Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts

Tom Bergan Dan Grossman Luis Ceze

University of Washington, Department of Computer Science & Engineering

Technical Report UW-CSE-13-08-01 *(revised 11/2013)*

Abstract

We describe an algorithm to perform symbolic execution of a multithreaded program starting from an arbitrary program context. We argue that this can enable more efficient symbolic exploration of deep code paths in multithreaded programs by allowing the symbolic engine to jump directly to program contexts of interest.

The key challenge is modeling the initial context with reasonable precision—an overly approximate model leads to exploration of many infeasible paths during symbolic execution, while a very precise model would be so expensive to compute that computing it would defeat the purpose of jumping directly to the initial context in the first place. We propose a *context-specific dataflow analysis* that approximates the initial context cheaply, but precisely enough to avoid some common causes of infeasible-path explosion. This model is necessarily approximate—it may leave portions of the memory state unconstrained, leaving our symbolic execution unable to answer simple questions such as "which thread holds lock A?" or "which object does pointer X refer to?" in all cases. For these cases, we describe novel algorithms for evaluating *symbolic pointers* and *symbolic synchronization* during symbolic execution. Our symbolic execution semantics are sound and complete up to the limits of the underlying SMT solver. We describe initial experiments on an implementation in Cloud9.

1. Introduction

Symbolic execution is a program analysis technique for systematically exploring all feasible execution paths. The idea is to execute programs with symbolic rather than concrete inputs and use an SMT (SAT Modulo Theory) solver to prune infeasible paths. On branches with more than one feasible resolution, the symbolic state is forked and all feasible resolutions are explored. The key advantage of this approach is precision—unlike other techniques, such as abstract interpretation, symbolic execution is generally free of false positives because its semantics are fully precise up to the limits of the underlying SMT solver, and recent advances in SMT solving have made symbolic execution faster and more practical.

Symbolic execution has been used to find bugs and generate high-coverage test cases with great success [5, 13, 14, 25]. Unfortunately, building scalable symbolic execution engines is difficult because of path explosion: the number of feasible execution paths is generally exponential in the length of an execution. Path explosion is even worse when symbolic execution is applied to multithreaded programs [4, 18], which suffer from an explosion of possible thread interleavings in addition to the explosion of single-threaded paths. Prior work has dealt with path explosion using summarization [12, 15, 23], path merging [16, 19], search heuristics [5, 21], and partial order reductions [10].

Our goal is scalable symbolic execution of multithreaded programs written in the C language and its derivatives. Our approach is to limit path explosion by symbolically executing relatively small fragments of a program in isolation—this reduces path length, which in turn reduces the potential for path explosion. Rather than exploring ways that program fragments might be selected, this paper focuses on a more basic question: how do we symbolically execute a fragment of a multithreaded program in isolation, *soundly* and *efficiently*? Prior work has largely assumed that symbolic execution will begin at one of a few natural starting points, such as program entry (for whole-program testing) or a function call (for single-threaded unit testing). We do not make such an assumption—we allow program fragments to begin anywhere—so our main challenge is to perform symbolic execution of multithreaded programs from *arbitrary* program contexts.

1.1 Problem Statement and Solution Overview

Specifically, we address the following problem: given an *initial program context*, which we define to be a set of threads and their program counters, how do we efficiently perform symbolic execution starting from that context while soundly accounting for all possible concrete initial states? We solve this problem in two parts. First, we use a *context-specific dataflow analysis* to construct an over-approximation of the initial state for the given program context. Second, we integrate that analysis with a novel *symbolic execution semantics* that can execute forward from an abstract initial state, even when precise information about pointers and synchronization is not available.

Constructing an Initial State. The most precise strategy is to symbolically execute all paths from program entry to the initial context, and then use path merging [16] to construct an initial state. This is not scalable—it suffers from exactly the sort of path explosion problems we are trying to avoid. Instead, we must *approximate* the initial state. The least precise approximation is to leave the initial state completely unconstrained, for example by assigning a fresh symbolic variable to every memory location. This is too conservative—it covers many memory states that never occur during any actual execution—and as a result, symbolic execution would investigate many infeasible paths.

Our solution represents a middle ground between the above two extremes: we use a *context-specific* dataflow analysis to construct a sound *over-approximation* of the initial state. We use an over-approximation to ensure that all feasible concrete initial states are included. Our analysis includes constraints on the initial memory state as well as constraints on synchronization, such as *locksets*, that together help symbolic execution avoid infeasible paths.

