Closed symbolic execution for verifying program termination

Germán Vidal
MiST, DSIC, Universitat Politècnica de València
Camino de Vera, S/N, 46022 Valencia (Spain)
Email: gvidal@dsic.upv.es

***Abstract*—Symbolic execution, originally introduced as a method for program testing and debugging, is usually incomplete because of infinite symbolic execution paths. In this work, we adapt some well-known notions from partial evaluation in order to have a complete symbolic execution scheme which can then be used to check liveness properties like program termination. We also introduce a representation of the symbolic transitions as a term rewrite system so that existing termination provers for these systems can be used to verify the termination of the original program.**

***Keywords*-program termination; symbolic execution; program analysis; term rewriting; partial evaluation;**

I. Introduction

There is a renewed interest in *symbolic execution* [1], [2], a well-known technique for program verification, testing, debugging, etc. In contrast to normal execution, symbolic execution considers that the values of some input data are unknown, i.e., some input parameters $x, y, \ldots$ take *symbolic values* $X, Y, \ldots$ Because of this, symbolic execution is often non-deterministic: at some control statements, we need to follow more than one execution path because the available information does not suffice to determine the validity of a control expression, e.g., symbolic execution may follow both branches of the conditional "`if` $(x{>}0)$ `then` $exp1$ `else` $exp2$" when the symbolic value X of variable x is not constrained enough to imply neither $x{>}0$ nor $\neg(x{>}0)$. Symbolic states include a *path condition* that stores the current constraints on symbolic values, i.e., the conditions that must hold to reach a particular execution state. E.g., after symbolically executing the above conditional, the derived states for $exp1$ and $exp2$ would add the conditions $X{>}0$ and $X{\leq}0$, respectively, to their path conditions.

Traditionally, formal techniques based on symbolic execution have enforced *soundness*: if a symbolic state is reached and its path condition is satisfiable, there must be a normal execution path that reaches the corresponding concrete state. In contrast, symbolic execution is *complete* when every reachable state in a normal execution is "covered" by some symbolic state. Completeness is important for verifying *liveness* properties—like program termination. For the general case of infinite state systems, completeness usually requires some kind of *abstraction* (as in infinite state model checking).

In this work, we follow well-known principles from partial evaluation [3] in order to design a *complete* symbolic execution scheme, that we call *closed* following the terminology of some partial evaluation literature. In particular, given an initial state with some missing input data, *online* partial evaluation constructs a *complete* representation—usually a graph—of all potential executions, and then extracts a residual program from the transitions in this graph. For this purpose, a symbolic execution method is augmented with *subsumption* and *abstraction* operators (similarly to those in [4], [5]) in order to guarantee that the computed representation is finite (see, e.g., Gallagher's basic algorithm parameterized by an unfolding rule and an abstraction operator [6]).

Analogously to the extraction of residual code in partial evaluation, we propose the extraction of *rewrite rules* [7] from the transitions in the symbolic execution graph. In this way, we obtain a rewrite system that can be used to analyze the termination (or other liveness properties) of the original program in a unified and well-known setting (where powerful termination provers exist for rewrite systems, e.g., AProVE [8]). An advantage of this approach is that one can "compile in" the language semantics in the symbolic execution graph, so that we extract residual rules from the *semantics* of the program rather than from its syntax. This pattern is well-known in the partial evaluation literature (a consequence of the first Futamura projection [9]).

We can find in the literature similar approaches to proving the termination of Haskell [10], Prolog with impure features [11] and Java bytecode [12], [13], [14] by transforming the original termination problem into the problem of analyzing the termination of a rewrite system. COSTA [15], [16], a cost and termination analyzer for Java bytecode, follows a similar pattern but produces a constraint logic program instead. All these transformational approaches share a similar pattern: they construct a finite-state representation of the program's computations (often called *termination graph* [13]), and a finite set of rules is extracted from this representation.

While all these approaches have proven useful in practice, they are tailored to the specific features of a programming

language. Unfortunately, this makes it rather difficult to grasp the key ingredients of the approach and, thus, it is not easy to design a termination tool for a different programming language by following the same pattern. In this paper, we aim at introducing a simpler but higher level scheme for proving liveness properties like program termination which is independent of the considered programming language. We do so by following the principles of partial evaluation.

This language-independent approach may ease the design of new program analyzers for different programming languages and promotes the reuse of existing analysis tools for rewrite systems. Another advantage of defining a unified higher level scheme is that common problems (and solutions!) can be better identified (e.g., scalability issues, improving accuracy, etc).

We show the viability of the scheme with a proof-of-concept implementation of a termination prover for simple imperative programs with integers, basic arithmetic, assignments, conditionals and jumps. Our preliminary results are encouraging and point out the usefulness of the approach.

The remainder of this paper is organized as follows. Section II introduces the first stage of our scheme, the construction of a finite-state symbolic execution graph for a given program. Then, Section III presents the extraction of a term rewrite system from the transitions of the symbolic execution graph. Section IV presents a proof-of-concept implementation of a termination prover that follows the ideas introduced in the previous sections. Finally, Section V discusses some related work and Section VI concludes and points out some directions for further research.

II. Closed Symbolic Execution

A. Programs and Computations

A program P is a tuple $\langle \Sigma, \Theta, \mathcal{T}, \rho \rangle$ where Σ is a (possibly infinite) set of states, $\Theta \subseteq \Sigma$ are the initial states, $\mathcal{T}$ is a finite set of transitions (corresponding to the program statements), and ρ is a function that assigns to each transition a binary relation over states: $\rho_\tau \subseteq \Sigma \times \Sigma,$ for $\tau \in \mathcal{T}$. States are modelled as pairs $\langle l, \sigma \rangle$ where l is the location of the next sentence to be executed and σ is a (finite) mapping from program variables to values (the heap). Formally, $\Sigma = \mathsf{Loc} \times (\mathsf{Var} \to \mathsf{Value})$, where Loc and Var are finite sets of program locations and variables, respectively, and Value is a (possibly infinite) set of values.

Transition relations are (possibly infinite) sets of pairs of states (s, s'), where s is the current state and s' is the next state. Transition relations can be compactly described as logical formulas over unprimed and primed variables corresponding to the variables of s and s' (so that variables not appearing in the formula are simply not constrained). We also introduce a *fresh* (i.e., not appearing in the program) variable pc which denotes the *program counter*.

l_0 : $x := \mathbf{input}();$
l_1 : **while** $x > 0$ **do**
l_2 : $\quad x := x - 1;$
l_3 : **done**

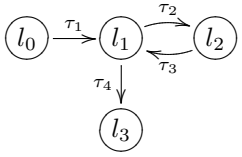

Figure 1. Program WHILE and its control flow graph.

Example 2.1: Consider the simple imperative program WHILE shown in Figure 1, where $\mathsf{Loc} = \{l_0, l_1, l_2, l_3\}$, $\mathsf{Var} = \{x\}$, and $\mathsf{Value} = \mathbb{Z} \cup \{\bot\}$.[1]

Here, we consider a single initial state $\Theta = \{\langle l_0, \{x \mapsto \bot\}\rangle\}$. We have four transitions, τ_1, τ_2, τ_3 and τ_4, as shown in the control flow graph depicted in Figure 1. The transition relations can be defined as follows:

$$\begin{array}{ll} \rho_{\tau_1}: & pc = l_0 \wedge pc' = l_1 \\ \rho_{\tau_2}: & pc = l_1 \wedge pc' = l_2 \wedge x > 0 \\ \rho_{\tau_3}: & pc = l_2 \wedge pc' = l_1 \wedge x' = x - 1 \\ \rho_{\tau_4}: & pc = l_1 \wedge pc' = l_3 \wedge x \leq 0 \end{array}$$

The transition relation R_P of a program P is then defined as the union of all transition relations: $R_P = \bigcup_{\tau \in \mathcal{T}} \rho_\tau$. Computations are (possibly infinite) maximal sequences of states $s_0, s_1, \ldots$ such that

- $s_0 \in \Theta$ is an initial state and
- $(s_i, s_{i+1}) \in R_P$ for all $i \geq 0$ (up to the length of the sequence if it is finite).

