The Journal of Systems and Software 85 (2012) 1418–1439

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A symbolic analysis framework for static analysis of imperative programming languages☆

Bernd Burgstaller[a,*], Bernhard Scholz[b], Johann Blieberger[c]

[a] Yonsei University, Seoul 120-749, Republic of Korea
[b] The University of Sydney, NSW 2006, Sydney, Australia
[c] Vienna University of Technology, 1040 Vienna, Austria

ARTICLE INFO

Article history:
Received 16 March 2010
Received in revised form 3 October 2011
Accepted 29 November 2011
Available online 8 December 2011

Keywords:
Static program analysis
Symbolic analysis
Path expression algebra
Programming language semantics

ABSTRACT

We present a generic symbolic analysis framework for imperative programming languages. Our framework is capable of computing all valid variable bindings of a program at given program points. This information is invaluable for domain-specific static program analyses such as memory leak detection, program parallelization, and the detection of superfluous bound checks, variable aliases and task deadlocks.

We employ path expression algebra to model the control flow information of programs. A homomorphism maps path expressions into the symbolic domain. At the center of the symbolic domain is a compact algebraic structure called supercontext. A supercontext contains the complete control and data flow analysis information valid at a given program point.

Our approach to compute supercontexts is based purely on algebra and is fully automated. This novel representation of program semantics closes the gap between program analysis and computer algebra systems, which makes supercontexts an ideal symbolic intermediate representation for all domain-specific static program analyses.

Our approach is more general than existing methods because it can derive solutions for arbitrary (even intra-loop and nested loop) nodes of reducible and irreducible control flow graphs. We prove the correctness of our symbolic analysis method. Our experimental results show that the problem sizes arising from real-world applications such as the SPEC95 benchmark suite are tractable for our symbolic analysis framework.

1. Introduction

Static program analysis is concerned with the design of algorithms that determine the dynamic behavior of programs without executing them. Symbolic analysis is an advanced static program analysis technique. It has been successfully applied to memory leak detection (Scholz et al., 2000), compilation of parallel programs (Fahringer and Scholz, 2003; Haghighat and Polychronopoulos, 1996; van Engelen et al., 2004; Blume and Eigenmann, 1998), detection of superfluous bound checks, variable aliases and task deadlocks (Rugina and Rinard, 2000; Blieberger and Burgstaller, 2003; Blieberger et al., 1999, 2000a), and to worst-case execution time analysis (Blieberger, 2002; Blieberger et al., 2000b). The results gained using symbolic analysis provide invaluable information for optimizing compilers, code generators, program verification, testing and debugging.

Symbolic analysis (Fahringer and Scholz, 2003; Pugh, 1994; Havlak, 1994; Burgstaller et al., 2006b) uses symbolic expressions to describe computations as algebraic formulæ over a program's problem space. Symbolic analysis consists of two steps:

(1) the computation of symbolic expressions that describe all valid variable bindings of a program at a given program point, and
(2) the formulation of a specific static analysis problem in terms of the computed variable bindings.

As an example, consider the statement sequence depicted in Fig. 1. After the declaration of two scalar variables, the read statement in line 2 assigns both variables a new value. The subsequent assignment statements change the values of both variables. Symbolic analysis applies symbolic values for program variables. Assuming that the read statement in line 2 yields the symbolic value $\underline{u}$ for

☆ A preliminary version of this work, excluding correctness proofs, treatment of nested loops, loop bounds and loop exit conditions, path expression generation and symbolic integer division has been presented at the JMLC'06 conference (Burgstaller et al., 2006b).

* Corresponding author. Tel.: +82 2 2123 5728; fax: +82 2 365 2579.
E-mail addresses: bburg@cs.yonsei.ac.kr (B. Burgstaller), scholz@it.usyd.edu.au (B. Scholz), blieb@auto.tuwien.ac.at (J. Blieberger).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.11.1039

		val_u	val_v
1	integer::u,v;		
2	read (u,v);	$\underline{u}$	$\underline{v}$
3	u := u + v;	$\underline{u}+\underline{v}$	$\underline{v}$
4	v := u - v;	$\underline{u}+\underline{v}$	$\underline{u}$
5	u := u - v;	$\underline{v}$	$\underline{u}$

Fig. 1. Statement sequence.

variable u, and $\underline{v}$ for variable v, then a simple sequence of forward substitutions and simplifications computes the symbolic values depicted in the table at the right of Fig. 1. Each row in the table denotes the symbolic values val_u and val_v of variables u and v after execution of the corresponding statement. These symbolic values describe the variable bindings that are valid at the corresponding program points.

Comparing the variable bindings depicted with line 2 and line 5, it is clear that the values of the variables u and v are swapped in the example in Fig. 1. Due to the symbolic nature of the analysis this is true irrespective of the concrete input values for u and v. Based on the computed variable bindings an optimizing compiler can derive that the expression u−v in line 4 of the example program will always yield $\underline{u}$, which makes an overflow check of this expression redundant. (Because variables u and v are of the same type.)

The above example reflects the clear-cut division of symbolic analysis into (1) the computation of valid variable bindings, and (2) the formulation of the specific analysis problem under consideration (e.g., range check elimination) in terms of those variable bindings.

In this paper we propose a generic symbolic analysis framework that automates step (1) above. The need for such a generic symbolic analysis framework stems from the observations

- that step (1) is a prerequisite common to all static analysis problems to be solved by symbolic analysis, and
- that existing approaches to this problem are of limited applicability (i.e., they cannot compute a solution for program points within loops, they are not applicable to irreducible control flow graphs, and they are often tailored to a specific application).

Our generic symbolic analysis framework extends the applicability of existing symbolic analyses to a larger class of programs. It allows the application of symbolic analysis to other static analysis problems.

Our symbolic analysis framework accurately models the semantics of imperative programming languages. We introduce a new representation of symbolic analysis information called *supercontext*, which is a comprehensive and compact algebraic structure describing the complete control and data flow analysis information valid at a given program point.

We encode the side-effect of a single statement's computation as a function from supercontexts to supercontexts. We then extend this functional description from single statements to program paths and sets of program paths. By doing so, we gain a functional description of the input program in the symbolic domain.

With our approach the control flow information of the input program is modeled by means of path expressions first introduced by Tarjan (1981). A path expression is a regular expression whose language is the set of paths emanating from the start node of a control flow graph to a given node. We provide a natural homomorphism that maps the regular expressions representing path sets into the symbolic domain. We define these mappings by reinterpreting the regular expression operators $\cdot$, +, and *. The technical part of our work shows that these mappings are indeed homomorphisms and that the symbolic functional representation is correct.

With our approach we represent the infinitely many program paths arising due to a loop by means of a closure context, which is an extension of a program context (cf. Fahringer and Scholz, 2003) that incorporates symbolic recurrence systems. In this way a supercontext consists of a finite number of closure contexts. Symbolic analysis at this stage reduces to the application of the functional representation of the input program to a closure context representing the initial execution environment.

The *contribution* of our paper is as follows: Our approach is the first to prove the semantic correctness of symbolic analysis with respect to the underlying standard semantics. Second, we show the correctness of the meet over all paths solution and the modeling of loops as symbolic recurrence systems. Third, our approach does not restrict symbolic analysis to reducible flowgraphs, and it can derive solutions for arbitrary graph nodes (even within nested loops). Fourth, our approach is purely algebra-based and fully automated. It closes the gap between static program analysis and computer algebra systems, which makes supercontexts an ideal symbolic intermediate representation for all domain-specific static program analyses. Fifth, the feasibility of our approach was proven by conducting experiments with the SPEC95 benchmark suite. A high portion (i.e., 94%) of the functions in SPEC95 has less than 10^5 closure contexts to analyze, with the majority of those 94% involving even fewer than 4000 closure contexts.

The paper is organized as follows: In Section 2 we outline notations and background material. In Section 3 we define syntax and semantics of a flow language that we use to develop our symbolic analysis methodology. In Section 4 we introduce the symbolic analysis domain and the notion of symbolic execution along program paths. Section 5 describes the main contribution of this paper, namely the mapping to the symbolic domain through path expressions. In Section 6 we discuss our prototype implementation and the experimental results from the SPEC95 benchmark suite. Section 7 surveys related work. Finally, in Section 8 we draw our conclusions and outline future work. The proofs of the theorems stated in the paper are given in the appendix.

2. Background and notation

We use $\mathbb{N}$ to denote the natural numbers, $\mathbb{Z}$ to denote the integers, and $\mathbb{B} = \{true, false\}$ to denote the truth values from Boolean algebra. The finite set of program variables is denoted by $\mathbb{V}$. *Dom* denotes the domain of a function. 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 $E \subseteq N \times N$. Each edge $e \in E$ has a *head* $h(e) \in N$ and a *tail* $t(e) \in N$. The set of incoming edges for a given node $n \in N$ is defined as $in(n) = \{e \in E : t(e) = n\}$. Likewise we define the set of outgoing edges for a node $n \in N$ as $out(n) = \{e \in E : h(e) = n\}$. *Entry* ($n_e$) and *Exit* ($n_x$) are distinguished CFG nodes used to denote the start and terminal node. The start node n_e has no incoming edges ($in(n_e) = \varnothing$), whereas the terminal node n_x has no outgoing edges ($out(n_x) = \varnothing$). We require that every node n is contained in a program path from n_e to n_x, where a *program path* $\pi = \langle e_1, e_2, \ldots, e_k \rangle$ is a sequence of edges such that $t(e_r) = h(e_{r+1})$ for $1 \le r \le k-1$.

Tarjan (1981) showed how program paths can be represented as regular expressions: Let Σ be a finite alphabet disjoint from $\{\Lambda, \varnothing, (,)\}$. A *regular expression* is any expression built by applying the following rules:

(1a) "Λ" and "$\varnothing$" 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.

pred	:	Predicate	assign	:	Assignment
pred	::=	true | false | not pred | pred or pred	assign	::=	id := exp
	|	pred and pred | exp rel-op exp			

Fig. 2. Syntactic domain of the Flow language.

In a regular expression, Λ denotes the empty string, $\varnothing$ denotes the empty set, + denotes union, $\cdot$ denotes concatenation, and * denotes reflexive, transitive closure under concatenation. We use $L(R)$ to denote the set of strings defined by the regular expression R over Σ. A regular expression R is *simple* if $R=\varnothing$ or R does not contain $\varnothing$ as a subexpression. 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. Standard algorithms such as Gaussian elimination can be applied to compute path expressions from a CFG (cf., e.g., Hopcroft and Ullman, 1979; Tarjan, 1981).

The following notational convention is used throughout the paper: to distinguish between corresponding entities from the standard semantic and symbolic domain, we subscript the first with the letter c and the latter with the letter s.

3. Standard semantic program execution

An environment *env* of our Flow language maps a program variable $v \in \mathbb{V}$ to its value $z \in \mathbb{Z}$. The set of possible environments can be represented by a function class $Env \subseteq \{env : \mathbb{V} \to \mathbb{Z}\}$. Functions $pred_c : E \to (Env \to \mathbb{B})$ and $\sigma_c : E \to (Env \to Env)$ associate with each edge $e \in E$ a branch-predicate and a side-effect. The syntax of Flow branch predicates and side-effects is depicted in Fig. 2.

The valuation functions $\text{pred}_c : \text{Predicate} \to (Env \to \mathbb{B})$ and $\text{assign}_c : \text{Assignment} \to (Env \to Env)$ map predicates and side-effects to the semantic domain; due to space considerations we refer to Burgstaller (2005, Section 3) for their definitions.

Control progresses from $h(e)$ to $t(e)$ iff $pred_c(e)(env) = true$, which means that the predicate associated with edge e evaluates to *true* within environment *env*. We require that for every node $n \neq n_x$ and environment *env* the branch-predicate of exactly one outgoing edge evaluates to true.

The transition function δ is of arity $(N \times Env) \to (N \times Env)$. Execution of a transition $(n, env) \to (n', env')$ via an edge e is defined as

$$\begin{aligned}(n,\ & env) \to (n', env') : \\ & \forall n \forall n' \forall env \forall env' \forall e : \\ & \quad (e \in \text{out}(n) \wedge t(e) = n' \wedge pred_c(e)(env)) \\ & \quad \Rightarrow env' = \sigma_c(e)(env),\end{aligned} \tag{1}$$

where $\Rightarrow$ denotes *implication*. The iterated transition function $\delta^* : (N \times Env) \to (N \times Env)$ is a partial function defined by $\delta^*(n_x, env) = (n_x, env)$ and $\delta^*(n, env) = \delta^*(\delta(n, env))$. For any graph $G=\langle N, E, n_e, n_x\rangle$ the environment env_x of the terminal node n_x represents the result of standard semantic program execution along the transition sequence $(n_e, env_e) \xrightarrow{*} (n_x, env_x)$. 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.

4. Semantics of symbolic program execution

The representation of variable values constitutes the main difference between standard semantics and symbolic semantics. Whereas with standard semantics the value of a variable is described by a concrete value $z \in \mathbb{Z}$, symbolic semantics employs symbolic expressions. The relation between standard semantics and semantics of symbolic program execution is depicted in (2).

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 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 $In, Out \in Env$. Given the standard semantics of Flow programs from Section 3, the computation of $S_{\text{con}}[\![P]\!](In)$ is equivalent to an application of the iterated transition function δ^* to start node n_e and environment *In*.

$$\begin{array}{ccc} In & \xrightarrow{\text{sym}} & In_s \\ {\scriptstyle S_{\text{con}}[\![P]\!]}\downarrow & \searrow & \downarrow{\scriptstyle S_{\text{sym}}[\![P]\!]} \\ Out & \xleftarrow{\text{con}} & Out_s \end{array} \tag{2}$$

Similarly we derive the semantics of symbolic program execution for 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 Env_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.

The diagram in (2) contains two additional functions, sym and con, needed to relate input and output of the functional descriptions $S_{\text{con}}[\![P]\!]$ and $S_{\text{sym}}[\![P]\!]$. The purpose of function sym is the transfer of a concrete environment to the symbolic domain, whereas function con *instantiates* (i.e., evaluates) a symbolic environment with the values of a concrete environment. For the definitions of functions sym and con we refer to Appendix B. The commutation of concrete and symbolic execution depicted in (2) can then be formalized as

$$S_{\text{con}}[\![P]\!](In) = \text{con}(In,\ S_{\text{sym}}[\![P]\!](\text{sym}(In))), \tag{3}$$

which means that the result of the symbolically executed program $S_{\text{sym}}[\![P]\!]$ over input $In_s = \text{sym}(In)$ and instantiated by *In* must be the same as the result from standard semantic program execution $S_{\text{con}}[\![P]\!](In)$.

4.1. The domain for symbolic program execution

To be able to distinguish between a variable and its *initial value*, we introduce the set $\underline{\mathbb{V}}$ of initial value variables. This set is isomorphic to $\mathbb{V}$. Its purpose is to represent 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 $\underline{\mathbb{V}}$. As a shorthand notation we write $\underline{v}$ for the application of the initial value operator to variable v.

The *standard semantic* model of the Flow language is based on *integer arithmetics*. Transferring this property to the symbolic domain requires symbolic expressions to be *integer-valued* as well.

Given the operations of addition and multiplication it follows that the multivariate polynomials from the ring $\mathbb{Z}[\mathbf{x}]$, with indeterminates $\mathbf{x} = (x_1, \ldots, x_v) \in \underline{\mathbb{V}}^v$, are integer-valued expressions.

To support division, the ring $\mathbb{Z}[\mathbf{x}]$ is extended to the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ (cf. Geddes et al., 1992). By means of the rounding operation Rnd we can "wrap" a rational function x/y to obtain an integer-valued expression $\text{Rnd}(x/y)$. Rounding involves floor and ceiling functions, which makes expressions resulting from symbolic division inflexible: we can move integer terms in and out of floor and ceiling functions, but little more (Graham et al., 1994). In Appendix A we therefore present simplifications for special cases of symbolic integer divisions. In the general case, we model integer division of two symbolic expressions x and y, $y \neq 0$, as $x \,\text{div}_s\, y = \text{Rnd}(x/y)$, where the symbolic division operator div_s

denotes the counterpart of the integer division operator div of the Flow standard semantics.

