Looper: Lightweight Detection of Infinite Loops at Runtime

Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
Berkeley, CA, USA
{jburnim,jalbert,chster,ksen}@cs.berkeley.edu

Abstract

When a running program becomes unresponsive, it is often impossible for a user to determine if the program is performing some useful computation or if it has entered an infinite loop. We present LOOPER*, an automated technique for dynamically analyzing a running program to prove that it is non-terminating.* LOOPER *uses symbolic execution to produce simple non-termination arguments for infinite loops dependent on both program values and the shape of heap. The constructed arguments are verified with an off-the-shelf SMT solver. We have implemented our technique in a prototype tool for Java applications, and we demonstrate our technique's effectiveness on several non-terminating benchmarks, including a reported infinite loop bug in open-source text editor jEdit. Our tool is able to dynamically detect infinite loops deep in the execution of large Java programs with no false warnings, producing symbolic arguments that can aid in debugging non-termination.*

1. Introduction

Non-termination bugs are a common reason for unresponsiveness in software applications. When a program becomes unresponsive, there is often no way for the user to tell if the program is doing useful computation or if it has entered an infinite loop. The user is forced to terminate her program after waiting for some time, never knowing if her work in progress could have been saved if she had waited just a little bit longer. Further, manual analysis of an unresponsive program can be a tedious and error-prone process for a software developer, potentially requiring stepping through thousands of program statements in a debugger or examining pages of program traces.

We present a technique for analyzing a running program and automatically proving that the program will never terminate. Our on-demand analysis, called LOOPER, combines concolic execution [1], [2] and basic invariant generation to construct simple non-termination arguments which are checked with a standard SMT solver.

Consider the program in Figure 1. It terminates for some inputs, but will loop forever if, for example, `foo` returns 6 and `bar` returns 1. This program poses several challenges for a static termination or non-termination prover. First, functions `foo` and `bar` may be large and complex or may involve system calls or other interaction with the external environment. Thus, static reasoning or exploration of these functions may be infeasible. Second, the analysis of the loop itself is non-trivial, as symbolic reasoning about the nonlinear update of `x` is difficult and `y` increases without bound. This difficulty is compounded if we replace `(x * x + 2) % 10` with a call `f(x)` to some pure, external function.

Dynamic non-termination analysis, however, started only once this program has become unresponsive, can overcome these challenges. Suppose LOOPER has been invoked on an execution when `foo` returned 6 and `bar` returned 1, and the `while`-loop has already run for some time.

LOOPER observes one iteration of the loop, say where `x` is initially 8 and `y` is initially 52. Parallel to the concrete execution, LOOPER symbolically executes one loop iteration. This symbolic execution treats variables `x` and `y` as symbolic inputs, and infers that: (1) whenever $\texttt{x} = 8$ and $\texttt{y} > 0$, the loop iterates once updating $\texttt{x} \mapsto 6$ and $\texttt{y} \mapsto \texttt{y} + 1$. Note that LOOPER symbolically abstracts `y`, but concretizes $\texttt{x} = 8$ to handle the non-linear arithmetic involving `x`. LOOPER then concretely and symbolically executes a second iteration, inferring that: (2) whenever $\texttt{x} = 6$ and $\texttt{y} > 0$, the loop iterates once updating $\texttt{x} \mapsto 8$ and $\texttt{y} \mapsto \texttt{y} + 1$.

Combining these two abstract iterations, LOOPER can easily prove that an iteration of type (1) or (2) must always be followed by another iteration. That is, after an iteration of either type is executed, `x` is either 6 or 8 and $\texttt{y} > 0$. Thus, `x` will never be 3, `y` will always be positive, and the loop will run forever.

As the above example shows, our approach has several advantages:

- LOOPER need only prove that one observed concrete state leads to non-termination. Thus, it can leverage the evolving concrete state of a running program to soundly simplify its symbolic execution, producing multiple simple abstractions for each observed path through a loop.
- These simple abstractions enable LOOPER to reason

```
main() {
   x = foo();
   y = bar();
   while (x != 3 && y > 0) {
      x = (x * x + 2) % 10;
      y++;
   }
}
```

Fig. 1: An Example Infinite Loop

about loops that depend on non-linear arithmetic, the shape of the heap, or side-effect-free external functions using only basic invariant generation and constraint solving.
- Because LOOPER is a dynamic, on-demand analysis, it need incur no cost until invoked by a user because the target program has become unresponsive. It can be used to analyze potentially-infinite loops deep in the execution of a large application.

We have built a prototype implementation of LOOPER for Java applications. Our prototype successfully detects non-termination in three real-world benchmarks, including an infinite loop bug in the jEdit text editor and a non-terminating Javascript program run in the Rhino interpreter. These loops execute up to 1600 statements per iteration, depend on the shape of the heap, and allocate new objects.

While necessarily incomplete, LOOPER can be effectively applied to automatically analyze infinite loops in real applications. This can allow a user to conclusively determine that a running program will never resume responding. Further, it can provide a programmer with an automatically-constructed non-termination proof, saving tedious and error-prone manual analysis and debugging.

2. Overview

In this section, we give an informal overview of LOOPER using the illustrative example in Figure 2. At a high level, LOOPER alternates between two stages: dynamic symbolic execution and theorem proving. Suppose that LOOPER has begun observing the execution of some program when it is at the head of a loop with concrete program state $\mathcal{M}_{c,0}$. LOOPER will symbolically execute the program in parallel with one full concrete execution through the loop along some path π_1. This process produces a two-part symbolic abstraction of the loop's concrete execution. First, it produces a *symbolic memory map* $\mathcal{M}_1$ – a description of the contents of the program's memory after the iteration as a function of the memory at the beginning of the iteration. Second, it produces a *path constraint* Φ_1 – a predicate over the initial memory contents detailing how the execution of π_1 depended on the program memory. In particular, whenever Φ_1 holds true at the head of the loop, then the program will execute exactly along path π_1 and will update memory exactly as described by $\mathcal{M}_1$. Note that $\mathcal{M}_{c,0}$ will always be one such input satisfying Φ_1.

