Symbolic Analysis of Imperative Programming Languages

Bernd Burgstaller[1], Bernhard Scholz[1], and Johann Blieberger[2]

[1] The University of Sydney*
[2] Technical University of Vienna

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 parallelisation, 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 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) nodes of reducible and irreducible control flow graphs. We prove the correctness of our symbolic analysis method. Our experimental results show that the problem sizes arising from real-world applications such as the SPEC95 benchmark suite are tractable for our symbolic analysis framework.

1 Introduction

Static program analysis is concerned with the design of algorithms that determine the dynamic behaviour of programs without executing them. Symbolic analysis is an advanced static program analysis technique. It has been successfully applied to memory leak detection [32], compilation of parallel programs [17, 22, 37, 10], detection of superfluous bound checks, variable aliases and task deadlocks [31, 13, 6, 7], and to worst-case execution time analysis [4, 8]. The results gained using symbolic analysis provide invaluable information for optimising compilers, code generators, program verification, testing and debugging.

* This work has been partially supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190 and the ARC Discovery Project Grant "Distributed Data Processing for Wireless Sensor Networks" under Contract DP 0664782.

Symbolic analysis [17, 29, 23] uses symbolic expressions to describe computations as algebraic formulæ over a program's problem space. Symbolic analysis consists of two steps:

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

As an example, consider the statement sequence depicted in Fig. 1. After the declaration of two scalar variables, the read statement in line 2 assigns both variables a new value. The subsequent assignment statements change the values of both variables. Symbolic analysis applies symbolic values for program variables. Assuming that the read statement in line 2 yields the symbolic value $\underline{u}$ for variable u, and $\underline{v}$ for variable v, then a simple sequence of forward substitutions and simplifications computes the symbolic values depicted in the table at the right of Fig. 1. Each row in the table denotes the symbolic values val_u and val_v of variables u and v after execution of the corresponding statement. These symbolic values describe the variable bindings that are valid at the corresponding program points. Comparing the variable bindings depicted with line 2 and line 5,

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

Fig. 1. Simple Statement Sequence

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 optimising compiler can derive that the expression u−v in line 4 of the example program will always yield $\underline{u}$, which makes an overflow check of this expression redundant. (Note that 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 analysis applications 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 modelled by means of path expressions first introduced in [34]. 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 $\cdot$, $+$, and $*$ operations used to construct regular expressions. 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. [17]) 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 modelling 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 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 analyse, with the majority of those 94% involving even fewer than 4000 closure contexts.

The paper is organised as follows: In Sect. 2 we outline notations and background material. In Sect. 3 we define syntax and semantics of a flow language that we use to develop our symbolic analysis methodology. In Sect. 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 Sect. 6 we discuss experimental results of the SPEC95 benchmark suite. Section 7 surveys related work. Finally, in Sect. 8 we draw our conclusions and outline future work. The proofs of the theorems stated in the paper have been made available in [11].

2 Background and Notation

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

It is shown in [34] how program paths can be represented as regular expressions: Let Σ be a finite alphabet disjoint from $\{\Lambda, \emptyset, (,)\}$. A *regular expression* is any expression built by applying the following rules:

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

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

In a regular expression, Λ denotes the empty string, $\emptyset$ denotes the empty set, $+$ denotes union, $\cdot$ denotes concatenation, and * denotes reflexive, transitive closure under concatenation. We use $L(R)$ to denote the set of strings defined by the regular expression R over Σ. A regular expression R is *simple* if $R = \emptyset$ or R does not contain $\emptyset$ as a subexpression. Given a CFG $G = \langle N, E, n_e, n_x \rangle$, we can regard any path π in G as a string over E, but not all strings over E are paths in G. A *path expression* P of *type* (v, w) is a simple regular expression over E such that every string in $L(P)$ is a program path from node v to node w. Standard algorithms such as Gaussian elimination can be applied to compute path expressions from a CFG (cf. e.g., [25, 34]).

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

3 Standard Semantic Program Execution

An environment *env* of our Flow language maps a program variable $v \in \mathbb{V}$ to its value $z \in \mathbb{Z}$. The set of possible environments can be represented by a

function class $Env \subseteq \{env : \mathbb{V} \to \mathbb{Z}\}$. Functions $pred_c : E \to \big(Env \to \mathbb{B}\big)$ and $\sigma_c : E \to (Env \to Env)$ associate with each edge $e \in E$ a branch-predicate and a side-effect. The syntax of Flow branch predicates and side-effects is depicted in Fig. 2.