Let $f^{(n)} \in \{+^{(2)}, -^{(2)}, -^{(1)}, .^{(2)}\}$ denote functions corresponding to the Flow arithmetic operations, where (n) denotes the respective arity. They 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.

Definition 1. The set of *integer-valued symbolic expressions* of the domain SymExpr is inductively defined by

- $\mathbb{Z}[\mathbf{x}] \subset$ SymExpr
- for all $f^{(n)}$ and all $e_1, \ldots, e_n \in$ SymExpr, we have $f^{(n)}(e_1, \ldots, e_n) \in$ SymExpr (i.e., application of functions $f^{(n)}$ to symbolic expressions yields symbolic expressions),
- for all $e_1, e_2 \in$ SymExpr, we have $e_1/e_2 \in$ SymExpr, iff e_1/e_2 is an integer-valued symbolic expression,
- for all $e_1, e_2 \in$ SymExpr, we have $\mathrm{Rnd}(e_1/e_2) \in$ SymExpr.

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., $\leq$: SymExpr × SymExpr → 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.

Definition 2. The set of *symbolic predicates* of SymPred, the symbolic predicate domain, is inductively defined as

- $\mathbb{B} \subset$ SymPred (i.e., *true* and *false* are symbolic predicates),
- for all f and all $e_1, e_2 \in$ SymExpr, we have $f(e_1, e_2) \in$ SymPred (i.e., application of relational connectives to symbolic expressions yields symbolic predicates),
- for all logical connectives l and all symbolic predicates $e_1, \ldots e_n \in$ SymPred, we have $l(e_1, \ldots, e_n) \in$ SymPred (i.e., application of logical connectives to symbolic predicates yields symbolic predicates).

The domain SymPred constitutes a Boolean algebra, as stated by Burgstaller (2005, Section 4.1).

Definition 3. A *state* $s \in S$ is a function that maps a program variable to the corresponding symbolic expression. The set of possible states can be represented by a function class $S \subseteq \{f : \mathbb{V} \to \text{SymExpr}\}$. A *clean slate* state s maps all variables in its domain to the corresponding initial value variables: $\forall v \in \mathcal{D}om(s) : s(v) = \underline{v}$. Note that if we restrict our interest to a subset of $\mathbb{V}$ then states are *partial* functions.

Definition 4. A *context* $c \in C \subseteq [S \times \text{SymPred}]$ is defined by an ordered tuple $[s, p]$ where s denotes a state, and pathcondition $p \in$ SymPred describes the condition for which the variable bindings specified through s hold (cf. Blieberger, 2002; Fahringer and Scholz, 2003). We make use of the functions $\mathrm{pc} : C \to$ SymPred and $\mathrm{st} : C \to S$ to access a context's pathcondition and state. A *clean slate* context consists of a clean slate state and a *true* pathcondition.

Standard semantic and symbolic side-effects and branch-predicates share the syntactic domain depicted in Fig. 2. Due to space considerations we refer to Burgstaller (2005, Section 4.2) for an exhaustive description of the valuation functions into the symbolic domain that are introduced in brief below. Eq. (4) defines valuation function assign_s which maps the derivation tree of an assignment statement to the corresponding side-effect in the symbolic domain. This side-effect is a function that transforms its argument context $[s, p]$ by updating the state s with a new symbolic expression at $\mathrm{id}[\![\mathrm{id}]\!]$.

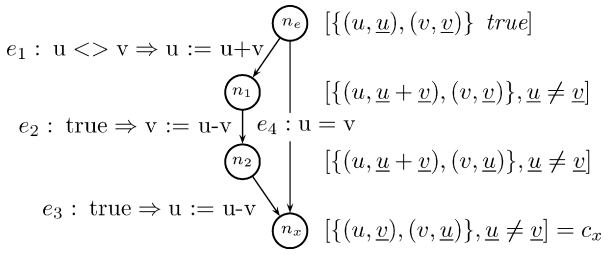


Fig. 3. Symbolic execution along path $\pi_1 = \langle e_1, e_2, e_3 \rangle$.

$$\begin{aligned} &\mathrm{assign}_s : \text{Assignment} \to (C \to C) \\ &\mathrm{assign}_s[[\mathrm{id} := \mathrm{exp}]](c) = \\ &\quad \lambda[s,p].[s[\mathrm{id}[[\mathrm{id}]] \mapsto \mathrm{exp}_s[[\mathrm{exp}]](s)], p](c) \end{aligned} \tag{4}$$

Branch-predicates are treated according to (5). The valuation function pred_s maps the derivation tree t of a branch-predicate to a function $f : C \to C$. Application of f to the argument-context $[s, p]$ results in a context $[s, p \wedge p']$, where $p' \in$ SymPred is a symbolic predicate corresponding to tree t.

$$\begin{aligned} &\mathrm{pred}_s : \text{Predicate} \to (C \to C) \\ &\ldots \\ &\mathrm{pred}_s[[\mathrm{pred}_1 \text{ and } \mathrm{pred}_2]](c) = \\ &\quad \lambda[s,p].[s, p \wedge (\ \mathrm{pc}\ (\mathrm{pred}_s[[\mathrm{pred}_1]]([s, true])) \\ &\qquad\qquad \wedge\ \mathrm{pc}\ (\mathrm{pred}_s[[\mathrm{pred}_2]]([s, true]))\ )](c) \end{aligned} \tag{5}$$

4.2. Single-edge symbolic execution

We express the effect of a computational step associated with a single edge e by a member of the function class $F_s \subseteq \{f : C \to C\}$. F_s contains the identity function ι which can be envisioned as a `null`-statement without any computational effect. We require F_s to be closed under composition, which allows us to compose the computational steps of edges along program paths.

An edge transition function $M_s : E \to F_s$ assigns a function $f \in F_s$ to each edge $e \in E$ of the CFG. The valuation function $\mathrm{edge}_s[\![\ldots]\!]$ maps syntactic constructs associated with CFG edges to the respective valuation functions for branch predicates and side-effects, which allows us to specify functions $f \in F_s$ as follows.

$$\begin{aligned} f &= M_s(e)(c) = \sigma_s(e) \circ \mathit{pred}_s(e)(c) = \\ &= \mathrm{edge}_s[[\mathrm{e} : \mathrm{pred} \Rightarrow \mathrm{assign}]](c) = \\ &= \mathrm{assign}_s[[\mathrm{assign}]](\mathrm{pred}_s[[\mathrm{pred}]](c)) \end{aligned} \tag{6}$$

It follows immediately from our denotational definitions of side-effects and branch predicates that functions specified in the above way fulfill the properties required for function class F_s.

Fig. 3 depicts our running example for which we determine the transition function f for edge e_1. Applying (6) and the valuation functions for branch predicates and side-effects, we get

$$\begin{aligned} f &= M_s(e_1)(c) = \mathrm{edge}_s[[\mathrm{e}_1 : \mathrm{u} <> \mathrm{v} \Rightarrow \mathrm{u} := \mathrm{u} + \mathrm{v}]](c) = \\ &= \mathrm{assign}_s[[\mathrm{u} := \mathrm{u} + \mathrm{v}]](\mathrm{pred}_s[[\mathrm{u} <> \mathrm{v}]](c)) = \\ &= \lambda[s,p].[s[u \mapsto s(u) + s(v)], p \wedge s(u) \neq s(v)](c). \end{aligned}$$

4.3. Single-path symbolic execution

For a forward data-flow problem we can extend the transition function M_s from edges e to program paths π as follows.

$$M_s(\pi) = \begin{cases} \iota, \text{ if } \pi \text{ is the empty path} \\ M_s(e_k) \circ \ldots \circ M_s(e_1), \text{ if } \pi = \langle e_1, \ldots, e_k \rangle \end{cases} \tag{7}$$

As a shorthand notation we may also use f_e for $M_s(e)$ and f_π for $M_s(\pi)$. Clearly if the computational effect of a single statement of a Flow program is described by a function $f \in F_s$, the computational effect of program execution along a path π is defined by $M_s(\pi)(c_e)$, where c_e denotes the initial context on entry to π. (*Proof by induction on the length of π omitted.*)

In the previous example we determined the result of the transition function $M_s(e_1)$ which represents the effect of symbolic program execution along edge e_1 of our running example. After evaluation of all edge transition functions along the program path $\pi_1 = \langle e_1, e_2, e_3 \rangle$ we use function $M_s(\pi_1)$ to calculate the effect of symbolic execution along path π_1. We assume that the initial context c_e passed as argument to $M_s(\pi_1)$ contains two program variables u and v holding their initial values $\underline{u}$ and $\underline{v}$. Then the contexts depicted in Fig. 3 illustrate the transformation of the initial context c_e during symbolic execution along program path π_1.[1] The context shown with node n_x represents the result for $M_s(\pi_1)(c_e)$.

4.4. Multi-path symbolic execution

In the preceding example we have omitted symbolic execution along edge e_4. As long as we cannot decide that this path is infeasible, we have to analyze it for our symbolic solution to be complete. Symbolic execution along edge e_4 yields a further program context

$$c'_x = [\{(u, \underline{u}), (v, \underline{v})\}, \underline{u} = \underline{v}].$$

As can be seen from this example, the description of the symbolic solution in terms of contexts increases with the number of program paths through a CFG; each program path from the entry node n_e to a given node n contributes one context to the symbolic solution at node n. As long as CFGs are acyclic, the number of 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 contexts of the symbolic solution at node n, becomes infinite. In order to describe the joint effects of execution along several program paths, we introduce a structure that allows us to aggregate contexts.

Definition 5. A *supercontext* $sc \in SC$ is a collection of contexts $c \in C$ and can be envisioned as a (possibly) infinite set

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

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, arriving at the following notation for supercontexts.

$$sc \in SC = \left[\bigcup_{k=0}^{\infty} [s_k, p_k]\right] \tag{8}$$

[1] As a notational convention we depict the *graphs* of the contained states instead of the states themselves.

Note that supercontexts correspond to the notion of *symbolic environments* used in the introduction of this section.

Because a supercontext consists of an arbitrary (even infinite) number of contexts, it can represent the result of symbolic execution along an arbitrary (even infinite) number of program paths. According to Hecht et al. (1977) the *meet over all paths (MOP) solution* for a given CFG node n is the maximum information, relevant to the problem at hand, which can be derived from every possible execution path from the entry node n_e to n. The MOP-solution of symbolic execution for a given node n can then be written as

$$\text{mop}(n) = \bigcup_{\pi \in \text{Path}(n_e, n)} M_s(\pi)(c_e), \tag{9}$$

with $\text{Path}(n_e, n)$ denoting the set of all program paths from node n_e to node n, $\cup$ denoting supercontext union, and c_e denoting the initial argument context. A correctness proof for the symbolic MOP-solution is given in Appendix B.

5. Symbolic evaluation

The symbolic execution approach of Section 4 is capable of computing the MOP-solution for arbitrary CFG nodes. It is however not constructive in the sense that we have not specified a method to obtain the set of program paths needed by this approach. Furthermore, the MOP-solution is infinite. In this section we define a method to compute the MOP-solution that is both constructive and finite. It is based on the regular expression algebra of Section 2, which we use to model the program paths of a given CFG. The structure of regular expressions imposes a horizontal functional decomposition of the CFG in contrast to the approach of the previous section in which our functional decomposition was vertically along whole program paths. As a consequence we have to extend domain and codomain of the function class F_s introduced in Section 4.2 from contexts to supercontexts, yielding a new function class F_{sc}:

$$F_{sc} \subseteq \{f_{sc} : SC \to SC\}. \tag{10}$$

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

$$\text{wrap} : (C \to C) \to (SC \to SC)$$
$$\text{wrap}(f_s)(sc) ::= f_{sc}\left(\left[\bigcup_{i=0}^{\infty} [s_i, p_i]\right]\right) = \left[\bigcup_{i=0}^{\infty} f_s([s_i, p_i])\right].$$

The function class F_{sc} has the following properties, which are easily verified from the definition of the wrapping operator, the properties of supercontexts from Definition 5, and the properties of the function class F_s on which F_{sc} is based.

F1) F_{sc} contains the identity function ι.
F2) F_{sc} is closed under $\cup$: $\forall f, g \in F_{sc} : (f \cup g)(x) = f(x) \cup g(x)$.
F3) F_{sc} is closed under composition: $\forall f, g \in F_{sc} : f \circ g \in F_{sc}$.
F4) F_{sc} is closed under iterated composition (with $f^0 = \iota$ and $f^i = f^{i-1} \circ f$):

$$f^*(x) = \left[\bigcup_{i \geq 0} f^i(x)\right]. \tag{11}$$

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

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

Based on the edge transition function M_s from Section 4.2 we define a new edge transition function M_{sc} that encapsulates the wrapping operator inside:

$$M_{sc} : E \to F_{sc}$$
$$M_{sc}(e) ::= \text{wrap}(M_s(e)). \tag{12}$$

We can compose edge transition functions from function class F_{sc} along program paths in the same way shown for function class F_s in (7). In a similar way we use the shorthand notation f_e for $M_{sc}(e)$, and f_π for $M_{sc}(\pi)$.

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

$$\phi(\Lambda) = \iota, \tag{13}$$

$$\phi(e) = M_{sc}(e) = f_e, \tag{14}$$

$$\phi(P_1 + P_2) = \phi(P_1) \cup \phi(P_2), \tag{15}$$

$$\phi(P_1 \cdot P_2) = \phi(P_2) \circ \phi(P_1), \tag{16}$$

$$\phi(P_1^*) = \phi(P_1)^*. \tag{17}$$

Lemma 1. *Let $P \neq \varnothing$ be a path expression of type (v, w). Then for all $x \in SC$,*

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

Proof in Appendix B. Based on Lemma 1 we establish that the mapping ϕ is a homomorphism from the regular expression algebra to the function class F_{sc} of (10), and that the computed solution corresponds to the MOP-solution for symbolic execution from (9).

Theorem 1. *For any node n let $P(n_e, n)$ be a path expression representing all paths from n_e to n. Then $\text{mop}(n) = \phi(P(n_e, n))(c_e)$, where c_e denotes the initial context[2] valid at entry node n_e.*

Proof in Appendix B. It should be noted that Theorem 1 does not impose a restriction on path expression $P(n_e, n)$. As a consequence, Theorem 1 holds for path expressions corresponding to CFGs with *irreducible* graph portions. Furthermore it holds for arbitrary graph nodes, even within loops and nested loops.

5.1. Finite supercontexts

It has been pointed out in Section 4.4 that the MOP solution becomes infinite with the introduction of CFG cycles. CFG cycles induce * operators in path expressions; due to the iterated composition that is implied by the right-hand side of (17), each * operator induces an infinite number of contexts in the resulting supercontext.

In changing the mapping ϕ by replacing (17) with

$$\phi(P_1^*) = \phi(P_1)^{\circledast}, \tag{18}$$

we introduce a new operation $\circledast$ which replaces the iterated composition operation from (11) by a composition operation that generates a finite representation for the result of symbolic evaluation of the CFG cycle corresponding to path expression P_1. This finite representation is an extension of a context by a system of symbolic recurrences (Lueker, 1980) and is called a *closure context*. As will be pointed out below, a system of symbolic recurrences makes a closure context an *exact* representation of the infinite set of contexts that is due to a CFG cycle. In this way (18) changes our representation of a supercontext from an *infinite set* of *contexts* to a *finite set* of *closure contexts*. The purpose of this change is to have a compact representation of supercontexts that facilitates domain-specific static program analyses and that can be implemented with CASs.

The remainder of this section is devoted to the definition of closure contexts and the $\circledast$ operation.

In analogy to the set $\mathbb{V}$ of program variables we define the set $\mathbb{L}$, $\mathbb{V} \cap \mathbb{L} = \varnothing$, of *loop index variables*. We use lowercase letters, e.g., l, m, n, to denote elements from $\mathbb{L}$. Conceptually a loop index variable can be envisioned as an artificial program variable that is assigned the value 0 upon entry of the loop body. After each iteration of the loop body, its value is increased by one.

