A symbolic analysis framework for static analysis of imperative programming languages

Bernd Burgstaller\textsuperscript{a,}\textsuperscript{*}, Bernhard Scholz\textsuperscript{b}, Johann Blieberger\textsuperscript{c}

\textsuperscript{a}Yonsei University, Seoul 120-749, Republic of Korea
\textsuperscript{b}The University of Sydney, NSW 2006, Sydney, Australia
\textsuperscript{c}Vienna University of Technology, 1040 Vienna, Austria

\textbf{ARTICLE INFO}

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

\textbf{Keywords:}
Static program analysis
Symbolic analysis
Path expression algebra
Programming language semantics

\textbf{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.

\textcopyright 2011 Elsevier Inc. All rights reserved.

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 Polychronopoulou, 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 formulae 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 $u$ for
variable $u$, and $y$ 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 $u$, 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 $\ast$. 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 notation 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 $V$. $Dom$ denotes the domain of a function. A control flow graph (CFG) is a directed labeled graph $G=(N, E, n_s, n_e)$ 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 ($n_s$) are distinguished CFG nodes used to denote the start and terminal node. The start node $n_e$ has no incoming edges ($in(n_e) = \emptyset$), whereas the terminal node $n_s$ has no outgoing edges ($out(n_s) = \emptyset$). We require that every node $n \in N$ is contained in a program path from $n_e$ to $n_s$, where a program path $\pi = (e_1, e_2, \ldots, e_k)$ is a sequence of edges such that $t(e_i) = h(e_{i+1})$ for $1 \leq i < k \leq n_s - 1$.

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

1a. “$\Lambda$” and “$\emptyset$” are atomic regular expressions; for any $a \in \Sigma$, “$a$” is an atomic regular expression.

1b. If $R_1$ and $R_2$ are regular expressions, then $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and $(R_1 \ast)$ are compound regular expressions.
3. Standard semantic program execution

An environment \( env \) of our Flow language maps a program variable \( v \in V \) to its value \( z \in Z \). The set of possible environments can be represented by a function class \( Env \subseteq (env : V \rightarrow Z) \). Functions \( \text{pred} : E \rightarrow (Env \rightarrow \mathbb{B}) \) and \( \sigma : E \rightarrow (Env \rightarrow Env) \) 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 \( \text{pred} : \) Predicate \( \rightarrow (Env \rightarrow \mathbb{B}) \) and \( \sigma : \) Assignment \( \rightarrow (Env \rightarrow Env) \) 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 \( \text{pred} \left( e \right) \left( env \right) = \text{true} \), which means that the predicate associated with edge \( e \) evaluates to \text{true} within environment \( env \). We require that for every node \( n \neq n_0 \) and environment \( env \), the branch-predicate of exactly one outgoing edge evaluates to \text{true}.

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

\[
(n, env) \rightarrow (n', env') : \quad \forall n' \forall env' \forall env \forall e : (e \in \text{out}(n) \land t(e) = n' \land \text{pred} \left( e \right) \left( env \right) \implies \text{env'} = \sigma \left( e \right) \left( env \right)),
\]

where \( \implies \) denotes implication. The extended transition function \( \delta' : (N \times Env) \rightarrow (N \times Env) \) is a partial function defined by \( \delta' \left( n_0, env \right) = \left( n_0, env \right) \) and \( \delta' \left( n, env \right) = \delta \left( n, env \right) \). For any graph \( G = (N, E, n_0, n_s) \) the environment \( env_s \) of the terminal node \( n_s \) represents the result of standard semantic program execution along the transition sequence \( \langle n_0, env_0 \rangle \rightarrow \ldots \rightarrow \langle n_s, env_s \rangle \). Depending on the structure of \( G \) and the initial environment \( env \), 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 \( z \in Z \), 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 \( S_{\text{con}} \) that takes a program \( P \) as argument and returns a standard-semantic functional description of the side-effect of \( P \). The side-effect \( S_{\text{con}}[P][I_n] \) is a function that maps a concrete input to a concrete output, written as \( S_{\text{con}}[P][I_n] = \text{Out} \), where \( I_n, \text{Out} \in \text{Env} \). Given the standard semantics of Flow programs from Section 3, the computation of \( S_{\text{con}}[P][I_n] \) is equivalent to an application of the iterated transition function \( \delta \) to start node \( n_0 \) and environment \( I_n \).

Similarly we derive the semantics of symbolic program execution for program \( P \), denoted by \( S_{\text{sym}}[P] \). It is the purpose of function \( S_{\text{sym}} \) 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 \( I_n \) to the corresponding symbolic output \( \text{Out}_s \). Symbolic input and output belong to the class \( Env_s \) of symbolic environments that replace the concrete environments \( env : V \rightarrow Z \) which are not able to bind identifiers to symbolic expressions.

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

\[
S_{\text{sym}}[P][\text{sym}(I_n)] = \text{con}(I_n, S_{\text{sym}}[P][\text{sym}(I_n)]),
\]

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

4.1. The domain for symbolic program execution

To be able to distinguish between a variable and its initial value, we introduce the set \( V \) of initial value variables. This set is isomorphic to \( V \). Its purpose is to represent the initial values for the variables in \( V \). The initial value operator \( \odot : V \rightarrow V \) maps a variable \( v \in V \) to the corresponding variable in \( V \). As a shorthand notation we write \( \odot \) for the application of the initial value operator to variable \( v \).

By 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 \( \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n \), are integer-valued expressions.

To support division, the ring \( \mathbb{Z}[\mathbf{x}] \) is extended to the quotient field \( \mathbb{Q}(\mathbb{Z}[\mathbf{x}]) \) (cf. Geddes et al., 1992). By means of the rounding operation \( \text{Rnd} \) 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 \( x \div y = \text{Rnd}(x/y) \), where the symbolic division operator \( \div \).
denotes the counterpart of the integer division operator div of the Flow standard semantics.

Let \( f^n \in \{ (n), (\), (\), (\), (\) \} \) 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 \( \text{Rnd} \).

**Definition 1.** The set of integer-valued symbolic expressions of the domain \( \text{SymExpr} \) is inductively defined by

- \( \mathbb{Z}[x] \subseteq \text{SymExpr} \)
- for all \( f^n \) and all \( e_1, \ldots, e_n \in \text{SymExpr} \), we have \( f^n(e_1, \ldots, e_n) \in \text{SymExpr} \) (i.e., application of functions \( f^n \) to symbolic expressions yields symbolic expressions),
- for all \( e_1, e_2 \in \text{SymExpr} \), we have \( \text{Rnd}(e_1/e_2) \in \text{SymExpr} \).

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 \( \text{SymExpr} \), and return values of the symbolic predicate domain \( \text{SymPred} \), e.g., \( \leq : \text{SymExpr} \times \text{SymExpr} \rightarrow \text{SymPred} \). Moreover, let \( f^n \in \{ (\wedge (2)), (\vee (2)), (\neg (1)) \} \) 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 \( \text{SymPred} \).

**Definition 2.** The set of symbolic predicates of \( \text{SymPred} \), the symbolic predicate domain, is inductively defined as

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

The domain \( \text{SymPred} \) 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 \( S \subseteq \{ f : V \rightarrow \text{SymExpr} \} \). A clean slate state \( s \) maps all variables in its domain to the corresponding initial value variables: \( \forall v \in \text{Dom}(s) : s(v) = v \). Note that if we restrict our interest to a subset of \( V \) then states are partial functions.