To illustrate, suppose we are asked to begin symbolic execution from the program context marked by arrows in Figure 1. This context includes two threads, each of which is about to execute line 11. Can lines 11 and 12 of `Foo` execute concurrently with lines 11 and 12 of `Bar`? To answer this we must first answer a different question: does thread t_2 hold any locks at the beginning of the program context (*i.e.*, at line 11)? Here we examine the locksets embedded in our initial state and learn that t_2 holds lock `b->lock`.

```
1  global int X,Y
2  global struct Node { Lock lock, int data } nodes[]
3
4  Thread 1                    Thread 2
5    void RunA() {               void RunB() {
6      i = ...                     k = ...
7      Foo(&nodes[i])              Bar(&nodes[k])
8    }                           }
9    void Foo(Node *a) {         void Bar(Node *b) {
10     for (x in 1..X) {           lock(b->lock)
11 ⇒     lock(a->lock)       ⇒   for (y in 1..Y)
12       ...                         ...
```

Figure 1. A simple multithreaded program that illustrates the challenges of beginning symbolic execution at an arbitrary program context. Especially notable are challenges that arise from explicit synchronization and from C-like pointers.

Next, we ask a second question: does `a==b`? Suppose our dataflow analysis determines that `i==k` at line 6, and that `Foo` and `Bar` are called from `RunA` and `RunB` only. In this case, we know that `a==b`, which means that line 11 of `Foo` cannot execute concurrently with line 11 of `Bar`.

Symbolic Execution Semantics. The input to symbolic execution is an abstract initial state constructed by our dataflow analysis. The output is a set of pairs $(\texttt{path}, \texttt{C})$, where `path` is a path of execution and `C` describes a *path constraint* such that when `C` is satisfied on the initial state, the `path` can be followed. To support multithreaded programs, we make each `path` a serialized (sequentially consistent) trace of a multithreaded execution.

The key novelty of our symbolic semantics is the way it integrates with our dataflow analysis. For example, we exploit locksets, as described above, along with other invariants to improve the precision of various *symbolic synchronization* primitives. We reason about the initial values of local variables by exploiting *reaching definitions* that our dataflow analysis computes. We additionally exploit a static points-to analysis to help reason about aliasing relationships between pairs of *symbolic pointers*. Notably, our semantics can reason about symbolic pointers that may refer to the interior bytes of an object.

However, our dataflow analysis is necessarily conservative, leaving us unable to precisely answer simple questions such as "which object does pointer X refer to?" in all cases. For example, suppose our dataflow analysis cannot determine if `i==k` at line 6. In this case, we must investigate two paths during symbolic execution: one in which `a==b`, and another in which `a!=b`. For this reason, the set of paths explored by symbolic execution may be a *superset* of the set of paths that are actually feasible.

Soundness and Completeness. Our symbolic semantics are sound and complete up to the limits of the underlying SMT solver. By *sound*, we mean that if our symbolic execution outputs a pair $(\texttt{path}, \texttt{C})$, then, from every concrete initial state that satisfies constraint `C`, concrete execution *must* follow `path` as long as context switches are made just as in `path`. By *complete*, we mean that symbolic execution outputs a set of pairs $(\texttt{path}, \texttt{C})$ sufficient to cover *all* possible concrete initial states that may arise during any valid execution of the program. However, our analysis is incomplete in practice: first, SMT solvers are incomplete in practice, and second, the set of feasible paths can be too large to practicably enumerate.

1.2 Applications

Our techniques have a variety of promising applications:

Focused Testing of Program Fragments. We can test an important parallel loop in the context of a larger program. Classic symbolic execution techniques require executing deep code paths from program entry to reach the loop in the first place, where these deep paths may include complex initialization code or prior parallel phases. Our techniques enable testing the loop directly, using a fast and scalable dataflow analysis to summarize the initial deep paths.

Testing Libraries. We would ideally test a concurrent library over all inputs and calling contexts, but as this is often infeasible, we instead might want to *prioritize* the specific contexts a library is called from by a specific program. One such prioritization strategy is to enumerate all pairs of calls into the library that may run concurrently, then treat each pair as a program context that can be symbolically executed using our techniques. Then do the same for every triple of concurrent calls, every quadruple, and so on.