After this loop iteration, LOOPER will attempt a non-termination proof. Specifically, it will try to prove that whenever Φ_1 holds at the loop head, Φ_1 will hold again immediately after the program executes one loop iteration. We write this $\Phi_1 \implies \Phi_1[\mathcal{M}_1]$ – that is, whenever Φ_1 holds in some concrete memory, then Φ_1 must also hold after the concrete memory has been updated as described by $\mathcal{M}_1$. (Essentially, we form $\Phi_1[\mathcal{M}_1]$ by replacing every occurrence of a variable v in Φ_1 with $\mathcal{M}_1(v)$ – the value of v at the end of the iteration as a function of the memory at the beginning of the iteration.) If this implication is a tautology, then the loop must infinitely execute the path π_1. LOOPER uses an SMT solver to check this implication.

```
 1 int find(Node head, int x) {
 2   int index = 0;
 3   Node p = head;
 4   while (p != null) {
 5     if (p.data == x)
 6       return index;
 7     index++;
 8     p = p.next;
 9   }
10   return -1;
11 }
```

Fig. 2: List Membership Function.

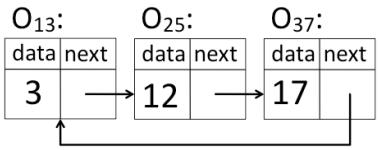

Fig. 3: Circular Linked List.

If the implication is not a tautology, LOOPER will concretely and symbolically execute additional loop iterations, producing $(\mathcal{M}_2, \Phi_2)$, $(\mathcal{M}_3, \Phi_3)$, etc. After each iteration, LOOPER will again try to prove that the program is stuck executing forever along one of the observed paths through the loop. That is, after k iterations, it will check if the following implication is a tautology:

$$\bigwedge_{i \in [1,k]} \left(\Phi_i \implies \bigvee_{j \in [1,k]} \Phi_j \left[\mathcal{M}_j \right] \right)$$

If LOOPER fails to prove non-termination after some fixed number of iterations, it reports that it cannot conclude whether or not the target program will terminate.

2.1. Example

Figure 2 is a Java implementation of a linked-list membership test. When called on a non-degenerate singly-linked list, the function returns the first index at which `x` occurs in the list or -1 if the list does not contain `x`. But when called on a *circular* linked list that does not contain `x`, the function will never terminate.

Consider an execution of the code in Figure 2 on the circular linked list depicted in Figure 3 and with `x` $= 7$. Suppose LOOPER begins its analysis when the program is at Line 5 with `p` pointing to object O_{13}.

Step 1: Symbolic Execution. In the first observed concrete execution of the loop, the program will check that `p` is not null, see that 3 (O_{13}.`data`) does not equal 7 (`x`), and then increment `index` and update `p` to point to O_{25}.

To execute this iteration symbolically, we treat the values of p, x, O_{13}.data, and O_{13}.next as symbolic inputs to the iteration, rather than concrete values. In terms of those inputs, the path constraint and symbolic memory map are:

$$\begin{aligned}\Phi_1 \;&=\; (p \neq \texttt{null}) \wedge (p = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x})\\ \mathcal{M}_1 \;&=\; \{\texttt{p} \mapsto O_{13}.\texttt{next},\;\; \texttt{x} \mapsto \texttt{x},\;\; \texttt{index} \mapsto \texttt{index} + 1,\\ &\quad\; O_{13}.\texttt{data} \mapsto O_{13}.\texttt{data},\;\; O_{13}.\texttt{next} \mapsto O_{13}.\texttt{next}\}\end{aligned}$$

These are obtained by following the concrete execution statement-by-statement, tracking its dependencies and effects in terms of the inputs. E.g., checking the condition at Line 4 adds the constraint (p $\neq$ null), because this condition was true in the concrete execution, while the increment at Line 7 updates $\mathcal{M}_1$ so that $\mathcal{M}_1(\texttt{index}) = \texttt{index} + 1$. (That is, the value of index at the end of the iteration equals one plus the value before the iteration.)

Note that our symbolic execution adds the constraint (p = O_{13}) when accessing a field of p at Line 5. That is, we specialize our symbolic abstraction of the iteration for this value of p, while leaving index fully symbolic.

Step 2: Attempt Non-Termination Proof. LOOPER then constructs the non-termination argument $\Phi_1 \implies \Phi_1[\mathcal{M}_1]$. Specifically, in $\Phi_1[\mathcal{M}_1]$ it substitutes $\mathcal{M}_1(p) = O_{13}.\texttt{next}$ for p and $\mathcal{M}_1(O_{13}.\texttt{data}) = O_{13}.\texttt{data}$ for O_{13}.data, yielding, leaving out the redundant (p $\neq$ null)'s:

$$\begin{aligned}&(p = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x}) \quad \implies\\ &\quad (O_{13}.\texttt{next} = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x})\end{aligned}$$

This implication is not a tautology – O_{13}.next is not O_{13} – so LOOPER continues.

Steps 3-5. Similarly, the symbolic execution of the second and third iterations, where p initially points to O_{25} and O_{37}, will produce:

$$\begin{aligned}\Phi_2 \;&=\; (p \neq \texttt{null}) \wedge (p = O_{25}) \wedge (O_{25}.\texttt{data} \neq \texttt{x})\\ \mathcal{M}_2 \;&=\; \{\texttt{p} \mapsto O_{25}.\texttt{next},\;\; \texttt{x} \mapsto \texttt{x},\;\; \texttt{index} \mapsto \texttt{index} + 1,\\ &\quad\; O_{25}.\texttt{data} \mapsto O_{25}.\texttt{data},\;\; O_{25}.\texttt{next} \mapsto O_{25}.\texttt{next}\}\end{aligned}$$

$$\begin{aligned}\Phi_3 \;&=\; (p \neq \texttt{null}) \wedge (p = O_{37}) \wedge (O_{37}.\texttt{data} \neq \texttt{x})\\ \mathcal{M}_3 \;&=\; \{\texttt{p} \mapsto O_{37}.\texttt{next},\;\; \texttt{x} \mapsto \texttt{x},\;\; \texttt{index} \mapsto \texttt{index} + 1,\\ &\quad\; O_{37}.\texttt{data} \mapsto O_{37}.\texttt{data},\;\; O_{37}.\texttt{next} \mapsto O_{37}.\texttt{next}\}\end{aligned}$$

