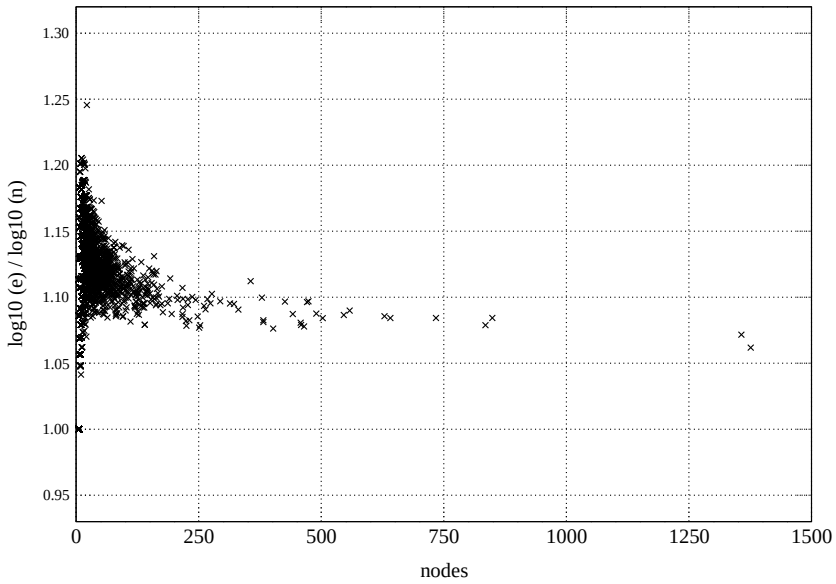


Figure 6.13: Edge/Node Ratio

is a linear function on the number of nodes.

Figure 6.14 graphs the loncp values of the SPEC95 procedures over the corresponding number of edges. Again we distinguish between reducible and irreducible control flow graphs. The fit for the data on reducible flowgraphs turns out to be a straight line for the logarithmic scale chosen for the loncp values. The fitted function is

$$\text{loncp}(e) = 2.9131 * \exp(0.076542 * e),$$

where e denotes the number of edges of a given procedure. The correlation coefficient of this function is 0.86041. This leads us to the following observation.

Observation 3: For the procedures of the SPEC95 benchmark suite the loncp metric is exponential in the number of CFG edges.

The fitted function for irreducible flowgraphs suggests that the loncp metric grows at a slower rate for irreducible graphs, but the sample of only 48 graphs of that kind is by no means representative.

Figure 6.15 (a) contains a quantile plot of the loncp values of the SPEC95 procedures. The f quantile $q(f)$ of a set of data is a value along the measurement scale of the data with the property that approximately a fraction f of the data are less than or equal to $q(f)$. The 0.25-, 0.50-, and 0.75-quantiles are distinguished values called lower quartile, median, and upper quartile respectively. If we order our observations of loncp values from smallest to largest, we can graph them against f. In this way quantile plots are an effective means of visualizing distributions. Coming back to our quantile plot depicted in Figure 6.15 (a), we arrive at the following observation.

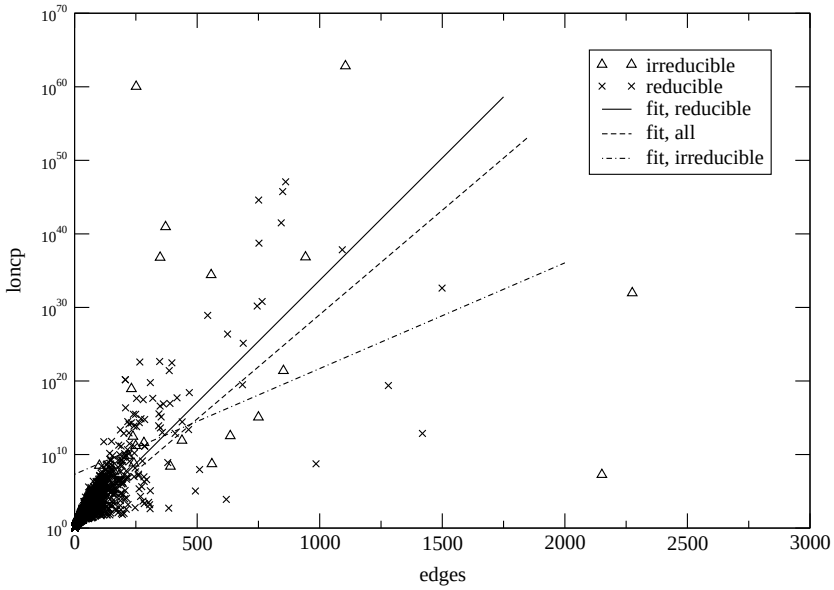


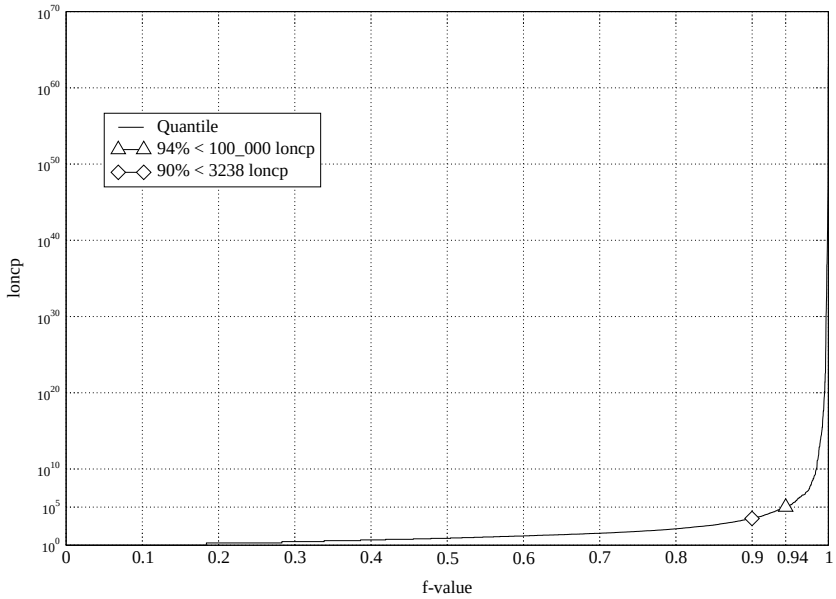
Figure 6.14: Regression

Observation 4: The distribution of loncp values starts at the lowest possible value (1) and increases modestly up to the 0.94 quantile. Thereafter we can observe an excessive increase of quantiles which suggests that the final 6 percent of our distribution represent costly outliers. Figure 6.15 (b) has been scaled and excludes outliers above 10^6. The two distinguished data points in the upper right corner represent the 0.9 quantile and the 0.94 quantile. Hence we learn that for 90 percent of the distribution the loncp-value is below 3238, and for 0.94 percent it is still below 100000. The bottom line of this observation is that for the major part of SPEC95 procedures the loncp metric yields surprisingly low values, but that a few outliers turn out to be very costly.