**Definition 4.** A context \( c \in \mathbb{C} \subseteq [S \times \text{SymPred}] \) is defined by an ordered tuple \( [s, p] \) where \( s \) denotes a state, and pathcondition \( p \in \text{SymPred} \) describes the condition for which the variable bindings specified through \( s \) hold (cf. Blieberger, 2002; Führinger and Scholz, 2003). We make use of the functions \( p : C \rightarrow \text{SymPred} \) and \( s : C \rightarrow S \) 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 \( \text{assign} \), 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 \( \text{Id}[\text{id}] \).

\[
\begin{align*}
\text{assign}_i & : \text{Assignment} \rightarrow (C \rightarrow C) \\
\text{assign}(\text{Id} := \text{exp})(s) & = \lambda[s, p] : \text{Id}[\text{id}](s) = \text{exp}(s, p)(c)
\end{align*}
\]

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

\[
\begin{align*}
\text{pred}_t & : \text{Predicate} \rightarrow (C \rightarrow C) \\
\text{pred}_t[s, p \wedge \text{pred}](c) & = \lambda[s, p] : \text{pred}(s, p, \text{true})(c)
\end{align*}
\]

**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 \( F_e \subseteq \{ f : C \rightarrow C \} \). \( F_e \) contains the identity function \( \text{id} \) which can be envisioned as a null-operation without any computational effect. We require \( F_e \) to be closed under composition, which allows us to compose the computational steps of edges along program paths.

An edge transition function \( M_e : E \rightarrow F_e \) assigns a function \( f \in F_e \) to each edge \( e \in E \) of the CFG. The valuation function \( \text{edge} \) maps syntactic constructs associated with CFG edges to the respective valuation functions for branch predicates and side-effects, which allows us to specify functions \( f \in F_e \) as follows.

\[
\begin{align*}
\text{edge}_t & : \text{Predicate} \rightarrow (C \rightarrow C) \\
\text{edge}_t[s, p \wedge \text{pred}](c) & = \lambda[s, p] : \text{pred}(s, p, \text{true})(c)
\end{align*}
\]

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 \( F_e \).
4.3. Single-path symbolic execution

For a forward data-flow problem we can extend the transition function $M_c$ from edges to program paths $\pi$ as follows.

$$M_c(\pi) = \begin{cases} 
1, & \text{if } \pi \text{ is the empty path} \\
M_c(e_1) \circ \ldots \circ M_c(e_k), & \text{if } \pi = (e_1, \ldots, e_k) 
\end{cases} \quad (7)$$

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

In the previous example we determined the result of the transition function $M_c(e_1)$ 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 $\pi_1 = (e_1, e_2, e_3)$ we use function $M_c(\pi_1)$ to calculate the effect of symbolic execution along path $\pi_1$. We assume that the initial context $c_e$ passed as argument to $M_c(\pi_1)$ contains two program variables $u$ and $v$ holding their initial values $u'$ and $v'$. Then the contexts depicted in Fig. 3 illustrate the transformation of the initial context $c_e$ during symbolic execution along program path $\pi_1$. The context shown with node $n_e$ represents the result for $M_c(\pi_1)(c_e)$.

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

$$c_e' = \{(u', v'), (v, v'), u = v\}.$$

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 $n_e$ and hence the number of contexts of the symbolic solution at node $n_e$ 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

$$sc = \{c_1, \ldots, c_k, \ldots\} = \{[S_1, p_1], \ldots, [S_k, p_k], \ldots\}.$$

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

$$sc \in SC = \bigcup_{k=0}^{\infty} [S_k, p_k] \quad (8)$$

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

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

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 $F$ introduced in Section 4.2 from contexts to supercontexts, yielding a new function class $F_{sc}$:

$$F_{sc} \subseteq \{f_{sc} : SC \rightarrow SC\}. \quad (10)$$

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

$$\text{wrap} : (C \rightarrow C) \rightarrow (SC \rightarrow SC)$$

$$\text{wrap}(f_{sc})(sc) := f_{sc} \left( \bigcup_{i=0}^{\infty} [S_i, p_i] \right) = \bigcup_{i=0}^{\infty} f([S_i, p_i]).$$

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 $F$ on which $F_{sc}$ is based.

F1 $F_{sc}$ contains the identity function $\iota$.

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 $f^1 = f \circ f$):

$$f(x) = \bigcup_{i=0}^{\infty} f^i(x). \quad (11)$$

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

$$\forall f \in F_{sc} \text{ and } X \subseteq SC : f(\cup X) = \bigcup_{x \in X} f(x).$$
Based on the edge transition function \( M_e \) from Section 4.2 we define a new edge transition function \( M_{sc} \) that encapsulates the wrapping operator inside:

\[
\begin{align*}
M_{sc} & : E \to F_{sc} \\
M_{sc}(e) & := \text{wrap}(M_e(e)).
\end{align*}
\] (12)

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

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

\[
\begin{align*}
\phi(L) &= t, \quad (13) \\
\phi(e) &= M_{sc}(e) = f_{sc}, \quad (14) \\
\phi(P_1 * P_2) &= \phi(P_1) \cup \phi(P_2), \quad (15) \\
\phi(P_1 \cdot P_2) &= \phi(P_2) \circ \phi(P_1), \quad (16) \\
\phi(P_1) &= \phi(P_1)^* \quad (17)
\end{align*}
\]

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

\[
\phi(P)(x) = \bigcup_{\pi: l \in (P)} f_{sc}(x).
\]

Proof in Appendix B. Based on Lemma 1 we establish that the mapping \( \phi \) 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 \( P(n_0, n) \) be a path expression representing all paths from \( n_0 \) to \( n \). Then \( \text{map}(n) = \phi(P(n_0, n))(c_e) \), 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 \( P(n_0, n) \). 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 \( \phi \) by replacing (17) with

\[
\phi(P_1) = \phi(P_1)^* \quad (18)
\]

we introduce a new operation \( \otimes \) 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 recursions (Lueker, 1980) and is called a closure context. As will be pointed out below, a system of symbolic recursions 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 \( \otimes \) operation.

In analogy to the set \( V \) of program variables we define the set \( L \), \( \forall \in L = \omega \), of loop index variables. We use lowercase letters, e.g., \( i, m, n \), to denote elements from \( L \). 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 \( i \) is a symbolic upper bound, denoted by \( l_i \). This symbolic upper bound represents the number of loop iterations. Specifically, an upper bound of \( l_i = 0 \) implies zero loop iterations, as can be inferred from Fig. 4.4. Endless loops can be modeled by defining \( l_i := +\infty \).

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

- \( L \subseteq \text{SymExpr} \) (i.e., loop index variables are symbolic expressions), and
- for all \( v_i \in V \), and \( l \in L \), \( v_i(0) \in \text{SymExpr} \), \( v_i(l) \in \text{SymExpr} \), \( v_i(l + 1) \in \text{SymExpr} \), and \( v_i(l - 1) \in \text{SymExpr} \) (i.e., dereferencing the value of a program variable to specify a recursion 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 \( 0 \leq i \leq l_i \), with loop index variable \( l \in L \), and \( l_i \) 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 \( l \in L \), \( 0 \leq l \leq l_i \subset \text{SymPred} \) (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

\[
r := \bigcup_{1 \leq j \leq k} rs(l_j).
\]

Reurrence 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 \( \leq \) on the elements of a recurrence system set in order to obtain the semantics of a list.

**Definition 8.** A closure context \( \tau \) is an element of the set \( \Omega = S \times \text{SymPred} \times R \), denoted by \( [s, p, r] \). For a clean state closure con-

\(^2\) Since program contexts are one-element supercontexts, \( c_e \) is a valid argument for functions from class \( F_e \).

\(^3\) 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 \( L \ldots U \) denotes the interval \([L, U] \).
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 \( r = \emptyset \). A supercontext consisting of a finite number of closure contexts is denoted by \( \mathcal{SC} \), for the set of all such finite supercontexts we write \( \mathcal{SC} \).

**Definition 9.** We define operation \( \circ \) of (18) in terms of the input/output-behavior of the function resulting from the application of operation \( \circ \) to \( \phi(P_1) \), that is, \( \phi(P_1)^{\circ} \). 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 \( \mathcal{C}_m = [s_m, p_m, r_m] \) we define the properties of the closure context \( \mathcal{C}_{out} = [s_{out}, p_{out}, r_{out}] \) resulting from the application of \( f \) to \( \mathcal{C}_m \), that is,

\[
\mathcal{C}_{out} = \phi(P_1)(\mathcal{C}_m) = \phi(P_1)^{\circ}(\mathcal{C}_m) = f^{\circ}(\mathcal{C}_m).
\]

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 \( \mathcal{C}_0 = [s_0, p_0, r_0] \) and compute the result of symbolic evaluation of one iteration of the loop body, denoted by \( \mathcal{C}_1 \).

\[
\mathcal{C}_1 = [s_1, p_1, r_1] = f(\mathcal{C}_0).
\]

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

\[
\sigma_{s,e} = [v_1 \mapsto v_1(e), \ldots, v_j \mapsto v_j(e)], \quad \text{with } v_i \in \text{Dom}(s).
\]

What follows is the description of \( \mathcal{C}_{out} \) in terms of its state \( s_{out} \), its pathcondition \( p_{out} \), and its recurrence system \( r_{out} \).