In Step 4, LOOPER will fail to prove that the first two observed iterations form an infinite loop. It will be able to prove $\Phi_1 \implies \Phi_1[\mathcal{M}_1] \vee \Phi_2[\mathcal{M}_1]$ – that an iteration of the first or second kind always follows an iteration of the first kind. However, it will find that $\Phi_2 \implies \Phi_1[\mathcal{M}_2] \vee \Phi_2[\mathcal{M}_2]$ is not a tautology:

$$\begin{aligned}&(\texttt{p} = O_{25}) \wedge (O_{25}.\texttt{data} \neq \texttt{x}) \quad \implies\\ &\quad (O_{25}.\texttt{next} = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x})\\ &\quad \vee \;\; (O_{25}.\texttt{next} = O_{25}) \wedge (O_{25}.\texttt{data} \neq \texttt{x})\end{aligned}$$

Step 6: Non-Termination Proof. After symbolically executing three iterations, LOOPER is able to prove that the program is in an infinite loop. It constructs the three implications $\Phi_i \implies \Phi_1[\mathcal{M}_i] \vee \Phi_2[\mathcal{M}_i] \vee \Phi_3[\mathcal{M}_i]$, one for each of $i = 1, 2, 3$. For example, for $i = 1$, omitting the redundant (p $\neq$ null)'s:

$$\begin{aligned}&(\texttt{p} = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x}) \implies\\ &\quad (O_{13}.\texttt{next} = O_{13}) \wedge (O_{13}.\texttt{data} \neq \texttt{x})\\ &\quad \vee \;\; (O_{13}.\texttt{next} = O_{25}) \wedge (O_{25}.\texttt{data} \neq \texttt{x})\\ &\quad \vee \;\; (O_{13}.\texttt{next} = O_{37}) \wedge (O_{37}.\texttt{data} \neq \texttt{x})\end{aligned}$$

In constructing these implications, LOOPER notes that x, the three data fields, and three next fields, are all read but never written. Thus, it can add as simple invariants that each of these symbolic inputs equals its fixed, concrete value. With these invariants – e.g. ($O_{13}.\texttt{next} = O_{25}$), (x = 7), and ($O_{25}.\texttt{data} = 12$) for the $i = 1$ case above – all three implications are clearly tautologies.

Discussion. Note that in the above proof we do not need a sophisticated theory of select-and-store or shape analysis, as the logical structure of the circular list is not modified in the infinite loop. Thus, we can leverage the dynamic state to concretize our non-termination argument so that it can be verified by a simple SMT solver handling only integer linear arithmetic. This same technique allows us to handle non-linear arithmetic and even calls to pure methods whose source is unavailable. We call our approach *lightweight* to contrast it with approaches requiring more powerful and more expensive reasoning and non-linear solving. In the next section, we formally present this technique.

3. Symbolic Execution for Non-Termination

In this section, we present the formal basis of our technique for using symbolic execution to reason about the potential non-termination of a running program. First, we present a simple imperative object-oriented language on which we will illustrate the technique. In Section 3.2, we discuss what it means precisely for a program in this language to be in an infinite loop, and sketch a way to reason abstractly about such loops. In Section 3.3, we describe the key aspects of our symbolic execution and show how it abstracts a loop's execution. In Section 3.4, we formalize our reasoning about these abstractions.

3.1. Programming Model and Concrete Semantics

We describe our technique on a simple Java-like imperative language with objects, fields, and dynamic allocation. It is not difficult to extend the model to support most Java features, such as method calls, exceptions, and arrays.

A program P is a sequence of labeled statements, with each statement one of: (1) a HALT statement, (2) an assignment $lv := e$ to lvalue lv of the value of side-effect-free expression e, (3) a conditional if e then goto l, where l is the label of another statement and e is a side-effect-free expression, or

(4) an allocation $lv :=$ `new` T assigning to lvalue lv a newly-allocated object of type T.

The base values in our language are objects, booleans, and unbounded integers. Lvalues lv are either program variables v or field references $v.f$. Expressions can include lvalues, constants, and arithmetic operators, such as + and <. As in Java, all lvalues must be assigned before they are first read.

The concrete state of a running program consists of a set of objects $\mathcal{O}$ and a *concrete memory map* $\mathcal{M}_c : X \sqcup (F \times \mathcal{O}) \rightarrow V$, where X denotes the set of program variables, F the set of field names, and V the set of concrete program values – i.e. unbounded integers, booleans, and references to the objects in $\mathcal{O}$.

Program statements have their natural semantics. Lvalues are looked up in the current state – $\mathcal{M}_c(x)$ for variable x and $\mathcal{M}_c(f, \mathcal{M}_c(v))$ for field $v.f$ – and assignments update $\mathcal{M}_c$. An allocation `new T()` returns an unused o from $\mathcal{O}$.

Note that we use somewhat abstracted semantics – unbounded integers and unlimited memory – rather than strict machine semantics. In Java or C, loops "`while (x > 0) { ...; x++; }`" or "`while (e) { x := new T; ... }`" are not technically infinite loops due to integer overflow and limited heap space. However, we believe it is valuable to report such "logical" or "algorithmic" infinite loops. Such algorithmic infinite loops are very likely bugs, especially when they degrade a program's responsiveness. Our technique can be applied to machine semantics, as well.

3.2. Proving a Loop is Infinite

Suppose we believe that a program is stuck infinitely taking paths $\pi_1, \ldots, \pi_k$ through a loop. Each of these π_i induces a partial map $\pi_i : \mathit{ConcreteStates} \rightarrow 2^{\mathit{ConcreteStates}}$, which takes a concrete memory which leads to execution along π_i to the set of memories that can result from running π_i. These induced maps are *partial* or *guarded transition functions*. (Allocation `new T()` introduces non-determinism because it can return any new, unique object from $\mathcal{O}$.)