Fig. 4 depicts a Flow example loop together with a textual representation where the loop index variable has been made explicit (cf. line 2 and 6). Associated with a loop index variable l is a *symbolic* upper bound, denoted by l_ω. This symbolic upper bound represents the number of loop iterations.[3] Specifically, an upper bound of $l_\omega = 0$ implies zero loop iterations, as can be inferred from Fig. 4.[4] Endless loops can be modeled by defining $l_\omega = +\infty$.

Definition 6. The set of symbolic expressions from Definition 1 is extended by

- $\mathbb{L} \subset \text{SymExpr}$ (i.e., loop index variables are symbolic expressions), and
- for all $v_i \in \mathbb{V}$, and $l \in \mathbb{L}$, $v_i(0) \in \text{SymExpr}$, $v_i(l) \in \text{SymExpr}$, $v_i(l+1) \in \text{SymExpr}$, and $v_i(l-1) \in \text{SymExpr}$ (i.e., dereferencing the value of a program variable to specify a recurrence relation yields a symbolic expression).

Note that loop index variables will be used to specify recurrence relations for the induction variables of a loop. E.g., induction variable v_i will be replaced by $v_i(l)$, which refers to the corresponding recurrence relation. For this reason, we need symbolic expressions $v_i(0)$, $v_i(l)$, $v_i(l+1)$ and $v_i(l-1)$ to refer to the first, current, next, and previous instance of a recurrence. A loop index variable is introduced with every cycle in the CFG, i.e., with (18) of the path expression homomorphism.

Definition 7. A *range expression* is a symbolic expression of the form $0 \le l \le l_\omega$, with loop index variable $l \in \mathbb{L}$, and l_ω being the symbolic upper bound of l. We extend the set of symbolic predicates of the domain SymPred (cf. Definition 2) by the following rule to include range expressions:

for all $l \in \mathbb{L}$, $0 \le l \le l_\omega \subset \text{SymPred}$ (i.e., range expressions constitute symbolic predicates).

We denote a *recurrence system* over loop index variable l by $rs(l)$. We can construct a *recurrence system set* r of k recurrence systems by

$$r ::= \bigcup_{1 \le j \le k} rs(l_j).$$

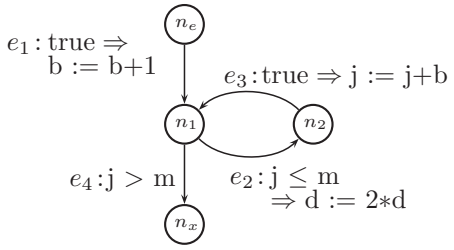
Recurrence system sets can be nested, and the set of all recurrence system sets is denoted by R. For our purpose it is furthermore beneficial to impose a total order $\le$ on the elements of a recurrence system set in order to obtain the semantics of a list.

Definition 8. A closure context $\bar{c}$ is an element of the set $\bar{C} = S \times \text{SymPred} \times R$, denoted by $[s, p, r]$. For a clean slate closure con-

[2] *Since program contexts are one-element supercontexts, c_e is a valid argument for functions from class F_{sc}.*

[3] Computing a *symbolic* upper bound for the number of loop iterations is beyond the scope of this paper. It is discussed, among others, by Fahringer and Scholz (2003) and Blieberger (1994).

[4] This contrasts the notion of range expressions in contemporary programming languages, where `range L..U` denotes the interval [L, U].




```
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;
```

Fig. 4. Example loop: implicit vs. explicit loop index variable.

text the state s is a clean slate state, p is a *true* pathcondition, and r is the empty set. A context can be considered a special case of a closure context with $r=\varnothing$. 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 9. We define operation $\circledast$ of (18) in terms of the input/output-behavior of the function resulting from the application of operation $\circledast$ to $\phi(P_1)$, that is, $\phi(P_1)^{\circledast}$. Let $f=\phi(P_1)$ be a functional description of the accumulated side-effect of one iteration of the loop body represented by the path expression P_1. For a given closure context $\overline{c_{in}}=[s_{in},p_{in},r_{in}]$ we define the properties of the closure context $\overline{c_{out}}=[s_{out},p_{out},r_{out}]$ resulting from the application of $f^{\circledast}$ to $\overline{c_{in}}$, that is,

$$\overline{c_{out}}=\phi(P_1^*)(\overline{c_{in}})=\phi(P_1)^{\circledast}(\overline{c_{in}})=f^{\circledast}(\overline{c_{in}}). \tag{19}$$

One iteration of the loop body determines the recurrence system that is due to the induction variables of the loop body. Hence we start with a clean slate closure context $\overline{c_0}=[s_0,p_0,r_0]$ and compute the result of symbolic evaluation of one iteration of the loop body, denoted by $\overline{c_1}$.

$$\overline{c_1}=[s_1,p_1,r_1]=f(\overline{c_0}). \tag{20}$$

A substitution $\sigma_{s,e}$ for a given state s and an expression $e\in$ SymExpr is defined as

$$\sigma_{s,e}=\{\underline{v_1}\mapsto v_1(e),\ldots,\underline{v_j}\mapsto v_j(e)\},\ \text{with}\ \underset{1\le i\le j}{v_i}\in\mathcal{D}om(s). \tag{21}$$

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 r_{out}.

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.

$$\forall v_i\in\mathcal{D}om(s_{in}):s_{out}::=s_{in}[v_i\mapsto v_i(\mathsf{l})] \tag{22}$$

Hence we get $\text{graph}(s_{out})=\{(v_1,v_1(\mathsf{l})),\ldots,(v_n,v_n(\mathsf{l}))\}$.

Pathcondition: The pathcondition p_{out} of closure context $\overline{c_{out}}$ has the form

$$p_{in}\wedge(0\le\mathsf{l}\le\mathsf{l}_\omega)\wedge\bigwedge_{1\le\mathsf{l}'\le\mathsf{l}}p(\mathsf{l}'-1). \tag{23}$$

Therein the term p_{in} constitutes the pathcondition of closure context $\overline{c_{in}}$. The second term is a range expression according to Definition 7. 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. It is a conjunction of l instances of the pathcondition p_1 from (20), where the l'^{th} instance corresponds to $\sigma_{s_{in},(\mathsf{l}'-1)}(p_1)$.

An example will illustrate this. Assume the pathcondition $p_1=j\le m$ from Fig. 4. After $\mathsf{l}>0$ iterations the third term in the above equation will read

$$\begin{aligned}&j(0)\le m(0)\wedge j(1)\le m(1)\wedge\cdots\wedge j(\mathsf{l}-1)\le m(\mathsf{l}-1)\\&=\bigwedge_{1\le\mathsf{l}'\le\mathsf{l}}(j(\mathsf{l}'-1)\le m(\mathsf{l}'-1)).\end{aligned}$$

Recurrence system: Let IV denote the set of induction variables of the loop under consideration. We set up a recurrence system over the loop index variable l, from which we construct a recurrence system set r as follows.

$$r=\left\{\begin{bmatrix}\forall v_i\in IV:\begin{cases}v_i(0)::=s_{in}(v_i) & (1)\\ v_i(\mathsf{l}+1)::=\sigma_{s_{in},\mathsf{l}}(s_1(v_i))\end{cases}\\ rc::=\sigma_{s_{in},\mathsf{l}}(p_1) \quad (2)\end{bmatrix}\right\} \tag{24}$$

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 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 a symbolic predicate obtained by replacing the initial value variables in the pathcondition p_1 from (20) by their recursive counterparts.

Having set up the recurrence system set r according to (24), the recurrence system set r_{out} of closure context $\overline{c_{out}}$ is derived from r_{in} by appending r to it.

A recurrence system set can be simplified if we are able to derive closed forms for the recurrence relations of the involved induction variables. There exists a vast body of literature on this topic, e.g., by Greene and Knuth (1982), Lueker (1980), van Engelen et al. (2004), van Engelen (2004), Haghighat and Polychronopoulos (1996) and Gerlek et al. (1995). These methods are directly applicable to the recurrence system sets of our symbolic analysis framework. Modern CASs such as Mathematica (Wolfram, 2003) provide an ideal platform for the implementation of these methods. Note that deriving closed forms for recurrence relations is undecidable in the general case. If a domain-specific program analysis requires a closed form that cannot be derived, the program analysis must resort to an approximation and analysis precision is lost.

For nested loops, the nesting relation is determined by the corresponding path expression. (Note that we do not restrict the structure of path expressions, consequently CFGs with irreducible graph portions (cycles) are covered as well.) The loop evaluation order implied by the mapping ϕ from (18) works from innermost

to outermost loops. By the time a loop is evaluated, its nested loops have already been analyzed. To install a nested loop within the containing loop, we rewrite the boundary conditions of the nested loop's recurrence system such that each initial value variable v_i is replaced by the variable's recurrence relation over the loop index variable of the containing loop. This corresponds to one application of substitution $\sigma_{s,e}$ from (21). For the details of this rewriting step we refer to Burgstaller (2005).

Returning to the example of Fig. 4, we seek the MOP-solution for node n_1, i.e., for path expression $e_1 \cdot (e_2 \cdot e_3)^*$ of type (n_e, n_1). From the clean slate closure context $\overline{c_e} = [s, p, r] = [\{(b, \underline{b}), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\}, true, \varnothing]$, we compute $\phi(e_1 \cdot (e_2 \cdot e_3)^*)(\overline{c_e}) = (f_{e_3} \circ f_{e_2})^{\circledast} \circ f_{e_1}(\overline{c_e})$. Function application $f_{e_1}(\overline{c_e})$ yields the closure context $\overline{c_{in}} = [\{(b, \underline{b}+1), (d, \underline{d}), (j, \underline{j}), (m, \underline{m})\}, true, \varnothing]$, which reduces our computation to $(f_{e_3} \circ f_{e_2})^{\circledast}(\overline{c_{in}})$. To apply operation $\circledast$ we proceed according to Definition 9. Due to (20) we have to compute the result of symbolic evaluation of one iteration of the loop body to derive the underlying recurrence relations. For this we can reuse the clean slate closure context $\overline{c_e}$ by defining $\overline{c_0} ::= \overline{c_e}$ and proceed with the calculation of $\overline{c_1} = (f_{e_3} \circ f_{e_2})(\overline{c_0}) = [\{(b, \underline{b}), (d, 2 \cdot \underline{d}), (j, \underline{j} + \underline{b}), (m, \underline{m})\}, \underline{j} \leq \underline{m}, \varnothing]$. The closure context $\overline{c_{out}}$ resulting from the computation of $\overline{c_{out}} = (f_{e_3} \circ f_{e_2})^{\circledast}(\overline{c_{in}})$ can then be described in terms of its state s_{out}, its pathcondition p_{out}, and its recurrence system set r_{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 induction variables $v_i \in IV = \{d, j\}$ by the value of the recurrence relation for v_i over loop index variable l. Hence we get $s_{out} = \{(b, \underline{b}+1), (d, d(\mathsf{l})), (j, j(\mathsf{l})), (m, \underline{m})\}$.

Pathcondition: According to (23) we get the pathcondition $p_{out} = true \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge_{1 \leq \mathsf{l}' \leq \mathsf{l}} (j(\mathsf{l}'-1) \leq \underline{m})$.

Recurrence system: According to (24) we arrive at the one-element recurrence system set r', with $s_{in} = \mathrm{st}(\overline{c_{in}})$ and $s_1 = \mathrm{st}(\overline{c_1})$ already substituted.

$$r' = \left\{ \left[\begin{array}{ll} \left\{ \begin{array}{l} d(0) ::= \underline{d} \\ d(\mathsf{l}+1) ::= 2 \cdot d(\mathsf{l}) \end{array} \right. & (1a) \\ \left\{ \begin{array}{l} j(0) ::= \underline{j} \\ j(\mathsf{l}+1) ::= j(\mathsf{l}) + \underline{b} + 1 \end{array} \right. & (1b) \\ rc ::= j(\mathsf{l}) \leq \underline{m} & (2) \end{array} \right] \right\}$$

Applying standard methods to solve the recurrence relations for the induction variables d and j, we arrive at

$$r = \left\{ \left[\begin{array}{ll} \left\{ d(\mathsf{l}) ::= 2^{\mathsf{l}} \cdot \underline{d} \right. & (1a) \\ \left\{ j(\mathsf{l}) ::= \underline{j} + \mathsf{l} \cdot (\underline{b}+1) \right. & (1b) \\ rc ::= j(\mathsf{l}) \leq \underline{m} & (2) \end{array} \right] \right\}. \tag{25}$$

Combining state s_{out}, pathcondition p_{out} and the recurrence system set r yields

$$[\{(\quad b, \underline{b}+1), (d, d(\mathsf{l})), (j, j(\mathsf{l})), (m, \underline{m})\}, \quad (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge_{1 \leq \mathsf{l}' \leq \mathsf{l}} (j(\mathsf{l}'-1) \leq \underline{m}), \{r\}] \tag{26}$$

as the solution for the closure context $\overline{c_{out}}$, which is also the MOP-solution for node n_1. The intuitive meaning of this closure context unveils if we consider the range expression $(0 \leq \mathsf{l} \leq \mathsf{l}_\omega)$ that is part of its pathcondition: as loop index variable l ranges from 0 to l_ω, the recurrence system r generates the variable bindings of the respective context[5] c_{l} valid after l loop iterations, i.e., $c_{\mathsf{l}} = (f_{e_3} \circ f_{e_2})^{\mathsf{l}} \circ f_{e_1}(\overline{c_e})$. Hence the closure context $\overline{c_{out}}$ represents a total number of $\mathsf{l}_\omega + 1$ contexts valid at node n_1. Formally the closure context $\overline{c_{out}}$ can be viewed as a predicate $\forall b \forall d \forall j \forall m \forall \mathsf{l} : \overline{c_{out}}$, where the set $\{x \mid 0 \leq x \leq \mathsf{l}_\omega)\} \subseteq \mathbb{N}$ is the universe of discourse for loop index variable l.

The above closure context describes all variable bindings valid at node n_1 of Fig. 4. It yields important information for static program analysis, e.g.,

- at node n_1 the variables b and m assume the values $\underline{b}+1$ and $\underline{m}$, respectively during all loop iterations,
- the induction variables d and j assume monotonically increasing/decreasing sequences of values (depending on the initial values of variables d and b),
- the symbolic values of the induction variables d and j during each iteration of the loop,
- a symbolic upper bound l_ω for the number of loop iterations (computed from the recurrence condition as described by Fahringer and Scholz (2003)), and hence
- symbolic lower and upper bounds for the induction variables d and j.

5.2. Loop bounds and closure context simplifications

It is instructive to consider how a closure context changes once control flow exits the associated loop L. (E.g., how in Fig. 4 the closure context from (26) valid at Node n_1 changes once control moves to the exit node n_x.)

Upon exit of a loop via edge e, the pathcondition p associated with e implies that $\mathsf{l} = \mathsf{l}_\omega$. In other words, the conjunction of p and the pathcondition of a closure context from node $\mathrm{h}(e)$ collapses the set of contexts represented by the resulting closure context to the single context valid after execution of the loop.

Returning to our example in Fig. 4, once we exit the loop via edge e_4, the fact that $\mathsf{l} = \mathsf{l}_\omega$ simplifies the closure context valid at node n_x to

$$[\{\quad (b, \underline{b}+1), (d, d(\mathsf{l}_\omega)), (j, j(\mathsf{l}_\omega)), (m, \underline{m})\}, \quad \bigwedge_{1 \leq \mathsf{l}' \leq \mathsf{l}_\omega} (j(\mathsf{l}'-1) \leq \underline{m}) \wedge j(\mathsf{l}_\omega) > \underline{m}, \{r\}], \tag{27}$$

which represents the single context valid after execution of the loop. It should be noted that the determination of loop exit edges is done based on path expressions, which makes the above simplification a purely mechanical step in our symbolic analysis method.