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

Fig. 2. Syntactic Domain of the Flow Language

The valuation functions pred_c : Predicate $\to (Env \to \mathbb{B})$ and assign_c : Assignment $\to (Env \to Env)$ map predicates and side-effects to the semantic domain; due to space considerations we refer to [11, Sect. 3] for their definitions.

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

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

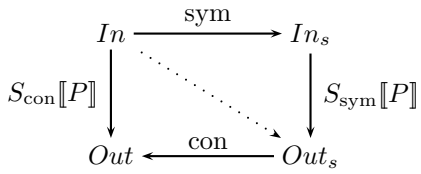
$$\begin{aligned}(n, env) \to &(n', env') : \\ &\big(\exists e \in \text{out}(n) : \ t(e) = n' \wedge pred_c(e)(env)\big) \\ &\Rightarrow env' = \sigma_c(e)(env),\end{aligned} \tag{1}$$

where $\Rightarrow$ denotes *implication*. The iterated transition function $\delta^* : (N \times Env) \to (N \times Env)$ is defined by $\delta^*(n_x, env) = (n_x, env)$ and $\delta^*(n, env) = \delta^*(\delta(n, env))$. For any graph $G = \langle N, E, n_e, n_x\rangle$ the environment env_x of the terminal node n_x represents the result of standard semantic program execution along the sequence of transitions $(n_e, env_e) \xrightarrow{*} (n_x, env_x)$. Depending on the structure of G and the initial environment env_e such a transition sequence may not exist. Deciding on its existence is in general equivalent to the halting problem.

4 Semantics of Symbolic Program Execution

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

The standard semantics of a program P is derived by the valuation function S_{con} that takes a program P as argument and returns a standard-semantic functional description of the side-effect of P. The side-effect $S_{\text{con}}[\![P]\!]$ is a function that maps a concrete input to a concrete output, written as $S_{\text{con}}[\![P]\!](In) = Out$, where $In, Out \in Env$. Given the standard semantics of Flow programs from Sect. 3, the computation of $S_{\text{con}}[\![P]\!](In)$ is equivalent to an application of the iterated transition function δ^* to start node n_e and environment In.



(2)

Similarly we derive the semantics of symbolic program execution for program P, denoted by $S_{\text{sym}}[\![P]\!]$. It is the purpose of function S_{sym} to transform P into a representation that is based on symbolic values instead of concrete ones. The side-effect $S_{\text{sym}}[\![P]\!]$ of this representation is therefore a function that maps a symbolic input In_s to the corresponding symbolic output Out_s. Symbolic input and output belong to the class Env_s of *symbolic environments* that replaces the concrete environments $env : \mathbb{V} \to \mathbb{Z}$ which are not able to bind identifiers to symbolic expressions.

The diagram in (2) contains two additional functions, sym and con, that we need in order to relate input and output of the functional descriptions $S_{\text{con}}[\![P]\!]$ and $S_{\text{sym}}[\![P]\!]$. Function sym transfers a concrete environment to the symbolic domain, whereas function con *instantiates* a symbolic environment with a concrete one. The commutation of concrete and symbolic execution depicted in (2) can then be formalised as

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

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

4.1 The Domain for Symbolic Program Execution

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

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

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

To support division, the ring $\mathbb{Z}[\mathbf{x}]$ is extended to the quotient field $Q(\mathbb{Z}[\mathbf{x}])$ (cf. [18]). By means of the rounding operation Rnd we can "wrap" a rational function x/y to obtain an integer-valued expression $\text{Rnd}(x/y)$[3]. Hence we

[3] Simplifications of expressions involving operation Rnd have been investigated in [11, Sect. 4.1], they are however outside the scope of this paper.

can model the integer division of two symbolic expressions x and y, $y \neq 0$, as $x \operatorname{div}_s y = \mathrm{Rnd}(x \,/\, y)$, where the symbolic division operator div_s denotes the counterpart of the integer division operator div of the Flow standard semantics.