Proposition 3.1. *Let $\pi_1, \ldots, \pi_k$ be paths through a loop in running program P and let $\mathcal{M}_{c,0}$ be the concrete state of P at the loop head. Let R denote the closure of $\{\mathcal{M}_{c,0}\}$ under the functions induced by the π_i. Then, the program will loop forever along paths $\pi_1, \ldots \pi_k$ iff*

$$R \subseteq \bigcup_i \mathit{dom}(\pi_i)$$

The set R in Proposition 3.1 is the set of all concrete states at the loop head that result from applying any of the π_i any number of times to initial state $\mathcal{M}_{c,0}$. The proposition says that the π_i form an infinite loop if and only if, however many times any of the π_i have been executed, it is always possible to execute another one of the π_i. In the language of [3], set R is *recurrent* for the π_i.

Rather than reason directly about transition functions π_i and set R, we will use symbolic execution to generate abstractions of the π_i. In Definition 3.2, we formally define such an abstraction.

Definition 3.2. *Function $f : \mathit{ConcreteStates} \rightarrow 2^{\mathit{ConcreteStates}}$ **abstracts** π iff*

$$\mathit{dom}(f) \subseteq \mathit{dom}(\pi) \;\; \mathit{and} \;\; \mathcal{M}_c \in \mathit{dom}(f) \implies \pi(\mathcal{M}_c) \subseteq f(\mathcal{M}_c)$$

*That is, f is applicable only when π is, and applying f always yields (at least) the actual concrete states. We call f an **abstract** guarded transition function.*

In Proposition 3.3, we show how such abstractions can be used to prove non-termination. In essence, it suffices to find a set of abstractions $f_1, \ldots, f_m$ of some given $\pi_1, \ldots, \pi_k$ so that the abstractions form a *closed* transition system. That is, for each f_h and for all $\mathcal{M}_c \in \mathrm{dom}(f_h)$, we have $f_h(\mathcal{M}_c) \subseteq \bigcup_j \mathrm{dom}(f_j)$.

Proposition 3.3. *Let $f_1, \ldots, f_m$ be **abstract** guarded transition functions, each abstracting one of $\pi_1, \ldots, \pi_k$. Suppose that $\mathcal{M}_{c,0} \in \bigcup_j \mathit{dom}(f_j)$ and that $f_1, \ldots, f_m$ is a **closed** transition system. Then,*

$$R \subseteq \bigcup_j \mathit{dom}(f_j) \subseteq \bigcup_i \mathit{dom}(\pi_i)$$

and thus, by Proposition 3.1, the π_i will loop forever.

The first inclusion is true by induction on the number of applications of any π_i to $\mathcal{M}_{c,0}$ and because $f_1, \ldots, f_m$ is closed. The second inclusion follows trivially from the fact that each $\mathrm{dom}(f_j)$ is contained in some $\mathrm{dom}(\pi_i)$.

3.3. Symbolic Execution

We use combined symbolic and concrete execution – i.e. concolic execution – to produce sound abstractions of the observed paths π through a loop in running program P. Recall that X, $\mathcal{O}$, and F denote the program variables, objects, and field names of an executing program P. Let $\mathcal{E}$ denote the set of all linear expressions over X and $F \times \mathcal{O}$.

Then, symbolic execution along some concretely-executed path π yields: (1) a partial **symbolic memory map** $\mathcal{M} : X \sqcup (F \times \mathcal{O}) \rightarrow \mathcal{E} \sqcup \{\bot\}$ which symbolically encodes the updates π makes to memory in terms of the memory before π is executed, and (2) a **path constraint** Φ, a predicate over $X \sqcup (F \times \mathcal{O})$.

The details of computing $\mathcal{M}$ and Φ during the statement-by-statement concolic execution of path π are standard [2], [1]. Initially, $\mathcal{M}$ is empty and $\Phi = \mathit{true}$. Assignment statements $lv := e$ set the mapping for $\mathcal{M}(lv)$ to $\mathcal{M}(e)$, the expression resulting from evaluating e in $\mathcal{M}$. Conditional statements `if` e `then goto` l update Φ to either $\Phi \wedge \mathcal{M}(e)$ or $\Phi \wedge \neg\mathcal{M}(e)$, depending on whether e is true or false in the concrete execution. Whenever some expression goes beyond the power of our decision procedure, linear arithmetic in this case, we simplify by replacing symbolic variables with their concrete values.

We introduce three key changes to our concolic execution:

1) We treat all memory read by path π as symbolic inputs. That is, whenever we find no mapping $\mathcal{M}(l)$ when looking up some location $l \in X \sqcup (F \times \mathcal{O})$, we introduce $\mathcal{M}(l) = l$.

2) Whenever we simplify an expression using concrete values, we constrain Φ so that the simplification is sound. For example, in executing `x = y*y`, we not only set `x` to the concrete value $\mathcal{M}_c(\texttt{y})^2$, but we add to Φ the constraint $(\mathcal{M}(\texttt{y}) = \mathcal{M}_c(\texttt{y}))$ that the symbolic expression $\mathcal{M}(\texttt{y})$ stored for `y` equals its concrete value $\mathcal{M}_c(y)$. Similarly, whenever we dereference a field `p.f`, to read or write it, we constrain $(\mathcal{M}(\texttt{p}) = \mathcal{M}_c(\texttt{p}))$. That is, the symbolic pointer $\mathcal{M}(\texttt{p})$ stored for `p` must point to its concrete target.
This is necessary so that our abstraction is sound – i.e. that whenever concrete inputs satisfy Φ, $\mathcal{M}$ accurately captures the updates made by π.

3) We also abstract any allocation `new T()` done by π. Specifically, we treat `new T()` as having symbolic value $\perp$. And we do not add to Φ any constraints involving $\perp$. This is sound because any constraint comparing a newly-allocated object to itself, another object, or `null`, must either always be true or always be false. (Because, e.g., `new T()` can never be a previously-allocated object.) Further, we remember all concrete objects $o_1, \ldots, o_n$ allocated by π. At the end of our concolic execution, we remove from $\mathcal{M}$ all symbolic mappings $\mathcal{M}(\texttt{f}, o_i)$ to fields of those newly-allocated objects.
This is also necessary for soundness – the symbolic abstraction of π must not depend on the particular object (i.e. address) returned by an allocation.