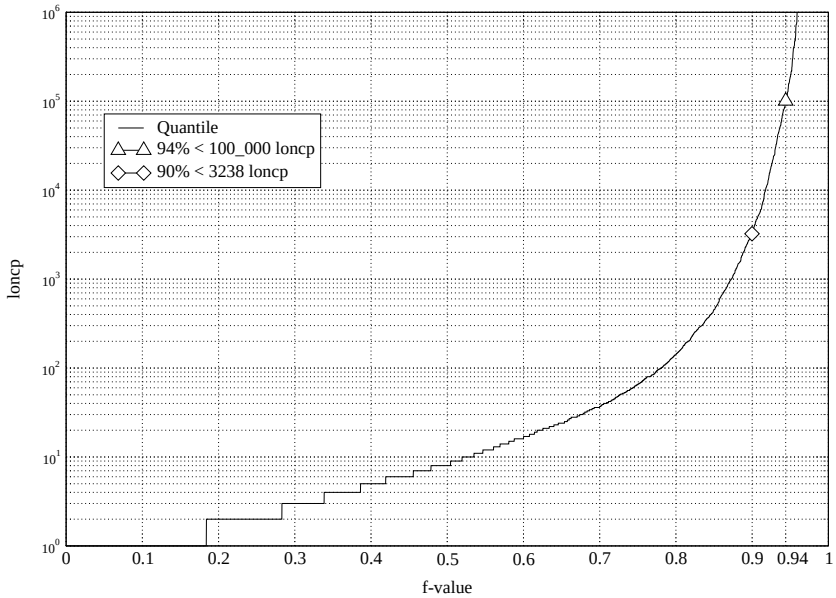
Observation (4) is also supported by the box plots of the loncp values depicted in Figure 6.16. For these plots the SPEC95 procedures have been grouped into the 18 benchmark programs, with an additional box plot titled "all" for the overall distribution. Note that benchmark programs 101 and 102 contain too few procedures to justify a box plot. For this reason their loncp values have been depicted as single data points. For benchmark 102 the data point at loncp = 17 actually represents three occurrences, which is in line with the total number of six procedures given in Figure 6.12.

Figure 6.16 (b) has again been scaled to exclude outliers above 10^6. In addition it contains a line connecting the medians of the respective plots. This line is helpful for locating the median for distributions like that of benchmark program 130 where the median falls together with one of the quartiles. Note that with all box plots the whiskers are drawn to the 0.1 and 0.9 quantiles.

Observation 5: Although the distributions of the SPEC95 benchmark programs have a large spread, their *center*, represented by the median, has a low loncp value. As an example consider the overall distribution depicted in the

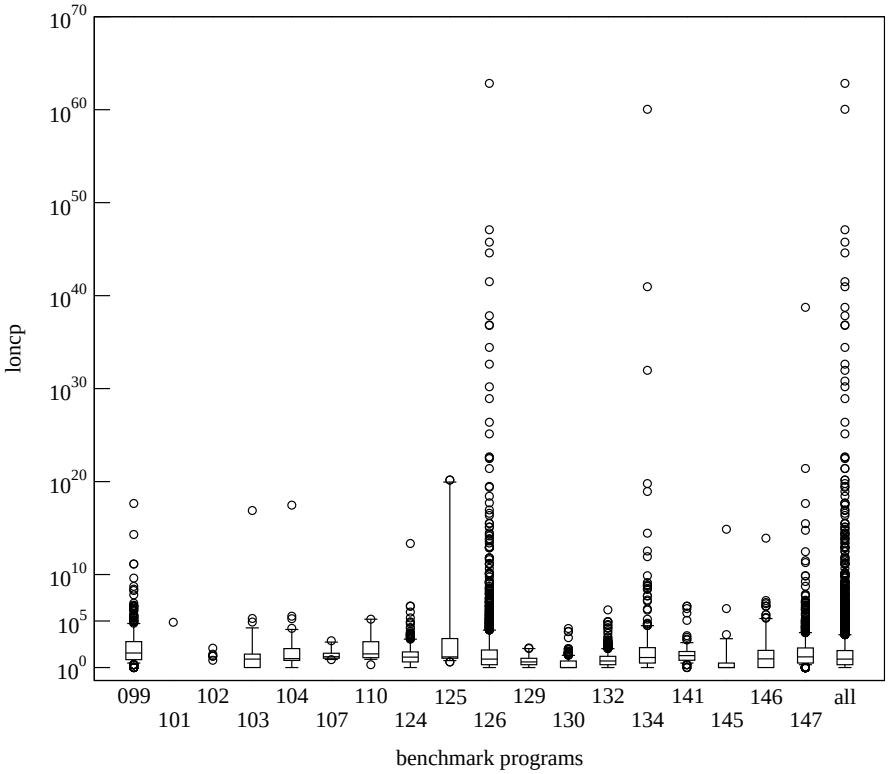


(a) Loncp Quantile Plot: complete distribution

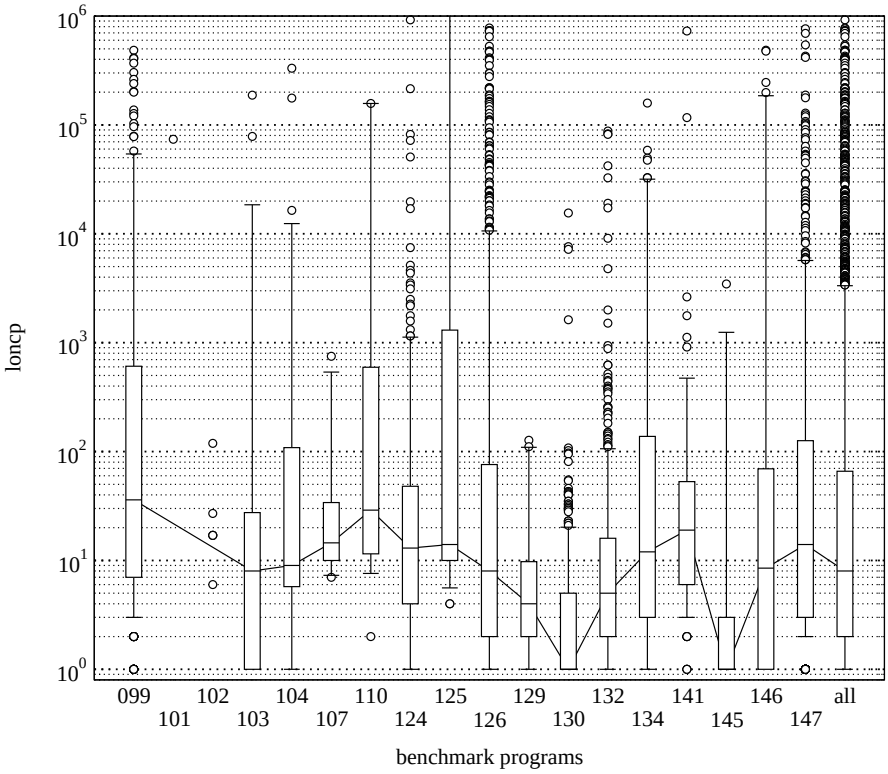


(b) Loncp Quantile Plot: outliers above 10^6 removed

Figure 6.15: Quantile Plot for SPEC95 Programs



(a) Loncp Box Plot: complete distribution



(b) Loncp Box Plot: outliers above 10^6 removed

Figure 6.16: Box Plot for SPEC95 Programs

rightmost column of Figure 6.16. It spreads across the interval of $[1, 10^{63}]$, whereas the median is at 8 (cf. also the 0.5 quantile of Figure 6.15 (b)).