We will denote computations as follows: $s_0 \xrightarrow{\tau_1}_{R_P} s_1 \xrightarrow{\tau_2}_{R_P} \ldots$ (we will omit the transition label and/or the program's transition relation when they are clear from the context).

Given a relation R, we let R^+ denote its transitive closure and R^* its transitive and reflexive closure. Finite computations $s_0 \to s_1 \to \ldots \to s_n$, $n \geq 0$, can be denoted by $s_0 \to^* s_n$ ($s_0 \to^+ s_n$ when it comprises at least one transition).

Example 2.2: Consider again program WHILE (shown in Fig. 1), where the transition relation $R_{\mathsf{WHILE}} = \rho_{\tau_1} \cup \rho_{\tau_2} \cup \rho_{\tau_3} \cup \rho_{\tau_4}$. An example computation follows:

$$\begin{array}{lllll} \langle l_0, \{x \mapsto \bot\}\rangle & \xrightarrow{\tau_1} & \langle l_1, \{x \mapsto 2\}\rangle & \xrightarrow{\tau_2} & \langle l_2, \{x \mapsto 2\}\rangle \\ & \xrightarrow{\tau_3} & \langle l_1, \{x \mapsto 1\}\rangle & \xrightarrow{\tau_2} & \langle l_2, \{x \mapsto 1\}\rangle \\ & \xrightarrow{\tau_3} & \langle l_1, \{x \mapsto 0\}\rangle & \xrightarrow{\tau_4} & \langle l_3, \{x \mapsto 0\}\rangle \end{array}$$

B. Symbolic Execution

Symbolic execution [1], [2], originally introduced in the context of program testing and debugging, extends normal execution in order to deal with variables bound to symbolic expressions (instead of concrete values). E.g., $\langle l_0, \{x \mapsto X, y \mapsto Y, z \mapsto 42\}, true\rangle$ is a symbolic state where x, y are program variables bound to symbolic values (denoted by capital letters), z is a local variable bound to the integer 42, and $true$ is a *path condition* (see below). Program variables can also be bound to symbolic

[1] As it is common practice, we denote by $\bot$ an undefined value.

expressions like $X + 2 * Y$ or arbitrary data structures (e.g., arrays, linked lists, etc) possibly including symbolic values denoting missing information. Control statements often involve (non-deterministically) exploring several paths. The *path condition* of symbolic states is then used to keep track of the assumptions made on the symbolic values in each computation thread. Therefore, the domain of symbolic states is now

$$\Sigma^\sharp = \mathsf{Loc} \times (\mathsf{Var} \to \mathsf{Value}^\sharp) \times \mathsf{PathCond}$$

where Loc and Var are the concrete (finite) sets of program locations and variables, respectively, $\mathsf{Value}^\sharp$ is a (possibly infinite) set of symbolic expressions, and PathCond is a domain of logic formulas over the symbolic values. We will denote symbolic states with $\mathcal{S}_1$, $\mathcal{S}_2$, etc.

There is a clear relation between concrete and symbolic states: a symbolic state represents the set of concrete states that can be obtained by replacing its symbolic values with concrete values that make the path condition true.

Definition 2.3 (concretization): Let $\langle l, \theta, pc\rangle$ be a symbolic state. We denote by $\mathsf{sol}(\theta, pc)$ the set of concrete heaps obtained from θ by replacing its symbolic values with concrete values that satisfy the path condition pc (and then evaluating the resulting symbolic expressions, if any). Then, the concretization function $\gamma : \Sigma^\sharp \mapsto \wp(\Sigma)$ is defined as follows: $\gamma(\langle l, \theta, pc\rangle) = \{\langle l, \sigma\rangle \mid \sigma \in \mathsf{sol}(\theta, pc)\}$.
Given a concrete state s and a symbolic state $\mathcal{S}$, we observe that $s \in \gamma(\mathcal{S})$ implies that s and $\mathcal{S}$ share the same program location. Analogously, $\gamma(\mathcal{S}) \subseteq \gamma(\mathcal{S}')$ implies that the symbolic states $\mathcal{S}$ and $\mathcal{S}'$ share the same program location too. Basically, only (symbolic) states that point to the same program location are comparable. We note that our symbolic states are similar to the notion of *region* in [17].

In the following, we assume a *decidable* partial order $\sqsubseteq_\gamma$ on symbolic states such that, if $\mathcal{S} \sqsubseteq_\gamma \mathcal{S}'$ then $\gamma(\mathcal{S}) \subseteq \gamma(\mathcal{S}')$ (the opposite direction does not generally hold in order to have a decidable approximation).

Definition 2.4 (symbolic program): Let $P = \langle \Sigma, \Theta, \mathcal{T}, \rho\rangle$ be a concrete program. We say that $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}^\sharp, \rho^\sharp\rangle$ is a symbolic version of P if the following conditions hold:

1) $\forall s \in \Sigma.\ \exists \mathcal{S} \in \Sigma^\sharp$ such that $s \in \gamma(\mathcal{S})$;
2) $\forall s \in \Theta.\ \exists \mathcal{S} \in \Theta^\sharp$ such that $s \in \gamma(\mathcal{S})$;
3) $\mathcal{T} = \mathcal{T}^\sharp$ (i.e., the program sentences are not changed);
4) $\forall (s, s') \in \rho_\tau$ and $\forall \mathcal{S} \in \Sigma^\sharp$ such that $s \in \gamma(\mathcal{S})$ there exists $(\mathcal{S}, \mathcal{S}') \in \rho^\sharp_\tau$ with $s' \in \gamma(\mathcal{S}')$ (completeness).

Note that no particular definition for $\rho^\sharp$ is given (which typically depends on the considered programming language); the definition above only shows the conditions that it must fulfill. Intuitively, conditions (1) and (2) imply that replacing some values by symbolic expressions do not change the nature of a state. Condition (3) means that symbolic execution does not change the source program (only the input data might be replaced by symbolic values). Finally, condition (4) states the basic completeness of symbolic execution, which guarantee that all concrete transitions have a counterpart in the symbolic program (which is essential to analyze liveness properties).

As before, the transition relation $R_{P^\sharp}$ of a symbolic program $P^\sharp$ is defined as the union of all transition relations: $R_{P^\sharp} = \bigcup_{\tau \in \mathcal{T}^\sharp} \rho^\sharp_\tau$. Symbolic computations are (possibly infinite) maximal sequences of symbolic states $\mathcal{S}_0, \mathcal{S}_1, \ldots$ such that

- $\mathcal{S}_0 \in \Theta^\sharp$ is an initial symbolic state and
- $(\mathcal{S}_i, \mathcal{S}_{i+1}) \in R_{P^\sharp}$ for all $i \geq 0$ (up to the length of the sequence if it is finite).

In the following we assume that, given a transition $(\langle l, \theta, pc\rangle, \langle l', \theta', pc'\rangle) \in \rho^\sharp_\tau$, the path condition pc' has the form $pc \wedge pc''$ where pc'' are the new constraints (if any) added to the path condition in the considered symbolic execution step. Moreover, we consider that the satisfiability of the path condition is checked at every step. If the domain of path conditions is not decidable, we can use a time bound so that if the constraints are not solved within this bound, the path condition is assumed satisfiable (to preserve completeness, which contrasts with traditional approaches where it is assumed *unsatisfiable* to preserve soundness).

We will denote symbolic computations as follows:

$$\mathcal{S}_0 \overset{\tau_1, pc_1}{\leadsto}_{R_{P^\sharp}} \mathcal{S}_1 \overset{\tau_2, pc_2}{\leadsto}_{R_{P^\sharp}} \mathcal{S}_2 \overset{\tau_3, pc_3}{\leadsto}_{R_{P^\sharp}} \cdots$$

where τ_i is the transition of the step and pc_i are the *new* constraints that are added to the path condition (we will omit the transition label, the path condition, and/or the program's transition relation when they are clear from the context).