Thus, our concolic execution guarantees that: (1) Every solution to Φ is an input that drives execution down π – i.e. in $\text{dom}(\pi)$. (2) Every update made by π is captured accurately by $\mathcal{M}$, except those to fields of newly-allocated objects. Thus, $\text{dom}(\mathcal{M})$ is the set of all locations read or written by π, except for fields of new objects. (3) $\mathcal{M}(l) = \perp$ for any location to which π stores the address of a newly-allocated object. We can now argue that this symbolic execution *abstracts* a path π in the sense of Definition 3.2:

Definition 3.4. *For path constraint and symbolic memory map $(\Phi, \mathcal{M})$, define an abstract guarded transition function $\alpha[\Phi, \mathcal{M}]$: ConcreteStates $\rightarrow 2^{\text{ConcreteStates}}$ by $\mathcal{M}'_c \in \alpha[\Phi, \mathcal{M}](\mathcal{M}_c)$ iff:*

- *Whenever $l \in$ dom$(\mathcal{M}_c)$ and $l \notin$ dom$(\mathcal{M})$, then $\mathcal{M}'_c(l) = \mathcal{M}_c(l)$.*
- *Whenever $l \in$ dom$(\mathcal{M})$ and $\mathcal{M}(l) \neq \perp$, then $\mathcal{M}'_c(l) = \mathcal{M}_c(\mathcal{M}(l))$*
  (where $\mathcal{M}_c(\mathcal{M}(l))$ denotes expression $\mathcal{M}(l)$ evaluated in $\mathcal{M}_c$.)

The domain of $\alpha[\Phi, \mathcal{M}]$ is the set of all concrete states $\mathcal{M}_c$ satisfying Φ. Note that Φ specifies only the domain, while $\mathcal{M}$ specifies the transition function itself.

Proposition 3.5. *Suppose $(\Phi, \mathcal{M})$ is the result of symbolically executing π as described in Section 3.3. Then, $\alpha[\Phi, \mathcal{M}]$ abstracts π.*

Proof: We do not formally prove this proposition, but the key to the argument is as follows. Let $\mathcal{M}_c$ be some concrete state in the domain of $\alpha[\Phi, \mathcal{M}]$. That is, $\mathcal{M}_c$ satisfies Φ, so $\mathcal{M}$ accurately records all updates made by executing π on $\mathcal{M}_c$ to variables and preexisting objects. Then, for any $\mathcal{M}'_c \in \pi(\mathcal{M}_c)$:

- If $l \in \text{dom}(\mathcal{M}_c)$ and $l \notin \text{dom}(\mathcal{M})$, then l is a variable or field of a preexisting object and thus is not set by π. Therefore, $\mathcal{M}'_c(l) = \mathcal{M}_c(l)$.
- If $l \in \text{dom}(\mathcal{M})$ and $\mathcal{M}(l) \neq \perp$, then $\mathcal{M}$ accurately captures the update made by π. Thus, $\mathcal{M}'_c(l) = \mathcal{M}_c(\mathcal{M}(l))$ – i.e. expression $\mathcal{M}(l)$ evaluated in $\mathcal{M}_c$.

So, by Definition 3.4, $\mathcal{M}'_c \in \alpha[\Phi, \mathcal{M}](\mathcal{M}_c)$. Further, every $\mathcal{M}_c$ satisfying Φ is in $\text{dom}(\pi)$, so $\text{dom}(\alpha[\Phi, \mathcal{M}]) \subseteq \text{dom}(\pi)$. Therefore, $\alpha[\Phi, \mathcal{M}]$ abstracts π. □

3.4. Constructing a Non-Termination Argument

Suppose we have symbolically executed several iterations $\pi_1, \ldots, \pi_k$ of a loop in a program P, yielding $(\Phi_1, \mathcal{M}_1), \ldots, (\Phi_k, \mathcal{M}_k)$. We wish to use these symbolic abstractions to prove, via Proposition 3.3, that the loop is nonterminating.

We now show how to construct a $\Phi_j[\mathcal{M}_i]$ and connect nontermination arguments $\Phi_i \implies \Phi_1[\mathcal{M}_i] \vee \cdots \vee \Phi_k[\mathcal{M}_i]$ to proofs via Proposition 3.3.

Definition 3.6. *For path constraint Φ_j and symbolic memory map $\mathcal{M}_i$, we define $\Phi_j[\mathcal{M}_i]$ as follows. Let L denote the set of all locations from $X \sqcup (F \times \mathcal{O})$ appearing in formula Φ_j. If for any $l \in L$ either $l \notin$ dom$(\mathcal{M}_i)$ or $\mathcal{M}_i(l) = \perp$, then $\Phi_j[\mathcal{M}_i] =$ false. Otherwise, $\Phi_j[\mathcal{M}_i]$ is the predicate obtained by substituting for every occurrence of each $l \in L$ the expression $\mathcal{M}_i(l)$.*

Proposition 3.7. *Suppose $\Phi_i \implies \Phi_1[\mathcal{M}_i] \vee \cdots \vee \Phi_k[\mathcal{M}_i]$ for each i. Then, the abstract transitions $\alpha[\Phi_1, \mathcal{M}_1], \ldots, \alpha[\Phi_k, \mathcal{M}_k]$ form a closed system.*

Proof: Let $\mathcal{M}_c$ be a concrete state in the domain of $\alpha[\Phi_i, \mathcal{M}_i]$ – i.e. a $\mathcal{M}_c$ satisfying Φ_i. Then, $\mathcal{M}_c$ must also satisfy some $\Phi_j[\mathcal{M}_i]$.

Let $\mathcal{M}'_c \in \alpha[\Phi_i, \mathcal{M}_i](\mathcal{M}_c)$. For every location l in Φ_j, we have $l \in \text{dom}(\mathcal{M}_i)$ and $\mathcal{M}_i(l) \neq \perp$, so by Definition 3.4 we have $\mathcal{M}'_c(l) = \mathcal{M}_c(\mathcal{M}_i(l))$. Therefore, $\Phi_j(\mathcal{M}'_c)$ must equal $\Phi_j[\mathcal{M}_i](\mathcal{M}_c)$.