**State:** The state \( s_{out} \) is computed from \( s_m \) 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 \( l \).

\[
\forall v_i \in \text{Dom}(s_m) : s_{out} := s_m[v_i \mapsto v_i(l)]
\]

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

**Pathcondition:** The pathcondition \( p_{out} \) of closure context \( \mathcal{C}_{out} \) has the form

\[
p_{in} \land (0 \leq l \leq l_m) \land \bigwedge_{1 \leq i \leq l} p(l - 1).
\]

Therein the term \( p_{in} \) constitutes the pathcondition of closure context \( \mathcal{C}_m \). 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 \([0, l_m]\). 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 \( l \) \( \text{th} \) instance corresponds to \( \sigma_{s_m, l(l-1), p_1} \).

An example will illustrate this. Assume the pathcondition \( p_1 = j \leq m \) from Fig. 4. After \( l > 0 \) iterations the third term in the above equation will read

\[
j(0) \leq m(0) \land j(1) \leq m(1) \land \ldots \land j(l-1) \leq m(l-1) = \bigwedge_{1 \leq j \leq l} j(l - 1) \leq m(l - 1).
\]

**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.

\[
\begin{align*}
\forall v_i \in IV : & \quad v_i(0) := s_m(v_i) \\
& \quad v_i(l + 1) := \sigma_{s_m, l(l)}(v_i(l))
\end{align*}
\]

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 \( \mathcal{C}_m \) 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 \( v_i \in IV \) by their recursive counterparts \( v_i(l) \), we obtain the bindings after iteration \( l + 1 \), denoted by \( v_i(l+1) \). 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 \( \mathcal{C}_{out} \) 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 \( \phi \) from (18) works from innermost
to outmost 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 $v_1$ is replaced by the variable’s recurrence relation over the loop index variable of the containing loop. This corresponds to one application of substitution $\sigma_{\text{loop}}$ 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 $e_1’ = (e_2’ \cdot e_3’)$ of type $(n_e, n_1)$. From the clean slate context $c_{\text{out}} = [s, p, r] = \langle \{ (b, b), (d, d), (j, j), (m, m) \}, \text{true}, \varnothing \rangle$, we compute $\phi(e_1’ = (e_2’ \cdot e_3’)) = (f_{e_2} \circ f_{e_3}) \circ f_1(c_{\text{out}})$. Function application $f_{e_3}$ yields the context $c_{\text{out}} = \langle \{ (b, b + 1), (d, d), (j, j + b), (m, m) \}, \text{true}, \varnothing \rangle$, which reduces our computation to $f_{e_2} \circ f_1(c_{\text{out}})$. To apply operation $\circ$ 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 context by defining $\bar{c}_{\text{out}} := c_{\text{out}}$ and proceed with the calculation of $\bar{c}_{\text{out}} = (f_{e_2} \circ f_1)(\bar{c}_{\text{out}}) = \langle \{ (b, b + 1), (d, d), (j, j + b), (m, m) \}, \text{true}, \varnothing \rangle$. The closure context $c_{\text{out}}$ from the computation of $\bar{c}_{\text{out}} = (f_{e_2} \circ f_1)(\bar{c}_{\text{out}})$ can then be described in terms of its state $s_{\text{out}}$, its pathcondition $p_{\text{out}}$, and its recurrence system set $r_{\text{out}}$. The loop index variable for this loop is $l$.

**State:** The state of $c_{\text{out}}$ is obtained from the state of $\bar{c}_{\text{out}}$ by replacing the symbolic expressions that describe the values of the induction variables $v_1 \in IV = \{ d, j \}$ by the value of the recurrence relation for $v_1$ over loop index variable $l$. Hence we get $s_{\text{out}} = \langle \{ (b, b + 1), (d, d), (j, j + b), (m, m) \}, \text{true}, \varnothing \rangle$.

**Pathcondition:** According to (23) we get the pathcondition $p_{\text{out}} = \langle \{ (b, b + 1), (d, d), (j, j + b), (m, m) \}, \text{true}, \varnothing \rangle$.

**Recurrence system:** According to (24) we arrive at the one-element recurrence system set $r$, with $s_{\text{out}} = s(\bar{c}_{\text{out}})$ and $s_1 = s(t(\bar{c}_{\text{out}}))$ already substituted.

$$r = \begin{cases} d(0) + d(1) + d(l) \quad & (1a) \\ j(0) + j(1) \quad & (1b) \\ r_e \quad & (2) \end{cases}$$

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

$$r = \begin{cases} d(0) + d(1) + d(l) \quad & (1a) \\ j(0) + j(1) \quad & (1b) \\ r_e \quad & (2) \end{cases}$$

(25)

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

$$\langle \{ (b, b + 1), (d, d), (j, j + b), (m, m) \}, \text{true} \rangle$$

(26)

as the solution for the closure context $c_{\text{out}}$, which is also the MOP-solution for node $n_1$. The intuitive meaning of this closure context unveils if we consider the range expression $0 \leq l \leq l_{\text{max}}$ that is part of its pathcondition: as loop index variable $l$ ranges from 0 to $l_{\text{max}}$, the recurrence system $r$ generates the variable bindings of the respective context $c_l$ valid after loop iterations, i.e., $c_l = (f_{e_2} \circ f_{e_3}) \circ f_1(c_{\text{out}})$. Hence the closure context $c_{\text{out}}$ represents a total number of $l_{\text{max}} + 1$ contexts valid at node $n_1$. Formally the closure context $c_{\text{out}}$ can be viewed as a predicate $\text{valid}(c_{\text{out}})$, where the set $\{ x \mid 0 \leq x \leq l_{\text{max}} \} \subseteq \mathbb{N}$ 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 $b + 1$ and $m$, 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 $l_{\text{max}}$ 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 $n_e$.)

Upon exit of a loop via edge $e$, the pathcondition $p$ associated with $e$ implies that $l = l_{\text{max}}$. In other words, the conjunction of $p$ and the pathcondition of a closure context from node $n_1$ 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 $l = l_{\text{max}}$ simplifies the closure context valid at node $n_1$ to

$$\langle \{ (b, b + 1), (d, d), (j, j), (m, m) \}, \text{true} \rangle$$

(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 $l_{\text{max}}$ for which the following holds:

$$e = \min \{ | r_e | \}$$

(28)

Plugging in the upper bound $l_{\text{max}}$ into the recurrence system set, we can compute the values $d(l_{\text{max}})$ and $j(l_{\text{max}})$ that induction variables $d$ and $j$ take on.
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 \( p_{0}(1) = e(1) > 0 \) \& \( e(1) \mod 2 = 1 \) and \( p_{1}(1) = e(1) > 0 \) \& \( e(1) \mod 2 \neq 1 \). Pathconditions for up to \( 1 \times 3 \) 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 as \( l = 3 \) 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 \( l = 3 \) iterations with the then-branch taken the initial 2 iterations is \( p_{0}(0) \times p_{1}(1) \times p_{1}(2) \). This execution corresponds to the overall loop-path \( p = 1 \) (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

\[
\begin{align*}
\left[(b, b+1), \left(d, d \cdot \frac{m-j+1}{b+1} \right)\right], \\
\left[(j, j+(b+1) \cdot \frac{m-j+1}{b+1} \right), (m, m)\right], \quad \left(j \leq l \right) \lor \left(j > l \right) \land \left(m \leq l \mid r \right).
\end{align*}
\]