Piecewise Program Testing. Rather than testing a whole program with one long symbolic execution, we can break the program into adjacent fragments and test each fragment in isolation. Such a piecewise testing scheme might enumerate fragments *dynamically* by queuing the next fragments found to be reachable from the current fragment. Fragments might end at loop backedges, for loops with input-dependent iteration counts, producing a set of fragments that are each short and largely acyclic. The key potential advantage is that we can explore fragments in parallel, as they are enumerated, enabling us to more quickly reach a variety of deep paths in the program's execution. The trade-off is a potential loss of precision, as our dataflow analysis may make conservative assumptions when constructing a fragment's initial abstract state.

Execution Reconstruction. We can record an execution with periodic lightweight checkpoints that include call stacks and little else. Then, on a crash, we can symbolically execute from a checkpoint onwards to reconstruct the bug. Variants of this approach include *bbr* [7] and *RES* [27]. However, *bbr* does not work for multithreaded programs, and both systems have less powerful support for pointers than does our semantics.

Input-Covering Schedules. Our symbolic execution techniques can be used as part of an algorithm for finding *input-covering schedules* [1]. For example, the algorithm from that prior work [1] partitions execution into adjacent fragments and uses symbolic execution to analyze each fragment in isolation. In §6, we evaluate the effectiveness of our symbolic execution techniques in the context of this algorithm.

1.3 Contributions and Outline

We propose a framework for solving the basic problem—symbolic execution from arbitrary multithreaded contexts. While prior work has largely focused on path explosion due to branches, our work focuses on path explosion due to pointers and synchronization, which are often symbolic in our context. In particular, our integration of dataflow analysis and symbolic execution (§2.3, §3.2, §4.4) and our algorithms for symbolically evaluating pointers (§3.1) and synchronization (§4.2–§4.3) are novel, to the best of our knowledge.

We start with a simple, single-threaded imperative language that has no pointers (§2). We then add pointers (§3) and threads (§4). At each step, we explain how we overcome the challenges introduced by each additional language feature. We then state soundness and completeness theorems (§5), discuss our implementation and empirical evaluation (§6), and end with related work (§7).

2. A Simple Imperative Language

Figure 2 gives the syntax of *Simp*, a simple imperative language that we use as a starting point. A program in this language contains a set of functions, including a distinguished `main` function for program entry. The language includes function calls, conditional branching, mutable local variables, and a set of standard arithmetic and boolean expressions (only partially shown). We separate side-effect-free expressions from statements. This simple language does *not* include pointers, dynamic memory allocation, or threads—those language features will be added in §3 and §4.

$$
\begin{array}{rll}
r \in \textit{Var} & & \textit{(local variables)} \\
x, y \in \textit{SymbolicConst} & & \textit{(symbolic constants)} \\
f \in \textit{FName} & & \textit{(function names)} \\
i \in \mathbb{Z} & & \textit{(integers)}
\end{array}
$$

$$
\begin{array}{rcl}
v \in \textit{Value} & ::= & f \mid i \\
e \in \textit{Expr} & ::= & v \mid r \mid x \mid e \wedge e \mid e \vee e \mid e < e \mid \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\gamma \in \textit{StmtLabel} & & \\
s \in \textit{Stmt} & ::= & r \leftarrow e(e^*) \\
& \mid & \texttt{br}\ e,\ \gamma_t,\ \gamma_f \\
& \mid & \texttt{return}\ e \\
\textit{Func} & ::= & \texttt{func}\ f(r^*)\{\ (\gamma : s;)^*\ \}
\end{array}
$$

Figure 2. Syntax of *Simp*. Asterisks (*) denote repetition.

The concrete semantics follow the standard pattern for imperative, lexically-scoped, call-by-value languages. We omit the detailed rules for brevity. Note that we use r to refer to local variables (or "registers"), while the metavariables x and y do not appear in the actual concrete language. Instead, x and y are used to name *symbolic constants* that represent unknown values during symbolic execution, as described below.

Challenges. Although this language is simple, it reveals two ways in which symbolic execution from arbitrary contexts can be imprecise. Specifically, we use this language to demonstrate imprecision due to unknown calling contexts (§2.2) and unknown values of local variables (§2.3). We also use this language to present basic frameworks that we will reuse in the rest of this paper.