Example 2.5: Consider again the program WHILE shown in Figure 1. Let $\mathsf{WHILE}^\sharp$ be its symbolic version. Given the initial symbolic state $\langle l_0, \{x \mapsto \bot\}, true\rangle$, we have for instance the following symbolic computation:

$$\begin{array}{rcl}
\langle l_0, \{x \mapsto \bot\}, true\rangle & \overset{\tau_1}{\leadsto} & \langle l_1, \{x \mapsto X\}, true\rangle \\
& \overset{\tau_2}{\leadsto} & \langle l_2, \{x \mapsto X\}, X > 0\rangle \\
& \overset{\tau_3}{\leadsto} & \langle l_1, \{x \mapsto X - 1\}, X > 0\rangle \\
& \overset{\tau_2}{\leadsto} & \langle l_2, \{x \mapsto X - 1\}, X > 1\rangle \\
& \overset{\tau_3}{\leadsto} & \langle l_1, \{x \mapsto X - 2\}, X > 1\rangle \\
& \overset{\tau_4}{\leadsto} & \langle l_3, \{x \mapsto X - 2\}, X = 2\rangle
\end{array}$$

Note that we have simplified the path condition $X > 0 \wedge X - 1 > 0$ to $X > 1$ and the path condition $X > 1 \wedge X - 2 = 0$ to $X = 2$.
Now, we lift the completeness of symbolic execution to computations.

Lemma 2.6 (completeness): Let P be a program and $P^\sharp$ a symbolic version of P. If there exists a (possibly infinite) computation of the form $s_0 \overset{\tau_1}{\to}_{R_P} s_1 \overset{\tau_2}{\to}_{R_P} \ldots$ then, for any symbolic state $\mathcal{S}_0$ such that $s_0 \in \gamma(\mathcal{S}_0)$, we have $\mathcal{S}_0 \overset{\tau_1}{\leadsto}_{R^\sharp_P} \mathcal{S}_1 \overset{\tau_2}{\leadsto}_{R^\sharp_P} \ldots$ where $s_i \in \gamma(\mathcal{S}_i)$ for all $i > 0$.

Proof: The claim follows straightforwardly by applying property (4) of Definition 2.4. Consider the first transition

$s_0 \xrightarrow{\tau_1}_{R_P} s_1$. By property (4), we have that, for all $\mathcal{S}_0 \in \Theta^\sharp$ such that $s_0 \in \gamma(\mathcal{S}_0)$, the transition $\mathcal{S}_0 \overset{\tau_1}{\rightsquigarrow}_{R_P^\sharp} \mathcal{S}_1$ holds with $s_1 \in \gamma(\mathcal{S}_1)$. The same reasoning can be applied repeatedly so that a symbolic computation mimicking the transitions of the concrete computation is built. ■

C. Closed Symbolic Execution

While previous work has emphasized the production of *underapproximations* of standard execution (so that no spurious errors are spotted), we are interested in producing *overapproximations* so that the termination of the original program (as well as other liveness properties) can be preserved through the transformation.

In general, symbolic computations do not terminate due to the use of symbolic values (even if the concrete program admits only finite computations).

Example 2.7: For instance, we might have the following infinite computation with the symbolic version WHILE$^\sharp$ of the program in Example 1:

$$\begin{array}{lll} \langle l_0, \{x \mapsto \bot\}, true\rangle & \overset{\tau_1}{\rightsquigarrow} & \langle l_1, \{x \mapsto X\}, true\rangle \\ & \overset{\tau_2}{\rightsquigarrow} & \langle l_2, \{x \mapsto X\}, X > 0\rangle \\ & \overset{\tau_3}{\rightsquigarrow} & \langle l_1, \{x \mapsto X-1\}, X > 0\rangle \\ & \overset{\tau_2}{\rightsquigarrow} & \langle l_2, \{x \mapsto X-1\}, X > 1\rangle \\ & \overset{\tau_3}{\rightsquigarrow} & \langle l_1, \{x \mapsto X-2\}, X > 1\rangle \\ & \overset{\tau_2}{\rightsquigarrow} & \ldots \end{array}$$

by always choosing transition τ_2 from location l_1.
Computations with a symbolic program can be represented by means of a tree-like structure as follows:

Definition 2.8 (symbolic execution graph):
Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp\rangle$ be a symbolic program. We represent the computations of $P^\sharp$ for an initial symbolic state $\mathcal{S}_0 \in \Theta^\sharp$ by means of a (possibly infinite) directed rooted node- and edge-labeled graph $\mathcal{G}_{P^\sharp}$:

- nodes are labeled with symbolic states from $\Sigma^\sharp$ and edges are labeled with transitions from $\mathcal{T}$ and logical formulas (denoting new path conditions);
- the root node is $\mathcal{S}_0$;
- there is an edge labeled with τ from a node labeled with $\mathcal{S}$ to a node labeled with $\mathcal{S}'$, denoted by $\mathcal{S} \xrightarrow{\tau, pc} \mathcal{S}'$, iff $\mathcal{S} \overset{\tau, pc}{\rightsquigarrow}_{R_{P^\sharp}} \mathcal{S}'$ (we will ignore τ and/or pc when they are clear from the context).

In the literature, one can find two basic operations to make the symbolic execution graph finite: subsumption and abstraction (see, e.g., [4], [5]). Basically, subsumption allows us to stop symbolic execution when we reach a state that is an *instance* of (i.e., it is *subsumed* by) a previous one. Formally,

Definition 2.9 (subsumption transformation): Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp\rangle$ be a symbolic program and $\mathcal{G}_{P^\sharp}$ a symbolic execution graph for $\mathcal{S}_0 \in \Theta^\sharp$. Let

$$\mathcal{S}_0 \longrightarrow \mathcal{S}_1 \longrightarrow \ldots \longrightarrow \mathcal{S}_n \longrightarrow \ldots$$

be a path in the graph with $n > 0$. If there exists a node labeled with $\mathcal{S}_i$, $0 \leq i < n$, such that $\mathcal{S}_n \sqsubseteq_\gamma \mathcal{S}_i$, we transform $\mathcal{G}_{P^\sharp}$ into $\mathcal{G}'_{P^\sharp}$ by deleting the children of $\mathcal{S}_n$ (and the edges from $\mathcal{S}_n$ to them). We assume an *implicit* edge labeled with sub from $\mathcal{S}_n$ to $\mathcal{S}_i$; we consider these edges implicit to formally keep the graph acyclic.

We say that $\mathcal{G}'_{P^\sharp}$ is obtained from $\mathcal{G}_{P^\sharp}$ by subsumption.

Example 2.10: Consider the infinite computation shown in Example 2.7. The infinite-state path in the graph can be made finite by subsumption as follows:

$$\begin{array}{lll} \langle l_0, \{x \mapsto \bot\}, true\rangle & \xrightarrow{\tau_1} & \langle l_1, \{x \mapsto X\}, true\rangle \\ & \xrightarrow{\tau_2} & \langle l_2, \{x \mapsto X\}, X > 0\rangle \quad \equiv \mathcal{S}_3 \\ & \xrightarrow{\tau_3} & \langle l_1, \{x \mapsto X-1\}, X > 0\rangle \\ & \xrightarrow{\tau_2} & \langle l_2, \{x \mapsto X-1\}, X > 1\rangle \equiv \mathcal{S}_5 \\ & \xrightarrow{\mathsf{sub}} & \mathcal{S}_3 \end{array}$$

since $\gamma(\mathcal{S}_3) = \gamma(\mathcal{S}_5) = \{\langle l_2, \{x \mapsto 1\}\rangle, \langle l_2, \{x \mapsto 2\}\rangle, \ldots\}$. For realistic examples, finding opportunities for subsumption requires appropriate heuristics. While subsumption allows one to produce finite-state symbolic execution graphs in many cases, this cannot be always ensured. In some cases, a form of *abstraction* is also required:

Definition 2.11 (abstraction operator): Let $\mathcal{S}$ be a symbolic state and let $\mathcal{C}$ be a set of symbolic states (e.g., a set of previous symbolic states). We say that $\alpha : \Sigma^\sharp \times \wp(\Sigma^\sharp) \mapsto \Sigma^\sharp$ is an abstraction operator if $\alpha(\mathcal{S}, \mathcal{C}) = \mathcal{S}'$ implies $\mathcal{S} \sqsubseteq \mathcal{S}'$. An abstraction operator *generalizes* a symbolic state, often taking into account the computation history (i.e., the previous states of the same computation).