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 \( f_{\otimes} \) 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.,

\[
\mathcal{X} = f_{\otimes}(c_{0}) = \bigcup_{s_{n} = 0}^{x_{n}} [s_{x}, p_{x}, r_{x}].
\]

where \( x_{n} \) 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.

\[
\begin{align*}
\forall v_{1} \in IV : v_{1}(0) := s_{a1}(v_{1}) & \quad (1) \\
\forall v_{1} \in IV : \forall [s_{x}, p_{x}, r_{x}] \in \mathcal{X} : v_{1}(1) = s_{a1}(x_{y} v_{1}) & \quad if \ s_{a1}(p_{x}) \quad (2) \\
rc := \bigcup_{1 \leq x \leq x_{n}} s_{a1}(p_{x}) & \quad (3) (31)
\end{align*}
\]

To compute closure contexts for multi-path loops, the set of symbolic expressions is extended by loop path selectors \( p \in \mathcal{P} \), i.e., \( \mathcal{P} \subset \text{SymExpr} \). Given \( l \) iterations of a loop, the loop path selector allows us to select one particular path from the \( x_{n} \) possible paths. Note that each of those \( x_{n} \) 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

\[
p_{0} \land (0 \leq l \leq \omega_{0}) \land (0 \leq p < x_{m}) \land \bigwedge_{1 \leq e \leq l} p(l - 1, p).
\]

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

\[
p(l - 1, p) = \sigma_{s_{a1}((l-1)p_{x})},
\]

where index \( k \) of pathcondition \( p_{x} \) is

\[
k = \left( p \times x_{n}^{(l-1)} \right) \mod x_{n}.
\]

As an example, consider overall loop-path \( p = 5 \) from Fig. 5. Inserting \( p = 5, x_{n} = 2, l = 3 \) and \( l' = (1, 2, 3) \) in (34) gives \( k = (1, 0, 1) \). Inserting pathconditions \( p_{x} \) in (33) yields

\[
\begin{align*}
p(0, 5) &= \sigma_{s_{a1}(0p_{x})} = e(0) > 0 \land e(0) \mod 2 \neq 1 \quad (33) \\
p(1, 5) &= \sigma_{s_{a1}(1p_{x})} = e(1) > 0 \land e(1) \mod 2 = 1 \\
p(2, 5) &= \sigma_{s_{a1}(2p_{x})} = e(2) > 0 \land e(2) \mod 2 \neq 1,
\end{align*}
\]

where \( e() \) refers to the recurrence system from (31). The overall loop-path for three iterations and loop-path selector \( p = 5 \) is \( p(0, 5) \land p(1, 5) \land p(2, 5) \).

5.3.1. Example

Given two positive numbers \( x \) and \( n \), the Ada95 procedure depicted in Fig. 6 computes \( x^{x} \). 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 \( e_{1}((e_{2} + e_{3}) \cdot e_{4} \cdot e_{5})^{7} \cdot e_{6} \), we will analyze the subexpression
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 \(\mathcal{C}\), symbolic analysis along edge \(e_1\) gives

\[
\mathcal{C}_{e_1} = \phi(e_1)(\mathcal{C}) = \{(y, 1), (e, e), (f, f), (1, 0), \text{true}, \varnothing\}.
\]  

(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 \(\mathcal{C}\), symbolic analysis of one iteration of the loop body gives

\[
\mathcal{C}_{e} = \phi((e_2 + e_3) \cdot e_4 \cdot e_5)(\mathcal{C}) = \{(y, y), (e, e), (1, 0), \varnothing\}
\]

\[
\cup \{(y, y), (e, e), (f, f), (1, 0), \varnothing\}.
\]

(36)

We combine closure context \(\mathcal{C}\) from (35) with the analysis result of the loop body, i.e., (36), to derive the conditional recurrence relation from (31):

\[
y(0) := 1
\]

\[
e(0) := e
\]

\[
f(0) := f
\]

\[
y(l + 1) := \begin{cases} f(l) \cdot y(l) & \text{if } e(1) \mod 2 = 1 \\ y(l) & \text{else} \end{cases}
\]

\[
e(1) := \begin{cases} e & \text{if } e < 0 \\ f & \text{else} \end{cases}
\]

\[
f(l) := f^2
\]

\[
c := e(1) > 0
\]

(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.

\[
\{(y, y), (e, e), (f, f), (1, 0), \text{true}, \varnothing\}
\]

\[
(0 \leq 1 \leq 10) \land (0 \leq p < 2^2) \land \bigwedge_{1 \leq i \leq 5} f(i) \land p\} \]  

(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 \(y^f(\text{line } 11)\) and \(f^f(\text{line } 13)\) are potentially outside the range of the Ada95 type \(\text{positive}\). (Note that the Ada95 type \(\text{positive}\) ranges from 1 to \(2^{63} - 1\) on a 64-bit architecture. Its lower and upper bounds are denoted by \(\text{positive'first}\) and \(\text{positive'last}\).) If the compile-time constant \(\text{Max}\) is small enough s.t.

\[
\text{Max}^{\text{Max}} \leq \text{positive'last}.
\]

(39)

then the upper bound of type \(\text{small_int}\) 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 \(\text{Max}^{\text{Max}}\cdot \text{Max}^{\text{Max}}\), overflow of expression \(f^f\) will not occur if

\[
\text{Max}^{\text{Max}} \leq \text{positive'last}.
\]

(40)

Moreover, expressions \(y^f\) and \(f^f\) are of type \(\text{positive}\) 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 ε > 0 and ε > 0, ε ∈ mod 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, we can plug in the upper bound ε = Max into the closure context from (38) and apply standard techniques from Fahrringer and Scholz (2003), Blieberger (1994) to derive \( l_u = \lfloor \text{ld Max} \rfloor + 1 \) as the upper bound for the number of loop iterations.

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

\[
f(l_u) = \sum_{j=0}^{l_u-1} \frac{\text{Max}}{j+1} \leq \sum_{j=1}^{\text{Max}} \leq \text{Max}^2 \text{Max}.
\]

This concrete value for the upper bound of expression \( f^*f \) is then compared with the value \( \text{positive last, according to } (40) \). If the upper bound is smaller than the value for \( \text{positive last, bounds-check R2} \) 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; Haghhiat 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 \( \text{order-sorted} \) framework, because we have \( \mathbb{Z} \subseteq \mathbb{Z}[x] \subseteq Q(\mathbb{Z}[x]) \). 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:

\[
R(n_e) = \Lambda \quad \text{(41)}
\]

\[
R(n) = R(n', R^2_{n,n}) \quad \text{(42)}
\]

Therein \( R_{n}^{M} \) 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 \( \text{intermediate} \) 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 loop-breaking rule that allows us to handle circularities in CFGs. We replace (41) and (42) by the following normal form for our data-flow problem,

\[
E : R(n) = \begin{cases} 
\sum_{m \in M \subseteq \text{sub}(\text{idom}(n))} [R(m) \cdot R^2_{\text{dom}(n),m} ; [m]] & \text{if } n \neq n_e, \\
\Lambda & \text{else,}
\end{cases}
\]

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.,

\[
E : R(n) = R(\text{idom}(n)) \cdot R^2_{\text{dom}(n),n} + R(n) \cdot R^2_{\text{dom}(n),n} \cdot R^2_{\text{sub}(n)}, n
\]

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

\[
e : R(n) = R(\text{idom}(n)) \cdot R^2_{\text{dom}(n),n} \cdot R^2_{\text{sub}(n)}, n
\]

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 program 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 \( \text{nc}(e)=1 \), \( \text{nc}(P_1 + P_2) = \text{nc}(P_1) + \text{nc}(P_2) \), and \( \text{nc}(P_1 \cdot P_2) = \text{nc}(P_1) \cdot \text{nc}(P_2) \). Every such program path induces the generation of one closure context during symbolic analysis.\(^7\) Our accumulated \( \text{nc} \) metric (\( \text{ancc} \))

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

\(^7\) Hence the name \( \text{nc} \) which stands for \text{number of closure contexts}.\)
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 $(n_l, n_h)$. 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 program 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 indexed arrays and access 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 Wonacott (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 analysis 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; Hajighat 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 supercontextuals an ideal symbolic intermediate representation for all domain-specific symbolic program analysis.