If we can derive a symbolic upper bound for the number of loop iterations from a loop's recurrence system set, a closure context can be simplified even further. E.g., for the recurrence system set r of (25), we are interested in an integer-valued expression e for l_ω for which the following holds.

$$\begin{aligned} e &= \min\{\mathsf{l} \mid \neg rc\} \\ &= \min\{\mathsf{l} \mid j(\mathsf{l}) > \underline{m}\} \\ &= \min\{\mathsf{l} \mid \underline{j} + \mathsf{l} \cdot (\underline{b}+1) > \underline{m}\} \\ &= \min\{\mathsf{l} \mid \mathsf{l} > \frac{\underline{m} - \underline{j}}{\underline{b}+1}\} \\ &= \left\lceil \frac{\underline{m} - \underline{j} + 1}{\underline{b}+1} \right\rceil \end{aligned} \tag{28}$$

Plugging in the upper bound l_ω into the recurrence system set, we can compute the values $d(\mathsf{l}_\omega)$ and $j(\mathsf{l}_\omega)$ that induction variables d

[5] Not to be mistaken with a closure context.

```
1  while e > 0 loop
2    if e mod 2 = 1 then
3      y := y * f;
4    end if;
5    f := f * f;
6    e := e / 2;
7  end loop;
```

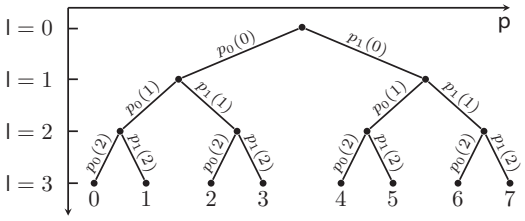


Fig. 5. Example: loop body with two program paths and pathconditions $p_0(\mathsf{l}) = e(\mathsf{l}) > 0 \wedge e(\mathsf{l}) \bmod 2 = 1$ and $p_1(\mathsf{l}) = e(\mathsf{l}) > 0 \wedge e(\mathsf{l}) \bmod 2 \neq 1$. Pathconditions for up to $\mathsf{l} = 3$ iterations are depicted in the tree to the right. Note that each loop iteration is symbolically evaluated in terms of the symbolic values of the previous loop iteration. I.e., the first loop iteration is based on recursion depth 0 of (31), hence pathconditions $p_0(0)$ and $p_1(0)$ at the root of the tree; the second iteration is based on recursion depth 1 aso. $\mathsf{l} = 3$ iterations of the loop yields 8 overall loop-paths. Overall pathconditions are a conjunction of pathconditions of individual loop iterations. E.g., the overall pathcondition for $\mathsf{l} = 3$ iterations with the then-branch taken the initial 2 iterations is $p_0(0) \wedge p_0(1) \wedge p_1(2)$. This execution corresponds to the overall loop-path $\mathfrak{p} = 1$ (overall loop-paths are numbered at the leaves of the tree).

and j assume at the loop exit. Inserting these results in the above closure context, finally gives

$$\left[\{(b, \underline{b}+1), \left(d, \underline{d} \cdot 2^{\left\lceil \frac{\underline{m}-\underline{j}+1}{\underline{b}+1} \right\rceil}\right), \left(j, \underline{j} + (\underline{b}+1) \cdot \left\lceil \frac{\underline{m}-\underline{j}+1}{\underline{b}+1} \right\rceil\right), (m, \underline{m})\}, \bigwedge_{\mathsf{l}'=1}^{\mathsf{l}_\omega} (j(\mathsf{l}'-1) \leq \underline{m}) \wedge \left(\underline{j} + (\underline{b}+1) \cdot \lceil \frac{\underline{m}-\underline{j}+1}{\underline{b}+1} \rceil \right) > \underline{m}, \{r\} \right]. \tag{29}$$

If we are interested only in the accumulated side-effect arising from symbolic execution of the loop, and not in the loop internals, we can discard the recurrence system set r and the associated part of the pathcondition.

5.3. Closure context computation

In this section we show how we compute closure contexts. We pay particular attention to loops where the loop body contains multiple program paths. We discuss an example with a loop with two program paths, and we show how the analysis information captured in closure contexts can be applied with domain-specific program analyses. Conceptually, closure contexts are computed when the operator $f^{\circledast}$ of the path homomorphism is evaluated, see (19).

Symbolic evaluation of a multi-path loop body yields one closure context for each program path through the loop body, i.e.,

$$\overline{sc} = f(\overline{c_0}) = \left[\bigcup_{x=1}^{x_\omega} [s_x, p_x, r_x] \right], \tag{30}$$

where x_ω denotes the number of program paths through the loop body.

A recurrence system for multi-path loops must support conditional recurrence relations. We divide the recurrence system in three parts for (1) boundary values, (2) recurrence relations, and (3) the recurrence condition of the loop.

$$r = \begin{cases} \forall v_i \in IV : v_i(0) ::= s_{in}(v_i) & (1) \\ \forall v_i \in IV; \forall [s_x, p_x, r_x] \in \overline{sc} : & \\ \quad v_i(\mathsf{l}+1) = \sigma_{s_{in},\mathsf{l}}(s_x(v_i)) \ \textbf{if} \ \sigma_{s_{in},\mathsf{l}}(p_x) & (2) \\ rc ::= \bigvee_{1 \leq x \leq x_\omega} \sigma_{s_{in},\mathsf{l}}(p_x) & (3) \end{cases} \tag{31}$$

To compute closure contexts for multi-path loops, the set of symbolic expressions is extended by loop path selectors $\mathfrak{p} \in \mathfrak{P}$, i.e., $\mathfrak{P} \subset \mathrm{SymExpr}$. Given l iterations of a loop, the loop path selector allows us to select one particular path from the x_ω^{l} possible paths. Note that each of those x_ω^{l} paths is a concatenation of l paths through the body of the loop (one path per iteration). If not clear from context, we use the term *overall loop-path* for paths across multiple iterations of a loop.

As an example, consider the loop body with two program paths p_0 and p_1 depicted in Fig. 5. l loop iterations yield 2^{l} overall loop-paths.

The pathcondition from symbolic analysis of a multipath loop body can be stated as

$$p_{in} \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge (0 \leq \mathfrak{p} < x_\omega^{\mathsf{l}}) \wedge \bigwedge_{1 \leq \mathsf{l}' \leq \mathsf{l}} p(\mathsf{l}'-1, \mathfrak{p}). \tag{32}$$

The range of the loop path selector $\mathfrak{p}$ is determined by loop index variable l. The overall loop-path condition is a conjunction of the pathconditions $p(\mathsf{l}'-1, \mathfrak{p})$, $1 \leq \mathsf{l}' \leq \mathsf{l}$, from single loop iterations. Note that the value of the loop path selector $\mathfrak{p}$ determines the overall loop-path under consideration (see the values of loop path selector $\mathfrak{p}$ at the bottom right of Fig. 5). Pathconditions $p(\mathsf{l}'-1, \mathfrak{p})$ are computed as

$$p(\mathsf{l}'-1, \mathfrak{p}) = \sigma_{s_{in},(\mathsf{l}'-1)}(p_\kappa), \tag{33}$$

where index κ of pathcondition p_κ is

$$\kappa = \left\lfloor \left(\mathfrak{p} \cdot x_\omega^{-(\mathsf{l}-\mathsf{l}')} \right) \right\rfloor \bmod \ x_\omega. \tag{34}$$

As an example, consider overall loop-path $\mathfrak{p} = 5$ from Fig. 5. Inserting $\mathfrak{p} = 5$, $x_\omega = 2$, $\mathsf{l} = 3$ and $\mathsf{l}' = \langle 1, 2, 3 \rangle$ in (34) gives $\kappa = \langle 1, 0, 1 \rangle$. Inserting pathconditions p_κ in (33) yields

$$\begin{aligned} p(0, 5) &= \sigma_{s_{in},0}(p_1) = e(0) > 0 \wedge e(0) \bmod 2 \neq 1 \\ p(1, 5) &= \sigma_{s_{in},1}(p_0) = e(1) > 0 \wedge e(1) \bmod 2 = 1 \\ p(2, 5) &= \sigma_{s_{in},2}(p_1) = e(2) > 0 \wedge e(2) \bmod 2 \neq 1, \end{aligned}$$

where $e(\mathsf{l})$ refers to the recurrence system from (31). The overall loop-path for three iterations and loop-path selector $\mathfrak{p} = 5$ is $p(0, 5) \wedge p(1, 5) \wedge p(2, 5)$.

5.3.1. Example

Given two positive numbers x and n, the Ada95 procedure depicted in Fig. 6 computes x^n. Expressions within braces denote computations protected by a run-time bounds-check. E.g., $\{y * f\}^{R1}$ in line 11 denotes bounds-check $R1$ which prevents the result of expression $y * f$ from overflowing the bounds of the Ada95 `positive` type. We will consider bounds-checks in Section 5.3.2.

The CFG of the above exponentiation procedure is depicted in Fig. 7. From the procedure's pathexpression $e_1((e_2 + e_3) \cdot e_4 \cdot e_5)^* \cdot e_6$, we will analyze the subexpression

```
1   Max : constant positive := ??;
2   subtype small_pos is positive range 1..Max;

3   procedure power (x,n : small_pos; y : out positive)
4   is
5       f : positive := x;
6       e : natural := n;
7   begin
8       y := 1;
9       while e > 0 loop
10          if e mod 2 = 1 then
11              y := {y*f}^R1;
12          end if;
13          f := {f*f}^R2;
14          e := e/2;
15      end loop;
16  end power;
```

Fig. 6. Procedure for raising to a power. Expressions encapsulated by $\{\ldots\}$ denote computations requiring a bounds-check at run-time.

corresponding to the loop, i.e., $((e_2+e_3)\cdot e_4\cdot e_5)^*$, to demonstrate the use of closure contexts with multi-path loop bodies. In Section 5.3.2, we will consider how this analysis result can be applied with a program transformation for the removal of redundant bounds-checks in Ada95 code.

Starting with a clean slate closure context $\bar{c}$, symbolic analysis along edge $e1$ gives

$$\overline{c_{in}} = \phi(e_1)(\bar{c}) = [\{(y,1),(e,\underline{e}),(f,\underline{f})\},\ true\ ,\varnothing]. \tag{35}$$

Because of the if-statement (lines 8–10), the loop body contains two program paths, $e_2\cdot e_4\cdot e_5$ and $e_3\cdot e_4\cdot e_5$. Re-using clean slate closure context $\bar{c}$, symbolic analysis of one iteration of the loop body gives

$$\begin{aligned}\overline{sc} &= \phi((e_2+e_3)\cdot e_4\cdot e_5)(\bar{c})\\ &= [\{(y,\underline{y}*\underline{f}),(e,\underline{e}/2),(f,\underline{f}^2)\},\ \underline{e}>0\wedge \underline{e}\bmod 2=1,\varnothing]\\ &\quad\cup[\{(y,\underline{y}),(e,\underline{e}/2),(f,\underline{f}^2)\},\ \underline{e}>0\wedge \underline{e}\bmod 2\neq 1,\varnothing\ ].\end{aligned} \tag{36}$$

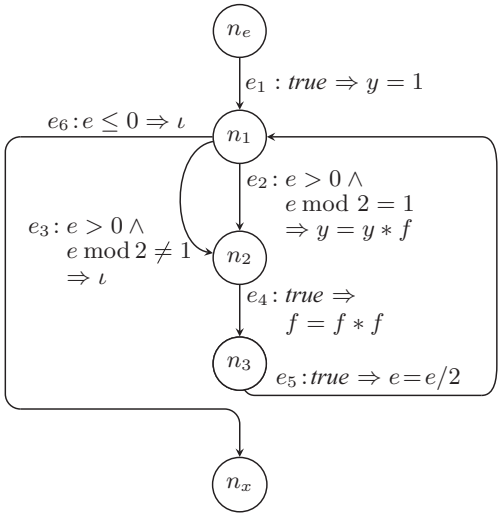


Fig. 7. CFG for procedure "power" from Fig. 6.

We combine closure context $\overline{c_{in}}$ from (35) with the analysis result of the loop body, i.e., (36), to derrive the conditional recurrence relation from (31):

$$r = \begin{cases} y(0) ::= 1 & \\ e(0) ::= \underline{e} & (1)\\ f(0) ::= \underline{f} & \\ y(\mathfrak{l}+1) ::= \begin{cases} f(\mathfrak{l})\cdot y(\mathfrak{l}) & \textbf{if } e(\mathfrak{l}) \bmod 2 = 1\\ y(\mathfrak{l}) & \textbf{else}\end{cases} & \\ e(\mathfrak{l}) ::= \left\lfloor \dfrac{\underline{e}}{2^{\mathfrak{l}}} \right\rfloor & (2)\\ f(\mathfrak{l}) ::= \underline{f}^{2^{\mathfrak{l}}} & \\ rc ::= e(\mathfrak{l}) > 0 & (3)\end{cases} \tag{37}$$

Note that variables e and f are not modified within the if-statement; hence their recurrence relations are unconditional and have been replaced by their closed-form expressions. Combining the above results yields the following closure-context, which is the MOP-solution for node n_1 of Fig. 7.

$$[\{(y,y(\mathfrak{l})),(e,e(\mathfrak{l})),(f,f(\mathfrak{l}))\},\ (0\le \mathfrak{l}\le \mathfrak{l}_\omega)\wedge(0\le \mathfrak{p}<2^{\mathfrak{l}})\wedge \bigwedge_{1\le \mathfrak{l}'\le \mathfrak{l}} p(\mathfrak{l}'-1,\mathfrak{p}),\{r\}] \tag{38}$$

5.3.2. Domain-specific analysis example: removal of redundant bounds-checks

It is the purpose of our symbolic analysis framework to compute all valid variable bindings of a program at given program points. This analysis information can then be applied with domain-specific static program analyses for the detection of program anomalies and for program transformations, esp. with optimizing compilers.

Although the comprehensive treatment of such domain-specific analyses is outside the scope of this paper, we motivate the benefit of our framework's analysis information for the removal of redundant Ada95 bounds-checks.

Bounds-checks are required by the Ada language to ensure type-safety with array indices and arithmetic expressions. All expressions that cannot be bound at compile-time require a costly bounds-check at run-time. A redundant bounds-check guards an expression which is bounded, but which cannot be proven so by the compiler. The program information provided by closure-contexts increases analysis precision, which in turn enables more aggressive removal of redundant bounds-checks.

As an example, consider the exponentiation procedure from Fig. 6. Expressions $y*f$ (line 11) and $f*f$ (line 13) are potentially outside the range of the Ada95 type `positive`. (Note that the Ada95 type `positive` ranges from 1 to $2^{63}-1$ on a 64 bit architecture. Its lower and upper bounds are denoted by `positive'first` and `positive'last`.) If the compile-time constant `Max` is small enough s.t.

$$\mathrm{Max}^{\mathrm{Max}} \le \mathrm{positive'last}, \tag{39}$$

then the upper bound of type `small_int` of variables x and n already implies that expression $y*f$ cannot overflow. Because expression $f*f$ "proactively" computes the multiplication factor for the next loop iteration (which will not be needed in case of the last loop iteration where it can grow to $\mathrm{Max}^{\mathrm{Max}}\cdot\mathrm{Max}^{\mathrm{Max}}$), overflow of expression $f*f$ will not occur if

$$\mathrm{Max}^{2\cdot\mathrm{Max}} \le \mathrm{positive'last}. \tag{40}$$

Moreover, expressions $y*f$ and $f*f$ are of type `positive` and strictly increasing, which guarantees them to be greater-equal to zero. Thus (39) and (40) are sufficient and statically checkable conditions whether bounds-checks $R1$ and $R2$ are redundant.

Setting Max = 1, GCC 4.4.5 under compiler option -O3 was not able to remove bounds-checks $R1$ and $R2$. Neither is the symbolic analysis method for bounds-check removal introduced in Blieberger and Burgstaller (2003), because its variable range analysis is restricted to information encoded in conditions of loops and if statements only. Obviously, the conditions $e > 0$ and $e > 0 \wedge e \bmod 2 = 1$ guarding lines 11 and 13 are insufficient to discharge bounds-checks $R1$ and $R2$.