Definition 2.12 (abstraction transformation):
Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp\rangle$ be a symbolic program and $\mathcal{G}_{P^\sharp}$ the symbolic execution graph for $\mathcal{S}_0 \in \Theta^\sharp$. Let α be an abstraction operator and let

$$\mathcal{S}_0 \longrightarrow \mathcal{S}_1 \longrightarrow \ldots \longrightarrow \mathcal{S}_n \longrightarrow \ldots$$

be a path in the graph with $n > 0$. We transform $\mathcal{G}_{P^\sharp}$ into $\mathcal{G}'_{P^\sharp}$ by deleting the children of $\mathcal{S}_n$ (and the edges from $\mathcal{S}_n$ to them) and adding a subgraph with the (possibly infinite) symbolic execution graph rooted with $\alpha(\mathcal{S}_n, \{\mathcal{S}_0, \ldots, \mathcal{S}_{n-1}\})$ and an edge labeled with abs from $\mathcal{S}_n$ to $\alpha(\mathcal{S}_n, \{\mathcal{S}_0, \ldots, \mathcal{S}_{n-1}\})$.

We say that $\mathcal{G}'_{P^\sharp}$ is obtained from $\mathcal{G}_{P^\sharp}$ by abstraction.
Defining appropriate heuristics for applying abstraction is far from trivial. There is a well-known trade off between accuracy and scalability: too much abstraction makes the analysis useless and too little prevents us from applying it to realistic programs. The definition of appropriate abstraction heuristics is an interesting topic for further research that is out of the scope of this paper.

Example 2.13: Consider the program LIST (slightly modified from [5]) shown in Figure 2 and the initial symbolic state $\mathcal{S}_0 \equiv \langle l_0, \{v \mapsto V,\ l \mapsto L,\ n \mapsto \bot\}, true\rangle$, where V denotes an arbitrary integer and L denotes an

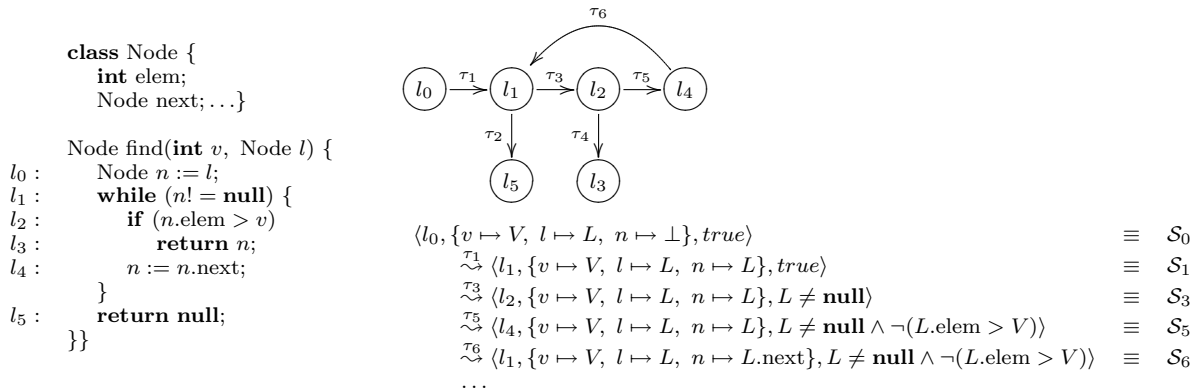

Figure 2. Program LIST, its control flow graph and an infinite symbolic execution.

object pointing to the head of an arbitrary *acyclic* singly linked list. Figure 2 shows one infinite symbolic computation starting from this initial state.[2] In contrast to the situation in Example 2.10, subsumption is not enough to stop this infinite computation since $\mathcal{S}_6 \not\sqsubseteq \mathcal{S}_1$: given the concrete state $s \equiv \langle l_1, \{v \mapsto 42, l \mapsto l_1, n \mapsto l_1.\text{next}\}\rangle$ where l_1 is an arbitrary value of type Node, we have $s \in \gamma(\mathcal{S}_6)$ but $s \notin \gamma(\mathcal{S}_1)$ (since both l and n should point to the same value in all instances of $\mathcal{S}_1$).

Here, we might consider an abstraction operator α that looks for the closest state with the same location and then generalizes the conflicting variables, e.g.,

$$\alpha(\mathcal{S}_6, \{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_5\}) = \\ \langle l_1, \{v \mapsto V, l \mapsto L', n \mapsto L.\text{next}\}, L \neq \textbf{null} \wedge L.\text{elem} > V\rangle$$

With this step, we lose the connection between variables l and n, which might imply a loss of accuracy. Note, however, that this connection is not needed, e.g., for proving program termination (as long as we know that the list is acyclic).

We observe that, in contrast to our approach, abstraction in symbolic execution is typically used to *underapproximate* the computation space, so it does not preserve completeness.[3]

Definition 2.14 (closed symbolic execution graph): Let P be a program and $P^\sharp$ a symbolic version of P. Let $\mathcal{G}$ be a *finite* graph obtained from the symbolic execution graph $\mathcal{G}_{P^\sharp}$ for $\mathcal{S}_0$ by a finite number of subsumption and abstraction transformations such that every leaf is a final state (i.e., no symbolic transition is possible) or it is subsumed by a previous symbolic state (i.e., there is an implicit edge to a previous state). Then, we say that $\mathcal{G}$ is a *closed* symbolic execution graph for P.[4]

[2]We have non-consecutive state numbers since this is only part of the symbolic execution space; the complete symbolic execution space will be seen later in Figure 3.

[3]Actually, [5] already suggests in the conclusion how to compute an *overapproximation* by also evaluating abstracted states, as we do.

[4]The terminology "closed" is taken from the partial evaluation literature.

The closedness of a symbolic execution graph guarantees that all symbolic executions are covered in the graph (i.e., completeness). We note that our closed symbolic execution graphs have some similarities with the *abstract reachability graphs* of [17]; however, completeness in [17] only holds when the graph is finite, which is not always ensured (though some strategies are discussed).

The construction of closed symbolic execution graphs is a well-known problem in the literature of partial evaluation, where appropriate subsumption and abstraction operators have been defined for many different programming languages (specially in the context of declarative programming languages, see e.g., [18], [19], [20], [21]).

Finally, we present the main result of this section, which shows that closed symbolic execution graphs are complete.

Theorem 2.15: Let P be a program and $P^\sharp$ a symbolic version of P. Let $\mathcal{G}$ be a closed symbolic execution graph for $\mathcal{S}_0$ and let $s_0 \in \gamma(\mathcal{S}_0)$. If there exists a (possibly infinite) computation $s_0 \overset{\tau_1}{\rightarrow}_{R_P} s_1 \overset{\tau_2}{\rightarrow}_{R_P} \ldots$ then there exists a (possibly infinite) path $\mathcal{S}_0 \longrightarrow^+ \mathcal{S}_1 \longrightarrow^+ \ldots$ in $\mathcal{G}$ such that $s_i \in \gamma(\mathcal{S}_i)$ for all $i \geq 0$ and each $\mathcal{S}_i \longrightarrow^+ \mathcal{S}_{i+1}$ is either

1) $\mathcal{S}_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$,
2) $\mathcal{S}_i \xrightarrow{\mathsf{sub}} \mathcal{S}'_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$ or
3) $\mathcal{S}_i \xrightarrow{\mathsf{abs}} \mathcal{S}'_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$.

Proof: Consider an arbitrary transition $s_i \overset{\tau_{i+1}}{\rightarrow}_{R_P} s_{i+1}$. By condition (4) of Definition 2.4, we have that, for any $\mathcal{S}_i$ such that $s_i \in \gamma(\mathcal{S}_i)$, the transition $\mathcal{S}_i \overset{\tau_{i+1}}{\rightsquigarrow}_{R^\sharp_P} \mathcal{S}_{i+1}$ holds with $s_{i+1} \in \gamma(\mathcal{S}_{i+1})$. Now, we assume that $\mathcal{S}_i$ belongs to the graph (which is trivial for the first transition, and an easy consequence of the reasoning below) and prove that either (1), (2) or (3) holds. We consider the following possibilities:

- The graph contains an edge $\mathcal{S}_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$. Then, the proof is done.
- The graph contains an edge $\mathcal{S}_i \xrightarrow{\mathsf{sub}} \mathcal{S}'_i$. By Definition 2.9, $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ and, thus, $\gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{S}'_i)$. Therefore, we have $s_i \in \gamma(\mathcal{S}'_i)$. By condition (4) of

Definition 2.4, the transition $\mathcal{S}'_i \overset{\tau_{i+1}}{\rightsquigarrow}_{R^\sharp_\mathcal{P}} \mathcal{S}_{i+1}$ holds with $s_{i+1} \in \gamma(\mathcal{S}_{i+1})$. Since no consecutive subsumption or abstraction steps are allowed, $\mathcal{S}'_i \overset{\tau_{i+1}}{\longrightarrow} \mathcal{S}_{i+1}$ belongs to the graph and the proof is done.