Let $f^{(n)} \in \{+^{(2)}, -^{(2)}, -^{(1)}, \cdot^{(2)}\}$ denote functions corresponding to the Flow arithmetic operations, where (n) denotes the respective arity. They constitute the corresponding operations on multivariate polynomials and rational functions, with the only extension that they do accept arguments "wrapped" by the rounding operator Rnd.

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

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

Let $f \in \{<, \leq, =, \geq, >\}$ denote functions corresponding to the relational connectives of the Flow language. They are extensions of their standard semantic counterparts which operate on values of the symbolic expression domain SymExpr, and return values of the symbolic predicate domain SymPred, e.g., $\leq$: SymExpr$\times$ SymExpr $\rightarrow$ SymPred. Moreover, let $l^{(n)} \in \{\wedge^{(2)}, \vee^{(2)}, \neg^{(1)}\}$ denote the logical connectives of conjunction, disjunction and negation. They are extensions of their standard semantic counterparts that operate on values of the symbolic predicate domain SymPred.

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

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

It is shown in [11, Sect. 4.1] that the domain SymPred constitutes a Boolean algebra.

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

Definition 4. *A* context $c \in C \subseteq [S \times \mathit{SymPred}]$ *is defined by an ordered tuple* $[s, p]$ *where* s *denotes a state, and pathcondition* $p \in \mathit{SymPred}$ *describes the condition for which the variable bindings specified through* s *hold (cf. [4, 17]). We make use of the functions* $\mathrm{pc} : C \to \mathit{SymPred}$ *and* $\mathrm{st} : C \to S$ *to access a context's pathcondition and state. A* clean slate *context consists of a clean slate state and a* true *pathcondition.*

Standard semantic and symbolic side-effects and branch-predicates share the syntactic domain depicted in Fig. 2. Due to space considerations we refer to [11, Sect. 4.2] for an exhaustive description of the valuation functions into the symbolic domain that are introduced in brief below. Equation (4) defines valuation function assign_s which maps the derivation tree of an assignment statement to the corresponding side-effect in the symbolic domain. This side-effect is a function that transforms its argument context $[s, p]$ by updating the state s with a new symbolic expression at $\mathrm{id}[\![\mathrm{id}]\!]$.

$$\begin{aligned} &\mathrm{assign}_s : \mathrm{Assignment} \to (C \to C) \\ &\mathrm{assign}_s[\![\mathrm{id{:=}exp}]\!](c) = \lambda\big[s, p\big].\big[\lambda s_1.s_1\big[\mathrm{id}[\![\mathrm{id}]\!] \mapsto \mathrm{exp}_s[\![\mathrm{exp}]\!](s_1)\big](s),\, p\big](c) \end{aligned} \tag{4}$$

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

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

4.2 Single-Edge Symbolic Execution

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

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

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

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

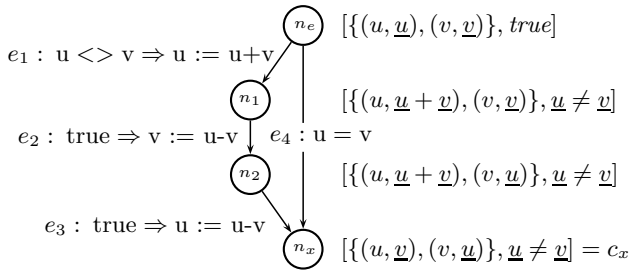


Fig. 3. Symbolic Execution along Path $\pi_1 = \langle e_1, e_2, e_3 \rangle$

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

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

4.3 Single-Path Symbolic Execution

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

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

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

In the previous example we determined the result of the transition function $M_s(e_1)$ which represents the effect of symbolic program execution along edge e_1 of our running example. After evaluation of all edge transition functions along the program path $\pi_1 = \langle e_1, e_2, e_3 \rangle$ we use function $M_s(\pi_1)$ to calculate the effect of symbolic execution along path π_1. We assume that the initial context c_e passed as argument to $M_s(\pi_1)$ contains two program variables u and v holding their initial values $\underline{u}$ and $\underline{v}$. Then the contexts depicted in Fig. 3 illustrate the transformation of the initial context c_e during symbolic execution along program path π_1[4]. The context shown with node n_x represents the result for $M_s(\pi_1)(c_e)$.

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

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 analyse it for our symbolic solution to be complete. Symbolic execution along edge e_4 yields a further program context $c'_x = [\{(u, \underline{u}), (v, \underline{v})\}, \underline{u} = \underline{v}]$.