Leaving out the details of a symbolic bounds-checks removal analysis due to space considerations, we conclude this section with an outline on how the analysis information captured by closure contexts can be utilized to capture bounds-checks such as those from Fig. 6. Assuming that ranges of types are available[6], we can plug in the upper bound $e = \text{Max}$ into the closure context from (38) and apply standard techniques from Fahringer and Scholz (2003), Blieberger (1994) to derive $l_\omega = \lfloor \text{ld Max} \rfloor + 1$ as the upper bound for the number of loop iterations.

Expression $f^* f$ can be bounded by plugging in l_ω and $f = \text{Max}$ into the recurrence system from (37), which gives

$$f(l_\omega) = \underline{f}^{2^{\lfloor \text{ld Max} \rfloor + 1}} \leq \underline{f}^{2 \cdot \text{Max}} \leq \text{Max}^{2 \cdot \text{Max}}.$$

This concrete value for the upper bound of expression $f^* f$ is then compared with the value `positive'last`, according to (40). If the upper bound is smaller than the value for `positive'last`, bounds-check $R2$ has been proven redundant. Bounds-check $R1$ can be treated similarly if we observe that the first case of the conditional recurrence relation for variable y in (37) is always larger than the second case, which means that restricting analysis to the first case gives a safe approximation for the upper bound of expression $y^* f$ to be used with the check in (39).

As outlined in this example, the analysis information provided by our symbolic analysis framework benefits domain-specific static analyses such as redundant bounds-check removal. Closure contexts capture all variable bindings valid at a given program point. Within loops, conditional recurrence relations are employed to model the behavior of induction variables across the different iterations of the loop. Standard techniques to solve recurrence relations (Greene and Knuth, 1982; Lueker, 1980; van Engelen et al., 2004; van Engelen, 2004; Haghighat and Polychronopoulos, 1996; Gerlek et al., 1995) can be applied to derive closed forms for induction variables. This analysis information can then be applied with domain-specific static program analyses.

6. Implementation and experimental results

With our method we transfer a standard-semantic program into the symbolic domain using a homomorphism on path expressions. The implementation of this transformation with a CAS is straightforward if we can ensure a unique normal form for our symbolic program representation. Based on the work of Buchberger and Loos (1982) we can specify the operational semantics of a convergent term rewrite system (Baader and Nipkow, 1998) to derive unique normal forms for multivariate polynomials, rational functions, and radical expressions. Our case constitutes an *order-sorted* framework, because we have $\mathbb{Z} \subseteq \mathbb{Z}[\mathbf{x}] \subseteq Q(\mathbb{Z}[\mathbf{x}])$. The generalization of many-sorted algebra to order-sorted algebra is shown by Goguen and Meseguer (2002). We have extended this convergent term rewrite system to include states, program contexts, closure contexts and finite supercontexts. Due to Birkhoff's Theorem (Baader and Nipkow, 1998) semantic equivalence coincides with syntactic equality and we can decide the first based on the latter.

The prototype implementation of our symbolic analysis framework constitutes a term rewrite system based on OBJ3 (Baader and Nipkow, 1998; Goguen et al., 1993). We formulate the computation of path expressions from CFGs as a data-flow problem stated by the following two equations.

$$R(n_e) = \Lambda \tag{41}$$

$$R(n) = \mathop{+}_{n' \in \text{Preds}(n)} [R(n') \cdot R^{\varnothing}_{n',n}] \tag{42}$$

Therein $R^M_{i,j}$ denotes the regular expression whose language is the set of strings r such that every path from CFG node i to node j is represented by a string r, with the additional constraint that r must not contain *intermediate* nodes $n \notin M$. This data-flow problem cannot be solved by standard iteration-based data-flow algorithms in the presence of CFG loops. To solve this data-flow problem, we apply the elimination-based algorithm of Sreedhar (1995). As pointed out by Paull (1988), we define a normal form for our equations and set up a loopbreaking rule that allows us to handle circularities in CFGs. We replace (41) and (42) by the following normal form for our data-flow problem,

$$E : R(n) = \begin{cases} \mathop{+}_{m \in M \subseteq \text{sub}(\text{idom}(n))} [R(m) \cdot R^{S_m \subseteq \text{sub}(m) \setminus \{m\}}_{m,n}] & \text{if } n \neq n_e, \\ \Lambda & \text{else,} \end{cases}$$

where $\text{sub}(x)$ denotes the set of nodes of the dominator subtree rooted at node x, and $\text{idom}(n)$ denotes the immediate dominator of node n. Contrary to Eq. (42), a data-flow equation in normal form can depend on other nodes than its control flow predecessors. This is due to substitutions that occur during elimination (Paull, 1988). Given a data-flow equation that depends on itself, i.e.,

$$E : R(n) = R(\text{idom}(n)) \cdot R^{S_n}_{\text{idom}(n),n} + R(n) \cdot R^{S'_n \subseteq \text{sub}(n) \setminus \{n\}}_{n,n},$$

our loop-breaking rule replaces this equation by an equation for which the left-hand side does not appear on the right-hand side:

$$e : R(n) = R(\text{idom}(n)) \cdot R^{S_n}_{\text{idom}(n),n} \cdot (R^{S'_n \subseteq \text{sub}(n) \setminus \{n\}}_{n,n})^*.$$

Our implementation of Sreedhar's elimination algorithm was done in C++. It relies on Mathematica (Wolfram, 2003) to perform substitution and loop-breaking of data-flow equations. Our framework, together with the analysis results of Flow sample programs, has been published online (Symbolic Analysis Framework, 2009).

The practicality of our symbolic analysis method critically depends on the size of the path expressions occurring in practice. We have therefore surveyed the problem sizes arising from the programs of the complete SPEC95 benchmark suite (Standard Performance Evaluation Corporation, 1995). The SPEC95 benchmark suite consists of 18 benchmark programs with GCC and the Perl interpreter among them. Overall, we investigated all 5053 procedures, in an attempt to make the survey representative both in quantity and in the problem sizes of the investigated programs. The technical part of this survey comprised the definition of a metric to compute the symbolic analysis problem sizes (i.e., the number of closure contexts resulting from a given path expression), and to apply this metric to the path expressions of the procedures from the SPEC95 benchmark code.

We compute the number of program paths of a path expression corresponding to an acyclic CFG through the mapping $\text{ncc}(e) = 1$, $\text{ncc}(P_1 + P_2) = \text{ncc}(P_1) + \text{ncc}(P_2)$, and $\text{ncc}(P_1 \cdot P_2) = \text{ncc}(P_1) \cdot \text{ncc}(P_2)$. Every such program path induces the generation of one closure context during symbolic analysis.[7] Our *accumulated* ncc metric (ancc)

[6] For an overview on type system information propagation through CFGs see, e.g., Blieberger and Burgstaller (2003).

[7] Hence the name ncc which stands for *number of closure contexts*.

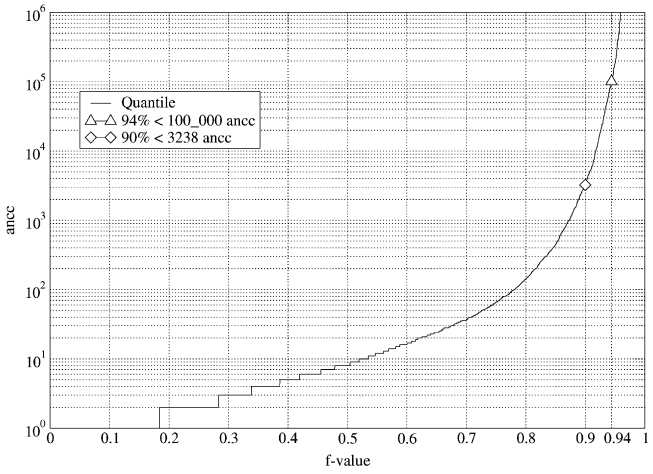


Fig. 8. Quantile plot for SPEC95 programs.

starts with the innermost nested loop of a path expression P and computes the ncc count for its body. Thereafter the subexpression in P that corresponds to this loop is replaced by a single edge and the ancc metric is applied to the resulting expression. This is done for all loops across all nesting levels, and for the topmost remaining loopless path expression itself. The ancc-value for P then equals the sum of the calculated ncc counts.

In our survey each SPEC95 procedure has been accounted for through its path expression of type (n_e, n_x). Fig. 8 contains a quantile plot of the ancc values of the SPEC95 procedures. It has been scaled to exclude outliers with an ancc-value above 10^6. It shows that the distribution of ancc 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 indicates that the final 6% of the distribution represent costly outliers. The two distinguished data points in the upper right corner represent the 0.9 quantile and the 0.94 quantile. It follows from those data points that 90% of the SPEC95 procedures show an ancc-value below 3238, and for 94% it is still below 100, 000. This means that the problem sizes of more than 94% of the procedures from the SPEC95 benchmark suite constitute no problem at all for symbolic analysis, and that the ancc values for 90% of all procedures are indeed very small. Due to space limitations we refer to Burgstaller et al. (2004) for a description of the whole range of experiments carried out on the SPEC95 benchmark suite.

7. Related work

Cousot and Cousot (1977) pioneered abstract interpretation as a theory of semantic approximation for semantic data and control flow analysis. The main differences between abstract interpretation and our symbolic analysis are as follows: our symbolic analysis framework precisely represents the values of program variables whereas abstract interpretation commonly approximates a program's computations. Second, path conditions guarding conditional variable values tend not to be included in abstract interpretation, although in principle this would be possible. Third, applications of abstract interpretation are faced with a trade-off between the level of abstraction and the precision of the analysis, and its approximated information may not be accurate enough to be useful.

Haghighat and Polychronopoulos (1996) base their symbolic analysis techniques on abstract interpretation. The information of all incoming paths to a node is intersected at the cost of analysis accuracy. Their method does not maintain predicates to guard the values of variables and it is restricted to reducible CFGs. No correctness proof of the used algorithms is given.

van Engelen et al. (2004) and van Engelen (2004) use chains of recurrences (Zima, 1995; Bachmann et al., 1994) to model symbolic expressions. Analysis is carried out directly on the CFG, with loops being analyzed in two phases. In the first phase recurrence relations are collected, whereas in the second phase the recurrence relations are solved in CR form. The analysis method is restricted to reducible CFGs, which makes it less general than our approach. In comparison, our algebra-centered approach uses only standard mathematical methods instead of specialized analysis algorithms. It provides for a seamless integration of the chains of recurrences algebra to solve recurrence relations, but it is not restricted to it.

The algorithms developed with both Haghighat's and van Engelen's approaches are tailored around the intended application (i.e., analysis problem). In contrast we advocate a generic method that allows the formulation of arbitrary domain-specific static analysis problems based on the MOP-solution.

Ammarguellat and Harrison (1990) employ abstract interpretation in conjunction with a template-based algorithm to recognize recurrence relations of programs containing `for` loops. Our approach is more general because it does not make any assumption on the CFG of a program, i.e., all loops and even irreducible CFGs are handled correctly. Our approach can compute solutions for nodes in the loop body, and for loop bodies with multiple program paths.

Blieberger (2002) uses symbolic evaluation for estimating the worst-case execution time of sequential real-time programs. Symbolic evaluation is set up as a data-flow problem, with equations describing the solutions at the respective CFG nodes. Fahringer and Scholz (2003) introduce a symbolic representation for program contexts. Closure contexts are an extension of this algebraic structure.

Tu and Padua (1995) developed a system for computing symbolic values of expressions using a demand-driven backward analysis based on G-SSA form. Their analysis can be more efficient than our approach if local analysis information suffices to obtain a result, otherwise they may have to examine large portions of a program. Tu and Padua require additional analysis to determine path conditions in contrast to our approach that directly represents path conditions in the context. For recurrences, Tu and Padua cannot directly determine the corresponding recurrence system from a given G-SSA form. With our approach the extraction of recurrence systems is an integral operation provided in the symbolic domain.

Menon et al. (2003) describe a technique for dependence analysis that verifies the legality of program transformations. They apply symbolic analysis to establish equality of a program and its transformation. Their symbolic analysis engine is limited to affine symbolic expressions and predicates consisting of conjunctions and disjunctions of affine inequalities. Blume and Eigenmann (1998) apply symbolic ranges to disprove carried dependences of permuted loop nests. They use abstract interpretation to compute the ranges for variables at each program point. Gerlek et al. (1995) describe a general induction variable recognition method based on a demand-driven SSA form. Rugina and Rinard (2000) carry out symbolic bounds analysis for accessed memory regions. With their method they set up a system of symbolic constraints that describe the lower and upper bounds of pointers, array indices, and accessed memory regions. This system of constraints is then solved using ILP. The Omega test by Pugh (1992) is an integer programming method that operates on a system of linear inequalities to determine whether a dependence between variables exists. It has been extended to nonlinear tests by Pugh and Wonnacott (1994) and Pugh (1994). Symbolic execution has been introduced in King (1976). With symbolic execution, program paths are executed based on symbolic input. Similar to symbolic analysis, symbolic expressions instead of actual data is used to present the values of program variables. The state of a symbolically executed program

includes the symbolic values of program variables, a path condition and the program counter. During execution, the path condition accumulates constraints which the inputs must satisfy for an execution to follow that particular program path. Symbolic execution is a mature technique that found widespread interest in the research community. A recent survey on symbolic execution with an emphasis on test generation and program analysis is given in Pasareanu and Visser (2009).

Person et al. (2008) proposes differential symbolic execution to compute behavioral characterizations of program changes. This analysis information can be used to automate software evolution or for checking the logical differences of two program versions, e.g., after re-factoring. The analysis is capable of demonstrating that two program versions are equivalent, or, if they are not, of characterizing the behavioral differences between versions by identifying the sets of inputs that cause a different effect.

Symbolic Pathfinder is a novel symbolic execution infrastructure for static analysis of Java bytecode. Pasareanu et al. (2008) developed this tool for unit-level symbolic execution of safety-critical software. Symbolic Java Pathfinder integrates symbolic execution with model checking to perform automated generation of test-cases and to check properties of programs during test-case generation.

Another recent approach with symbolic execution extends the sequential execution model to cover complex input data and concurrency. It has been shown by Khurshid et al. (2003) how dynamically allocated data, arrays and concurrency can be handled with Java bytecode. To reduce analysis effort, input data structures with uninitialized fields are initialized lazily, i.e., fields are initialized when they are first accessed during symbolic execution. The Kiasan tool by Khurshid et al. (2003), Belt et al. (2011) and Deng et al. (2006b) that builds on the Bogor model checking framework applies so-called k-bounded lazy initialization, where heap reference chains are bounded up to length k. Initialization of uninitialized fields is further delayed until fields are used.

Symbolic execution can only prove program correctness if the corresponding symbolic execution tree is finite. Because most programs do not exhibit bounds on the number of possible loop iterations, their execution trees are infinite. To prove correctness of such programs using symbolic execution, execution trees must be processed inductively rather than explicitly, as proposed by Hantler and King (1976). Loop invariants are required, either through user annotations or by automatic discovery. The Bogor/Kiasan tool by Deng et al. (2006a) employs such reasoning with Java programs.

A method that combines symbolic execution with model checking for the verification of parallel numerical programs has been proposed by Siegel et al. (2008). Because parallel programs potentially execute floating-point computations in a different order than the corresponding sequential program, parallel numerical programs may behave differently with the same input vector. The proposed method establishes functional equivalence of the sequential and parallel versions of a program. Floating-point operations are replaced by corresponding operations in the symbolic domain. The model checker employs symbolic execution to record all possible paths of the sequential program, up to a certain loop bound. In a second step, the model checker explores all possible paths of the parallel program that are consistent with the sequential execution to check for functional equivalence. In Siegel and Rossi (2008), the method has been applied for the verification of an MPI-based scientific program for the solution of two-dimensional Navier–Stokes equations.