- The graph contains an edge $\mathcal{S}_i \overset{\mathsf{abs}}{\longrightarrow} \mathcal{S}'_i$. By Definition 2.12, $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ and, thus, $\gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{S}'_i)$. Therefore, we have $s_i \in \gamma(\mathcal{S}'_i)$ and the proof proceeds as in the preceding case.

■

Note that a closed symbolic execution graph can always be computed with a finite number of subsumption and abstraction steps (e.g., by fixing a bound on the number of times a program location can be visited).

III. Generation of Rewrite Rules

In this section, we extract a term rewriting system (TRS in the following) from the closed symbolic execution graph, so that we can prove the termination of the original program using the generated TRS.

Actually, the use of term rewriting is not essential and other rule-based formalisms could be used. For instance, while some approaches consider the translation to term rewriting systems (e.g., [13], [14], [10], [22], [12], [23], [24], [11] or [25], [26]), other approaches consider a rule-based language similar to constraint logic programming (e.g., [15], [16]). We have chosen to generate TRSs because of the extensive literature on the termination of these systems and the active research on the development of termination provers (as witnessed by the annual *termination competition* [27]).

A. Integer Term Rewriting

In particular, we consider *integer term rewrite systems* (ITRS), originally introduced in [28]. These systems extend the usual rewrite systems with integers and some basic pre-defined operators. Here, we consider that the TRS's signature is split into three disjoint subsets: $\mathcal{F}$, the defined symbols of the system, $\mathcal{C}$ the data constructors (e.g., the list constructors nil and $cons$), and $\mathcal{F}_{int}$, that contains the integers $\mathbb{Z} = \{0, -1, 1, -2, 2, \ldots\}$, the Boolean values $\mathbb{B} = \{true, false\}$, and the following pre-defined operations

- arithmetic operations (like $+$, $-$, $*$, etc),
- relational operations (like $>$, $\geq$, $<$, etc) and
- Boolean operations (like $\wedge$, $\vee$, etc).

These operators suffice to express path conditions on integer symbolic values. Constraints on data structures like arrays or lists can be expressed by means of terms (see below). In the following, we denote by $\mathsf{Term}(\mathcal{C}, \mathcal{V})$ the (possibly infinite) set of constructor terms with variables and by $\mathsf{Term}(\mathcal{F}_{int}, \mathcal{V})$ the (possibly infinite) set of arithmetic, relational and Boolean expressions with variables.

The rules of an ITRS have the form $l \to r \mid b$, where the following conditions hold:

- The left-hand side l has the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$ is a defined function symbol and $t_i \in \mathsf{Term}(\mathcal{C}, \mathcal{V}) \cup \mathbb{Z} \cup \mathbb{B}$ is a term made of constructor symbols and variables, an integer or a Boolean value, for all $i = 1, \ldots, n$.
- The right-hand side r has the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$ is a defined function symbol and either $t_i \in \mathsf{Term}(\mathcal{C}, \mathcal{V})$ is a constructor term or $t_i \in \mathsf{Term}(\mathcal{F}_{int}, \mathcal{V})$ is an integer term, $i = 1, \ldots, n$. Observe that no nested defined functions are allowed in both the left- and right-hand sides.
- The condition b is an integer constraint including variables, integers, and pre-defined operators.

A rule of the form $l \to r \mid true$ is simply denoted by $l \to r$. We denote variables with capital letters.

Example 3.1: The following ITRS returns a tuple with the maximum element and the sum of all elements from a list of positive integers (built using the list constructors nil and $cons$):

$$\begin{aligned} mslist(L) &\to msl(L, 0, 0) \\ msl(nil, M, S) &\to (M, S) \\ msl(cons(H, T), M, S) &\to msl(T, M, S + H) \mid H \leq M \\ msl(cons(H, T), M, S) &\to msl(T, H, S + H) \mid H > M \end{aligned}$$

By considering integer and Boolean values a special type of 0-ary constructor symbols, and by assuming implicitly that every ITRS contains an infinite set of pre-defined rules $\mathcal{PD}$ for the pre-defined operations on integers and Booleans, the semantics of ITRSs is a simplified form of innermost rewriting (i.e., the counterpart of call-by-value evaluation in functional programming).

For instance, given the ITRS of Example 3.1 above and the initial term

$$mslist(cons(1, cons(3, cons(2, nil))))$$

we have the following reduction sequence (the reduced subterm is underlined):

$$\begin{aligned} & \underline{mslist(cons(1, cons(3, cons(2, nil))))} \\ \to\ & \underline{msl(cons(1, cons(3, cons(2, nil))), 0, 0)} \\ \to\ & msl(cons(3, cons(2, nil)), 1, \underline{0 + 1}) \\ \to\ & \underline{msl(cons(3, cons(2, nil)), 1, 1)} \\ \to\ & msl(cons(2, nil), 3, \underline{1 + 3}) \\ \to\ & \underline{msl(cons(2, nil), 3, 4)} \\ \to\ & msl(nil, 3, \underline{4 + 2}) \\ \to\ & \underline{msl(nil, 3, 6)} \\ \to\ & (3, 6) \end{aligned}$$

B. From Symbolic Execution Graphs to ITRSs

We now introduce a generic transformation that takes a closed symbolic execution graph and returns a finite ITRS. Basically, we produce an ITRS that mimics the transitions of the closed symbolic execution graph. For this purpose, we first introduce a function that produces a term representation for states:

Definition 3.2 (term representation): We introduce a function $\mathsf{tr} : \mathsf{Var} \times \Sigma^\sharp \mapsto \mathcal{T}(\mathcal{C},\mathcal{V}) \cup \mathbb{Z} \cup \mathbb{B}$ that computes the term representation $\mathsf{tr}(x,\mathcal{S})$ for a program variable x according to a symbolic state $\mathcal{S}$. We denote by $\mathsf{tr}(x_1,\ldots,x_n,\mathcal{S})$ the sequence $\mathsf{tr}(x_1,\mathcal{S}),\ldots,\mathsf{tr}(x_n,\mathcal{S})$.

Function tr is extended to symbolic states by: $\mathsf{tr}(\mathcal{S}) = \mathsf{f}_\mathcal{S}(\mathsf{tr}(x_1,\ldots,x_n,\mathcal{S}))$ where $\mathsf{f}_\mathcal{S} \in \mathcal{F}$ is a fresh function symbol uniquely associated to $\mathcal{S}$. We also extend tr to concrete states in the natural way: $\mathsf{tr}(x,\langle l,\sigma\rangle) = \mathsf{tr}(x,\langle l,\sigma,true\rangle)$, i.e., we apply tr to the symbolic state $\langle l,\sigma,true\rangle$ that just represents $\langle l,\sigma\rangle$.