That is, $\alpha[\Phi_i, \mathcal{M}_i](\mathcal{M}_c)$ is contained in the domain of some $\alpha[\Phi_j, \mathcal{M}_j]$ for every such $\mathcal{M}_c$. Thus, the $\alpha[\Phi_1, \mathcal{M}_1], \ldots, \alpha[\Phi_k, \mathcal{M}_k]$ form a closed system. □

Therefore, if we prove each $\Phi_i \implies \Phi_1[\mathcal{M}_i] \vee \cdots \vee \Phi_k[\mathcal{M}_i]$ is a tautology, then by Proposition 3.3, program P will loop forever along paths $\pi_1, \ldots, \pi_k$.

Algorithm 1 LOOPER ($\mathcal{P}$)

```
1: Input: Executing program P
2: repeat
3:    head ⇐ select a new statement for potential loop head
4:    run P until it reaches statement head
5:    I ⇐ {} // initialize the set of abstract iterations
6:    repeat
7:       (Φ, M) ⇐ symbolically and concretely execute P until
         head is reached again
8:       I ⇐ I ∪ (Φ, M)
9:       Γ ⇐ generateInvariants(I)
10:      σ ⇐ isNonTerminating(I, Γ) // package and send to SMT
         solver
11:      if σ then
12:         return σ // report an infinite loop
13:      end if
14:   until iteration limit is reached
15: until every possible loop head statement has failed
16: return false // report failure
```

3.5. Limitations

Although LOOPER handles some infinite loops which modify the heap (e.g. the jEdit case study loop grows a linked list), it typically cannot reason about loops where non-termination depends on the particulars of heap mutation in every loop iteration. This is because our symbolic execution is conservative in concretizing pointers, and our symbolic reasoning insufficiently powerful. We believe combining our techniques with shape analysis and more powerful invariant generation and proving would be valuable future work.

Further, LOOPER does not currently handle infinite recursion. However, dealing with recursion is a fairly straightforward extension.

4. Algorithm

We present the overall LOOPER algorithm in Algorithm 1. LOOPER takes as input a running program $\mathcal{P}$, identifies potential loop heads and loop iterations, symbolically executes these iterations, and repeatedly attempts to prove that the program will never terminate. We describe these steps below.

Loop Head Selection. A loop head is a statement in program $\mathcal{P}$ to which the observed execution repeatedly returns. We must select a loop head in order to get paths π that loop back to the head – i.e. the iterations. Note that, for certain loops, some loop heads may allow for the construction of a non-termination proof while others may not.

LOOPER iterates through the possible loop head statements one by one, attempting to prove that each gives rise to abstract iterations for which we can construct a non-termination proof.

Symbolic Execution. Once we have chosen a loop head, we identify loop iterations by watching executing program $\mathcal{P}$ until it returns to the loop head. We perform symbolic execution on each observed iteration starting with an empty symbolic state, and then add the resulting abstract iteration to set $\mathcal{I}$. After each such iteration, we attempt to construct a non-termination proof from all of the iterations we have collected so far.

Invariant Generation. In this step, we attempt to generate simple arithmetic invariants about each symbolic variable. The invariants are global to the set of iterations we have in $\mathcal{I}$. Any invariant generation procedure can be plugged in here, although we needed only simple invariant templates of the form $x = n$, $x \leq n$, and $x \geq n$ for our case studies. If we are able to determine that a symbolic variable remains fixed for all iterations in $\mathcal{I}$, we replace it with its concrete value, maintaining soundness and simplifying theorem proving

Proving Non-Termination. Once we have a set of invariants, we construct the implications asserting that our abstract iterations form a closed abstract transition system, as described in the previous section. We then use an off-the-shelf SMT solver to check if these implications are tautologies.

5. Implementation

Our prototype implementation targets Java programs. We use the Soot compiler framework [4] to instrument Java classes at the bytecode level. Our prototype uses a heuristic to identify loop heads. Conceptually, we iterate over all potential loop heads.

We designed our prototype to enable easy substitution of invariant generators and SMT solvers. We currently have implemented our own simple template-based invariant generation and use the off-the-shelf Yices [5] SMT solver to verify invariants and generate non-termination proofs. The simple invariants that we generate are of the form $x = n$, $x \leq n$, and $x \geq n$, and we compute n by simply observing the concrete values of x. Due to our use of the concrete program state to simplify our non-termination arguments, as well as the simple form of our invariants, the generation of invariants and proof of non-termination is relatively inexpensive—i.e. *lightweight*.

Real-time performance was not a design goal for LOOPER because the identification of infinite loops is intended to be performed *on-demand*. The unoptimized LOOPER prototype has a runtime overhead up to 10,000X because of I/O costs of printing a trace statement for each instruction. We expect that an optimized implementation of LOOPER would have a more typical 10-100X slowdown.

Further, LOOPER can be implemented so that it incurs no overhead until activated by the user. With dynamic class reloading, uninstrumented classes can be replaced with instrumented classes during program execution. We believe that dynamic class reloading would make a 100X overhead acceptable as it would only need to be paid when the program becomes unresponsive, and the typical infinite loop iterations we have observed are reasonably short.

Our prototype models Java features which have the potential to jeopardize the soundness of our proof generation. The key is to ensure that a memory satisfying a generated path constraint, Φ_k, will always follow the same path through iteration k. Our implementation soundly handles object field accesses, array accesses, virtual method calls, calls to certain pure native methods, and exceptions.

```
1  int offset = start;
2  HyperSearchResult lastResult = null;
3  for (int counter = 0; ; counter++) {
4     int line = buffer.getLineOfOffset(offset);
5     boolean startOfLine =
6       (buffer.getLineStartOffset(line) == offset);
7
8     buffer.getText(offset,end - offset,text);
9     Match match = matcher.nextMatch(
10        new SegmentCharSequence(text, false),
11        startOfLine, endOfLine, counter==0, false);
12    if (match == null) break;
13
14    newLine =
15      buffer.getLineOfOffset(offset + match.start);
16    if ((lastResult == null)
17        || (lastResult.line != newLine)) {
18      lastResult = new HyperSearchResult(...);
19      ...
20    }
21
22    lastResult.addOccur(offset + match.start,
23                        offset + match.end);
24    offset += match.end;
25    resultCount++;
26 }
```

Fig. 4: jEdit regular expression bug

6. Case Studies

We tested our LOOPER prototype on three Java benchmarks: a reported non-termination bug in the jEdit text editor, a reported non-termination bug extracted from the Ganymede network directory management system, and the infinite traversal of a cyclic linked-list in a Javascript program run in the Rhino Javascript interpreter. These benchmarks provide evidence that our technique can be effectively applied to detect infinite loops in large, real-world Java applications.