The main difference between symbolic execution and our method is that symbolic execution is performed on individual program paths. In contrast, our method computes the complete control and data flow analysis information valid at a given program point. In this way symbolic analysis can cover program paths which are intractable for symbolic execution. However, in the presence of recurrence relations for which our symbolic analysis method cannot derive a closed form, it has to revert to an approximation, which induces loss of analysis precision. Symbolic execution can be used to selectively and accurately analyze program behavior on a subset of all program paths up to a given path length, but it cannot obtain the complete program information for larger programs. As a result, symbolic execution shows good results with analyses that look for a single, particular program instance, e.g., with software testing. Symbolic analysis on the other hand attempts to gain a holistic view of the program, which, if possible to achieve, is more suitable e.g., for program verification. Recent symbolic execution approaches, e.g., by Deng et al. (2006a), constitute promising approaches to overcome the limitation to finite execution trees through the inclusion of inductive reasoning. It has yet to be decided whether loop-invariants can be derived for a sufficiently large class of programs.

Symbolic execution methods differ from our symbolic analysis framework in their specialization towards a particular analysis goal. With our framework, we propose a two-stage approach where the computation of a generic set of analysis information is provided, which can then be tailored and/or applied for particular domain-specific static analyses. If successful, this approach will be similar to model-checking, where there is a clear boundary between models and model checkers.

Symbolic execution is a very mature field, with many tools readily available, e.g., by Deng et al. (2006a), Bush et al. (2000), Tomb et al. (2007) and Csallner et al. (2005). Symbolic execution techniques are advanced to the stage where dynamic data-structures, multi-threading and interprocedural analysis is covered. At this stage, our symbolic analysis framework is restricted to the analysis of single procedures with assignment statements. Nevertheless, our analysis supports arbitrary CFG structures, including irreducible graphs. Our proposed analysis method has been proven correct, and it is solely based on a fully-automated path homomorphism from path expressions into the symbolic domain, which provides straight-forward implementation opportunities with contemporary CAS systems.

8. Conclusions and future work

In this paper we have presented a generic symbolic analysis framework for imperative programming languages. At the center of our framework is a comprehensive and compact algebraic structure called supercontext. Supercontexts describe the complete control and data flow analysis information valid at a given program point. This information is invaluable for all kinds of static program analyses, such as memory leak detection (Scholz et al., 2000), program parallelization (Fahringer and Scholz, 2003; Haghighat and Polychronopoulos, 1996; van Engelen et al., 2004; Blume and Eigenmann, 1998), detection of superfluous bound checks, variable aliases and task deadlocks (Rugina and Rinard, 2000; Blieberger and Burgstaller, 2003; Blieberger et al., 1999, 2000a), and for worst-case execution time analysis (Blieberger, 2002; Blieberger et al., 2000b).

At present our framework accurately models assignment statements, branches, and loop constructs of imperative programming languages. It can easily be extended to the inter-procedural case as proposed by Fahringer and Scholz (2003) and Blieberger et al. (1999).

Our approach is more general than existing methods because it can derive solutions for arbitrary nodes (even within loops and nested loops) of reducible and irreducible CFGs.

We proved the correctness of our symbolic analysis method using a two-step verification based on the MOP-solution for symbolic execution and path-expression-based symbolic evaluation.

Our approach is based purely on algebra and is fully automated. The detection of recurrences is decoupled from the process of finding closed forms. This separation facilitates the extension of our recurrence solver with new classes of recurrence relations. Our novel representation of program semantics closes the gap between program analysis and computer algebra systems, which makes supercontexts an ideal symbolic intermediate representation for all domain-specific static program analyses.

The experiments conducted with our prototype implementation showed that the problem sizes of real-world programs such as those from the SPEC95 benchmark suite are tractable for our symbolic analysis framework. It has been shown by Burgstaller et al. (2006a) that symbolic analysis has a vast improvement potential in the area of contemporary data-flow based analyses of sequential and concurrent programs. We are therefore facing two research tiers that we plan to pursue in the future, namely (1) the extension of our method to incorporate concurrent programming language constructs, and (2) the application of our method to domain-specific static program analysis problems.

Acknowledgements

Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) under grant No. 2010-0005234, and the OKAWA Foundation Research Grant (2009). Research partially supported by the Australian Research Council through ARC DP grants DP1096445 and DP0560190.

Appendix A. Symbolic integer division and remainder operations

With 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

$$e_1(\mathbf{x}) \operatorname{div}_s e_2(\mathbf{x}) = \mathrm{Rnd}(e_1(\mathbf{x}) / e_2(\mathbf{x})), \tag{A.1}$$

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

$$e_1(\mathbf{x}) \operatorname{rem}_s e_2(\mathbf{x}) = e_1(\mathbf{x}) - e_2(\mathbf{x}) \cdot (e_1(\mathbf{x}) \operatorname{div}_s e_2(\mathbf{x})) \tag{A.2}$$

for the symbolic remainder operator rem_s. Since the rounding operation Rnd involves floor and ceiling functions, the expressions resulting from the symbolic division and remainder operations are inflexible: we can move integer terms in and out of floor and ceiling functions, but little more (Graham et al., 1994). The following special cases of symbolic integer division allow us to avoid floor and ceiling functions.

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 Geddes et al. (1992, Theorem 2.5 (v)) that $\mathbb{Q}(x)$ is an Euclidean domain.

On the analogy of standard-semantic integer division, 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

$$a(x) = q(x) \cdot b(x) + r(x), \deg(r(x)) < \deg(b(x)), \tag{A.3}$$

where deg(...) 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 Knuth (1997, Section 4.6).

We are interested in the quotient polynomial $q(x)$ to simplify the symbolic division and remainder operations of Eqs. (A.1) and (A.2).

Table A.1
Resulting symbolic expressions for univariate polynomials.

Case	Expressions		Condition		
	$a(x)\operatorname{div}_s b(x)$	$a(x)\operatorname{rem}_s b(x)$	$a(x)$	$b(x)$	$r'(x)$
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	

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*

$$\left|\frac{r(x)}{b(x)}\right| \leq \frac{1}{2}. \tag{A.4}$$

Lemma 2. *Given the side-condition from Eq. (A.4), we have*

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

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

Proof. From Eq. (A.3) we get

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

Due to the side-condition stated in Eq. (A.4), 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 $u = \mathrm{RndN}(q(x))$. Then $(a(x)/b(x)) \in [q(x)-0.5, q(x)+0.5]$ implies that $\mathrm{Rnd}(a(x)/b(x)) \in \{u-1, u, u+1\}$, which satisfies Lemma 2.

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 $\mathrm{RndN}(q(x)) \in \{u_1, u_2\}$, and due to the side-condition, $\mathrm{Rnd}(a(x)/b(x)) \in \{u_1, u_2\}$. □

Lemma 2 establishes that, given the side-condition of Eq. (A.4), the integer-valued result of $a(x)\operatorname{div}_s b(x)$ is an expression among the set

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

Let $q'(x) = \mathrm{RndN}(q(x))$ and $r'(x) = a(x) - (b(x) \cdot q'(x))$. 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 A.1.

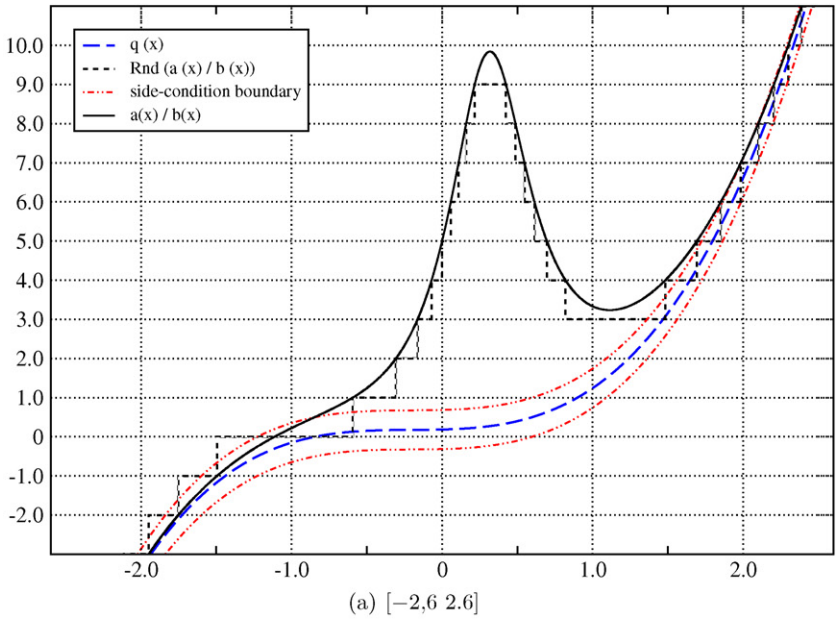
Cases (1) and (2) compute the same expression for the symbolic quotient (and also for the remainder), so we can combine these cases using the conjunction of both conditions. The same holds for cases (3) and (4). The solutions stated in Eqs. (A.1) and (A.2) account for the additional case of an invalid 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 1. 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

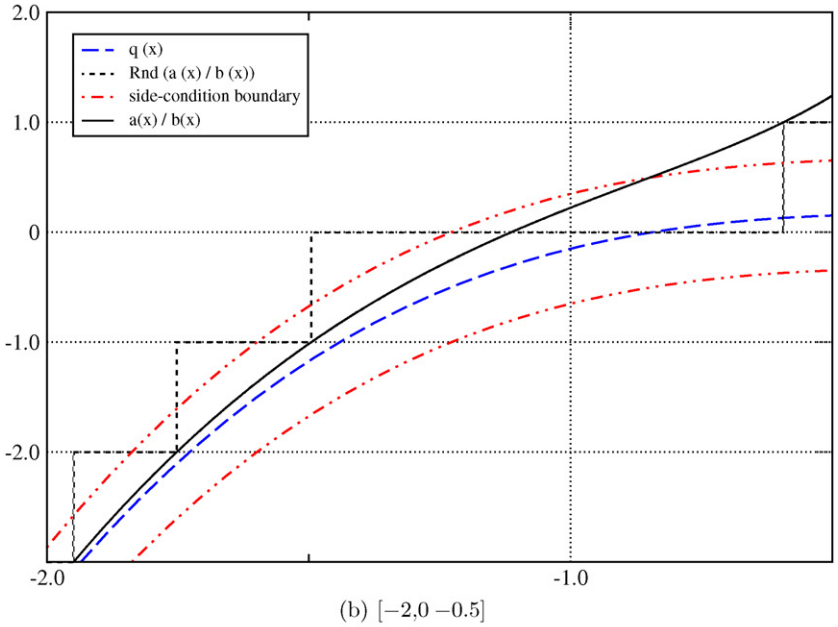
$$q(x) = \frac{3x^3}{5} + \frac{9x^2}{25} + \frac{12x}{125} + \frac{116}{625} \tag{A.5}$$

and the remainder polynomial

$$r(x) = \frac{913x}{625} + \frac{3009}{625}. \tag{A.6}$$



(a) $[-2{,}6\ 2.6]$



(b) $[-2{,}0\ -0.5]$

Fig. A.9. Example of Polynomial Integer Arithmetic

An illustration of this example is given in Fig. A.9. 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 Fig. A.9(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 2div9 = 0 and 2rem9 = 2 for quotient and remainder computed from the integer arithmetic of the Flow concrete semantics (cf. also Fig. A.9(b)).

Furthermore, we have $q'(-1) = \mathrm{RndN}(-0.1504) = 0$, and $r'(-1) = 2$. From the values of $a(-1)$, $b(-1)$, and $r'(-1)$ it follows that Case (5) of Table A.1 is 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 (Geddes et al., 1992, 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 (1995, Section 3.4)). In this case we have to fall back to the solutions of Eqs. (A.1) and (A.2).

Appendix B. Correctness Proofs

In this section we give a proof of correctness for the MOP solution stated in Eq. (9). 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.

Definition 10. 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 $\sigma : \underline{\mathbb{V}} \to$ 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.

$$\widehat{\sigma}(x) = \sigma(x), \text{ for } x \in \underline{\mathbb{V}} \tag{B.1}$$

$$\widehat{\sigma}(f(e_1, \ldots, e_n)) = \begin{array}{l} f(\widehat{\sigma}(e_1), \ldots, \widehat{\sigma}(e_n)), \\ \text{for a non-variable expression} \end{array} \tag{B.2}$$

From Eq. (B.2) 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. Note that the concept of SymExpr-substitutions is closely related to substitutions on terms presented by Baader and Nipkow (1998).

Definition 11. An **extended environment** $\overline{env} \in Env \times \mathbb{B}$ is an ordered tuple $[env, b]$ where env denotes an environment from the Flow standard semantics, and $b \in \mathbb{B}$ is a boolean value denoting the standard semantic pathcondition.

Definition 12. To describe the standard semantic computational effect of an *arbitrary* edge e, we use the function class

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

with its member functions f_c defined as

$$f_c = M_c(e)([env, b]) = ([\sigma_c(e)(env), b \wedge pred_c(e)(env)]), \tag{B.3}$$

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 Eq. (B.3) is equivalent to δ in terms of the computed environment $\sigma_c(e)(env)$ as well as the computed pathcondition $b \wedge pred_c(e)(env)$. (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 $pred_c(e)(env) = true$). On the other hand, for values outside the domain of δ we note that the function class of Eq. (B.3) 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 $pred_c(e)(env) = false$.

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 Eq. (7), we get

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

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 13. Function **sym** transfers an extended environment $[env, b]$ to a program context in the symbolic domain.

$$\begin{array}{l} \text{sym} : Env \times \mathbb{B} \to C \\ \text{sym}([env, b]) = \lambda[env\prime, b\prime]. \\ \quad [\lambda env''.s[\forall v \in \mathcal{D}om(env'') : v \mapsto \underline{v}](env\prime), b\prime]([env, b]) \end{array}$$

The extended environment $[env, b]$ and the state of the resulting program context $[s, b]$ coincide on the contained program variables, that is, $\mathcal{D}om(env) = \mathcal{D}om(s)$. 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 14. Function **con** takes an environment env and a program context c as input and returns an extended environment $\overline{env}$ which is the result of evaluation of c with the values provided by env.

$$\begin{array}{l} \text{con} : Env \times C \to Env \times \mathbb{B} \\ \text{con}(env, c) = \lambda[env\prime, [s, p]]. \\ \quad [env\prime[\forall v \in \mathcal{D}om(s) : v \mapsto \sigma_{env}(s(v))], \sigma_{env}(p)](env, c) \end{array}$$

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

$$\sigma_{env} = \{\underline{v_1} \to env(v_1), \ldots, \underline{v_n} \to env(v_n)\}, \tag{B.4}$$

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 $\underline{\mathbb{V}}$). 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 Fig. B.10, 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

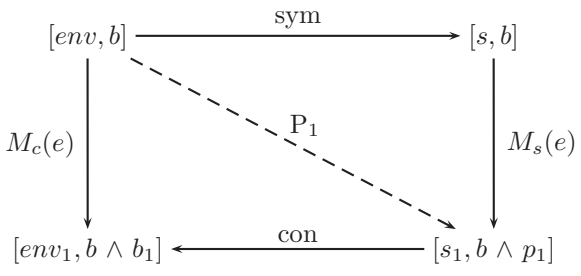


Fig. B.10. Commutation of single-edge concrete and symbolic execution.

with Definition 12 this coincides with the transition function δ iff $b = true$ and $b_1 = pred_c(e)(env) = true$).

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 (see Eq. (6)).
- For the pathcondition $b \wedge p_1$ we have $b \in \mathbb{B}$ and $p_1 \in$ SymPred. p_1 represents the branch predicate $pred_s(e)$ of the symbolic domain associated with edge e (see again Eq. (6)).

Transforming 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 Fig. B.10, 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 Fig. B.10 is generalized in the following lemma.

Lemma 3. *Given a* Flow *program with control flow graph* $G = \langle N, E, n_e, n_x \rangle$. *For any edge* $e \in E$ *of* G, *the condition*

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

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

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

$$\mathrm{con}(env, \mathrm{sym}([env, b])) = [env, b]. \tag{B.5}$$

From the valuation functions for assignment statements (cf. lines 1–3 of Table B.2) 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 "id⟦id⟧", to be specific), that for a given side-effect the concrete and symbolic valuation functions update the *same* program variable.