Let us now introduce the extraction of rewrite rules from a closed symbolic execution graph:

Definition 3.3 (ITRS generation): Let $\mathcal{G}$ be a closed symbolic execution graph for a program P. We construct an ITRS as follows:

- The set of defined function symbols $\mathcal{F}$ contains a function symbol $\mathsf{f}_\mathcal{S}$ associated to every symbolic state $\mathcal{S}$ in $\mathcal{G}$.
- We produce a rule[5] $\mathsf{tr}(\mathcal{S})\vartheta_{pc} \to \mathsf{tr}(\mathcal{S}')\vartheta_{pc}\vartheta_\tau \mid i(pc)$, for each edge $\mathcal{S} \xrightarrow{\tau,pc} \mathcal{S}'$, where
  - $\vartheta_{pc} : \mathsf{Var} \mapsto \mathsf{Term}(\mathcal{C},\mathcal{V}) \cup \mathbb{Z} \cup \mathbb{B}$ is a substitution that depends on the path condition pc and might bind some variables to constructor terms, integers or Booleans. For instance, it might bind some variable L to a list $cons(H,T)$ if pc includes the constraint $L \neq \mathbf{null}$. It is intended to *backpropagate* the path condition to the left-hand side of the rule.
  - $\vartheta_\tau : \mathsf{Var} \mapsto \mathsf{Term}(\mathcal{F}_{int},\mathcal{V})$ is a substitution that depends on the transition τ and might bind some variable to an arithmetic expression. For instance, it might bind a variable X to $X+1$ if this is the effect of transition τ on this variable.
  - Finally, $i(pc)$ denotes the integer constraints of the path condition pc (note that we might have non-integer constraints like $L \neq \mathbf{null}$ that are dealt with by instantiating variables using ϑ_{pc}).
- We produce a rule of the form $\mathsf{tr}(\mathcal{S}) \to \mathsf{tr}(\mathcal{S}')$, for each edge $\mathcal{S} \xrightarrow{\mathsf{abs}} \mathcal{S}'$.
- We produce a rule of the form $\mathsf{tr}(\mathcal{S}) \to \mathsf{f}_{\mathcal{S}'}(\mathsf{tr}(x_1,\ldots,x_n,\mathcal{S}))$, for each edge $\mathcal{S} \xrightarrow{\mathsf{sub}} \mathcal{S}'$, where $x_1,\ldots,x_n$ are the program variables.

Observe that the substitutions ϑ_{pc} are used to encode data objects by means of terms (as it is done, e.g., in [12], [14]). This is very natural in the context of term rewriting and gives rise to ITRSs that accurately represent the transitions of the original program.

In our context, we are only interested in *safe* extraction methods:

[5]As it is common in term rewriting, we use postfix notation for substitution application and write $t\vartheta$ instead of $\vartheta(t)$.

Definition 3.4: Let $\mathcal{P}$ be a program, $\mathcal{G}$ be a closed symbolic execution graph and $\mathcal{R}$ be the ITRS extracted from $\mathcal{G}$ according to Definition 3.3 and using a term representation function tr. We say that the extraction method is *safe* if the following conditions hold:

1) $s \in \gamma(\mathcal{S})$ implies that $\mathsf{tr}(s)$ matches $\mathsf{tr}(\mathcal{S})$ (i.e., there exists a variable substitution ϑ such that $\mathsf{tr}(s) = \mathsf{tr}(\mathcal{S})\vartheta$).
2) for all concrete states s, s' such that $s \xrightarrow{\tau}_{R_P} s'$ and for all $\mathcal{S} \xrightarrow{\tau} \mathcal{S}'$ with $s \in \gamma(S)$, $s' \in \gamma(\mathcal{S}')$ and associated rewrite rule $\mathsf{tr}(\mathcal{S})\vartheta_{pc} \to \mathsf{tr}(\mathcal{S}')\vartheta_{pc}\vartheta_\tau \mid i(pc)$, we have

$$\mathsf{tr}(s) = \mathsf{tr}(\mathcal{S})\vartheta_{pc}\delta \to \mathsf{tr}(\mathcal{S}')\vartheta_{pc}\vartheta_\tau\delta \to^*_{\mathcal{PD}} t = \mathsf{tr}(s')$$

where the subsequence $\mathsf{tr}(\mathcal{S}')\vartheta\vartheta'\delta \to^*_{\mathcal{PD}} t$ is used to evaluate integer expressions to values (either integers or variables).

In general, one should require tr to preserve the observable property one is interested in (for proving termination, though, safeness is enough).

Example 3.5: Let us consider the closed symbolic execution graph for program LIST shown in Figure 3. The graph is made finite using the abstraction step described in Example 2.13.

Here, we consider a simple term representation function $\mathsf{tr}(x,\langle l,\theta,pc\rangle)$ that returns the value of a variable x using the bindings of θ. In particular, linked lists are represented with a list data structure built from nil and $cons$ (i.e., nil denotes an empty list and $cons(h,t)$ denotes a list with head h and tail t).

Given a path condition $L = \mathsf{null}$ we produce a substitution $\vartheta_{pc} = \{L \mapsto nil\}$. In contrast, if the path condition is $L \neq \mathsf{null}$ we have $\vartheta_{pc} = \{L \mapsto cons(H,T)\}$ for some fresh symbolic variables H and T. Substitutions ϑ_τ are not used in this example since we have no update on integer variables.

Using this term representation, we get the ITRS depicted in Figure 4. The termination of this TRS can be proved using the termination prover AProVE [8] and its extension for ITRSs [28]. The correctness of our approach then guarantees that the original program LIST is also terminating.

Our final result states the correctness of the overall scheme for proving termination (similar results could be proved for other observable properties).

Theorem 3.6: Let P be a program and $P^\sharp$ a symbolic version of P. Let $\mathcal{G}$ be a closed symbolic execution graph for $\mathcal{S}_0$. Let $\mathcal{R}$ be the ITRS obtained from $\mathcal{G}$ using a safe extraction method. Let $s_0 \in \gamma(\mathcal{S}_0)$. If there exists a (possibly infinite) computation $s_0 \xrightarrow{\tau_1}_{R_P} s_1 \xrightarrow{\tau_2}_{R_P} \ldots$ then there exists a (possibly infinite) reduction sequence in $\mathcal{R}$ starting from $\mathsf{f}_{\mathcal{S}_0}(\mathsf{tr}(x_1,\ldots,x_n,s_0))$.

Proof: By Theorem 2.15, we have that there exists a (possibly infinite) path $\mathcal{S}_0 \longrightarrow^+ \mathcal{S}_1 \longrightarrow^+ \ldots$ in $\mathcal{G}$ such that $s_i \in \gamma(\mathcal{S}_i)$ for all $i \geq 0$ and each $\mathcal{S}_i \longrightarrow^+ \mathcal{S}_{i+1}$ is either (1) $\mathcal{S}_i \xrightarrow{\tau_{i+1},pc_{i+1}} \mathcal{S}_{i+1}$, (2) $\mathcal{S}_i \xrightarrow{\mathsf{sub}} \mathcal{S}'_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$ or