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

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

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

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

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

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 [24] 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

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

with $\mathrm{Path}(n_e, n)$ 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 [11].

5 Symbolic Evaluation

The symbolic execution approach of Sect. 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 delivered 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 Sect. 2, which we use to model the program paths of a given CFG. The structure of regular expressions imposes a horizontal functional decomposition of the CFG in contrast to the approach of the previous section in which our functional decomposition was vertically along whole program paths. As a consequence we have to extend domain and codomain of the function class F_s introduced in Sect. 4.2 from contexts to supercontexts, yielding a new function class F_{sc}:

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

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

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

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

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

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

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

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

Based on the edge transition function M_s (cf. Sect. 4.2) we define a new edge transition function M_{sc} that encapsulates the wrapping operator inside:

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

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

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

$$\phi(\Lambda) = \iota, \tag{13}$$
$$\phi(e) = M_{\text{sc}}(e) = f_e, \tag{14}$$
$$\phi(P_1 + P_2) = \phi(P_1) \cup \phi(P_2), \tag{15}$$
$$\phi(P_1 \cdot P_2) = \phi(P_2) \circ \phi(P_1), \tag{16}$$
$$\phi(P_1^*) = \phi(P_1)^*. \tag{17}$$

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

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

Proof in [11]. Based on Lemma 1 we establish that the mapping ϕ is a homomorphism from the regular expression algebra to the function class F_{sc} of (10), and that the computed solution corresponds to the MOP-solution for symbolic execution of (9).

Theorem 1. *For any node n let $P(n_e, n)$ be a path expression representing all paths from n_e to n. Then* $\text{mop}(n) = \phi\big(P(n_e, n)\big)(c_e)$*, where c_e denotes the initial context*[5] *valid at entry node n_e.*

Proof in [11]. It should be noted that Theorem 1 does not impose a restriction on path expression $P(n_e, n)$. As a consequence, Theorem 1 holds for path expressions corresponding to CFGs with *irreducible* graph portions. Furthermore it holds for arbitrary graph nodes, even within loops and nested loops.

5.1 Finite Supercontexts

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

In changing the mapping ϕ by replacing (17) with

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

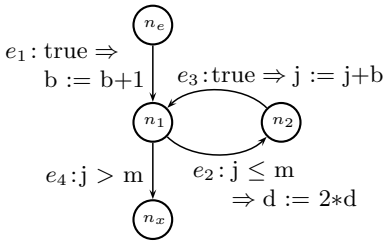
we introduce a new operation $\circledast$ which replaces the iterated composition operation from (11) by a composition operation that generates a finite representation

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

for the result of symbolic evaluation of the CFG cycle corresponding to path expression P_1. This finite representation is an extension of a context by a system of symbolic recurrences [26] and is called a *closure context*. As will be pointed out below, a system of symbolic recurrences makes a closure context an *exact* representation of the infinite set of contexts that is due to a CFG cycle. In this way (18) changes our representation of a supercontext from an *infinite set* of *contexts* to a *finite set* of *closure contexts*. The purpose of this change is to have a compact representation of supercontexts that facilitates domain-specific static program analyses and that can be implemented with CASs.

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

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




```
1  b := b + 1;
2  l := 0;
3  while j <= m loop
4    d := 2*d;
5    j := j+b;
6    l := l+1;
7  end loop;
```

Fig. 4. Example Loop: Implicit vs. Explicit Loop Index Variable

Figure 4 depicts a Flow example loop together with a textual representation where the loop index variable has been made explicit (cf. line 2 and 6). Associated with a loop index variable l is a *symbolic* upper bound, denoted by l_ω. This symbolic upper bound represents the number of loop iterations[6]. Specifically, an upper bound of $\mathsf{l}_\omega = 0$ implies zero loop iterations, as can be inferred from Fig. 4[7]. Endless loops can be modelled by defining $\mathsf{l}_\omega = +\infty$.

Definition 6. *The set of symbolic expressions (cf. Definition 1) is extended by*

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

[6] Computing a *symbolic* upper bound for the number of loop iterations is beyond the scope of this paper. It is discussed, among others, in [17, 5].