6.1. jEdit

jEdit (www.jedit.org) is a text editor oriented towards programmers containing 100K lines of code. We tested LOOPER on an infinite loop bug reported to the project's bug tracker on version jEdit4.3pre6. The bug is triggered by searching through a file using the one-character regular expression for beginning (^) or end of line ($). This causes jEdit to enter a search loop where it updates an offset in the text buffer with the length of each found match. Since that length is zero for these particular inputs, the search never progresses and results in an infinite loop.

For this case study, we instrumented the jEdit byte code, opened the program, and manually triggered the infinite loop by searching for the ^ symbol. We terminated the execution after several seconds. This allowed us to generate a trace of the infinite loop, which we then fed to the LOOPER back end for proof generation.

Figure 4 shows part of the jEdit code that the infinite loop traverses. The actual search for the regular expression pattern occurs in the call to the `SearchMatcher` method `nextMatch` in line 9. The result of the call is the object `match` that contains the start and end offset of the text that matches the regular expression.

When the search request is a regular expression whose match in the line has size 0 then the result of the call will satisfy `match.start == match.end`. Examples of such regular expressions are the caret (^) and dollar ($) symbols that match the beginning and end of a line respectively.

The only way to leave the loop is the result of the call to the `nextMatch` method to be null or equivalently the regular expression to not have any more matchings in the text.

The variable `lastResult` maintains a list of search occurrences of the search item in the current line. If it is the first time through the loop, `lastResult` will be null and will be initialized on line 18. If a match is found in a new line, i.e. the `newLine` offset is different than the current `lastResult` line, a new `lastResult` will be allocated as well. Thus, the code between the lines 14 and 20 is not relevant in the steady state of the infinite loop case.

The new match is inserted in the list of matchings in line 22 and the offset is updated in 24. The loop becomes infinite when the size of the match is equal to 0. In that case, `match.start == match.end == 0` and `offset` in line 24 will not be updated. Therefore we will get to the start of the loop having the same offset value, passing the same input to `nextMatch` and having increased the size of the matchings list `lastResult` by one.

Our LOOPER prototype generates a proof of the non-termination of this loop. The size of the detected infinite loop, including all statements that are executed inside the method calls of the loop, is 1356 symbolic execution statements which translates to approximately 350 java statements. LOOPER generates 52 path constraints and 37 symbolic variables. It fixes 32 of those symbolic variables to their concrete values and produces 4 simple invariants.

Every match found in the loop is represented by an object that is added to the front of a list of occurrences of the search. In our case, an infinite number of these objects are allocated. Note here that we distinguish between abstract non-termination and machine semantic non-termination. Eventually, this loop will terminate due to lack of heap space. However, this loop does not terminate in an abstract sense, and we feel this is important to report to the user. This case study also demonstrates LOOPER's robustness in the face of heap mutation. Each iteration mutates the heap, but our tool is able to ignore these changes because they do not flow symbolically into the path constraints and therefore they do not affect the non-termination proof.

6.2. Ganymede

Our second case study of is an infinite loop present in the Ganymede project. Ganymede is a network directory management system containing 100K lines of Java code and 250 distinct Java classes. In an older version of the program,

```
1   public XMLItem getNextItem()
2       throws SAXException
3   {
4     XMLItem item = null;
5     while (item == null) {
6       item = reader.getNextItem();
7       if (item instanceof XMLError)
8         throw new SAXException(item.toString());
9       if (item instanceof XMLWarning) {
10        err.println("Warning!: " + item);
11        item = null;
12      }
13    }
14    return item;
15  }
```

Fig. 5: Ganymede Infinite Loop

there was a bug that led to an infinite loop when the system was given an end-of-file XML stream to parse.

Figure 5 shows the buggy helper method in the server which reads an item from an XML stream and checks it for any errors or warnings. When given an end-of-file XML stream, the `reader` object can get into a state where it will always return null on a call to `getNextItem()`. The null return will fail both `instanceof` checks and the loop will go on to the next iteration, only to continually get null back from `reader`.

For this case study, we extracted the relevant portions of the loop from the server source and created a test harness that would activate the loop.

We applied the LOOPER prototype to an execution of our test harness. This loop was small compared to the previous case study, made up of 90 symbolic execution statements. The loop followed a single path of execution and only one iteration was needed to reason that the loop was infinite. LOOPER generated five path constraints, and introduced seven symbolic variables. All of these were concretized and no invariants were required for the proof.

6.3. Rhino

Rhino (www.mozilla.org/rhino) is a Javascript engine written entirely in Java. It is comprised of roughly 182 classes and 62K lines of code. Since Rhino interprets Javascript code, any Javascript code that has infinite loops will make the Rhino interpreter loop infinitely as well. The Javascript code that we used as input to Rhino is a Javascript implementation of the Java circular linked list example in Section 2 shown in Figure 6.

LOOPER generates three loop abstractions consisting of roughly 1600 Java statements each (6768 intermediate symbolic execution statements) over the course of generating the non-termination proof. The path constraints generated have 863 conjuncts mostly due to a very large switch statement whose variable is the type of the current bytecode interpreted. Every unsuccessful comparison to a case expression results in an additional constraint. In total, 83 symbolic variables are introduced, all but one of which are concretized.

```
1   function member(head, x) {
2       var p = head;
3       while (p !== null) {
4           if (p.data === x) {
5               return true;
6           }
7           p = p.next;
8       }
9       return false;
10  }
```

Fig. 6: Javascript Circular Linked List Example

The body of this loop is found very deep in the execution because the Rhino engine has a very big set-up phase. The first invocation of the function which interprets the loop occurs after approximately 150,000 symbolic execution statements. This illustrates the benefits of dynamic detection of infinite loops as finding this loop could pose a challenge for static methods.