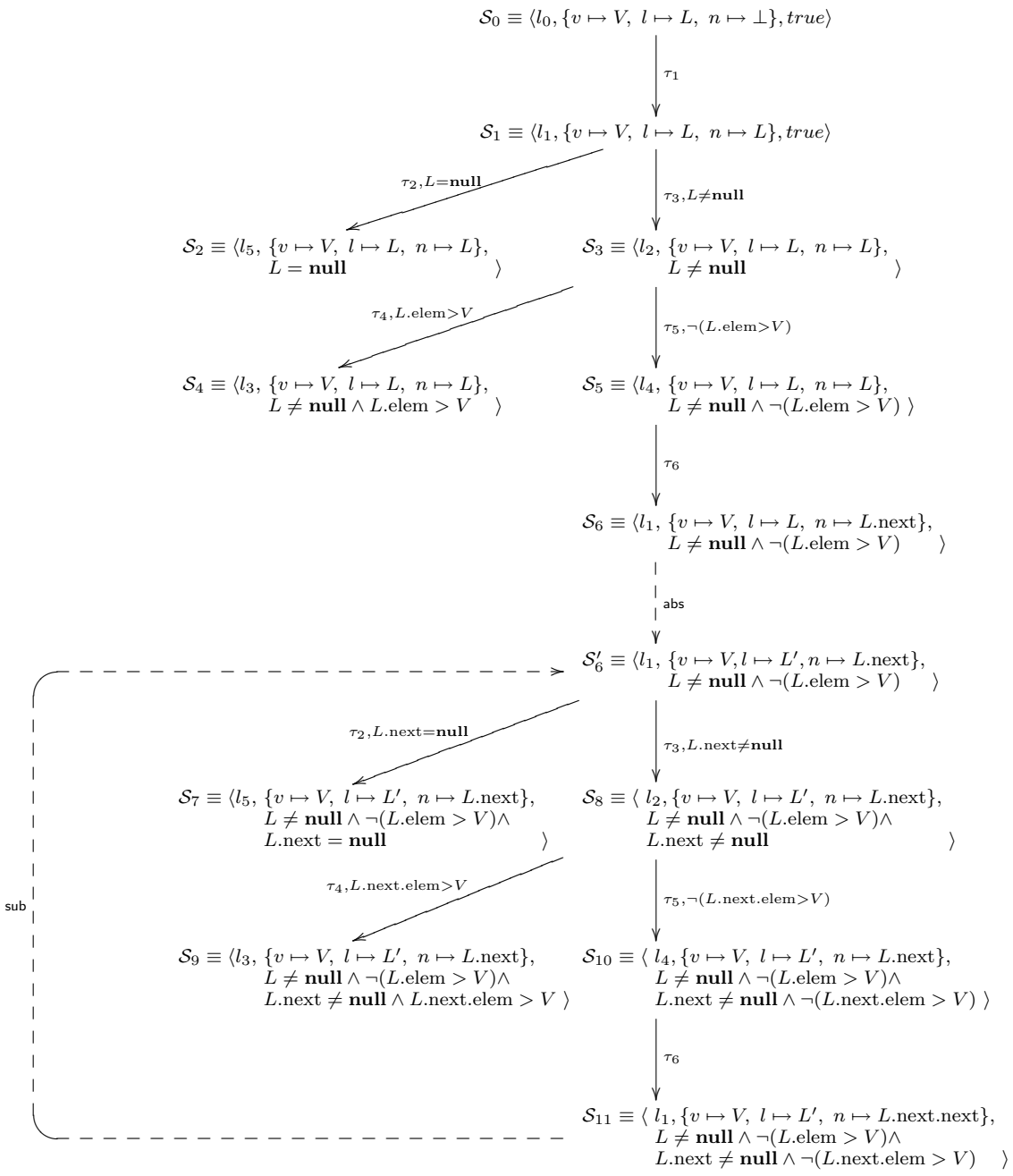

Figure 3. Closed symbolic execution graph for program LIST

$$
\begin{array}{rcll}
\mathsf{f}_{\mathcal{S}_0}(V, L, \bot) & \to & \mathsf{f}_{\mathcal{S}_1}(V, L, L) & \\
\mathsf{f}_{\mathcal{S}_1}(V, nil, nil) & \to & \mathsf{f}_{\mathcal{S}_2}(V, nil, nil) & \\
\mathsf{f}_{\mathcal{S}_1}(V, cons(H,T), cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_3}(V, cons(H,T), cons(H,T)) & \\
\mathsf{f}_{\mathcal{S}_3}(V, cons(H,T), cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_4}(V, cons(H,T), cons(H,T)) & \mid \ H > V \\
\mathsf{f}_{\mathcal{S}_3}(V, cons(H,T), cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_5}(V, cons(H,T), cons(H,T)) & \mid \ H <= V \\
\mathsf{f}_{\mathcal{S}_5}(V, cons(H,T), cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_6}(V, cons(H,T), T) & \\
\mathsf{f}_{\mathcal{S}_6}(V, cons(H,T), T) & \to & \mathsf{f}_{\mathcal{S}'_6}(V, L', T) & \\
\mathsf{f}_{\mathcal{S}'_6}(V, L', nil) & \to & \mathsf{f}_{\mathcal{S}_7}(V, L', nil) & \\
\mathsf{f}_{\mathcal{S}'_6}(V, L', cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_8}(V, L', cons(H,T)) & \\
\mathsf{f}_{\mathcal{S}_8}(V, L', cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_9}(V, L', cons(H,T)) & \mid \ H > V \\
\mathsf{f}_{\mathcal{S}_8}(V, L', cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_{10}}(V, L', cons(H,T)) & \mid \ H <= V \\
\mathsf{f}_{\mathcal{S}_{10}}(V, L', cons(H,T)) & \to & \mathsf{f}_{\mathcal{S}_{11}}(V, L', T) & \\
\mathsf{f}_{\mathcal{S}_{11}}(V, L', T) & \to & \mathsf{f}_{\mathcal{S}'_6}(V, L', T) &
\end{array}
$$

Figure 4. ITRS extracted from the closed symbolic execution graph of Figure 3

(3) $\mathcal{S}_i \xrightarrow{\mathsf{abs}} \mathcal{S}'_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$. Now, case (1) follows straightforwardly by the soundness of the extraction method: for every transition $s_i \overset{\tau_{i+1}}{\to} s_{i+1}$ we have $\mathsf{tr}(s_i) = \mathsf{tr}(\mathcal{S}_i)\vartheta_{pc_{i+1}}\delta \to \mathsf{tr}(\mathcal{S}_{i+1})\vartheta_{pc_{i+1}}\vartheta_{\tau_{i+1}}\delta \to^*_{\mathcal{PD}} \mathsf{tr}(s_{i+1})$. Cases (2) and (3) are immediate consequences of the soundness of the extraction method and the fact that $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ and, thus, $\gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{S}'_i)$ (so every term $\mathsf{tr}(s_i)$ that is an instance of $\mathsf{tr}(\mathcal{S}_i)$ is also an instance of $\mathsf{tr}(\mathcal{S}'_i)$). ■

IV. Symbolic Execution-based Termination Tool

In order to check the viability of the ideas presented so far, we have developed a proof-of-concept implementation of a termination prover for simple imperative programs with integers, basic arithmetic, assignments, conditionals and jumps (there is no explicit iteration but it can easily be encoded with conditionals and jumps). The implemented tool is called SETT: *Symbolic Execution-based Termination Tool*. In its current version, only subsumption has been implemented (nevertheless, we succeeded in all the considered examples even without abstraction steps). A web interface to test the tool is available from `http://kaz.dsic.upv.es/sett/`. Let us illustrate the application of the tool over a couple of simple (though not trivial) examples.

Our first example is taken from [29] (here, `input()` returns a random value provided by the user):

```
while x>0 and y>0 do
    if input() = 1 then
        x := x-1;
        y := input();
    else
        y := y-1;
    fi
done
```

Proving the termination of this program is difficult because there is no ranking function into the natural numbers that can prove its termination. Our tool successfully computed a closed symbolic execution and, then, produced the following ITRS:

```
fun3(x,y) -> if (x>0 and y>0)
                then fun4(x,y)
                else fun10(x,y)
fun4(x,y) -> if (input=1)
                then fun5(x,y)
                else fun8(x,y)
fun5(x,y) -> fun6(x-1,y)
fun6(x,y) -> fun7(x,input)
fun7(x,y) -> fun9(x,y)
fun8(x,y) -> fun9(x,y-1)
fun9(x,y) -> fun3(x,y)
```

where `x`, `y`, and `input` are variables. The termination of this ITRS can be automatically proved using AProVE [8].

Another (difficult) termination problem is taken from [30]:

```
while x>0 and y>0 do
    if (input()) then
        x := x-1;
        y := x;
    else
        x := y-2;
        y := x+1;
    fi
done
```

Again, our tool successfully computed a closed symbolic execution and produced the following ITRS:

```
fun3(x,y) -> if (x>0 and y>0)
                then fun4(x,y)
                else fun11(x,y)
fun4(x,y) -> if (input=1)
                then fun5(x,y)
                else fun8(x,y)
fun5(x,y) -> fun6(x-1,y)
```

```
fun6(x,y) -> fun7(x,x)
fun7(x,y) -> fun10(x,y)
fun8(x,y) -> fun9(y-2,y)
fun9(x,y) -> fun10(x,x+1)
fun10(x,y) -> fun3(x,y)
```

whose termination was also proved using AProVE [8].

More details and examples can be found in the tool webpage `http://kaz.dsic.upv.es/sett/`.

V. Related Work

As mentioned in the introduction, there are already several approaches to proving the termination of programs which follow a similar scheme as the one we have presented. This is the case, e.g., of the works that consider the termination of Haskell [10], Prolog with impure features [11] and Java bytecode [12], [13], [14] by transforming the original termination problem into the problem of analyzing the termination of a rewrite system. COSTA [15], [16], a cost and termination analyzer for Java bytecode, follows a similar pattern but produces a constraint logic program instead.