Observation 6: The interquartile ranges of the SPEC95 benchmark programs are small. This means that the middle 50 percent of the data are tightly packed around the median. As an example we consider again the overall distribution depicted in the rightmost column of Figure 6.16 (b); It has a lower quartile of 2 and an upper quartile of 66 which results in an interquartile range of only 64.

Small interquartile ranges and low loncp values for the median support our argument stated in Observation (4) that the major part of SPEC95 procedures yield low loncp values, but that a few outliers are very costly.

Observation 7: The distributions of the majority of surveyed benchmark programs are skewed towards *larger* loncp values. This observation is supported by three facts regarding median, whiskers, and outliers of the plots.

1. For most benchmark programs the median is closer to the lower than to the upper quartile.

2. For every benchmark program the lower whisker ranging from the 0.1 to the 0.25 quantiles is smaller than the upper whisker ranging from the 0.75 to the 0.9 quantiles.

3. For every benchmark program the lower relative range of outliers (up to the 0.1 quantile) is by orders of magnitudes smaller than the corresponding upper relative range.

It should be noted that similarly to the interquartile range, the whiskers provide summaries of spread and shape of our benchmark data. But contrary to the interquartile range, the whiskers are not concerned with the center but with the extremes (or tails) of a distribution. The outliers, in turn, provide insight on the spread and shape in the extreme tails.

The remaining observations deal with the improvement of the loncp metric over the ncp metric. For all procedures except procedure `toke_c_yylex` from benchmark program 134 we have determined both ncp and loncp values[‡]. Figure 6.17 depicts the results of this survey.

Observation 8: From the 5052 examined SPEC95 procedures

- 18 percent had already the minimum metric value of 1,
- 22 percent yielded a smaller result for the loncp metric than for the ncp metric,
- and 60 percent could not be improved with the loncp metric.

In addition to the reduction in absolute values we also computed the *relative* reduction r' according to the formula

$$\text{r'} = \text{round}\big[10 * (100 - \frac{\text{loncp}(P)}{\text{ncp}(P) * 10^{-2}})\big] * 10^{-1}, \tag{6.30}$$

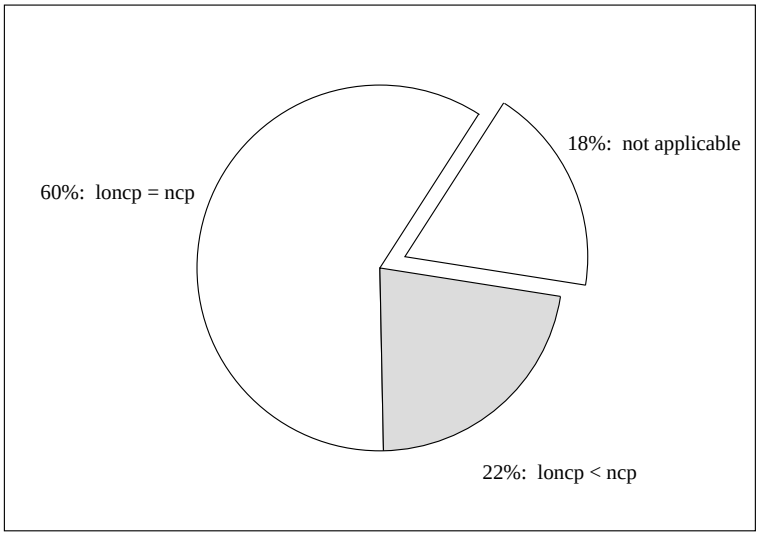


Figure 6.17: Loncp Metric: Number of Unaffected vs. Improved Procedures

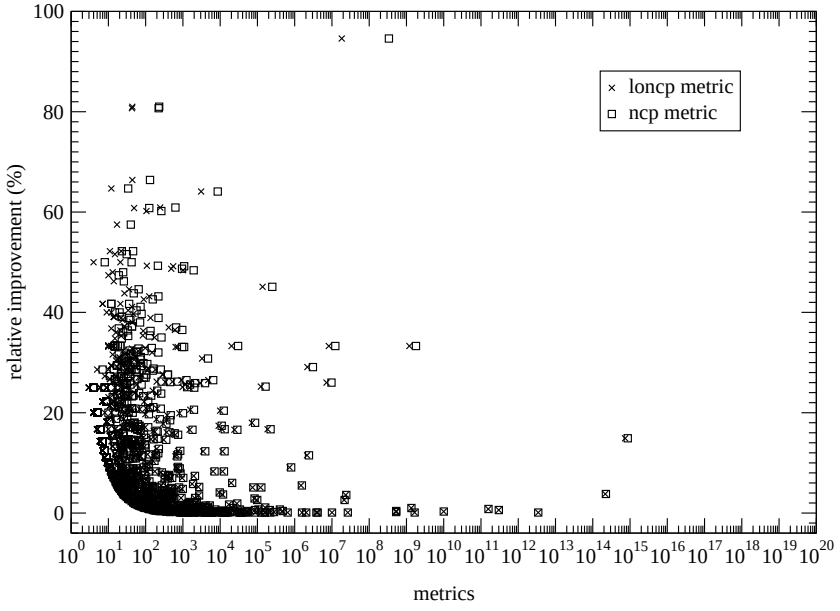


Figure 6.18: Relative Improvement of Loncp over Ncp Metric

where "round" denotes the *round to nearest* rounding function. It became then apparent that there exist cases where the loncp metric achieves substantial improvements in terms of absolute values, but the relative reduction is comparatively small.

Figure 6.18 lists the ncp and loncp values (x-axis) of all procedures with a relative reduction (y-axis) greater than zero. This figure already suggests that higher ncp values result in lower relative reduction.

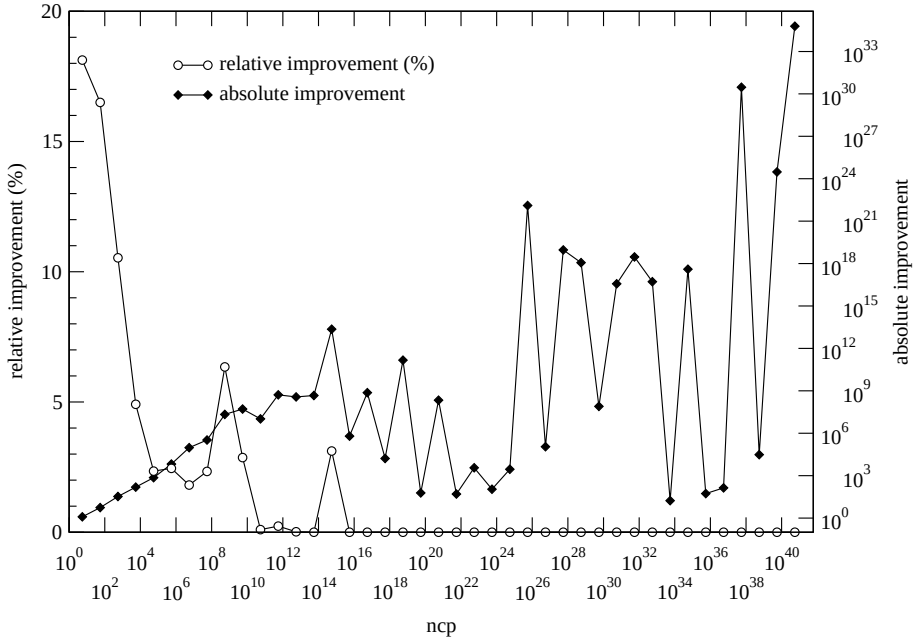


Figure 6.19: Comparison of Relative and Absolute Improvement

We derive the confirmation of this observation from Figure 6.19, where we directly compare absolute and relative reduction for ncp values. The x-axis of Figure 6.19 corresponds to the ncp values of procedures with an absolute reduction greater than zero. The y-axis on the left-hand side of this figure lists the relative reduction, whereas the y-axis on the right-hand side gives the absolute reduction. Every datapoint represents the average of the ncp-interval $[i, 10 * i)$, with $1 \leq i \leq 10^{40}$. Note that the 0.75 quantile for this distribution is below 10^5, which means that datapoints above this value represent comparatively few outliers. This is also in line with Observation 7. In this way the representative part of the graphs is located in the ncp interval $[1, 10^5]$. This leads us to our final observation.