[7] This contrasts the notion of range expressions in contemporary programming languages, where `range L..U` denotes the interval $[\mathtt{L}, \mathtt{U}]$.

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

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

$$r ::= \bigcup_{1 \leq j \leq k} rs(\mathsf{l}_j).$$

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

Definition 8. *A closure context* $\overline{c}$ *is an element of the set* $\overline{C} = S \times SymPred \times R$*, denoted by* $[s, p, r]$*. For a clean slate closure context 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* $\overline{sc}$*, for the set of all such finite supercontexts we write* $\overline{SC}$*.*

Definition 9. *We define operation* $\circledast$ *of* (18) *in terms of the input/output-behaviour of the function resulting from the application of operation* $\circledast$ *to* $\phi(P_1)$*, that is,* $\phi(P_1)^\circledast$*. Let* $f = \phi(P_1)$ *be a functional description of the accumulated side-effect of one iteration of the loop body represented by the path expression* P_1*. For a given closure context* $\overline{c_{in}} = [s_{in}, p_{in}, r_{in}]$ *we define the properties of the closure context* $\overline{c_{out}} = [s_{out}, p_{out}, r_{out}]$ *resulting from the application of* $f^\circledast$ *to* $\overline{c_{in}}$*, that is,*

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

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

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

A substitution $\sigma_{s,e}$ *for a given state* s *and an expression* $e \in SymExpr$ *is defined such that* $\sigma_{s,e} = \{\underline{v_1} \mapsto v_1(e), \ldots, \underline{v_j} \mapsto v_j(e)\}$*, with* $\underset{1 \leq i \leq j}{v_i} \in \mathcal{D}om(s)$*.*

What follows is the description of $\overline{c_{out}}$ in terms of its state s_{out}, its pathcondition p_{out}, and its recurrence system set r_{out}.

State: The state s_{out} is computed from s_{in} by replacing the symbolic expressions that describe the values of the variables v_i by the value of the recurrence relation for v_i over loop index variable l.

$$\forall v_i \in \mathcal{D}om(s_{in}) : \; s_{out} ::= s_{in}[v_i \mapsto v_i(\mathsf{l})] \tag{21}$$

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

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

$$p_{in} \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge_{1 \leq \mathsf{l}' \leq \mathsf{l}} p(\mathsf{l}' - 1). \tag{22}$$

Therein the term p_{in} constitutes the pathcondition of closure context $\overline{c_{in}}$. The second term is a range expression according to Definition 7. It defines the value of the loop index variable l to be in the interval $[0, \mathsf{l}_\omega]$. The third term denotes the pathcondition accumulated during l iterations of the loop. It is actually a conjunction of l instances of the pathcondition p_1 from (20), where the l'^{th} instance corresponds to $\sigma_{s_{in},(\mathsf{l}'-1)}(p_1)$.

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

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

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

$$r = \left\{ \begin{bmatrix} \forall v_i \in IV : \begin{cases} v_i(0) ::= s_{in}(v_i) & \\ v_i(\mathsf{l}+1) ::= \sigma_{s_{in},\mathsf{l}}\big(s_1(v_i)\big) & (1) \end{cases} \\ rc ::= \sigma_{s_{in},\mathsf{l}}(p_1) \qquad (2) \end{bmatrix} \right\} \tag{23}$$

Part (1) denotes the recurrence for induction variable v_i. The boundary value of a variable upon entry of the loop body is the variable's value from the "incoming" context ($\overline{c_{in}}$ in our case). We derive the recurrence relation for variable v_i as follows. State s_1 contains the variable bindings after the first iteration of the loop body. In replacing all occurrences of the initial value variables $\underline{v_i} \in \underline{\mathbb{V}}$ by their recursive counterpart $v_i(\mathsf{l})$, we obtain the bindings after iteration $\mathsf{l}+1$, denoted by $v_i(\mathsf{l}+1)$. If we can derive a closed form for the recurrence relation of variable v_i, Part (1) consists only of a symbolic closed form expression over loop index variable l. Part (2) holds the recurrence condition rc for this recurrence system. The condition is basically a symbolic predicate obtained by replacing the initial value variables in the pathcondition p_1 (cf. (20)) by their recursive counterparts.

Having set up the recurrence system set r according to (23), the recurrence system set r_{out} of closure context $\overline{c_{out}}$ is derived from r_{in} by appending r to it.