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 program language constructs, and (2) the application of our method to domain-specific symbolic 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 $\div$ of the quotient field $Q(\mathbb{Z}[x])$, we can model the integer division of two symbolic expressions $e_1(x)$ and $e_2(x)$, $e_2(x) \neq 0$, as

$$e_1(x) \div e_2(x) = \text{Rnd}(e_1(x) / e_2(x)), \quad (A.1)$$

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

$$e_1(x) \text{rem} e_2(x) = e_1(x) - e_2(x) \cdot (e_1(x) \div e_2(x)) \quad (A.2)$$

for the symbolic remainder operator $\text{rem}$. Since the rounding operation $\text{Rnd}$ involves floor and ceiling functions, the expressions resulting from the symbolic division and remainder operations are infeasible: 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:** $e_1(x) = a(x)$ and $e_2(x) = b(x) \neq 0$ 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

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

where $\text{deg}(\ldots)$ denotes the degree of a polynomial. Contrary to the integer division, the quotient and remainder polynomials satisfying the above relation are unique. They can be determined by polynomial long division, using e.g., Algorithm D of 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>
<thead>
<tr>
<th>Case</th>
<th>Expressions</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a(x) \div b(x)$</td>
<td>$r(x) \rem b(x)$</td>
</tr>
<tr>
<td>2</td>
<td>$q(x) - 1$</td>
<td>$r'(x) + b(x)$</td>
</tr>
<tr>
<td>3</td>
<td>$q(x) - 1$</td>
<td>$r'(x) + b(x)$</td>
</tr>
<tr>
<td>4</td>
<td>$q(x) - 1$</td>
<td>$r'(x) - b(x)$</td>
</tr>
<tr>
<td>5</td>
<td>$q(x) - 1$</td>
<td>$r'(x)$</td>
</tr>
</tbody>
</table>

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

$$|r(x)| < \frac{1}{2}.$$  \hspace{1cm} (A.4)

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

$$|\text{Rnd}(\frac{a(x)}{b(x)}) - \text{Rnd}(q(x))| \leq 1,$$

with $\text{Rnd}$ denoting the round to nearest integer rounding function.

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

$$a(x) = q(x) + r(x) \div b(x).$$

Due to the side-condition stated in Eq. (A.4), the result of dividing polynomial $a(x)$ by $b(x)$ in the quotient field $Q(\mathbb{Q}[x])$ 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 $u = \text{Rnd}(q(x))$. Then $[a(x) / b(x)] \in [q(x) - 0.5, q(x) + 0.5]$ implies that $\text{Rnd}(a(x) / b(x)) \in \{u_1, u_2\}$, and due to the side-condition, $\text{Rnd}(a(x) / b(x)) \in \{u_1, u_2\}$. \hfill $\square$

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

$$\text{Rnd}(q(x)) - 1, \text{Rnd}(q(x)), \text{Rnd}(q(x)) + 1.$$

Let $q'(x) = \text{Rnd}(q(x))$ 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 $a(x) = 3x^3 + x^2 + x + 5$ and $b(x) = 5x^2 - 3x + 1$. Performing polynomial long division results in the quotient polynomial

$$q(x) = \frac{3x^3 + 9x^2 + 12x + 116}{5} \div \frac{5x^2 - 3x + 1}{625} = \frac{3x^3 + 9x^2 + 12x + 116}{5},$$

and the remainder polynomial

$$r(x) = \frac{913x - 3009}{625}.$$  \hspace{1cm} (A.6)
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 \( 2 \text{div} 9 = 0 \) and \( 2 \text{rem} 9 = 2 \) for quotient and remainder computed from the integer arithmetic of the Flow concrete semantics (cf. also Fig. A.9(b)).

Furthermore, we have \( q'(-1) = \text{RndN}(-0.1504) = 0 \), 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:** \( e_1(x) = a(x) \) and \( e_2(x) = b(x) \neq 0 \) are multivariate integer-valued polynomials in the polynomial domain \( \mathbb{Q}(x) \) over the field \( \mathbb{Q} \) of rational numbers, and \( a(x) \) is a multiple of \( b(x) \), i.e., \( a(x) = q(x) \cdot b(x) \). Since \( \mathbb{Q} \) is a field, it follows from (Geddes et al., 1992, Theorem 2.7 (v)) that \( \mathbb{Q}(x) \) 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 \( a(x) \) and \( b(x) \) in a unique way into irreducible polynomials and thus determine whether polynomial \( a(x) \) is a multiple of \( b(x) \). If this is the case, then the result of dividing \( a(x) \) by \( b(x) \) is \( q(x) \), and the remainder is zero.
It should be noted that the quotient polynomial \( q(x) \) need not be integer-valued; in this case we have to apply the rounding function \( \text{RND} \) to obtain an integer-valued result.

If \( a(x) \) is not a multiple of \( b(x) \), 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 \( \sigma_e \) to denote the standard-semantic side-effect associated with an edge \( e \), and \( \sigma_b \) to denote its symbolic counterpart. Likewise we will use \( \text{pred}_e \) and \( \text{pred}_b \) to distinguish between standard-semantic and symbolic branch predicates.

**Definition 10.** Given the set \( \mathbb{V} \) of initial value variables and \( \text{SymExpr} \), the domain of integer-valued symbolic expressions. A \text{SymExpr}-\text{substitution} – or simply substitution, if the domain \text{SymExpr} is irrelevant or clear from the context of use – is a function \( \sigma : \mathbb{V} \to \text{SymExpr} \) such that \( \sigma(x) \neq x \) for only finitely many \( x \). The finite set of variables that \( \sigma \) does not map to themselves is called the domain of \( \sigma \): \( \text{Dom}(\sigma) := \{ x \in \mathbb{V} \mid \sigma(x) \neq x \} \). If \( \text{Dom}(\sigma) = \{ x_1, \ldots, x_n \} \), then we may write \( \sigma \) as

\[
\sigma = [x_1 \to \sigma(x_1), \ldots, x_n \to \sigma(x_n)].
\]

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

\[
\hat{\sigma}(x) = \sigma(x), \quad \text{for } x \in \mathbb{V} \quad \text{(B.1)}
\]

\[
\hat{\sigma}(f(e_1, \ldots, e_n)) = f(\hat{\sigma}(e_1), \ldots, \hat{\sigma}(e_n)), \quad \text{for a non-variable expression} \quad \text{(B.2)}
\]

From Eq. (B.2) it follows that the extension \( \hat{\sigma} \) is an endomorphism on the domain \( \text{SymDom} \) that coincides with the identity mapping on almost all variables. To simplify notation we distinguish between a substitution \( \sigma \) and its extension \( \hat{\sigma} \) only at places where this distinction is crucial. Note that the concept of \text{SymExpr}-substitutions is closely related to substitutions on terms presented by Baader and Nipkow (1998).

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

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

\[
F_e \subseteq \{ f_e : \text{Env} \times \mathbb{B} \to \text{Env} \times \mathbb{B} \}
\]

with its member functions \( f_e \) defined as

\[
f_e = M_e(e)[\text{env}, b] = ([\sigma_e(e)(\text{env}), b \land \text{pred}_e(e)(\text{env})]), \quad \text{(B.3)}
\]

where \( M_e : E \to F_e \) represents the standard semantic edge transition function. It is easy to see that for the domain of the transition function \( \delta \) the function class of Eq. (B.3) is equivalent to \( \delta \) in terms of the computed environment \( \sigma_e(e)(\text{env}) \) as well as the computed program condition \( b \land \text{pred}_e(e)(\text{env}) \). (The latter due to the fact that with the transition function \( \delta \) we have the implicit condition which is always true, since \( \delta \) is only defined on edges \( e \) such that \( \text{pred}_e(e)(\text{env}) = \text{true} \). On the other hand, for values outside the domain of \( \delta \) 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 \( \text{pred}_e(e)(\text{env}) = \text{false} \).