2.1 Symbolic Semantics Overview

We now describe an algorithm to perform symbolic execution of *Simp* programs. Our algorithm operates over symbolic states that contain the following domains (also illustrated in Figure 3):

- $\overline{\mathcal{Y}}$, which is a stack of local variable bindings. A new stack frame is pushed by each function call and popped by the matching return. Variables are bound to either function arguments (for formal parameters) or the result of a statement (as in $r \leftarrow f()$).
- *CallCtx*, which names the current calling context, where the youngest stack label is the thread's current program counter and older labels are return addresses.
- *path*, which records an execution trace.
- *C*, an expression that records the current path constraint.

Constructing an Initial State. Recall from §1.1 that our job is to perform symbolic execution from an arbitrary program context that is specified by a set of program counters, one for each thread. As *Simp* is single-threaded, the initial program context for *Simp* programs contains just one program counter, γ_0.

Given γ_0, where γ_0 is a statement in function f_0, we must construct an initial symbolic state, S_{init}, from which we can begin symbolic execution. A simple approach is: $\textit{path}_{init} = \textit{empty}$; $\textit{C}_{init} = \text{true}$; $\textit{CallCtx}_{init} = \{\gamma_0\}$; and $\overline{\mathcal{Y}}_{init}$ contains one stack frame that maps each $r_i \in f_0$ to a distinct symbolic constant x_i. We describe a more precise approach in §2.3.

Correspondence of Concrete and Symbolic States. Note that we use *symbolic constants*, such as x_i, above, to represent unknown parts of a symbolic state. This allows each symbolic state to represent a *set* of concrete states. Specifically, the set of concrete states represented by S_{init} can be found by enumerating the total set of assignments of symbolic constants x_i to values v_i—each such assignment corresponds to a concrete state in which $x_i = v_i$.

$$
\begin{array}{rcll}
\overline{\mathcal{Y}} & : & \text{Stack of } (\textit{Var} \rightarrow \textit{Expr}) & \textit{(local variables)} \\
\textit{CallCtx} & : & \text{Stack of } \textit{StmtLabel} & \textit{(calling context)} \\
\textit{path} & : & \text{List of } \textit{StmtLabel} & \textit{(execution trace)} \\
\textit{C} & : & \textit{Expr} & \textit{(path constraint)}
\end{array}
$$

Figure 3. Symbolic state for *Simp*.

Symbolic Execution. At a high level, symbolic execution is straightforward. We begin from the initial state, S_{init}. We execute one statement at a time using $step$, which is defined below. At branches, we use an SMT solver to determine which branch edges are feasible and we fork as necessary. We repeatedly execute $step$ on non-terminated states until all states have terminated or until a user-defined resource budget has been exceeded. We define $step$ as follows, and we also make use of an auxiliary function $eval$ to evaluate side-effect-free expressions:

- $step : (\textit{State} \times \textit{Stmt}) \rightarrow \text{Set of } \textit{State}$
  Evaluates a single statement under an initial state and produces a set of states, as we may *fork* execution at control flow statements to separately evaluate each feasible branch. The type of each *State* is given by Figure 3.
- $eval : ((\textit{Var} \rightarrow \textit{Expr}) \times \textit{Expr}) \rightarrow \textit{Expr}$
  Given $eval(\mathcal{Y}, e)$, we evaluate expression e under binding $\mathcal{Y}$, where $\mathcal{Y}$ represents a single stack frame. We expect that $\mathcal{Y}$ has a binding for every local variable referenced by e. Note that $eval$ returns an *Expr* rather than a *Value*, as we cannot completely reduce expressions that contain symbolic constants.

Our algorithm's final result is a set of *State*s from which we can extract (*path*, *C*) pairs that represent our final output. For each such pair, *C* is an expression that constrains the initial symbolic state, S_{init}, such that when *C* is satisfied, program execution *must* follow the corresponding *path*.

SMT Solver Interface. Our symbolic semantics relies on an SMT solver that we query using the following interface. The function *isSat*(C, e), shown below, determines if boolean expression e is satisfiable under the constraints given by expression C, where C is a conjunction of assumptions. In addition to *isSat*, we use *mayBeTrue* and *mustBeTrue* as syntactic sugar, as defined below.

$$
\begin{array}{l}
\textit{isSat}(C, e) = \text{true iff } e \text{ is satisfiable under } C \\
\textit{mayBeTrue}(C, e) = \textit{isSat}(C, e) \\
\textit{mustBeTrue}(C, e) = \neg\textit{mayBeTrue}(C, \neg e)
\end{array}
$$

If a query *isSat*(C, e) cannot be solved, then our symbolic execution becomes *incomplete*. In this case, we concretize enough subexpressions of e so the query becomes solvable and we can make forward progress (similarly to Pasareanu *et al.* [22]).