A recurrence system set can be simplified if we are able to derive closed forms for the recurrence relations of the involved induction variables. There exists a vast body of literature on this topic, e.g., [21, 26, 37, 36, 22, 19]. These methods are directly applicable to the recurrence system sets of our symbolic analysis framework. Modern CASs such as Mathematica [38] provide an ideal platform for the implementation of these methods.

Due to space limitations we refer to [11] for the details involved with the construction of recurrence system sets for nested loops.

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

State: The state of $\overline{c_{out}}$ is obtained from the state of $\overline{c_{in}}$ by replacing the symbolic expressions that describe the values of the induction variables $v_i \in IV = \{d, j\}$ by the value of the recurrence relation for v_i over loop index variable l. Hence we get $s_{out} = \{(b, \underline{b} + 1), (d, d(\mathsf{l})), (j, j(\mathsf{l})), (m, \underline{m})\}$.

Pathcondition: According to (22) we get the pathcondition $p_{out} = true \wedge (0 \leq \mathsf{l} \leq \mathsf{l}_\omega) \wedge \bigwedge\limits_{1 \leq \mathsf{l}' \leq \mathsf{l}} \big(j(\mathsf{l}' - 1) \leq \underline{m}\big)$.

Recurrence System: According to (23) we arrive at the one-element recurrence system set r', with $s_{in} = \mathrm{st}(\overline{c_{in}})$ and $s_1 = \mathrm{st}(\overline{c_1})$ already substituted.

$$r' = \left\{ \begin{bmatrix} \begin{cases} d(0) ::= \underline{d} & (1a) \\ d(\mathsf{l}+1) ::= 2 \cdot d(\mathsf{l}) \end{cases} \\ \begin{cases} j(0) ::= \underline{j} & (1b) \\ j(\mathsf{l}+1) ::= j(\mathsf{l}) + \underline{b} + 1 \end{cases} \\ rc ::= j(\mathsf{l}) \leq \underline{m} \quad (2) \end{bmatrix} \right\}$$

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

$$r = \left\{ \begin{bmatrix} \{ d(\mathsf{l}) ::= 2^{\mathsf{l}} \cdot \underline{d} & (1a) \\ \{ j(\mathsf{l}) ::= \underline{j} + \mathsf{l} \cdot (\underline{b} + 1) & (1b) \\ rc ::= j(\mathsf{l}) \leq \underline{m} \quad (2) \end{bmatrix} \right\}.$$

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

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

as the solution for the closure context $\overline{c_{out}}$, which is also the MOP-solution for node n_1. The intuitive meaning of this closure context unveils if we consider the range expression $(0 \leq \mathsf{l} \leq \mathsf{l}_\omega)$ that is part of its pathcondition: as loop index variable l ranges from 0 to l_ω, the recurrence system r generates the variable

bindings of the respective context[8] c_{l} valid after l loop iterations, i.e. $c_{\mathsf{l}} = (f_{e_3} \circ f_{e_2})^{\mathsf{l}} \circ f_{e_1}(\overline{c_e})$. Hence the closure context $\overline{c_{out}}$ represents a total number of $\mathsf{l}_\omega + 1$ contexts valid at node n_1. Formally the closure context $\overline{c_{out}}$ can be viewed as a predicate $\forall b \forall d \forall j \forall m \forall \mathsf{l} : \overline{c_{out}}$, where the set $\{x \mid 0 \leq x \leq \mathsf{l}_\omega)\} \subseteq \mathbb{N}$ is the universe of discourse for loop index variable l.

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

- at node n_1 the variables b and m assume the values $\underline{b}+1$ and $\underline{m}$ respectively during all loop iterations,
- the induction variables d and j assume monotonically increasing/decreasing sequences of values (depending on the initial values of variables d and b),
- the symbolic values of the induction variables d and j during each iteration of the loop,
- a symbolic upper bound l_ω for the number of loop iterations (computed from the recurrence condition as described in [17]), and therefore
- symbolic lower and upper bounds for the induction variables d and j.

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

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

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

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.

6 Experiments

The prototype implementation of our symbolic analysis framework constitutes a term rewrite system based on OBJ3 ([3, 20]) and Mathematica [38]. Together with the analysis results of Flow sample programs, we have made it available at [14].