On the analogy of symbolic program execution we can extend the standard semantic edge transition function \( M_e \) from edges \( e \) to program paths \( \pi \). Resembling Eq. (7), we get

\[
M_e(\pi) = \begin{cases} \{i, \text{if } \pi \text{ is the empty path} \\
M_e(e_1) \to M_e(e_{i-1}) \to \cdots \to M_e(e_1) \to M_e(e_{i}) \\
\text{for } \pi = (e_1, \ldots, e_i) \end{cases}
\]

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

**Definition 13.** Function \( \text{sym} \) transfers an extended environment \( [\text{env}, b] \) to a program context in the symbolic domain.

\[
\text{sym} : \text{Env} \times \mathbb{B} \to C
\]

\[
\text{sym}([\text{env}, b]) = [\lambda][\text{env}, b] = \lambda[\text{env}, b]
\]

\[
\hat{\text{sym}} : \forall \nu \in \text{Dom}(\text{env}) : v \mapsto \sigma_{\text{env}}(s(v)), \sigma_{\text{env}}(p)[\text{env}, c]
\]

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

**Definition 14.** Function \( \text{con} \) takes an environment \( \text{env} \) and a program context \( c \) as input and returns an extended environment \( \text{env} \) which is the result of evaluation of \( c \) with the values provided by \( \text{env} \).

\[
\text{con} : \text{Env} \to \text{Env} \times \mathbb{B}
\]

\[
\text{con}([\text{env}, c]) = [\lambda\{\text{env}, c\} = \lambda[\text{env}, [s, b]]
\]

\[
\text{con}([\forall v \in \text{Dom}(s) : v \mapsto \sigma_{\text{env}}(s(v))], \sigma_{\text{env}}(p))\left[\text{env}, c\right]
\]

Therein the \text{SymExpr}-substitution \( \sigma_{\text{env}} \) can be written as

\[
\sigma_{\text{env}} = [v_1 \mapsto \text{env}(v_1), \ldots, v_n \mapsto \text{env}(v_n)], \quad \text{(B.4)}
\]

which means that occurrences of the initial value variables \( v_i \) are replaced by the concrete value of the corresponding variable \( c(v_i) \) in \( \text{env} \) (recall that \text{SymDom} contains expressions in terms of the initial value variables from \( \mathbb{V} \)). For this to work we need the side-condition that the domains of \( \text{env} \) and \( s \) coincide, that is, \( \text{Dom}(\text{env}) = \text{Dom}(s) \).

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 \( [\text{env}, b] \) passed as argument to function \( M_e(e) \) representing the standard semantic computational effect associated with edge \( e \). This results in a new extended environment \( [\text{env}_1, b \land b_1] \) 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.](image-url)
with Definition 12 this coincides with the transition function \( \delta \) iff \( b = \text{true} \) and \( b_1 = \text{pred}_1(e) \).

On the other hand, transferring the initial extended environment \( [\text{env}, b] \) via function sym to the symbolic domain gives us a program context \( [s, b] \). We can apply function \( M_0(e) \) that represents the computational effect in the symbolic domain associated with edge \( e \). (Note that we use \( M_0 \) to distinguish the edge transition function into the symbolic domain from its standard semantic counterpart \( M_0 \).) The result is a program context \( [s_1, b \land p_1] \) 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 \( \sigma_\text{sym}(e) \) of edge \( e \) (see Eq. (6)).
- For the pathcondition \( b \land p_1 \), we have \( b \in \mathcal{B} \) and \( p_1 \in \text{SymPred} \). \( p_1 \) represents the branch predicate \( \text{pred}_1(e) \) of the symbolic domain associated with edge \( e \) (see again Eq. (6)).

Transforming the program context \( [s_1, b \land p_1] \) from the symbolic to the standard semantic domain via function \( \text{con} \) uses environment \( \text{env} \) of the initial extended environment \( [\text{env}, b] \) 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 \( G = (N, E, n_s, n_e) \). For any edge \( e \in E \) of \( G \), the condition

\[
M_0(e)([\text{env}, b]) = \text{con}([\text{env}, M_0(e)(\text{sym}([\text{env}, b]))])
\]

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

**Proof.** It follows from the definitions of functions \( \text{sym} \) and \( \text{con} \) that the transformation of an extended environment \( [\text{env}, b] \) into the symbolic domain is invertible using function \( \text{con} \) and the contained environment \( \text{env} \), that is

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

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 \( i \) 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 \( \text{id}[\text{id}] \)), 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 \( \text{env}_{V_1} \) coincides with environment \( \text{env} \) in all but one program variable, and that this is consistent with commutation. Without loss of generality we may assume \( V_1 \) as the updated program variable.

In order to prove that concrete and symbolic execution commute on environment \( \text{env}_{V_1} \), we must show that they update program variable \( V_1 \) consistently. The notion of consistent update can be formalized based on the initial environment \( \text{env} \), the corresponding state \( s \), the concrete and symbolic valuation functions computing the new value for \( V_1 \) (cf. line 3 of the corresponding definitions of Table B.2), and the substitution \( \sigma_\text{env} \) (cf. Eq. (B.4)) that is part of function \( \text{con} \). Due to the involved concrete and symbolic valuation functions \( \text{exp} \) and \( \text{exp}_s \), the notion of consistent update is named \( \text{exp}^c \)-consistency.

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

Recall that in this equation “exp” denotes a derivation tree corresponding to a Flow expression. We prove \( \text{exp}^c \)-consistency by structural induction on the Flow expression derivation trees according to lines 5–10 of the \( \text{exp} \) and \( \text{exp}_s \) 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 \( \text{exp}_s([\text{exp}](\text{env})) = \sigma_\text{env}(\text{exp}_s([\text{exp}](\text{env}))) \) holds if \( \text{env}(\text{id}[\text{id}]) = \sigma_\text{env}(\text{sym}([\text{id}[\text{id}]])) \) which follows from Eq. (B.5). Furthermore we must show that \( \text{exp}_s([\text{num}](\text{env})) = \text{exp}_s([\text{num}](\text{env})) \) which is trivially true since the valuation function \( \text{num}([\text{num}]) \) for numbers is common to the concrete and symbolic domain (cf. Table B.2).

**Induction:** In the inductive step we must show that

\[
\text{exp}_s([\text{exp}_1 \text{binop} \text{exp}_2](\text{env})) = \sigma_\text{env}(\text{exp}_s([\text{exp}_1 \text{binop} \text{exp}_2](\text{env}))).
\]

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

\[
e_{c_2} = \text{exp}_c([\text{exp}_1](\text{env})), \quad e_{s_2} = \text{exp}_s([\text{exp}_2](\text{env})),
\]

we must show that

\[
e_{c_1} \text{binop} [\text{binop} e_{c_2}] = \sigma_\text{env}(e_{s_1} \text{binop} [\text{binop} e_{s_2}]). \tag{B.7}
\]
For the inductive step we may assume that \( e_1 = \sigma_{env}(e_1) \), and that \( e_2 = \sigma_{env}(e_2) \). Based on the binary operator \( \text{binop}[\text{binop}] \) we can then distinguish the following three cases.

**Case 1:** Let \( \text{binop}[\text{binop}] \in \{+, -, \ldots \} \). From Definition 10 of substitution \( \sigma_{env} \) we may write

\[
\sigma_{env}(e_1) \text{ binop } [\text{binop}] \sigma_{env}(e_2) = \sigma_{env}(e_1) \text{ binop } [\text{binop}] \sigma_{env}(e_2).
\]

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

\[
e_1 \text{ binop } [\text{binop}] e_2 = \sigma_{env}(e_1) \text{ binop } [\text{binop}] \sigma_{env}(e_2).
\]

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:** \( \text{binop}[\text{binop}] = \text{div} \) and \( \text{binop}[\text{binop}] = \text{div} \).