2.2 Dealing with an Underspecified *CallCtx*

Recall that the initial program context is simply a single program counter, γ_0. If γ_0 is not a statement in the `main` function, then the initial state S_{init} does *not* have a complete call stack. How do we reconstruct a complete call stack?

We could start with a single stack frame and then lazily expand older frames, forking as necessary to explore all paths through the static call graph. However, we consider this overkill for our anticipated applications, and instead opt to exit the program when the initial stack frame returns. Our rationale is that, for each application listed in §1.2, either the program fragment of interest will be lexically scoped, in which case we never return from the initial stack frames anyway, or complete call stacks will be provided, which we can use directly (*e.g.*, we expect that complete call stacks will be available during execution reconstruction, as in *bbr* [7]).

2.3 Initializing Local Variables with Reaching Definitions

The simple approach for constructing S_{init}, as described above, is imprecise. Specifically, the simple approach assigns each local variable a unique symbolic constant, x_i, effectively assuming that each local variable can start with *any* initial value. This is often not the case. For example, consider thread t_1 in Figure 1. In this example, assuming that `RunA` is the only caller of `Foo`, the value of local variable `a` is known precisely. Even when the initial value of a variable cannot be determined precisely, we can often define its initial value as a symbolic function over other variables.

Our approach is to initialize $\overline{\mathcal{Y}}_{init}$ using an interprocedural dataflow analysis that computes *reaching definitions* for all local variables. We use a standard iterative dataflow analysis framework with function summaries for scalability, and we make the framework *context-specific* as follows: First, we combine a static call graph with each function's control-flow graph to produce an interprocedural control-flow graph, *CFG*. Then, we compute the subset of this graph, $CFG_0 \subseteq CFG$, that includes only those flow edges that might occur on some interprocedural path that starts at `main` and ends at the initial program counter, γ_0. We analyze flows in CFG_0 only, effectively summarizing all paths that end at γ_0.

We use a standard *must-reach* analysis applied to CFG_0. Specifically, we compute a set of pairs $R_{local} = \{(r_i, e_i)\}$, where each r_i is a local variable in $\overline{\mathcal{Y}}_{init}$ such that the assignment $r_i \leftarrow e_i$ must-reach the initial program context. That is, r_i's value at the initial program context *must* match expression e_i. We compute R_{local} using standard flow functions for reaching definitions, then assign each e_i to r_i in $\overline{\mathcal{Y}}_{init}$. Some variables may not have a must-reach assignment—these variables, r_k, do not appear in R_{local}, and they are assigned a unique symbolic constant x_k in $\overline{\mathcal{Y}}_{init}$, as before.

As we compute R_{local}, each assignment $r \leftarrow e$ generates a must-reach definition $(r, eval(R_{local}, e))$ on its outgoing dataflow edge. Note that we use $eval$ to reduce expressions. Thus, given $r_1 \leftarrow r_2{+}5$, where $(r_2, x) \in R_{local}$, we generate the definition $(r_1, x{+}5)$ to express that r_1 and r_2 are functions of the same value.

Must-Reach vs. May-Reach. Must-reach definitions provide a sound *over-approximation* of $\overline{\mathcal{Y}}_{init}$, as any variable not included in the must-reach set may have *any* initial value. More precision could be achieved through *may-reach* definitions; however, this would result in a symbolic state with many large disjunctions that are expensive to solve in current SMT solvers [16, 19].

3. Adding Pointers

Figure 4 shows the syntax of *SimpHeaps*, which adds pointers and dynamic memory allocation to *Simp*. As a convention, we use p to range over expressions that should evaluate to pointers.