Since the practicality of our symbolic analysis method critically depends on the size of the path expressions occurring in practice, we have surveyed the problem sizes arising from the programs of the complete SPEC95 benchmark suite (cf. [33]). The SPEC95 benchmark suite consists of 18 benchmark programs with GCC and the Perl interpreter among them. Overall, we investigated all 5053

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

procedures, in an attempt to make the survey representative both in quantity and in the problem sizes of the investigated programs.

The technical part of this survey comprised the definition of a metric to compute the symbolic analysis problem sizes (i.e., the number of closure contexts resulting from a given path expression), and to apply this metric to the path expressions of the procedures from the SPEC95 benchmark code.

We compute the number of program paths of a path expression corresponding to an acyclic CFG through the mapping $\mathrm{ncc}(e) = 1$, $\mathrm{ncc}(P_1 + P_2) = \mathrm{ncc}(P_1) + \mathrm{ncc}(P_2)$, and $\mathrm{ncc}(P_1 \cdot P_2) = \mathrm{ncc}(P_1) \cdot \mathrm{ncc}(P_2)$. Every such program path induces the generation of one closure context during symbolic analysis[9]. Our *accumulated* ncc metric (ancc) 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.

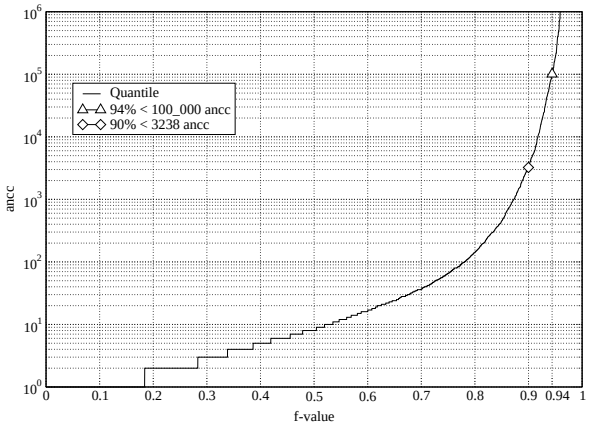


Fig. 5. Quantile Plot for SPEC95 Programs

In our survey each SPEC95 procedure has been accounted for through its path expression of type (n_e, n_x). Figure 5 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 percent 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 percent of the

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

SPEC95 procedures show an ancc-value below 3238, and for 0.94 percent it is still below $100,000$. This means that the problem sizes of more than 94 percent of the procedures from the SPEC95 benchmark suite constitute no problem at all for symbolic analysis, and that the ancc values for 90 percent of all procedures are indeed very small. Due to space limitations we refer to [12] for a description of the whole range of experiments carried out on the SPEC95 benchmark suite.

7 Related Work

P. and R. Cousot [16] 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 are not included in abstract interpretation. 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 [22] 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. [37, 36] use chains of recurrences [39, 2] to model symbolic expressions. Analysis is carried out directly on the CFG, with loops being analysed 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 specialised 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.

In [4] symbolic evaluation is used 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. In [17] a symbolic representation for contexts is introduced. Closure contexts are an extension of this algebraic structure.

Tu and Padua [35] 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 suf-

fices 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. [27] 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 [10] 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. [19] describe a general induction variable recognition method based on a demand-driven SSA form. Rugina and Rinard [31] carry out symbolic bounds analysis for accessed memory regions. With their method they set up a system of symbolic constraints that describe the lower and upper bounds of pointers, array indices, and accessed memory regions. This system of constraints is then solved using ILP. The Omega test [28] developed by W. Pugh 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 in [30, 29].

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 [32], program parallelisation [17, 22, 37, 10], detection of superfluous bound checks, variable aliases and task deadlocks [31, 13, 6, 7], and for worst-case execution time analysis [4, 8].

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 discussed in [17, 6]).

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

We proved (cf. also [11]) 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 su-

percontexts an ideal intermediate representation for all domain-specific static program analyses.