Observation 9: For the majority of SPEC95 procedures that can be improved by the loncp metric, we have a negative correlation between ncp values and relative improvement, and a positive correlation between ncp values and absolute improvement.

The reason for this observation is at the moment unclear and needs further investigation. One possible cause could be that the optimization achieved with

[‡]For procedure `toke_c__yylex` only the loncp value has been computed.

the loncp metric is based on the elimination of loops, and that programmer-supplied loops rarely exceed a given size. This in turn also limits the potential of a method targeting at loops. The loops contained in path expressions with high ncp values could be due to the use of `goto` statements. Another source for such loops (also possibly due to gotos) could be machine-generated code (e.g., scanners and parsers). It is likely that source code of this kind displays other characteristics as man-made goto-less code. It should however be noted that static program analysis methods are, for obvious reasons, mainly applied to the latter category.

Chapter 7

Related Work

7.1 Early Entrepreneurs

As early as in 1976 J. King [Kin76] introduced symbolic execution along distinct program paths as a software testing and program verification methodology. Execution proceeds as with standard semantic execution, except that the values of expressions may be symbolic formulas over the input symbols. The state of the computation is represented by the set of program variables, a statement counter, and a pathcondition. Program verification is facilitated by allowing the programmer to specify boolean predicates at certain program points, which are then evaluated using the current state of the computation. Overall, the approach can be viewed as an early attempt to automate static analysis, in this case restricted to classes of input resulting in the same program path taken.

Around the same time T. E. Cheatham and J. A. Townley started work on a program development system that aimed at the provision of tools and facilities for program development and maintenance. One of the planned tools was the Symbolic Evaluator (SE) which was needed to generate a program database for a given program. The database contained the original program, annotated with symbolic values for each construct, plus further known facts on the program such as constraints. The operation of the SE on several example-loops is sketched in [CT76]), with a focus on the determination of induction variables, recurrence relations, closed forms, and bounds for the number of loop iterations. The paper already mentions the computation of normal forms for symbolic expressions. Further descriptions of the program development systems are given in [CTH79, CHT81].

In [CHT79] symbolic evaluation with the Symbolic Evaluator is used as a means of static program analysis for sequential EL1 programs. The approach considers all paths that reach a program point thus implying exponential growth. The used program representation is somewhat costly in the sense that program statements, predicates of conditional assignments, and variable values are all separately represented and linked via context numbers and time labels. A list of values ever assigned to a variable is explicitly stored and retrieved via time stamps. Techniques to solve recurrence relations and to simplify systems of constraints are introduced, with a restriction to linear problems. Path conditions are not employed with this approach. In [Plo79] the Symbolic Evaluator is

extended to support interprocedural symbolic analysis by so-called templates that describe the accumulated side-effects of a procedure as a function of its input parameters and the initial values of free variables. Procedures are analyzed as if the call-environment were unknown. However, for cases where this generic template mechanism would create undue complexity, information from the call-site is used to create a template of restricted applicability. At a procedure call site a template is therefore checked for applicability before it is instantiated with the call environment.

In the work of E. Ploedereder [Plo80] a comprehensive semantic model for symbolic analysis and program verification based on denotational semantics is presented. The basic model provides semantic descriptions for constants, identifiers, pure functions, selection, assignment, sequenced and conditional expressions, compound expressions, declarations, input/output functions, assertions, procedures, and loops. The basic model is then extended to the semantics of EL1. A further extension includes recursive procedure calls, pointer-related semantics, and run-time exception detection. Despite the provision of semantic functions for weakest symbolic preconditions and strongest symbolic postconditions in the sense of Dijkstra's predicate transformers (cf. [Dij76]), there is a clear-cut distinction between symbolic evaluation and the utilization of its results, e.g., by a program verification tool.

7.2 Abstract Interpretation

Paralleling the ongoing research on symbolic evaluation, P. and R. Cousot initiated work in the area of abstract interpretation (cf. [CC77]), which can be considered a theory of semantic approximation for the analysis of sequential programs. The need for semantic approximations arose from the fact that standard semantic valuation functions are not in general computable. Hence the idea arose to use a simplified non-standard semantics whose valuation functions are computable. In order to obtain standard-semantic meaning from the non-standard semantics of a program, the two meanings must be related by the abstraction relation that describes the abstracted aspects as well as the properties of the standard-semantic meaning that can be deduced from the non-standard semantics. There is a clear trade-off between the level of abstraction and the precision of the computed results. Moreover, the non-standard semantics have to be carefully tailored to the needs of the particular application. Due to its approximations abstract interpretation is incomplete, but it may find solutions to problems which can tolerate imprecise results.

An example of an application of abstract interpretation is the extraction and propagation of variable range constraints across a program. It is used by F. Bourdoncle in [Bou93] to determine the range of scalar variables in Pascal programs (although the constraints are not symbolic in this work). Blume and Eigenmann [BE96] use it to determine constraints about variable ranges in shared memory parallel programs. Their way of determining the ranges of program values employs data-flow analysis involving iteration to a fixed point. To make this computation feasible, widening operations are employed in order to reduce the number of iterations. Widening must be considered a further conservative approximation disturbing the accuracy of the computed results. In order to compensate for the loss of accuracy due to widening a subsequent

narrowing step is introduced which discards infinite bounds but does not tighten finite bounds.

P. Cousot and N. Halbwachs (cf. [CH78]) apply abstract interpretation to determine linear equality and inequality relations among variables of a sequential program. With their approach, sets of constraints between variables are represented as n-dimensional convex polyhedrons computed by abstract interpretation.

M. Haghighat and C. Polychronopoulos [HP96] describe several symbolic analysis techniques that are based on abstract interpretation. The information of all incoming program paths to a statement is intersected at the cost of analysis accuracy. We describe the symbolic domain used by this approach in Section 7.3, the symbolic differencing method for induction expressions is discussed in Section 7.5.

Comparing abstract interpretation to our approach of symbolic analysis, it has to be noted that symbolic analysis precisely represents the values of program variables which contrasts the approximations produced by abstract interpretation. As already mentioned, applications of abstract interpretations have to face the trade-off between the level of abstraction and the precision of the computed results. The representations of the two methods are also fundamentally different: while abstract interpretation is only concerned with state information, our symbolic evaluation method is based on the notion of supercontexts which also include information from pathconditions to determine under which conditions the symbolic values encapsulated in a program state reach a certain flow graph node.