Further simplifications yield

\[
\sigma_{env} (e_1) \text{ binop } [\text{binop}] \sigma_{env} (e_2) = \sigma_{env}(\text{Rnd}(e_1) / e_2) = \text{Rnd}(\sigma_{env}(e_1) / \sigma_{env}(e_2)) = \begin{cases} \sigma_{env}(e_1) / \sigma_{env}(e_2), & \text{if } \sigma_{env}(e_1) / \sigma_{env}(e_2) \geq 0 \\ \sigma_{env}(e_1) / \sigma_{env}(e_2) \end{cases}.
\]

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:** \( \text{binop}[\text{binop}] = \text{rem} \) and \( \text{binop}[\text{binop}] = \text{rem} \).

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 \( \text{exp}_2 \)-consistency of Flow expression derivation trees as postulated in Eq. (B.6). This means that concrete and symbolic execution commute on environment \( env \), as depicted in Fig. B.10, and it remains to show that they also commute regarding the pathcondition \( b \land b_1 \).

From Fig. B.10 we observe that with the pathcondition \( b \land b_1 \) of the extended environment \( [env, b \land b_1] \) 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 \( b \land b_1 \), we must show that they commute on \( b \) (Strictly speaking, we only need to show that they commute on \( b \) if \( b = \text{true} \), for \( b_1 = \text{false} \).)

On the analogy of \( \text{exp}_2 \)-consistency we proceed with the definition of \( \text{pred}_2 \)-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.

\[
\text{pred}_2 \text{-consistency } \iff b \land \text{pred}_2[[\text{pred}]](\text{env}) = \text{pc}(\sigma_{env}(\text{pred}_2 [[\text{pred}]])(\text{env})).
\]

We will establish \( \text{pred}_2 \)-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 \( \text{pred}_2 \) 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 \( \text{pred}_2 \)-consistency of Flow predicates as outlined above.

### Table B.3

<table>
<thead>
<tr>
<th>Comparison: concrete vs. symbolic branch-predicates.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Concrete domain</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

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

\[
b \land \text{pred}_2[[\text{true}]](\text{env}) = \text{pc}(\sigma_{env}(\text{pred}_2[[\text{true}]])(\text{env}))) \\
b \land \text{true} = \text{pc}(\sigma_{env}(\text{true}))(\text{env}))
\]

Likewise for Case 2:

\[
b \land \text{pred}_2[[\text{false}]](\text{env}) = \text{pc}(\sigma_{env}(\text{pred}_2[[\text{false}]])(\text{env}))) \\
b \land \text{false} = \text{pc}(\sigma_{env}(\text{false}))(\text{env}))
\]

false \land \text{false} = \text{false}.

Case 6 requires us to show that

\[
b \land \text{pred}_2[[\text{rel} \lor \text{exp2}[[\text{exp}]](\text{env})] = \text{pc}(\sigma_{env}(\text{pred}_2[[\text{rel} \lor \text{exp2}[[\text{exp}]](\text{env})))
\]

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

\[
e_1 = \text{exp1}[[\text{exp}]](\text{env}), \quad e_2 = \text{exp1}[[\text{exp}]](\text{env}), \quad e_1 = \text{exp2}[[\text{exp}]](\text{env}), \quad e_2 = \text{exp2}[[\text{exp}]](\text{env}),
\]

we get

\[
b \land (e_1 \land \text{rel} \lor \text{exp1}[[\text{exp}]](\text{env}) = \text{pc}(\sigma_{env}(\text{rel} \lor \text{exp1}[[\text{exp}]](\text{env})))
\]

It is the \( \text{exp}_2 \)-consistency that we have stated in Eq. (B.6) which provides that \( e_1 = \sigma_{env}(e_1) \), and that \( e_2 = \sigma_{env}(e_2) \). Knowing that the relational connective \( \text{rel} \lor \text{exp1}[[\text{rel} \lor \text{exp}]](\text{env}) \) 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
\[
\begin{align*}
\mathit{B} \land (\text{pred}_1[[\text{pred}_1 \land \text{pred}_2]]([\text{env}]) &= \text{ pc}(\sigma_{\text{env}}(\text{pred}_1[[\text{pred}_1 \land \text{pred}_2]])[s, \mathit{B}])).
\end{align*}
\]
Substituting the right-hand sides of Case 3 from Table
B.3 and using the following shorthand notations
\[
\begin{align*}
\text{pc}_1 &= \text{ pred}_1[[\text{pred}_1]]([\text{env}]), \\
\text{pc}_2 &= \text{ pred}_2[[\text{pred}_2]]([\text{env}]), \\
\text{pc}_{12} &= \text{ pred}_1[[\text{pred}_2]]([s, \text{true}]),
\end{align*}
\]
we get
\[
\begin{align*}
\mathit{B} \land (\text{pc}_1 \land \text{pc}_2) &= \text{ pc}(\sigma_{\text{env}}(s, \mathit{B} \land (\text{ pc}(\text{pc}_1) \land \text{ pc}(\text{pc}_2)))) \\
&= \mathit{B} \land (\sigma_{\text{env}}(\text{pc}(\text{pc}_1)) \land \sigma_{\text{env}}(\text{pc}(\text{pc}_2))).
\end{align*}
\]
The inductive hypothesis implies that \text{true} \land \text{pc}_1 = 
\sigma_{\text{env}}(\text{pc}(\text{pc}_1)) and \text{true} \land \text{pc}_2 = \sigma_{\text{env}}(\text{pc}(\text{pc}_2)), 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 \text{pc}_1 and \text{pc}_2
are joined by disjunction.

Case 5 follows along the above lines:
\[
\begin{align*}
\mathit{B} \land \neg(\text{pred}_1[[\text{not pred}]]([\text{env}]) &= \text{ pc}(\sigma_{\text{env}}(\text{pred}_1[[\text{not pred}]])[s, \text{true})))) \\
&= \mathit{B} \land \neg\sigma_{\text{env}}(\text{pc}(\text{pred}_1[[\text{not pred}]])[s, \text{true}))).
\end{align*}
\]
Considering the fact that from the inductive hypothesis we have
\[
\begin{align*}
\text{true} \land \text{pred}_1[[\text{not pred}]]([\text{env}]) &= \sigma_{\text{env}}(\text{pc}(\text{pred}_1[[\text{not pred}]])[s, \text{true}])))
\end{align*}
\]
closes the case.

With the proof of Case 5 we have established \text{pred}_2^-\text{consistency}
of Flow predicate derivation trees as postulated in Eq. (B.8).
Combining this result with \text{exp}_1^-\text{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 \(s\) can be further
extended to a mapping \(\widehat{s}\) : \(S \to S\) in the following way.
\[
\widehat{s}(s)(s) = \lambda s.\lambda v.\{v \in \text{Dom}(s): v \mapsto \widehat{s}(s)(v)\}
\]

**Lemma 4.** Given a Flow program with control flow graph \(G=\langle N, E, n_s, n_a\rangle\). For any program path \(\pi\) through \(G\), the condition
\[
M(\llbracket\pi\rrbracket([\text{env}, b])) = \text{con}(\text{env}, M_1(\llbracket\text{sym}\rrbracket([\text{env}, b])))
\]
holds, which means that concrete and symbolic Flow program execution commute for program paths.

**Proof.** By induction on the length of program path \(\pi\). 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 \(\pi\) 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 \([s, b \otimes p_1 \land \cdots \land p_k]\). Due to the assumption of the inductive step we know that concrete and symbolic execution along edges \(e_1, \ldots, e_k\) commute, hence
\[
[s, b \land b_1 \land \cdots \land b_k] = \text{con}(\text{env}, [s, b \land p_1 \land \cdots \land p_k]).
\]
(In Fig. B.11 the environment \text{env} of the above equation is provided across projection \(P_{11}\).) 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 \([\text{env}_k, b \land b_1 \land \cdots \land b_k]\) to the symbolic domain, we get the program context \(c_k' = [s_k', b \land b_1 \land \cdots \land b_k]\). Symbolic execution along edge \(e_{k+1}\) then gives us
\[
M(e_{k+1})([s_k, b \land b_1 \land \cdots \land b_k]) = [s_k', b \land b_1 \land \cdots \land b_k \land p_{k+1}] = c_{k+1}.'
\]
The resulting program context \(c_{k+1}'\) can be transformed back to the concrete domain, using the environment \(\text{env}_k\) provided through projection \(P_{11}\). Due to Lemma 3 the result coincides with the result we get from concrete execution along edge \(e_{k+1}\).