The experiments conducted with our prototype implementation showed that the problem sizes of real-world programs such as those from the SPEC95 benchmark suite are tractable for our symbolic analysis framework. It has been shown in [9] that symbolic analysis has a vast improvement potential in the area of contemporary data-flow based analyses of sequential and concurrent programs. We are therefore facing two research tiers that we plan to pursue in the future, namely (1) the extension of our method to incorporate concurrent programming language constructs, and (2) the application of our method to domain-specific static program analysis problems.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. O. Bachmann, P. S. Wang, and E. V. Zima. Chains of Recurrences — A Method to Expedite the Evaluation of Closed-Form Functions. In *Proc. of the Internat. Symposium on Symbolic and Algebraic Computation*, pages 242–249. ACM Press, 1994.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, 1998.
4. J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22:183–227, 2002.
5. J. Blieberger. Discrete Loops and Worst Case Performance. *Computer Languages*, 20(3):193–212, 1994.
6. J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 136–145, Santander, Spain, June 1999.
7. J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 225–237. Springer-Verlag, 2000.
8. J. Blieberger, T. Fahringer, and B. Scholz. Symbolic Cache Analysis for Real-Time Systems. *Real-Time Systems*, 18(2/3):181–215, 2000.
9. B. Burgstaller, J. Blieberger, and R. Mittermayr. Static Detection of Access Anomalies in Ada95. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 40–55. Springer-Verlag, 2006.
10. W. Blume and R. Eigenmann. Nonlinear and Symbolic Data Dependence Testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, 1998.
11. B. Burgstaller. Symbolic Evaluation of Imperative Programming Languages. Technical Report 183/1-138, Department of Automation, Vienna University of Technology, June 2005. `http://www.auto.tuwien.ac.at/~bburg/reports.html`.
12. B. Burgstaller, B. Scholz, and J. Blieberger. Tour de Spec — A Collection of Spec95 Program Paths and Associated Costs for Symbolic Evaluation. Technical Report 183/1-137, Department of Automation, Vienna University of Technology, June 2004. `http://www.auto.tuwien.ac.at/~bburg/reports.html`.
13. J. Blieberger and B. Burgstaller. Eliminating Redundant Range Checks in GNAT Using Symbolic Evaluation. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 153–167, Toulouse, France, June 2003.
14. `http://www.it.usyd.edu.au/~bburg/symanalysis.html`.
15. L. A. Clarke and D. J. Richardson. Symbolic Evaluation Methods for Program Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 264–300. Prentice-Hall, 1981.
16. P. Cousot and R. Cousot. Abstract Intrepretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*, pages 238–252, January 1977.

17. T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers*, volume 2628. LNCS, Springer-Verlag, 2003.
18. K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.
19. M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *TOPLAS*, 17(1):85–122, January 1995.
20. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. Draft, Oxford University Computing Laboratory, 1993.
21. D. Greene and D. E. Knuth. *Mathematics For the Analysis of Algorithms*. Birkhäuser, second edition, 1982.
22. M. R. Haghighat and C. D. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *TOPLAS*, 18(4):477–518, July 1996.
23. P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994.
24. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.
25. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1979.
26. George S. Lueker. Some Techniques for Solving Recurrences. *ACM Computing Surveys (CSUR)*, 12(4):419–436, 1980.
27. V. Menon, K. Pingali, and N. Mateev. Fractal Symbolic Analysis. *TOPLAS*, 25(6):776–813, 2003.
28. W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.
29. W. Pugh. Counting Solutions To Presburger Formulas: How and Why. In *Proc. of PLDI*, pages 121–134, 1994.
30. W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. Technical report, College Park, MD, USA, 1994.
31. R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proc. of PLDI*, pages 182–195, 2000.
32. B. Scholz, J. Blieberger, and T. Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00)*, Boston, January 2000.
33. SPEC CPU95 Benchmark Suite, Version 1.10, August 1995.
34. R. E. Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, 1981.
35. P. Tu and D. A. Padua. Gated SAA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *International Conference on Supercomputing*, pages 414–423, 1995.
36. R. A. van Engelen. The CR# Algebra and its Application in Loop Analysis and Optimization. Technical Report TR-041223, Department of Computer Science, Florida State University, December 2004.
37. R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *ICS '04: Proc. of the 18th Annual International Conference on Supercomputing*, pages 106–115. ACM Press, 2004.
38. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Incorporated, 2003.
39. E. V. Zima. Simplification and Optimization Transformations of Chains of Recurrences. In *ISSAC '95: Proc. of the 1995 International Symposium on Symbolic and Algebraic Computation*, pages 42–50. ACM Press, 1995.
40. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1991.