Finally, the introduction of an approximation is also an option for symbolic evaluation to treat those cases where an exact solution is not feasible. However, with symbolic evaluation we can precisely delimit the application of an approximation to the unsolvable subpart of a problem (e.g., the recurrence relation of an induction variable v for which no closed form can be derived), while still being elevated by the exact results "surrounding" the unsolvable subpart (e.g., closed forms of induction variables that are part of the recurrence relation of v). In minimizing the number of approximations we can expect to keep the precision of the analysis at a maximum.

7.3 Symbolic Domains

Geddes et al. [GCL92, Chapter 2] survey the algebraic structures that our symbolic domain introduced in Section 4.1.1 is based upon. Chapter 3 of [GCL92] addresses various computer representations of algebraic objects, together with a treatment of the zero equivalence problem and normal forms. Involved is the issue of *simplification* of algebraic objects, which has already been raised by Moses [Mos71]. Geddes et al. introduce three levels of abstraction in order to distinguish between mathematical objects and their computer representations. With our work we have adopted this hierarchy (cf. Section 4.1).

In [GCL92] there is already a distinction between equivalence and identity of expressions, which is however not due to the distinction between syntax and semantics, but to the use of an equivalence operator $=$ on the object level and an identity operator $\equiv$ on the form level. A semantics-based transformation function converts expressions within the same equivalence class.

As already mentioned, there is no clear distinction between syntax and semantics of symbolic expressions. Furthermore this approach does not make use of syntax-based term rewriting, and there is no notion of a symbolic predicate domain.

The abstract symbolic domain of Haghighat presented in [Hag95] is based on Geddes et al. [GCL92]. It is implementation-centered in the sense that it is mainly concerned with the data-structure level. There is no clear-cut distinction between syntax and semantics of symbolic expressions, but the zero equivalence problem is solved on the syntactic level, based on a unique normal form for symbolic expressions. It is however not mentioned how symbolic expressions are transformed into canonical form. As this approach does not use pathconditions, there is also no notion of a symbolic predicate domain.

The method of denotational semantics is often used to map derivation trees of a given programming language to a semantic domain where symbolic evaluation can then be performed. As noted in Section 2.2.2, semantic domains are algebras, which, according to the hierarchy of Section 4.1, are located on the *object* level. Obviously an underlying implementation is however restricted to the form or data structure level. Even if this distinction is not made, the presence of the form level is usually manifested through symbolic expression simplification facilities which, among other purposes, have to derive unique normal forms for symbolic expressions.

Initially our approach also employs denotational semantics, but we establish a clear-cut distinction between object and form level, as outlined in Section 5.5.

7.4 Symbolic Evaluation

In [Bli02] symbolic evaluation is employed to estimate the worst-case execution time of sequential real-time programs. With this approach programs are mapped to control flow graphs such that graph nodes represent the basic blocks of the program, and edges denote the transfer of control between basic blocks. Symbolic evaluation is set up as a data-flow problem, with equations describing the solutions at the respective CFG nodes. Figure 7.1 shows the Ada-code of our running example from Section 5.4 together with the associated control flow graph.

```
1   b := b + 1;              -- node n1
2   while j <= m loop        -- node n2
3      d := 2*d;             -- node n3
4      j := j+b;             -- node n3
5   end loop;
```

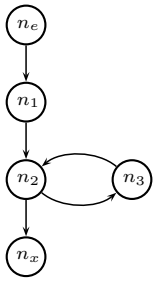


Figure 7.1: Example Program and Control Flow Graph

A variant of denotational semantics is used to derive the branch predicates and side-effects from the source-code of the program. Equations (7.1)–(7.5) constitute the data-flow equations for this example, where X_{n_i} denotes the

data-flow equation for node n_i.

$$X_{n_e} = \left[\{(b, \underline{b}), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\},\ true\right] \tag{7.1}$$

$$X_{n_1} = X_{n_e} \mid \{(b, b+1)\} \tag{7.2}$$

$$X_{n_2} = X_{n_1} \cup X_{n_3} \tag{7.3}$$

$$X_{n_3} = (j \leq m) \odot X_{n_2} \mid \{(d, 2 \cdot d), (j, j+b)\} \tag{7.4}$$

$$X_{n_e} = \neg(j \leq m) \odot X_{n_2} \tag{7.5}$$

The graph of the program state together with the pathcondition makes up a program context, as can be seen in Equation (7.1)*. The effect of a basic block is denoted by the term after the "|"-operator, branch predicates are followed by the symbol "$\odot$".

With this notation Equation (7.4) reads as follows: Equation X_{n_3} for node n_3 depends under the branch predicate $j \leq m$ on Equation X_{n_2} for node n_2, with the local effect of setting the value of variable d to $2 \cdot d$, and the value of variable j to $j + b$.

In order to solve this data-flow problem by an elimination algorithm, a normal form for equations, a procedure to insert one equation into another, and a loopbreaking rule to resolve circular equational dependencies are introduced. During a loopbreaking step the induction variables of the loop body are determined, a system of recurrence relations is set up, and closed forms are derived where possible.

The elimination algorithm reduces the control flow graph to the dominator tree, while performing equation insertions and loop breaking in parallel. Once the control flow graph is reduced to the dominator tree, each data-flow equation depends only on its immediate dominator. The solution to the data-flow equation at the root node of the dominator tree (X_{n_e} in the above example) can then be propagated along the dominator tree to derive the data-flow solution for the exit node.

In formulating symbolic evaluation as a data-flow problem this approach differs from ours since we set up a data-flow problem to calculate path expressions (cf. Section 6.2) that are then interpreted by redefining the operators of the regular expression algebra. Another difference concerns the flow graph nodes, for which a solution is computed: we can compute the MOP solution for any node in the control flow graph (also for nodes within a loop), while in [Bli02] the solution for the exit node is computed. The emphasis on the exit node is due to the fact that for worst-case execution time analysis the time spent until and including the exit note is of sole interest.

Despite its initial orientation towards worst-case execution time analysis this approach has been successfully applied to many other application areas [BB98, BBS99, BBS00, BB03]. The technique of symbolic instrumentation ([Bli02, Definition 15]) is furthermore an elegant example of an auxiliary semantics (cf. [WM95, Section 10.2.3]) that can easily be adopted by our approach.