With this argument and Eq. (B.5) we establish that in the diagram of Fig. B.10 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 B.2), and the substitution σ_{env} (cf. Eq. (B.4)) that is part of function con. Due to the involved concrete and symbolic valuation functions exp_c and exp_s, the notion of consistent update is named exp_s^c-consistency.

Table B.2
Comparison: concrete vs. symbolic side-effects.

(a) Concrete domain	
1	$\mathrm{assign}_c : \mathrm{Assignment} \to Env \to Env$
2	$\mathrm{assign}_c[\![\mathrm{id} := \mathrm{exp}]\!](env_1) =$
3	$\lambda env_2 \,.\, env_2[\mathrm{id}[\![\mathrm{id}]\!] \mapsto \mathrm{exp}_c[\![\mathrm{exp}]\!](env_2)](env_1)$
4	$\mathrm{exp}_c : \mathrm{Expression} \to Env \to \mathbb{Z}$
5	$\mathrm{exp}_c[\![\mathrm{exp}_1 \ \mathrm{binop} \ \mathrm{exp}_2]\!](env) =$
6	$\mathrm{exp}_c[\![\mathrm{exp}_1]\!](env) \ \mathrm{binop}_c[\![\mathrm{binop}]\!] \ \mathrm{exp}_c[\![\mathrm{exp}_2]\!](env)$
7	$\mathrm{exp}_c[\![-\mathrm{exp}]\!](env) = -\mathrm{exp}_c[\![\mathrm{exp}]\!](env)$
8	$\mathrm{exp}_c[\![\mathrm{id}]\!](env) = env(\mathrm{id}[\![\mathrm{id}]\!])$
9	$\mathrm{exp}_c[\![\mathrm{num}]\!](env) = \mathrm{num}[\![\mathrm{num}]\!]$
10	$\mathrm{exp}_c[\![(\mathrm{exp})]\!](env) = \mathrm{exp}_c[\![\mathrm{exp}]\!](env)$
11	$\mathrm{id} : \mathrm{Identifier} \to \mathbb{V}$ (omitted)
12	$\mathrm{binop}_c : \mathrm{BinaryOperator} \to \{+, -, \cdot, \mathrm{div}, \mathrm{rem}\}$ (omitted)
(b) Symbolic domain	
1	$\mathrm{assign}_s : \mathrm{Assignment} \to C \to C$
2	$\mathrm{assign}_s[\![\mathrm{id} := \mathrm{exp}]\!]([s_1, p_1]) =$
3	$\lambda[s_2, p_2] \,.\, [\lambda s_3 \,.\, s_3[\mathrm{id}[\![\mathrm{id}]\!] \mapsto \mathrm{exp}_s[\![\mathrm{exp}]\!](s_3)](s_2), p_2]([s_1, p_1])$
4	$\mathrm{exp}_s : \mathrm{Expression} \to S \to \mathrm{SymExpr}$
5	$\mathrm{exp}_s[\![\mathrm{exp}_1 \ \mathrm{binop} \ \mathrm{exp}_2]\!](s) =$
6	$\mathrm{exp}_s[\![\mathrm{exp}_1]\!](s) \ \mathrm{binop}_s[\![\mathrm{binop}]\!] \ \mathrm{exp}_s[\![\mathrm{exp}_2]\!](s)$
7	$\mathrm{exp}_s[\![-\mathrm{exp}]\!](s) = -\mathrm{exp}_s[\![\mathrm{exp}]\!](s)$
8	$\mathrm{exp}_s[\![\mathrm{id}]\!](s) = s(\mathrm{id}[\![\mathrm{id}]\!])$
9	$\mathrm{exp}_s[\![\mathrm{num}]\!](s) = \mathrm{num}[\![\mathrm{num}]\!]$
10	$\mathrm{exp}_s[\![(\mathrm{exp})]\!](s) = \mathrm{exp}_s[\![\mathrm{exp}]\!](s)$
11	$\mathrm{id} : \mathrm{Identifier} \to \mathbb{V}$ (omitted)
12	$\mathrm{binop}_s : \mathrm{BinaryOperator} \to \{+, -, \cdot, \mathrm{div}_s, \mathrm{rem}_s\}$ (omitted)

$$\mathrm{exp}_s^c \quad -\text{consistency} \Leftrightarrow \mathrm{exp}_c[[\mathrm{exp}]](env) = \sigma_{env}(\mathrm{exp}_s[[\mathrm{exp}]](s)) \tag{B.6}$$

Recall that in this equation "exp" denotes a derivation tree corresponding to a Flow expression. We prove exp_s^c-consistency by structural induction on the Flow expression derivation trees according to lines 5–10 of the exp_c and exp_s valuation functions of Table B.2. Lines 8 and 9 constitute the cases of the induction basis, as they represent the leaves of expression derivation trees. 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: We show that $\mathrm{exp}_c[\![\mathrm{id}]\!](env) = \sigma_{env}(\mathrm{exp}_s[\![\mathrm{id}]\!](s))$. This holds if $env(\mathrm{id}[\![\mathrm{id}]\!]) = \sigma_{env}(s(\mathrm{id}[\![\mathrm{id}]\!]))$ which follows from Eq. (B.5). Furthermore we must show that $\mathrm{exp}_c[\![\mathrm{num}]\!](env) = \mathrm{exp}_s[\![\mathrm{num}]\!](s)$ which is trivially true since the valuation function num⟦num⟧ for numbers is common to the concrete and symbolic domain (cf. Table B.2).

Induction: In the inductive step we must show that

$$\mathrm{exp}_c \quad [[\mathrm{exp}_1 \, \mathrm{binop} \, \mathrm{exp}_2]](env) = \sigma_{env}(\mathrm{exp}_s[[\mathrm{exp}_1 \, \mathrm{binop} \, \mathrm{exp}_2]](s)).$$

Substituting the right-hand sides of lines 6 from Table B.2 and using the shorthand notations

$$e_{c1} = \mathrm{exp}_c[[\mathrm{exp}_1]](env), \quad e_{s1} = \mathrm{exp}_s[[\mathrm{exp}_1]](s),$$
$$e_{c2} = \mathrm{exp}_c[[\mathrm{exp}_2]](env), \quad e_{s2} = \mathrm{exp}_s[[\mathrm{exp}_2]](s),$$

we must show that

$$e_{c1} \ \mathrm{binop}_c[[\ \mathrm{binop}]] \ e_{c2} = \sigma_{env}(e_{s1} \ \mathrm{binop}_s[[\mathrm{binop}]] \ e_{s2}). \tag{B.7}$$

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 binop[[binop]] we can then distinguish the following three cases.

Case 1: Let binop[[binop]] $\in \{+, -, \cdot\}$. From Definition 10 of substitution σ_{env} we may write

$$\sigma_{env}(e_{s1}\ \text{binop}_s[[\text{binop}]]\ e_{s2}) = \sigma_{env}(e_{s1})\ \text{binop}_s[[\text{binop}]]\ \sigma_{env}(e_{s2}).$$

Inserting this identity into Eq. (B.7) yields

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

This is 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: $\text{binop}_c[[\text{binop}]] = \text{div}$ and $\text{binop}_s[[\text{binop}]] = \text{div}_s$. Further simplifications yield

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

Again we have used rounding towards zero. Inserting the above right-hand side into Eq. (B.7) concludes the proof for Case 2 due to the assumption of the inductive step.

Case 3: $\text{binop}_c[[\text{binop}]] = \text{rem}$ and $\text{binop}_s[[\text{binop}]] = \text{rem}_s$. 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 exp_s^c-consistency of Flow expression derivation trees as postulated in Eq. (B.6). This means that concrete and symbolic execution commute on environment env_1 as depicted in Fig. B.10, and it remains to show that they also commute regarding the pathcondition $b \wedge b_1$.

From Fig. B.10 we observe that with the pathcondition $b \wedge b_1$ of the extended environment $[env_1, b \wedge b_1]$ variable b is due to the extended input-environment $[env, b]$. Moreover, 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 $b = true$, for $false \wedge b_1 = false$.)

On the analogy of exp_s^c-consistency we proceed with the definition of 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 Fig. B.10.

$$\begin{aligned}\text{pred}_s^c\quad & -\text{consistency} \Leftrightarrow \overline{b} \wedge \text{pred}_c[[\text{pred}]](env) = \\ &= \text{pc}(\sigma_{env}(\text{pred}_s[[\text{pred}]]([s, \overline{b}])))\end{aligned} \tag{B.8}$$

We will establish pred_s^c-consistency of Flow predicates by structural induction on the predicate derivation trees according to the corresponding valuation functions listed in Table B.3. In the by-case definition of the valuation function pred_s in Table B.3 (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 B.3 (b).

What follows is the inductive proof of pred_s^c-consistency of Flow predicates as outlined above.

Table B.3
Comparison: concrete vs. symbolic branch-predicates.

(a) Concrete domain

1 $\text{pred}_c : \text{Predicate} \to Env \to \mathbb{B}$
2 $\text{pred}_c[[\text{true}]](env) = true$
3 $\text{pred}_c[[\text{false}]](env) = false$
4 $\text{pred}_c[[\text{pred}_1 \text{ and } \text{pred}_2]](env) = \text{pred}_c[[\text{pred}_1]](env) \wedge \text{pred}_c[[\text{pred}_2]](env)$
5 $\text{pred}_c[[\text{pred}_1 \text{ or } \text{pred}_2]](env) = \text{pred}_c[[\text{pred}_1]](env) \vee \text{pred}_c[[\text{pred}_2]](env)$
6 $\text{pred}_c[[\text{not pred}]](env) = \neg\, \text{pred}_c[[\text{pred}]](env)$
7 $\text{pred}_c[[\text{exp}_1 \text{ rel} - \text{op exp}_2]](env) = \text{exp}_c[[\text{exp}_1]](env)$
8 $\qquad \text{rel} - \text{op}_c[[\text{rel} - \text{op}]]\text{exp}_c[[\text{exp}_2]](env)$

(b) Symbolic domain

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

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

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

Likewise for Case 2:

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

Case 6 requires us to show that

$$\begin{aligned}&\overline{b} \wedge (\text{pred}_c[[\text{exp}_1 \text{ rel} - \text{op exp}_2]](env)) = \\ &= \text{pc}(\sigma_{env}(\text{pred}_s[[\text{exp}_1 \text{ rel} - \text{op exp}_2]]([s, \overline{b}]))).\end{aligned}$$

Substituting the right-hand sides of Case 6 from Table B.3 and using the following shorthand notations

$$\begin{aligned}e_{c1} &= \text{exp}_c[[\text{exp}_1]](env), \quad e_{s1} = \text{exp}_s[[\text{exp}_1]](s), \\ e_{c2} &= \text{exp}_c[[\text{exp}_2]](env), \quad e_{s2} = \text{exp}_s[[\text{exp}_2]](s),\end{aligned}$$

we get

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

It is the exp_s^c-consistency that we have stated in Eq. (B.6) which provides that $e_{c1} = \sigma_{env}(e_{s1})$, and that $e_{c2} = \sigma_{env}(e_{s2})$. Knowing that the relational connective $\text{rel} - \text{op}_s[[\text{rel} - \text{op}]]$ 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

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

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

$$p_{c1} = \mathrm{pred}_c[[\mathrm{pred}_1]](env),\\ p_{s1} = \mathrm{pred}_s[[\mathrm{pred}_1]]([s, true]),\\ p_{c2} = \mathrm{pred}_c[[\mathrm{pred}_2]](env),\\ p_{s2} = \mathrm{pred}_s[[\mathrm{pred}_2]]([s, true]),$$

we get

$$\bar{b} \wedge (p_{c1} \wedge p_{c2}) = \\ = \mathrm{pc}(\sigma_{env}([s, \bar{b} \wedge (\mathrm{pc}(p_{s1}) \wedge \mathrm{pc}(p_{s2}))])) \\ = \bar{b} \wedge (\sigma_{env}(\mathrm{pc}(p_{s1})) \wedge \sigma_{env}(\mathrm{pc}(p_{s2}))).$$

The inductive hypothesis implies that $true \wedge p_{c1} = \sigma_{env}(\mathrm{pc}(p_{s1}))$, and $true \wedge p_{c2} = \sigma_{env}(\mathrm{pc}(p_{s2}))$, 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 follows along the above lines:

$$\bar{b} \wedge \mathrm{pred}_c[[\mathrm{not\ pred}]](env) = \\ = \mathrm{pc}(\sigma_{env}(\mathrm{pred}_s[[\mathrm{not\ pred}]]([s, true]))) \\ \bar{b} \wedge \neg\mathrm{pred}_c[[\mathrm{pred}]](env) = \\ = \bar{b} \wedge \neg\sigma_{env}(\mathrm{pc}(\mathrm{pred}_s[[\mathrm{pred}]]([s, true]))).$$

Considering the fact that from the inductive hypothesis we have

$$true \wedge \mathrm{pred}_c[[\mathrm{pred}]](env) = \\ \sigma_{env}(\mathrm{pc}(\mathrm{pred}_s[[\mathrm{pred}]]([s, true])))$$

concludes the case.

With the proof of Case 5 we have established pred_s^c-consistency of Flow predicate derivation trees as postulated in Eq. (B.8). Combining this result with exp_s^c-consistency proves the single-edge commutation property of concrete and symbolic execution as depicted in Fig. B.10. □

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

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

Lemma 4. *Given a* Flow *program with control flow graph* $G = \langle N, E, n_e, n_x \rangle$. *For any program path* π *through G, the condition*

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

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 3.

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 Fig. B.11 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

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

(In Fig. B.11 the environment env of the above equation is provided across projection P_{1_1}.) Moreover it follows from Lemma 3 that concrete and symbolic execution along edge $e_k + 1$ commute. This is depicted in the lower half of Fig. B.11: 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

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

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 3 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

$$\sigma_{s_k} = \{\underline{v_1} \mapsto s_k(v_1), \ldots, \underline{v_j} \mapsto s_k(v_j)\}, \\ \text{with } \underline{v_i}_{1 \le i \le j} \in \mathcal{D}om(s_k). \tag{B.9}$$

Observe from Fig. B.11 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 use 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 Fig. B.11.