Memory Interface. We represent pointers as pairs $\texttt{ptr}(l, i)$, where l is the base address of a heap object and i is a non-negative integer offset into that object. Pointers may also be `null`. Pointer arithmetic is supported with the $\texttt{ptradd}(p, e)$ expression, which is evaluated as follows in the concrete language:

$$\frac{eval(\mathcal{Y}, p) = \texttt{ptr}(l, i) \qquad eval(\mathcal{Y}, e) = i'}{eval(\mathcal{Y}, \texttt{ptradd}(p, e)) = \texttt{ptr}(l, i + i')}$$

The heap is a mapping from locations to objects, and each object includes a sequence of fields. Our notion of a field encompasses the common notions of array elements and structure fields. To simplify the semantics, we assume that each field has a uniform size that is big enough to store any value. Following that assumption, we define i to be the offset of the $(i{+}1)$th field (making 0 the offset of the first field), and we define the size of an object to be its number of fields. Heap objects are allocated with `malloc`, which returns $\texttt{ptr}(l, 0)$ with a fresh location l, and they are deallocated with `free`.

$$\begin{array}{rl} l \in \textit{Loc} & \textit{(heap locations)} \\ v \in \textit{Value} & ::= \ldots \mid \texttt{null} \mid \texttt{ptr}(l, i) \\ e, p \in \textit{Expr} & ::= \ldots \mid \texttt{ptr}(l, e) \mid \texttt{ptradd}(p, e) \\ s \in \textit{Stmt} & ::= \ldots \mid r \leftarrow \texttt{load}\ p \mid \texttt{store}\ p,\ e \\ & \quad \mid r \leftarrow \texttt{malloc}(e) \mid \texttt{free}(p) \end{array}$$

Figure 4. Syntax additions for *SimpHeaps*.

$$\begin{array}{rl} \mathcal{H} & : \textit{Loc} \rightarrow \{\text{fields} : (\textit{Expr} \rightarrow \textit{Expr})\} \\ \mathcal{A} & : \text{List of } \{\text{x} : \textit{SymbolicConst},\ \text{primary} : \textit{Loc},\ \text{n} : \textit{PtrNode}\} \end{array}$$

Figure 5. Symbolic state additions for *SimpHeaps*, including a *heap* ($\mathcal{H}$) and a list of *aliasable objects* ($\mathcal{A}$).

Memory Errors. Out-of-bounds memory accesses, uninitialized memory reads, and other memory errors have undefined behavior in C [17]. We treat these as runtime errors in our semantics to simplify the notions of *soundness* and *completeness* of symbolic execution. The details of dynamic detectors for these errors are orthogonal to this paper and are not discussed in detail.

Challenges. In the concrete language, `load` and `store` statements always operate on values of the form $\texttt{ptr}(l, i)$. The symbolic semantics must consider three additional kinds of pointer expressions: $\texttt{ptr}(l, e)$, in which the offset e is symbolic; and x and $\texttt{ptradd}(x, e)$, in which the heap location is symbolic as well.

3.1 Symbolic Semantics

We now extend our symbolic execution algorithm for *SimpHeaps*. As shown in Figure 5, we add two fields to the symbolic state: a heap, $\mathcal{H}$, which maps concrete locations to dynamically allocated heap objects, and a list $\mathcal{A}$, which tracks aliasing information that is used to resolve symbolic pointers. Key rules for the semantics described in this section are given in Figure 6. The $\overset{mem}{\Longrightarrow}$ relation is used by $step$ to evaluate memory statements. (Note that memory operations never fork execution in the absence of memory errors, and we elide those error-checking details from this paper.)

Accessing Concrete Locations. We first consider accessing pointers of the form $\texttt{ptr}(l, e)$. In this case, l uniquely names the heap object being accessed, so we simply construct an expression in the theory of arrays [11] to load from or store to offset e of that object's `fields` array.

Accessing Symbolic Locations. Now we consider accessing pointers of the form x and $\texttt{ptradd}(x, e)$. This case is more challenging since the pointer x may refer to an unknown object. Following [7], our approach is to assign each symbolic pointer x a unique *primary object* in the heap, then use aliasing constraints to allow multiple pointers to refer to the same object. This effectively encodes multiple concrete memory graphs into a single symbolic heap. We allocate the primary object for x *lazily*, the first time x is accessed. In this way, we lazily expand the symbolic heap and are able to efficiently encode heaps with unboundedly many objects.

Stores to x update x's primary object, l_x, and also conditionally update all other objects that x may-alias. For example, suppose pointers x and y may point to the same object. To write value v to pointer x, we first update l_x by writing v to address $\texttt{ptr}(l_x, x_{off})$, and we then update l_y by writing the expression $(x = \texttt{ptr}(l_y, x_{off}))\ ?\ v : e_{old}$ to address $\texttt{ptr}(l_y, e)$, where e_{old} is the previous value in l_y (this makes the update *conditional*) and x_{off} is a symbolic offset that will be described shortly. Loads of x access $\texttt{ptr}(l_x, x_{off})$ directly—since stores update all aliases, it is unnecessary for loads to access aliases as well.