In [Sch01, FS03] a novel representation for program contexts is introduced. It consists of a state s, a state condition t, and a path condition p. While s and p are comparable to our notions of program state and pathcondition, the

*Although in [Bli02] the word "context" corresponds to our notion of a "supercontext".

state condition t is used to store additional assumptions on the values of variables in s, such as constraints between variable values or user assertions. The representation of program contexts is a sparse representation that makes use of a folding operator $\odot$ in order to collapse analysis information at confluence nodes. It joins two contexts $[s', t', p']$ and $[s'', t'', p'']$ and generates a symbolic state composition $[s''', t''', p''']$, written as

$$[s', t', p'] \odot [s'', t'', p''] = [s''', t''', p'''].$$

Therein

$$s' = \{v_1 = e_1, \ldots, v_n = e_n\},$$
$$s'' = \{v_1 = f_1, \ldots, v_n = f_n\},$$

and the context $[s''', t''', p''']$ resulting from the folding operation is defined such that

$$\begin{aligned}
g_i &= \begin{cases} e_i & : \text{ if } e_i = f_i, \\ \text{new symbol} & : \text{ otherwise,} \end{cases} \\
s''' &= \{v_1 = g_1, \ldots, v_i = g_i, \ldots, v_n = g_n\}, \\
\Omega' &= \bigwedge_{1 \le i \le n, e_i \neq f_i} (g_i = e_i), \\
\Omega'' &= \bigwedge_{1 \le i \le n, e_i \neq f_i} (g_i = f_i), \\
t''' &= (t' \vee t'') \wedge ((p' \wedge \Omega') \vee (p'' \wedge \Omega'')), \\
p''' &= p' \vee p''.
\end{aligned}$$

The definition of the folding operator $\odot$ considers only variables with different symbolic values in state s' and state s''. For those variables with different symbolic values new symbols are introduced and bound either to the symbolic value of s' or the symbolic value of s'', depending on the path condition.

Furthermore, [Sch01, FS03] contain denotational definitions for assignments, input/output operations, conditional statements, loops, procedures, and dynamic data structures. An array algebra models symbolic read and write operations on array variables. An algorithm for the generation of program contexts from a reducible control flow graph extended by loop preheader-, postbody-, and postexit-nodes of a program is also presented.

Alone from the number of supported programming language constructs this approach is very comprehensive and therefore hardly comparable to the approach chosen with our simple, yet Turing-equivalent Flow language. A major difference is clearly rooted in the fact that with our approach we employ path expressions to capture the control flow information of programs, which also works for irreducible control flow graphs. Admittedly our correctness proof is very much facilitated by the simplicity of the Flow language. The sparse context representation is an efficient alternative to the path-enumerating representation used with our approach. The rich set of supported programming language constructs, esp. the formalism for symbolic arrays, can be considered valuable extensions of our approach.

7.5 Induction Variable Substitution

Symbolic analysis methods used by current state-of-the-art restructuring compilers require the availability of closed-form expressions at the source-code level, which involves treatment of induction variables in loops. Generalized induction variables (GIVs) are a general class of induction variables that form polynomial and geometric progressions through loop iterations [GSW95]. The recognition of GIVs and the subsequent replacement of GIV-uses with semantically equivalent closed form expressions is called *induction variable substitution* (IVS).

Haghighat's symbolic differencing method [Hag95] recognizes generalized induction variables and determines closed form expressions by a method called *symbolic differencing*. Therein a generalized induction variable (GIV) is characterized by its function $\mathcal{X}$ defined by

$$\mathcal{X}(n) = \varphi(n) + ra^n, \tag{7.6}$$

where n is the loop iteration number, φ is a polynomial of order k, and a and r are loop-invariant expressions. The forward difference operator Δ is defined by

$$\Delta\mathcal{X}(n) = \mathcal{X}(n+1) - \mathcal{X}(n).$$

Each term in the successive differences of φ is the sum of two parts, one arising from $\varphi(n)$, and the other from ra^n. Since $\varphi(n)$ is a polynomial of order k, the part arising from it will vanish after $k+1$ differences, and the remaining parts in the differences will form a geometric progression with the common ration r.

Conversely, if the first few terms of a series are known, and if the $(k+1)^{\text{th}}$ differences of these terms form a geometric progression whose common ratio is r, then it is assumed that the general term of the given series is of the form stated in Equation (7.6), where φ is an integer-valued polynomial in n of degree k.

From this an algorithm for the recognition and substitution of generalized induction variables can be devised[†]. Given a loop L, where f_{sc} denotes the accumulated effect of one iteration of the loop body of L. We can compute the effect of the first i iterations of the loop body on a clean slate program context c, denoted by

$$c_1 = f^1_{\text{sc}}(c),\ c_2 = f^2_{\text{sc}}(c), \ldots,\ c_j = f^j_{\text{sc}}(c).$$

Suppose we are interested in variable v_i. After constructing a difference table for the expressions $\text{st}(c_\nu)(v_j), 1 \leq \nu \leq i$, Newton's formula for forward interpolation (cf. [Bac96, Equation 3.1], [Hag95], [Knu97, pp. 503–505]) will give us an interpolating formula for a closed form expression for variable v_j.

Example 7.1 Consider again our running example from Section 5.4. The program contexts c_ν for $1 \leq \nu \leq 2$ are given in Equations 5.7 and 5.8. Suppose we are interested in variable j, which gets assigned the expression $j + b$ of order $k = 1$. Figure 7.2 depicts the difference table for the expressions

$$\text{st}(c_\nu)(j),\ 1 \leq \nu \leq 3,$$

describing the value of variable j during the first three iterations of the example loop. Employing forward interpolation (cf. [Hag95, Section 4.3.1]) yields

[†]The explanation of this algorithm will already be given in terms of our symbolic evaluation approach.

k	symbolic differences		
0	$\underline{j} + \underline{b}$	$\underline{j} + 2 \cdot \underline{b}$	$\underline{j} + 3 \cdot \underline{b}$
1	$\underline{b}$	$\underline{b}$	
2	0		

Figure 7.2: Difference Table for Variable j

the closed form expression for variable j that we have already established in Equation 5.9.

$$\mathcal{X}(i) = \underline{j} + \underline{b} + \underline{b} \cdot \binom{i-1}{1} = \underline{j} + i \cdot \underline{b}$$

It is pointed out in [vE01] that the method of symbolic differencing is unsafe if the number of computed loop iterations, which is a user-supplied constant, is set too low. In that case an order-underestimate of recognized induction variables happens, which in turn causes wrong interpolations. Consequently the user must be knowledgeable about the type of program analyzed to make a wise decision.

Another method of induction variable substitution closely related to symbolic differentiation is based on *chains of recurrences* (CR). Chains of recurrences originated in the work of Bachmann, Zima, and Wang [BWZ94, Bac96, Zim95], they expedite the evaluation of closed form functions on regular grids (for further references cf. [Bac97, KMZ98, Zim01]). The CR-based IVS method recently developed by R. van Engelen et al. [vEBS$^+$04, vE00, vE01] is more general than the symbolic differencing method because it recognizes progressions due to polynomials, exponentials, factorials, and compositions of these.

Chapter 8

Conclusion and Future Work

Ever tried. Ever failed. No matter.
Try again. Fail again. Fail better.
— Samuel Beckett, quoted in "The Lure of the Quest",
a report on the dog sled race "Yukon Quest", written by John Balzar.

In this work we have taken a novel approach to symbolic analysis based on path expressions. We have shown that by means of path expressions we can capture the control flow information that is inherent in a program. By reinterpreting path expressions we have obtained a mapping from the path expression algebra into the symbolic analysis domain. At the heart of this domain is the supercontext, an algebraic structure capable of describing all possible variable bindings valid at a well-defined program point. For a possible implementation we have outlined a finite representation for supercontexts. Furthermore we have set up a data-flow problem in order to compute path expressions from arbitrary (i.e., reducible and irreducible) control flow graphs. We have defined metrics on path expressions in order to asses the required analysis effort. We have proved the minimality of the generated path expressions with respect to a given metric. Furthermore, the empirical data collected in an extensive survey show that symbolic evaluation is a methodology capable of coping with the considerable problem sizes that arise from contemporary real-world applications.