The novelty of our approach is twofold. On the one hand, we propose a language-independent approach that may ease the design of new program analyzers for different programming languages by clarifying some common principles of these approaches. On the other hand, we reformulate the scheme using well-known principles from partial evaluation, so that the vast literature on constructing finite symbolic executions can be reused (rather than starting from scratch, as some of the above works have done).

Proving that a program terminates for all possible inputs is undoubtedly a fundamental problem that has been extensively studied in the context of term rewriting [7], [31] and logic programming [32], where powerful termination provers exist (see, e.g., the results from the last *termination competition* [27]). In contrast, proving the termination of imperative programs has been mostly overlooked for decades. Recent progress in this area, however, has changed the picture and powerful—and usable—tools have emerged [33].

One popular branch of work is based on the notion of *transition invariants* [34] and applies to both sequential and concurrent programs (see [29] for a recent survey). These techniques aim at identifying a set of invariants that approximate the closure of the transition relation of a program, so that if these invariants are well founded, the considered program is terminating. The main advantage of this method is that his divide-and-conquer approach allows one to search for different well-founded relations rather than a single, monolithic one for the complete program (which is much more difficult in practice). these methods, however, rely on the construction of ranking functions and, thus, our symbolic execution-based approach may be advantageous when the control flow is complex (but can be represented with a finite number of states without losing too much precision). Actually, our preliminary experimental results showed that our scheme succeeds for some typical examples from the transition invariants literature. Unfortunately, a detailed comparison is quite difficult since the main tool based on transition invariants, TERMINATOR, is not publicly available.

Another alternative approach considers the termination of C programs by translating the original program to a term rewrite system [26]. However, in contrast to our approach, the rewrite rules are extracted from the program's syntax. Consequently, it is (faster but) much less accurate since no information is propagated forward in the computations. In order to alleviate this problem, additional static analyses are proposed, though their impact is difficult to measure.

VI. Concluding Remarks

In this paper, we have presented a language-independent approach to proving liveness properties by constructing a closed symbolic execution of the program. Then, we have proposed a method for proving program termination by extracting a rewrite system that reproduces the transitions of symbolic execution. We have illustrated the usefulness of our approach by implementing a proof-of-concept termination prover for imperative programs with integers, basic arithmetic, assignments, conditionals and jumps. Our preliminary results are encouraging and point out the practicality of the approach. Hopefully, this higher level approach will be useful to design new analysis tools—by reusing existing techniques for term rewriting—and to get new insights on the overall process.

Complete symbolic execution is a relatively new area, so there are plenty of topics for further research. In particular, we want to design refined heuristics for subsumption and abstraction. Also, it would be worth studying the definition of an instance of the scheme presented in this paper for a *dynamic* programming language (like JavaScript or Erlang). Proving termination in dynamic languages is a challenging task, but our approach based on symbolic execution might be useful to track reachable states (as witnessed by the success of [10], [11], [12], [13], [16], [14]). An inherent limitation of the current approach is the use of integers, since floats cannot be represented using ITRSs. We plan to consider other rule-based representations in order to overcome this limitation.

Acknowledgment

The author would like to thank the anonymous referees for many useful comments and suggestions. This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Sec. Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

References

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] L. Clarke, "A program testing system," in *Proceedings of the 1976 Annual Conference (ACM'76)*, 1976, pp. 488–491.

[3] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[4] S. Anand, C. S. Pasareanu, and W. Visser, "Symbolic execution with abstract subsumption checking," in *Proc. of SPIN'06*, ser. Lecture Notes in Computer Science, A. Valmari, Ed., vol. 3925. Springer, 2006, pp. 163–181.

[5] ——, "Symbolic execution with abstraction," *STTT*, vol. 11, no. 1, pp. 53–67, 2009.

[6] J. Gallagher, "Tutorial on Specialisation of Logic Programs," in *Proc. of PEPM'93*. ACM, New York, 1993, pp. 88–98.

[7] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] J. Giesl, P. Schneider-Kamp, and R. Thiemann, "AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework," in *Proc. of IJCAR'06*. Springer LNCS 4130, 2006, pp. 281–286.

[9] Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler," *Systems, Computers, Controls*, vol. 2, no. 5, pp. 45–50, 1971.

[10] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann, "Automated termination proofs for Haskell by term rewriting," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, p. 7, 2011.

[11] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann, "Automated termination analysis for logic programs with cut," *TPLP*, vol. 10, no. 4-6, pp. 365–381, 2010.

[12] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl, "Automated Termination Analysis of Java Bytecode by Term Rewriting," in *Proc. of RTA 2010*, ser. LIPIcs, C. Lynch, Ed., vol. 6. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 259–276.

[13] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl, "Termination Graphs for Java Bytecode," in *Verification, Induction, Termination Analysis*, ser. Lecture Notes in Computer Science, S. Siegler and N. Wasser, Eds., vol. 6463. Springer, 2010, pp. 17–37.

[14] M. Brockschmidt, C. Otto, and J. Giesl, "Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting," in *Proc. of RTA 2011*, ser. LIPIcs, vol. 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 155–170.

[15] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode," in *Proc. of FMCO'07*. Springer LNCS 5382, 2008, pp. 113–132.

[16] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini, "Termination analysis of Java bytecode," in *Proc. of FMOODS'08*, ser. Lecture Notes in Computer Science, G. Barthe and F. S. de Boer, Eds., vol. 5051. Springer, 2008, pp. 2–18.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. of POPL*, 2002, pp. 58–70.

[18] A. Glenstrup and N. Jones, "Termination analysis and specialization-point insertion in offline partial evaluation," *ACM TOPLAS*, vol. 27, no. 6, pp. 1147–1215, 2005.

[19] M. Leuschel, B. Martens, and D. De Schreye, "Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 1, pp. 208–258, 1998.

[20] B. Martens and J. Gallagher, "Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance," in *Proc. of ICLP'95*. MIT Press, 1995, pp. 597–611.

[21] G. Vidal, "A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs," in *Proc. of LOPSTR'11*, ser. Lecture Notes in Computer Science, M. Alpuente, Ed., vol. 6564. Springer, 2011, pp. 200–214.

[22] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann, "Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages," in *Proc. of RTA 2006*, F. Pfenning, Ed. Springer LNCS 4098, 2006, pp. 297–312.

[23] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann, "Automated termination analysis for logic programs by term rewriting," in *Proc. of LOPSTR'06*, ser. Lecture Notes in Computer Science, G. Puebla, Ed., vol. 4407. Springer, 2006, pp. 177–193.

[24] ——, "Automated termination proofs for logic programs by term rewriting," *ACM Trans. Comput. Log.*, vol. 11, no. 1, 2009.

[25] S. Falke and D. Kapur, "A term rewriting approach to the automated termination analysis of imperative programs," in *Proc. of CADE'09*, ser. Lecture Notes in Computer Science, R. A. Schmidt, Ed., vol. 5663. Springer, 2009, pp. 277–293.

[26] S. Falke, D. Kapur, and C. Sinz, "Termination Analysis of C Programs Using Compiler Intermediate Languages," in *Proc. of RTA'11*, ser. LIPIcs, M. Schmidt-Schauß, Ed., vol. 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 41–50.

[27] "Annual international termination competition." [Online]. Available: http://www.termination-portal.org/wiki/Termination_Competition

[28] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke, "Proving termination of integer term rewriting," in *Proc. of RTA'09*, ser. Lecture Notes in Computer Science, R. Treinen, Ed., vol. 5595. Springer, 2009, pp. 32–47.

[29] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, no. 5, pp. 88–98, 2011.

[30] A. Podelski and A. Rybalchenko, "Transition invariants and transition predicate abstraction for program termination," in *Proc. of TACAS'11*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 3–10.

[31] Terese, *Term Rewriting Systems*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 55.

[32] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987, second edition.

[33] G. Stix, "Send in the Terminator," *Scientific American Magazine*, November 2006.

[34] A. Podelski and A. Rybalchenko, "Transition invariants," in *Proc. of LICS'04*. IEEE Computer Society, 2004, pp. 32–41.