Now let \(\sigma_{s}\) be a substitution that transforms state \(s\) into state \(s_k\), such that \(\sigma_{s}(s) = s_k\). The substitution \(\sigma_{s}\) can then be written as
\[
\sigma_{s} = \{v_1 \mapsto s_k(v_1), \ldots, v_j \mapsto s_k(v_j)\} \quad \text{with} \quad v_i \in \text{Dom}(s).
\]
Observe from Fig. B.11 that because \(\sigma_{s}\) 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 \text{sym} in conjunction with the fact that \(\text{Dom}(\text{env}) = \text{Dom}(\text{env}_k)\) that \(s = s_k\), and therefore \(\sigma_{s}(s_k) = s_k\).

We use single-edge commutation along edge \(e_{k+1}\) together with the inductive hypothesis to show that the program context \(c_{k+1} = [s_{k+1}, b \land p_1 \land \cdots \land p_{k+1}]\) 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 \(\sigma_{s}\) as indicated in Fig. B.11.

To begin with, we note that the endomorphism-property of \(\sigma_{s}\) that is due to Eq. (B.2) together with the fact that \(\sigma_{s}(s_k) = s_k\) ensures that
\[
\sigma_{s}(\sigma_{s}(s_k)) = \sigma_{s}(s_k) = \sigma_{s}(s_k) = s_k.
\]
In other words, starting with the clean slate state \(s_k\), to which we apply the side-effect \(\sigma_{s}\) of edge \(e_{k+1}\) results in state \(s_k\). Applying substitution \(\sigma_{s}\) to this state yields the same result as if we would start with the clean slate program state \(s\), apply substitution \(\sigma_{s}\), and subject the result to the side-effect associated with edge \(e_{k+1}\). As a consequence, the endomorphism-property of \(\sigma_{s}\) also ensures that
\[
\sigma_{s}(p_{k+1}1) = p_{k+1}.
\]
This identity is sufficient to determine the correctness of the program condition \(b \land p_1 \land \cdots \land p_k\) of program context \(c_{k+1}\), since the correctness of the subcondition \(b \land p_1 \land \cdots \land p_k\) 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 \(\text{env}\) provided via the projection function \(P_{11}\), coincides
with the extended environment \( \overline{env}_{k+1} = [env_{k+1}, b \land b_1 \land \cdots \land b_{k+1}] \). In that case we have
\[
\text{con}(\text{env}, [s_{k+1}, b \land p_1 \land \cdots \land p_{k+1}]) = \overline{env}_{k+1}. \tag{B.10}
\]
To show this we will use the already established single-edge commutation along edge \( e_{k+1} \) from which it follows that
\[
\text{con}(\text{env}_k, [s'_{k+1}, b \land b_1 \land \cdots \land b_k \land p_{k+1}]) = \overline{env}_{k+1}. \tag{B.11}
\]
To make our point we will relate the program contexts \( c'_k \) and \( c_{k+1} \) and use the commutation of context \( c'_k \) and environment \( env_k \) as stated in Eq. (B.11) to prove the commutation of context \( c_{k+1} \) and environment \( env_k \) 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
\[
\sigma_{\text{env}_k}(s'_{k}) = \sigma_{\text{env}}(s_k) \tag{B.12}
\]

\( \exists \forall \nu \in \text{Dom}(s) : \sigma_{\text{env}_{k+1}}(s'_{k+1}(\nu)) = \sigma_{\text{env}}(s_k(\nu)) \) \tag{B.13}
\( \exists \forall \nu \in \text{Dom}(s) : \sigma_{\text{env}_k}(\nu) = \sigma_{\text{env}}(s_k(\nu)) \) \tag{B.14}
\( \exists \forall \nu \in \text{Dom}(s) : \sigma_{\text{env}_k}(\nu) = \sigma_{\text{env}}(s_k(\nu)) \) \tag{B.15}
\( \exists \forall \nu \in \text{Dom}(s) : \sigma_{\text{env}_k}(\nu) = \sigma_{\text{env}}(s_k(\nu)) \) \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 \( \nu \) to the corresponding initial value variable \( \nu \). The step from Eq. (B.14) to Eq. (B.15) is due to the properties of substitution \( \sigma_{env} \). 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 \( \nu \) to the corresponding initial value variable \( \nu \). A similar relation between state \( s_{k+1} \) and state \( s'_{k+1} \) is required for the commutation stated in Eq. (B.10):
\[
\sigma_{\text{env}_{k+1}}(s'_{k+1}) = \sigma_{\text{env}}(s_{k+1}) \tag{B.17}
\]
\( \exists \forall \nu \in \text{Dom}(s) : \sigma_{\text{env}_{k+1}}(s_{k+1}(\nu)) = \sigma_{\text{env}}(s_{k+1}(\nu)) \). \tag{B.18}
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 \( v_w \) 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 \( v_w \). 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 \( e'_{k}(x)=\exp[\exp(s'_{k})] \) as the corresponding symbolic expression, and once within state \( s_{k} \), yielding \( e_{k}(x)=\exp[\exp(s_{k})] \).

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 \( \sigma_{\bar{s}} \) applies only to the leaves of expression trees where it replaces the initial value variables \( v_j \) with the expression (sub)trees corresponding to the respective symbolic expression \( \sigma_{\bar{s}}(v_j) \). 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 \( e'_{k}(x) \) and \( e_{k}(x) \) to satisfy Eq. (B.18) regarding program variable \( v_w \).

\[
\sigma_{\bar{s}}(e'_{k}(x)) = \sigma_{\bar{s}}(e_{k}(x)) \tag{B.19}
\]

\[
\forall v \in \text{Dom}(s) : \sigma_{\bar{s}}(v) = \sigma_{\bar{s}}(\sigma_{\bar{s}}(v)) \tag{B.20}
\]

Eq. (B.20) is suggested by the already mentioned “leaf-ness” of substitution \( \sigma_{\bar{s}} \), 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 \( \sigma_{\bar{s}}(v) = \sigma_{\bar{s}}(\sigma_{\bar{s}}(v)) \) required by Eq. (B.20) already follows from the inductive hypothesis of this lemma, which closes the case for program variable \( v_w \).

Because symbolic predicates are composed of symbolic expressions, we can close the case of the relation of predicates \( p_{k+1} \) and \( p_k \) 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 computation-property postulated in Eq. (B.10), which also ends the proof of Lemma 4.

We noted with the definition of substitution \( \sigma_{\bar{s}} \) in Eq. (B.9) that due to the transformation of state \( s \) to state \( \bar{s} \), \( \sigma_{\bar{s}} \) 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 \( G = (N, E, n_r, R) \). For any program path \( \pi = (e_1, e_2, \ldots, e_k) \) through \( G \), the program context \( \left[ s_k, p_k \right] = M_{\pi}(\pi)[s, p] \) that results from symbolic execution along \( \pi \) is a partial function of arity \( \text{Env} \times \mathbb{S} \rightarrow \text{Env} \times \mathbb{S} \) that represents the function composition \( M_{\pi}(e_k) \circ M_{\pi}(e_{k-1}) \circ \cdots \circ M_{\pi}(e_1) \).

This partial function is defined for those environments \( \text{env} \) for which the standard-semantic iterated transition function \( \delta \) is defined (e.g., not in case of division by zero or other program anomalies). It coincides with the standard-semantic iterated transition function \( \delta \) for those environments \( \text{env} \) for which \( \sigma_{\bar{s}}(p_k) = \text{true} \).

**Theorem 2.** Correctness of the MOP solution.

\[
\bigcup_{\pi \in \text{Path}(n, n)} \left( \text{con}(\text{env}, M_{\pi}(\pi)[\text{sym}([\text{env}, b]]) = M_{\pi}(\pi)[\text{env}, b] \right) \bigcup_{\pi \in \text{Path}(n, n)} \text{MOP}(\pi, \text{env}).
\]

**Proof.** Immediate from Lemma 4. □

**References**