To begin with, we note that the endomorphism-property of σ_{s_k} that is due to Eq. (B.2) together with the fact that $\sigma_{s_k}(s'_k) = s_k$ ensures that

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

In other words, starting with the clean slate 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 sufficient to determine the correctness of the pathcondition $b \wedge \cdots \wedge p_{k+1}$ 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

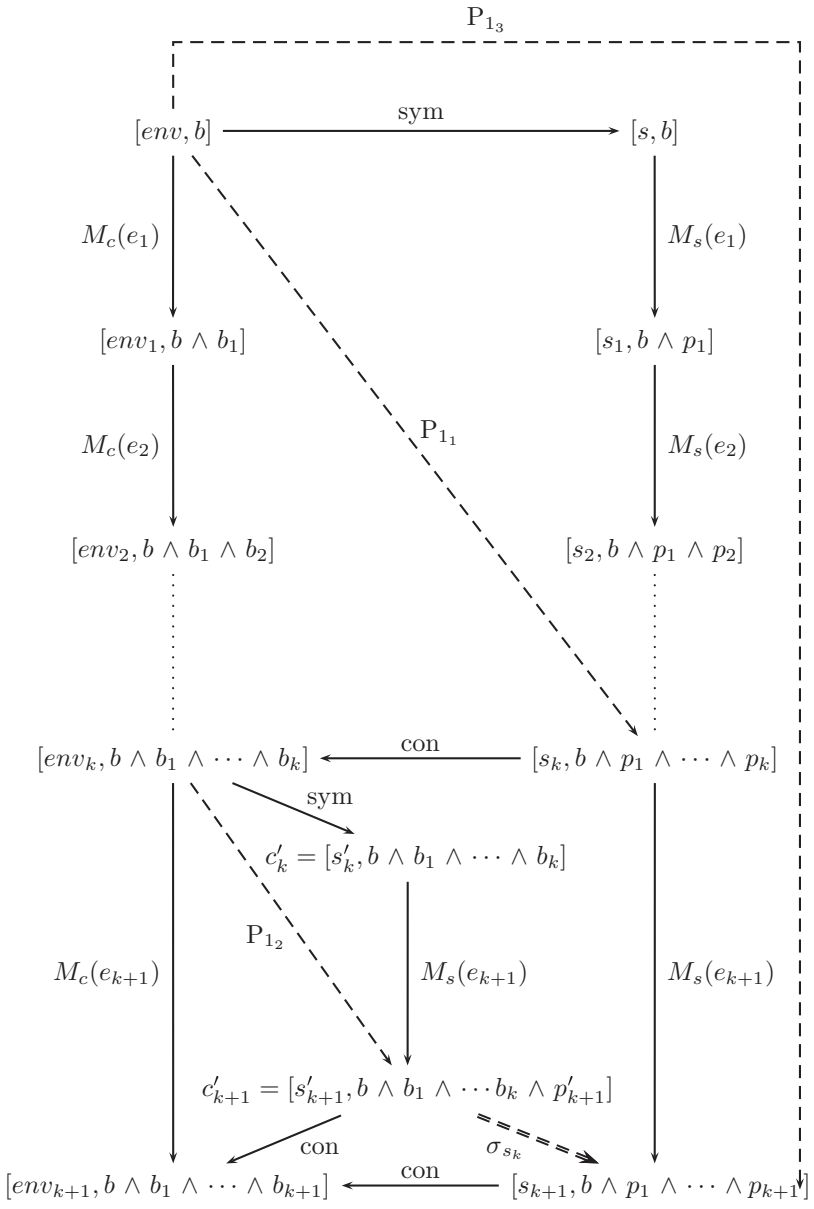


Fig. B.11. Commutation of Single-Path Concrete and Symbolic Execution

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

$$\text{con}(env, [s_{k+1}, b \wedge p_1 \wedge \cdots \wedge p_{k+1}]) = \overline{env}_{k+1}. \tag{B.10}$$

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

$$\text{con}(env_k, [s'_{k+1}, b \wedge b_1 \wedge \cdots \wedge b_k \wedge p_{k+1\prime}]) = \overline{env}_{k+1}. \tag{B.11}$$

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 Eq. (B.11) to prove the commutation of context c_{k+1} and environment env stated in Eq. (B.10).

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

$$\sigma_{env_k}(s'_k) = \sigma_{env}(s_k) \tag{B.12}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(s'_k(v)) = \sigma_{env}(s_k(v)) \tag{B.13}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}(s_k(v)) \tag{B.14}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(s(v))) \tag{B.15}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(\underline{v})). \tag{B.16}$$

In the above sequence of transformations the step from Eq. (B.13) to Eq. (B.14) is due to the fact that state s'_k maps a variable v to the corresponding initial value variable $\underline{v}$. The step from Eq. (B.14) to Eq. (B.15) is due to the properties of substitution σ_{s_k}. Finally, the step from Eq. (B.15) to Eq. (B.16) 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 Eq. (B.10):

$$\sigma_{env_k}(s'_{k+1}) = \sigma_{env}(s_{k+1}) \tag{B.17}$$

$$\Leftrightarrow \forall v \in \mathcal{D}om(s) : \quad \sigma_{env_k}(s'_{k+1}(v)) = \sigma_{env}(s_{k+1}(v)). \tag{B.18}$$

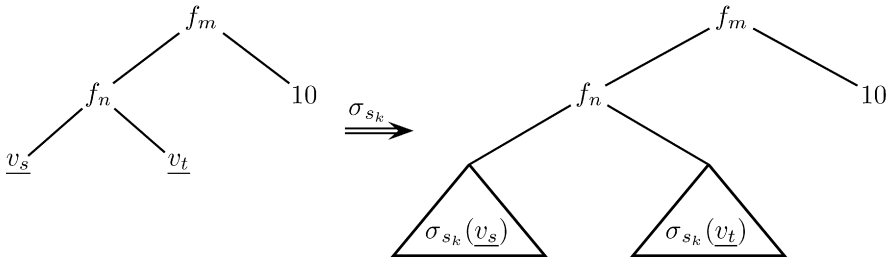


Fig. B.12. Application of Substitution σ_{s_k} to Expression $f_m(f_n(v_s, v_t), 10)$

We have already noted in the proof of Lemma 3 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, Eq. (B.13) would already imply Eq. (B.18). Take now a given derivation tree exp ∈ Expression constituting the syntactic representation of the update-value of the before-mentioned program variable v_ω. A final look at Fig. B.11 confirms that for our proof the side-effect of edge e_{k+1} is evaluated twice: once within state s'_k, yielding $e'_k(\mathbf{x}) = \exp_s[\![\exp]\!](s'_k)$ as the corresponding symbolic expression, and once within state s_k, yielding $e_k(\mathbf{x}) = \exp_s[\![\exp]\!](s_k)$.

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 Fig. B.12.

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 Eq. (B.18) regarding program variable v_ω.

$$\sigma_{env_k}(e'_k(\mathbf{x})) = \sigma_{env}(e_k(\mathbf{x})) \tag{B.19}$$

$$\Leftrightarrow \forall \underline{v} \in \mathcal{D}om(s) : \sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(\underline{v})) \tag{B.20}$$

Eq. (B.20) is suggested by the already mentioned "leaf-ness" of substitution σ_{s_k}, but the fact that Equation (B.19) holds when Eq. (B.20) holds follows immediately by induction on the structure of symbolic expressions (cf. also Fig. B.12). The condition $\sigma_{env_k}(\underline{v}) = \sigma_{env}(\sigma_{s_k}(\underline{v}))$ required by Eq. (B.20) 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 correctness of the remaining subcondition $b \wedge p_1 \wedge \cdots \wedge p_k$ is already due to the inductive hypothesis, which finally establishes the commutation-property postulated in Eq. (B.10), which also ends the proof of Lemma 4. □

We noted with the definition of substitution σ_{s_k} in Eq. (B.9) 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 4.

Corollary 1. *Given a* Flow *program with 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* π *is a partial function of arity* $Env \times \mathbb{B} \to Env \times \mathbb{B}$ *that represents the function composition* $M_s(e_k) \circ M_s(e_{k-1}) \circ \ldots \circ M_s(e_1)$. *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* $\sigma_{env}(p_k) = true$.

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 4. □

References

Ammarguellat, Z., Harrison III, W.L., 1990. Automatic recognition of induction variables and recurrence relations by abstract interpretation. SIGPLAN Notices 25 (6), 283–295.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press, New York.

Bachmann, O., Wang, P.S., Zima, E.V.,1994. 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. ACM Press, pp. 242–249.

Belt, J., Hatcliff, J., Robby, Chalin, P., Hardin, D., Deng, X.,2011. Bakar Kiasan: flexible contract checking for critical systems using symbolic execution. In: Third International Symposium on NASA Formal Methods. Springer LNCS, pp. 58–72.

Blieberger, J., 1994. Discrete loops and worst case performance. Computer Languages 20 (3), 193–212.

Blieberger, J., 2002. Data-flow frameworks for worst-case execution time analysis. Real-Time Systems 22, 183–227.

Blieberger, J., Burgstaller, B., 2003 June. Eliminating redundant range checks in GNAT using symbolic evaluation. In: Proc. of the Ada-Europe International Conference on Reliable Software Technologies, Toulouse, France, pp. 153–167.

Blieberger, J., Burgstaller, B., Scholz, B., 1999 June. Interprocedural symbolic evaluation of Ada programs with aliases. In: Proc. of the Ada-Europe International Conference on Reliable Software Technologies, Santander, Spain, pp. 136–145.

Blieberger, J., Burgstaller, B., Scholz, B.,2000a. Symbolic data flow analysis for detecting deadlocks in Ada tasking programs. In: Proc. of the Ada-Europe International Conference on Reliable Software Technologies. Springer-Verlag, pp. 225–237.

Blieberger, J., Fahringer, T., Scholz, B., 2000b. Symbolic cache analysis for real-time systems. Real-Time Systems 18 (2/3), 181–215.

Blume, W., Eigenmann, R., 1998. Nonlinear and symbolic data dependence testing. IEEE Transactions on Parallel and Distributed Systems 9 (12), 1180–1194.

Buchberger, B., Loos, R., 1982. Algebraic simplification. In: Buchberger, B., Collins, G.E., Loos, R. (Eds.), Computer Algebra: Symbolic and Algebraic Computation. Vol. 4 of Computing. Supplementum. Springer, Wien/New York, pp. 11–43.

Burgstaller, B., June 2005. Symbolic Evaluation of Imperative Programming Languages. Technical Report 183/1-138, Department of Automation, Vienna University of Technology.

Burgstaller, B., Scholz, B., Blieberger, J., June 2004. 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.

Burgstaller, B., Blieberger, J., Mittermayr, R., 2006a. Static detection of access anomalies in Ada95. In: Pinho, L.M., Harbour, M.G. (Eds.), Ada-Europe. Vol. 4006 of Lecture Notes in Computer Science. Springer, pp. 40–55.

Burgstaller, B., Scholz, B., Blieberger, J., 2006b. Symbolic Analysis of Imperative Programming Languages. In: Lightfoot, D.E., Szyperski, C.A. (Eds.), JMLC. Vol. 4228 of Lecture Notes in Computer Science. Springer, pp. 172–194.

Bush, W.R., Pincus, J.D., Sielaff, D.J., 2000 June. A static analyzer for finding dynamic programming errors. Software - Practice and Experience 30, 775–802.

Cousot, P., Cousot, R., 1977 January. Abstract interpretation: 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, Los Angeles, CA, pp. 238–252.

Csallner, C., Smaragdakis, Y., 2005. Check 'n' crash: combining static checking and testing. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05. ACM, New York, NY, USA, pp. 422–431.
Deng, X., Lee, J., Robby,2006a. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 157–166.
Deng, X., Robby, Hatcliff, J.,2006b. Kiasan: a verification and test-case generation framework for Java based on symbolic execution. In: ISoLA. IEEE, p. 137.
Fahringer, T., Scholz, B., 2003. Advanced Symbolic Analysis for Compilers. Vol. 2628. LNCS. Springer-Verlag.
Geddes, K.O., Czapor, S.R., Labahn, G., 1992. Algorithms for Computer Algebra. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands.
Gerlek, M.P., Stoltz, E., Wolfe, M., 1995 January. 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.
Goguen, J., Winkler, T., Meseguer, J., Futatsugui, F., Jouannaud, J., 1993. Introducing OBJ. Draft. Oxford University Computing Laboratory.
Goguen, J.A., Meseguer, J., 2002 March. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations.
Graham, R.L., Knuth, D.E., Patashnik, O., 1994. Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Addison-Wesley, Reading, MA, USA.
Greene, D., Knuth, D.E., 1982. Mathematics For the Analysis of Algorithms, 2nd ed. Birkhäuser, Cambridge, MA, USA/Berlin, Germany; Basel, Switzerland.
Haghighat, M.R., 1995. Symbolic Analysis for Parallelizing Compilers. Kluwer Academic.
Haghighat, M.R., Polychronopoulos, C.D., 1996 July. Symbolic analysis for parallelizing compilers. ACM Transactions on Programming Languages and Systems 18 (4), 477–518.
Hantler, S.L., King, J.C., 1976 September. An introduction to proving the correctness of programs. ACM Computing Surveys 8, 331–353.
Havlak, P., 1994 May. Interprocedural Symbolic Analysis. Ph.D. Thesis, Dept. of Computer Science, Rice University, also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.
Hecht, M.S., 1977. Flow Analysis of Computer Programs. Elsevier Science Inc.
Hopcroft, J.E., Ullman, J.D., 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, N. Reading, MA.
Khurshid, S., Pasareanu, C.S., Visser, W.,2003. Generalized symbolic execution for model checking and testing. In: TACAS. Springer LNCS, pp. 553–568.
King, J.C., 1976. Symbolic execution and program testing. Communications of the ACM 19 (7), 385–394.
Knuth, D.E., 1997. Seminumerical Algorithms, 3rd ed. Vol. 2 of The Art of Computer Programming. Addison-Wesley, Reading MA.
Lueker, G.S., 1980. Some techniques for solving recurrences. ACM Computing Surveys (CSUR) 12 (4), 419–436.
Menon, V., Pingali, K., Mateev, N., 2003. Fractal symbolic analysis. ACM Transactions on Programming Languages and Systems 25 (6), 776–813.
Pasareanu, C.S., Visser, W., 2009 October. A survey of new trends in symbolic execution for software testing and analysis. International Journal of Software Tools for Technology Transfer 11, 339–353.
Paull, M.C., 1988. Algorithm Design: A Recursion Transformation Framework. Wiley-Interscience.
Person, S., Dwyer, M.B., Elbaum, S., Pasareanu, C.S.,2008. Differential symbolic execution. In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 226–237.
Pugh, W., 1992 August. Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 35 (8), 102–114.
Pugh, W., 1994. Counting solutions to Presburger formulas: how and why. In: PLDI, pp. 121–134.
Pugh, W., Wonnacott, D., 1994. Nonlinear array dependence analysis. Tech. rep., College Park, MD, USA.
Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.,2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, pp. 15–26.
Rugina, R., Rinard, M., 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: Proc. of PLDI, pp. 182–195.
Scholz, B., Blieberger, J., Fahringer, T., 2000 January. Symbolic pointer analysis for detecting memory leaks. In: ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00), Boston.
Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A., 2008 May. Combining symbolic execution with model checking to verify parallel numerical programs. ACM Transactions on Software Engineering and Methodology 17, 10:1–10:34.
Siegel, S.F., Rossi, L.F.,2008. Analyzing BlobFlow: a case study using model checking to verify parallel scientific software. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag, Berlin, Heidelberg, pp. 274–282.
Sreedhar, V.C., 1995. Efficient program analysis using DJ graphs. Ph.D. thesis, School of Computer Science, McGill University, Montréal, Québec, Canada.
Standard Performance Evaluation Corporation, 1995 August. SPEC CPU95 Benchmark Suite, Version 1.10.
Symbolic Analysis Framework, 2009. http://www.it.usyd.edu.au/bburg/symanalysis.html.
Tarjan, R.E., 1981. A unified approach to path problems. Journal of the ACM (JACM) 28 (3), 577–593.
Tomb, A., Brat, G., Visser, W.,2007. Variably interprocedural program analysis for runtime error detection. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. ISSTA '07. ACM, New York, NY, USA, pp. 97–107.
Tu, P., Padua, D.A., 1995. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In: International Conference on Supercomputing, pp. 414–423.
van Engelen, R.A., 2004. The CR# algebra and its application in loop analysis and optimization. Tech. Rep. TR-041223, Department of Computer Science, Florida State University.
van Engelen, R.A., Birch, J., Shou, Y., Walsh, B., Gallivan, K.A.,2004. A unified framework for nonlinear dependence testing and symbolic analysis. In: ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing. ACM Press, pp. 106–115.
Wolfram, S., 2003. The Mathematica Book. Wolfram Media, Incorporated.
Zima, E.V.,1995. Simplification and optimization transformations of chains of recurrences. In: ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation. ACM Press, pp. 42–50.

Bernd Burgstaller received the PhD degree in static program analysis from the Vienna University of Technology, Austria, in 2005. He was a postdoc at the University of Sydney, Australia, until 2007. He is currently an assistant professor in the Department of Computer Science, Yonsei University, Seoul, Korea. Before pursuing an academic career, he was a software engineer and software architect for Philips Consumer Electronics, Vienna.

Bernhard Scholz received the Dipllng (eq. MEng) and PhD degrees from Vienna University of Technology, Austria, in 1997 and 2001, respectively. He is currently a senior lecturer in the School of Information Technologies, University of Sydney, Australia. His research interests include programming languages and parallel and embedded systems.

Johann Blieberger received the Dipllng (eq. MEng) and PhD degrees from Vienna University of Technology, Austria, in 1984 and 1985, respectively. In 2000 he got his habilitation and since then he is Associate Professor at the Institute of Computer-Aided Automation, Vienna University of Technology, Austria. His research interests include static analysis, real-time and embedded systems.