Figure 6 shows the detailed semantics for accessing symbolic pointers of the form $\texttt{ptradd}(x, e_{off})$ (we treat x as $\texttt{ptradd}(x, 0)$).

Symbolic heap interface, including conditional *put*:

$$\frac{(l, \{\text{fields}\}) \in \mathcal{H} \qquad \mathit{read}(\text{fields}, e_{\mathit{off}}) = e}{\mathit{heapGet}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}})) = e}$$

$$\frac{(l, \{\text{fields}\}) \in \mathcal{H} \qquad \mathit{read}(\text{fields}, e_{\mathit{off}}) = e_{\mathit{old}} \qquad e_{\mathit{val}} = e_{\mathit{cond}}\ ?\ e : e_{\mathit{old}} \qquad \mathit{write}(\text{fields}, e_{\mathit{off}}, e_{\mathit{val}}) = \text{fields}' \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\text{fields}'\}]}{\mathit{heapPut}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}}), e_{\mathit{cond}}, e) = \mathcal{H}'}$$

$$\boxed{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathit{Stmt} \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}'; C'; \mathcal{A}'}$$

Load/store of a concrete location:

$$\frac{\mathit{eval}(\mathcal{Y}, p) = \texttt{ptr}(l, e_{\mathit{off}}) \qquad \mathit{heapGet}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}})) = e}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \texttt{load}\ p \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}; \mathcal{Y}[r \mapsto e]; C; \mathcal{A}}$$

$$\frac{\mathit{eval}(\mathcal{Y}, p) = \texttt{ptr}(l, e_{\mathit{off}}) \qquad \mathit{eval}(\mathcal{Y}, e) = e' \qquad \mathit{heapPut}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}}), \text{true}, e') = \mathcal{H}'}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \texttt{store}\ p,\ e \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}; C; \mathcal{A}}$$

Load/store of a symbolic location:

$$\frac{\mathit{eval}(\mathcal{Y}, p) = \texttt{ptradd}(x, e_{\mathit{off}}) \qquad \mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \texttt{ptr}(l_x, x_{\mathit{off}})) \qquad \mathit{heapGet}(\mathcal{H}', \texttt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}})) = e}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \texttt{load}\ p \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto e]; C'; \mathcal{A}'}$$

$$\frac{\begin{array}{c}\mathit{eval}(\mathcal{Y}, p) = \texttt{ptradd}(x, e_{\mathit{off}}) \qquad \mathit{eval}(\mathcal{Y}, e) = e' \\ \mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \texttt{ptr}(l_x, x_{\mathit{off}})) \\ \mathit{lookupAliases}(\mathcal{A}', x) = \{l_1 ... l_n\} \\ \mathit{heapPut}(\mathcal{H}', \quad \texttt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}}), \text{true}, \qquad e') = \mathcal{H}''_0 \\ \mathit{heapPut}(\mathcal{H}''_0, \quad \texttt{ptr}(l_1, x_{\mathit{off}} + e_{\mathit{off}}), (x = \texttt{ptr}(l_1, x_{\mathit{off}})), e') = \mathcal{H}''_1 \\ \cdots \\ \mathit{heapPut}(\mathcal{H}''_{n-1}, \texttt{ptr}(l_n, x_{\mathit{off}} + e_{\mathit{off}}), (x = \texttt{ptr}(l_n, x_{\mathit{off}})), e') = \mathcal{H}''_n\end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \texttt{store}\ p,\ e \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}''_n; \mathcal{Y}; C'; \mathcal{A}'}$$

Allocate and free:

$$\frac{l = \mathit{fresh\ loc} \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\lambda i.\texttt{undef}\}]}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \texttt{malloc}(e_{\mathit{size}}) \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto \texttt{ptr}(l, 0)]; C; \mathcal{A}} \qquad \frac{\mathit{true}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \texttt{free}(p) \overset{\mathit{mem}}{\Longrightarrow} \mathcal{H}; \mathcal{Y}; C; \mathcal{A}}$$

Figure 6. Representative rules from the symbolic heap semantics. In these rules, $\mathcal{Y}$ refers to the current stack frame (namely, the youngest stack frame in $\overline{\mathcal{Y}}$), and $\mathit{read}(A, e_{\mathit{off}})$ and $\mathit{write}(A, e_{\mathit{off}}, e_{\mathit{val}})$ are standard constructors from the theory of arrays (*e.g.*, see [11]).