In the following we outline several topics for future work.

8.1 Induction Variables and Recurrences

Finding closed form expressions for the recurrence relations of a system of recurrences can be considered a major challenge for every symbolic analysis approach. Since this problem is undecidable in the general case, it is necessary to gain knowledge on the type of recurrence relations that occur in applications. Solution procedures will have to take into account symbolic differencing and the chains of recurrences method outlined in the related work. The latter is especially interesting due to its possible applicability to conditional recurrences.

8.2 Parallel Execution Within Flow

In Equation (3.3) on page 20 we required that for a given state s of our Flow execution-model the branch predicate of exactly one outgoing edge evaluates to true. We can weaken this requirement and allow more than one predicate to evaluate to true:

$$1 \leq | \bigcup_{e_i \in \mathrm{out}(s)} \{pred(e_i)(env) = true\} | \,. \tag{8.1}$$

This will incorporate the parallel execution-model into Flow. Two topics that have to be solved with this step are enumerated below.

1. Parallel threads of execution are bound to race-conditions (remember that all Flow variables are global). Therefore all possible interleavings between threads have to be generated as contexts.

2. Introduce a *synchronization primitive* that allows Flow program segments to enforce mutual exclusion in order to tackle the before-mentioned problem.

8.3 Flow Extensions and Implementation

We have proved in Section 3.5 that the computational model of the language Flow is equivalent to the computational model of a Turing machine. Yet there are several programming language features that, once incorporated into Flow, would facilitate the symbolic analysis of programs.

An important feature is the capability to analyze arrays. In [Sch01] an array algebra for symbolic arrays is described. Symbolic arrays can also be incorporated into our method of symbolic analysis. Despite their importance as a data-structure, arrays can also be used to model the memory of a computer. In this way arrays can be considered a prerequisite for an effective analysis targeting at the detection of aliases, memory leaks (cf. [SBF00]), and buffer overflows. Moreover, the pointer arithmetic of contemporary programming languages, and also of the intermediate language that the GCC uses, can be modeled with symbolic arrays. Mapping the intermediate language of GCC to Flow would enable us to symbolically analyze programs written in all programming languages for which a GCC frontend exists.

Currently we work on an implementation to automatically generate finite supercontexts from path expressions and side-effects. This implementation is based on OBJ3 [GWM+93]. We are planning to incorporate the path expression generation mechanism described in Chapter 6 to obtain an integrated framework that takes an input program, computes the path expressions and side-effects for its control flow graph edges, and generates the MOP solution for arbitrary flow graph nodes. We plan to investigate the application of this framework to several areas of static program analysis.

I keep riding, into the highest peering hills, up the pitch of a mountainside,
where green leaves quiver in the cold sun.
— Lance Armstrong.

Bibliography

[AC76] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137, 1976.

[Ada95] ISO/IEC 8652. *Ada Reference Manual*, 1995.

[All86] L. Allison. *A Practical Introduction to Denotational Semantics.* Cambridge Computer Science Texts. Cambridge-University-Press, 1986.

[AS03] J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations.* Cambridge University Press, 2003.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bac96] O. Bachmann. *Chains of Recurrences.* PhD thesis, Kent State University, Kent, Ohio - 44240, U.S.A., December 1996.

[Bac97] O. Bachmann. MPCR: An Efficient and Flexible Chains of Recurrences Server. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(1):15–21, March 1997.

[BB98] J. Blieberger and B. Burgstaller. Symbolic Reaching Definitions Analysis of Ada Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 238–250, Uppsala, Sweden, June 1998.

[BB03] J. Blieberger and B. Burgstaller. Eliminating Redundant Range Checks in GNAT Using Symbolic Evaluation. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 153–167, Toulouse, France, June 2003.

[BBS99] J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 136–145, Santander, Spain, June 1999.

[BBS00] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 225–237. Springer-Verlag, 2000.

[BBS02] J. Blieberger, B. Burgstaller, and G.-H. Schildt. *Informatik*. Springer, 4th edition, 2002.

[BE96] W. Blume and R. Eigenmann. Demand-Driven, Symbolic Range Propagation. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 141–160. Springer-Verlag, 1996.

[BEGO71] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in Graphs and Expressions. *IEEE Transactions on Computers*, 20(2):149–153, February 1971.

[BJ66] C. Boehm and G. Jacopini. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[BL82] B. Buchberger and R. Loos. Algebraic Simplification. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, volume 4 of *Computing. Supplementum*, pages 11–43, Wien / New York, 1982. Springer.

[BL96] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[Bli02] J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22:183–227, 2002.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, 1998.

[Bou93] F. Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.

[BSB04] B. Burgstaller, B. Scholz, and J. Blieberger. Tour de Spec — A Collection of Spec95 Program Paths and Associated Costs for Symbolic Evaluation. Technical Report 183/1-137, Department of Automation, Vienna University of Technology, June 2004.

[BWZ94] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of Recurrences - A Method to Expedite the Evaluation of Closed-Form Functions. In *ISSAC '94: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 242–249. ACM Press, 1994.

[BZZ03] R. Bagnara, A. Zaccagnini, and T. Zolo. The Automatic Solution of Recurrence Relations. I. Linear Recurrences of Finite Order with Constant Coefficients. Quaderno 334, Dipartimento di Matematica, Università di Parma, Italy, 2003.

[CC77] P. Cousot and R. Cousot. Abstract Intrepretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM*

Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, CA, January 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[CHT79] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, 5(4):403–417, July 1979.

[CHT81] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Program Refinement by Transformation. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 430–437. IEEE Press, 1981.

[Cle93] W. S. Cleveland. *Visualizing Data*. Hobart Press, 1993.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[CPU95] SPEC CPU95 Benchmark Suite, Version 1.10, August 1995.

[CR81] L. A. Clarke and D. J. Richardson. Symbolic Evaluation Methods for Program Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 264–300. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1981.

[CT76] T. E. Cheatham and J. A. Townley. Symbolic Evaluation of Programs: a Look at Loop Analysis. In *SYMSAC '76: Proceedings of the 3rd ACM Symposium on Symbolic and Algebraic Computation*, pages 90–96. ACM Press, 1976.

[CT04] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, San Francisco, 2004.

[CTH79] T. E. Cheatham, J. A. Townley, and G. H. Holloway. A System for Program Refinement. In *ICSE '79: Proceedings of the 4th International Conference on Software Engineering*, pages 53–62. IEEE Press, 1979.

[Dav82] M. Davis. *Computability and Unsolvability*. Dover, 1982.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[DST93] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, New York, NY, USA, second edition, 1993. With a preface by Daniel Lazard. Translated from the French by A. Davenport and J. H. Davenport. With a foreword by Anthony C. Hearn.

[Fah98] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.

[Feh89] E. Fehr. *Semantik von Programmiersprachen*. Studienreihe Informatik. Springer-Verlag, first edition, 1989.

[FS03] T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers*, volume 2628. LNCS, Springer-Verlag, 2003.

[GCL92] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1992.

[GD94] J. Goguen and R. Diaconescu. An Oxford Survey of Order Sorted Algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.

[Gin67] A. Ginzburg. A Procedure for Checking Equality of Regular Expressions. *J. ACM*, 14(2):355–362, 1967.