6.4. Discussion

To further validate our approach, we would like to evaluate LOOPER on additional benchmarks. So far, these three infinite loops are the only ones which we have been able to reproduce and on which we have been able to run our Java tracing infrastructure.

For these three case studies, the cost of running LOOPER was dominated by the overhead of our Java instrumentation for producing a trace to be symbolically executed. Generating invariants, constructing a non-termination argument, and proving the non-termination all together required less than a second for each case study.

7. Related Work

Gupta et al. [3] propose a method of searching for non-terminating program executions and implement it in a tool called TNT. TNT uses concolic test generation [1], [2] to find candidate executions containing loops. For analysis of programs with machine semantics – i.e. bounded integers and bitwise operators – a bit-precise SMT solver is used to determine if there are inputs which cause the loop to execute infinitely. For linear programs with unbounded integers, TNT uses sophisticated invariant generation like that of [6], [7], [8] to find loop invariants that lead to non-termination and which hold for some program input. A backtracking nonlinear constraint solver is used to compute such an invariant.

Although Gupta et al. focus on exhaustively searching for non-terminating executions, their techniques can be applied at runtime, as well. Similarly, our techniques could perhaps be combined with an exhaustive enumeration of candidate loops using dynamic test generation.

TNT's sophisticated invariant generation enables it to detect more linear arithmetic loops than our work. By using the

concrete program state to simplify our non-termination arguments, as well as by using only simple invariants, LOOPER can use an off-the-shelf and highly optimized SMT solver rather than more powerful and more expensive reasoning. Further, by generating multiple simple abstractions for each observed path through a loop, LOOPER can reason about loops whose non-termination depends on the shape of the heap, such as the infinite traversal of a circular linked list. TNT cannot currently catch non-termination that depends on shape assumptions on data structures. LOOPER can also reason about loops combining bounded nonlinear or bitwise arithmetic with unbounded linear arithmetic, or loops which irregularly alternate between different loop paths.

Velroyen et al. [9] have proposed another approach for statically finding non-terminating inputs for a program. Their technique relies heavily on automated, iterative invariant generation, and may be less scalable than TNT.

Proving the termination of programs is a complementary technique which can be used in concert with our approach. Terminator [10] combines work from [11] to reduce the problem of proving termination to checking the disjunction of a set of rank functions using binary reachability analysis. There have also been extensions of Terminator to target heap manipulating programs [12], [13] and multithreaded programs [14]. Other work on proving program termination includes [15], [16].

There has also been work in proving non-termination and termination in other areas, such as term-rewrite systems [17] and functional programs [18].

As described in Section 3.3, we use a concolic execution similar to that of DART [1] and CUTE [2]. Like in those works, we use the concrete state to simplify pointer and nonlinear expressions, but we add additional constraints to ensure that our symbolic abstraction is sound. And, as our concolic execution is on only a piece of a whole, running program, we treat all memory read in each loop iteration as symbolic inputs, rather than having pre-specified inputs. Further, our abstraction of allocation appears to be novel.

8. Conclusion

We have presented a lightweight, dynamic algorithm, LOOPER, that can prove loops are non-terminating at runtime. LOOPER uses a combination of symbolic execution, concrete execution, and theorem proving to soundly diagnose loops. We have described a prototype implementation of LOOPER and our experience running on infinite loops found in several large, real-world programs. Although we have implemented LOOPER for Java it can be easily extended to other programming languages. We believe that LOOPER is a simple and practical tool that can prove non-termination in real-life programs and provide useful debugging information.

9. Acknowledgements

This work supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906 and CCF-0747390, and by a DoD NDSEG Graduate Fellowship.

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Programming language design and implementation*. ACM, 2005, pp. 213–223.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE-13*. ACM, 2005, pp. 263–272.

[3] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," in *POPL*. ACM, 2008, pp. 147–158.

[4] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java optimization framework," in *CASCON 1999*, 1999, pp. 125–135.

[5] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *Computer Aided Verification*, ser. LNCS, vol. 4144, 2006, pp. 81–94.

[6] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," *Computer Aided Verification*, 2003.

[7] D. Kapur, "Automatically generating loop invariants using quantifier elimination," in *IMACS Intl. Conf. on Applications of Computer Algebra*, vol. 116, 2004.

[8] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using gröbner bases," in *POPL '04*. New York, NY, USA: ACM, 2004, pp. 318–329.

[9] H. Velroyen and P. Rümmer, "Non-termination checking for imperative programs," in *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, ser. LNCS, vol. 4966. Springer, 2008, pp. 154–170.

[10] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *Programming language design and implementation*. ACM, 2006, pp. 415–426.

[11] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," *Lecture notes in computer science*, pp. 239–251, 2003.

[12] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn, "Automatic termination proofs for programs with shape-shifting heaps," *Computer Aided Verification*, 2006. [Online]. Available: http://dx.doi.org/10.1007/11817963_35

[13] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn, "Variance analyses from invariance analyses," in *Proceedings of the 2007 POPL Conference*, vol. 42, no. 1. ACM New York, NY, USA, 2007, pp. 211–224.

[14] B. Cook, A. Podelski, and A. Rybalchenko, "Proving thread termination," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM New York, NY, USA, 2007, pp. 320–330.

[15] A. R. Bradley, Z. Manna, and H. B. Sipma, "The polyranking principle," *Automata, Languages and Programming*, pp. 1349–1361, 2005.

[16] P. Cousot, "Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming," in *Proc. VMCAI*, vol. 3385. Springer, 2005, pp. 1–24.

[17] J. Giesl, P. Schneider-Kamp, and R. Thiemann, "AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4130, p. 281, 2006.

[18] C. Lee, N. Jones, and A. Ben-Amram, "The Size-Change Principle for Program Termination," in *Annual Symposium on Principles of Programming Languages: Proceedings of the 28 th ACM SIGPLAN-SIGACT symposium on Principles of programming languages: London, United Kingdom*, vol. 2001, 2001, pp. 81–92.