[GK82] D. Greene and D. E. Knuth. *Mathematics For the Analysis of Algorithms*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, second edition, 1982.

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[GM96] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, Mass., 1. edition, 1996.

[GM02] J. A. Goguen and J. Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations, March 08 2002.

[Gog80] J. A. Goguen. How to Prove Algebraic Inductive Hypotheses without Induction, with Applications to the Correctness of Data Type Implementation. In W. Bibel and R. Kowalski, editors, *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 356–373, Les Arcs, France, July 1980. Springer.

[Grä68] G. Grätzer. *Universal Algebra*. Van Nostrand Reinhold, Princeton, 1968.

[GS90] C. A. Gunter and D. S. Scott. Semantic Domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 633–674. Elsevier, Amsterdam, 1990.

[GSW95] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):85–122, January 1995.

[GWM+93] J. Goguen, T. Winkler, J. Meseguer, F. Futatsugui, and J. Jouannaud. Introducing OBJ. Draft, Oxford University Computing Laboratory, 1993.

[Hag95] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic, 1995.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[Hil00] D. Hilbert. Mathematische Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900. *Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, pages 253–297, 1900.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 2nd edition edition, 2001.

[HP96] M. R. Haghighat and C. D. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[HR04] M. R. A. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1979.

[Jac74] N. Jacobson. *Basic Algebra I*. W. H. Freeman and Company, San Francisco, CA, USA, 1974. ISBN 0-7167-0453-6 (v. 1.).

[Kil73] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[Kin76] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KMZ98] V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional Chains of Recurrences. In *ISSAC '98: Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 199–206. ACM Press, 1998.

[Knu97] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1997.

[Lei03] A. Leitsch. Algorithmen–, Rekursions– und Komplexitätstheorie, Skriptum zur Vorlesung. TU Wien, Institut für Computersprachen, AG Theoretische Informatik und Logik, 2003.

[LN73] H. Lausch and W. Nöbauer. *Algebra of Polynomials*. North-Holland, Amsterdam, 1973.

[Lou93] K. C. Louden. *Programming Languages – Principles and Practice*. PWS-Kent, Boston, Massachusetts, 1993.

[Lue80] G. S. Lueker. Some Techniques for Solving Recurrences. *ACM Computing Surveys (CSUR)*, 12(4):419–436, 1980.

[MG86] J. Meseguer and J. A. Goguen. Initiality, Induction, and Computability. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1986.

[Mos71] J. Moses. Algebraic Simplification: A Guide for the Perplexed. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 282–304. ACM, March 1971. Also available in Communications of the ACM, 14(8), 527–537, August 1971 (Section on Lexicographic Ordering not in revised version).

[Mos90] P. D. Mosses. Denotational Semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 575–631. Elsevier, Amsterdam, 1990.

[MP94] V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. In *Proc. of the International Conference on Parallel and Vector Processing*, pages 737–748, Linz, Austria, 1994.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of Data Flow Frameworks. A Unified Model. *Acta Informatica*, 28(2):121–163, December 1990.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[Pau88] M. C. Paull. *Algorithm Design: A Recursion Transformation Framework*. Wiley-Interscience, 1988.

[Plo79] E. Ploedereder. Pragmatic Techniques for Program Analysis and Verification. In *4th International Conference on Software Engineering (ICSE '79)*, pages 63–72. IEEE Press, September 1979.

[Plo80] E. Ploedereder. *A Semantic Model for the Analysis and Verification of Programs in General, Higher-Level Languages*. PhD thesis, Division of Applied Sciences, Harvard University, 1980.

[Pug92] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[Ram96] G. Ramalingam. Data Flow Frequency Analysis. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 267–277. ACM Press, 1996.

[Ram99] G. Ramalingam. Identifying Loops in Almost Linear Time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, 1999.

[Rog87] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, 1987.

[Ros95] K. H. Rosen. *Discrete Mathematics And Its Applications (3rd ed.)*. McGraw-Hill, Inc., 1995.

[RP86] B. G. Ryder and M. C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.

[RW94] D. Raymond and D. Wood. Grail: A C++ Library for Automata and Expressions. *Journal of Symbolic Computation*, 17(4):341–350, 1994.

[Sal66] A. Salomaa. Two Complete Axiom Systems for the Algebra of Regular Events. *J. ACM*, 13(1):158–169, 1966.

[SBF00] B. Scholz, J. Blieberger, and T. Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00)*, Boston, January 2000.

[Sch86] D. A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sch01] B. Scholz. *Symbolic Analysis of Programs and its Applications*. PhD thesis, Institute of Computer Languages, Vienna University of Technology, Vienna, Austria, 2001.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1988.

[Sho79] R. E. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[Sma96] N. Smart. *The Algorithmic Resolution of Diophantine Equations*. Cambridge University Press, November 1996.

[Sre95] V. C. Sreedhar. *Efficient Program Analysis Using DJ Graphs*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.

[Tar81] R. E. Tarjan. A Unified Approach to Path Problems. *Journal of the ACM (JACM)*, 28(3):577–593, 1981.

[Ten76] R. D. Tennent. The Denotational Semantics of Programming Languages. *Journal of the ACM (JACM)*, 19(8):437–453, August 1976.

[vE00] R. A. van Engelen. Symbolic Evaluation of Chains of Recurrences for Loop Optimization. Technical Report TR-000102, Florida State University, Computer Science Deptartment, 2000.

[vE01] R. A. van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. *Lecture Notes in Computer Science*, 2027:118–132, 2001.

[vEBS+04] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 106–115. ACM Press, 2004.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics*. International Series in Computer Science. Prentice Hall, 1st edition, 1991.

[WCHP01] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic Evolution: An Alternative to Induction Variable Substitution for Dependence Analysis. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 78–91. ACM Press, 2001.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1991.

[Zim95] E. V. Zima. Simplification and Optimization Transformations of Chains of Recurrences. In *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, pages 42–50. ACM Press, 1995.

[Zim01] E. V. Zima. On Computational Properties of Chains of Recurrences. In *ISSAC '01: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, pages 345–352. ACM Press, 2001.

Curriculum Vitae

Bernd Burgstaller

Contact Information

Institute of Computer Aided Automation
Vienna University of Technology
Treitlstr. 1–3/4
A-1040 Vienna, Austria
bburg@auto.tuwien.ac.at

Education

06/2002–02/2005	Doctoral Dissertation "Symbolic Evaluation of Imperative Programming Languages", Supervisor: Prof. J. Blieberger.
1997	M.S. (with distinction), Vienna University of Technology. "The WOOP Preprocessor—An Implementation of Discrete Loops in Ada95", Supervisors: Prof. J. Blieberger and Prof. U. Schmid.
1989–1997	Studies in Computer Science

Positions

02/2000–01/2004	Assistant Professor, Institute of Computer Aided Automation, Vienna University of Technology, Vienna, Austria.
03/1997–01/2000	Software Engineer and Software Architect, Philips Consumer Electronics, Vienna, Austria. Architecture, design and implementation of embedded software for VCRs and DVD+RWs, CMM level 2 and 3 process definition and implementation.
10/1988–05/1989	Military Service

Scholarship

01/1995–12/1996	FWF Austrian Science Fund, sponsored M.S. thesis.