Restricting Aliasing with a Points-To Analysis. Recall that symbolic constants like x represent values that originate in our initial program context. That is, if x is a valid pointer, then x *must* point to some object that was allocated *before* our initial program context. In the worst case, the set of possible aliases includes all primary objects that have been previously allocated for other symbolic pointers. This list of objects is recorded in $\mathcal{A}$ (Figure 5) and kept up-to-date by *addPrimary*.

In practice, we can narrow the set of aliases using a static points-to analysis. On the first access to x, we add the record $\{x, l_x, n_x\}$ to $\mathcal{A}$, where n_x is the representative node for x in the static points-to graph. The set of objects that x *may-alias* is found by enumerating all $\{y, l_y, n_y\} \in \mathcal{A}$ for which n_y and n_x may point-to the same object according to the static points-to graph—this search is performed by *lookupAliases*. Note that, in practice, the search for aliases can be implemented efficiently by exploiting the structure of the underlying points-to graph.

We use a *field-sensitive* points-to analysis so we can additionally constrain the offset being accessed. For each symbolic pointer x, we query the points-to analysis to compute a range of possible offsets for x, and then construct a fresh symbolic constant x_{off} that is constrained to that range. (This is the same x_{off} used above in the discussion of loads and stores.) For example, if x is known to point at a specific field, then x_{off} is fixed to that field. If a range of offsets cannot be soundly determined, x_{off} is left unconstrained.

Heap Invariants. On the first access of symbolic pointer x, *addPrimary* allocates a primary object at l_x, appends the record $\{x, l_x, n_x\}$ to $\mathcal{A}$, and emits a *heap invariant* that we describe now.

Suppose the first access of x is a load, and suppose that x may-alias some other symbolic pointer y. For soundness, we *must* ensure that every load of x satisfies the following invariant: $x = y \implies \texttt{load}(x) = \texttt{load}(y)$. Making matters more complicated is the fact that we may have performed stores on y before our first access of x—we must ensure that these stores are visible through x as well. Our approach is to define the initial fields of l_x as follows:

$$\begin{array}{l}\mathit{Initial}\ \texttt{fields}\ \mathit{of}\ l_x \\ \quad \equiv\ (x = \texttt{ptr}(l_y, x_{\mathit{off}}))\ ?\ \texttt{fields}_y\ :\ \mathit{fresh}\end{array} \tag{1}$$

where $\texttt{fields}_y$ is the current fields array of object l_y, which is the primary object for y, and where *fresh* is a symbolic array that maps each field $\mathit{fresh}(i)$ to a fresh symbolic constant—this represents the unknown initial values of l_x in the case that x and y do not alias. In general x may have more than one alias, in which case we initialize the `fields` of l_x similarly to the above, but we use a chain of conditionals that compares x with all possible aliases.

Memory Allocation. Semantics for $\texttt{malloc}(e_{\mathit{size}})$ are shown in Figure 6. Since each object has its own symbolic `fields` array, we naturally support allocations of unbounded symbolic size.

Memory Error Checkers. Since this paper elides memory error-checking details, we treat $\texttt{free}(p)$ as a no-op in Figure 6.

Briefly, to detect memory errors, we might add *size* and *isLive* attributes to each object in $\mathcal{H}$. On $\texttt{malloc}(e)$, we would set $\mathit{size} = e$ and $\mathit{isLive} = \text{true}$. On $\texttt{free}(p)$, we would *conditionally* free all objects that p may-alias by conditionally setting $\mathit{isLive} = \text{false}$ in all aliases, much in the same way that $\texttt{store}(p, e)$ conditionally writes e to all aliases of p. Error checkers such as out-of-bounds and use-after-free would then ensure that, for each access at $\texttt{ptr}(l, e_{\mathit{off}})$, $0 \leq e_{\mathit{off}} < \mathcal{H}(l).\mathit{size}$ and $\mathcal{H}(l).\mathit{isLive} = \text{true}$.

Compound Symbolic Pointer Expressions. Figure 6 shows rules for load and store statements where the pointer p evaluates to an expression of the form $\texttt{ptr}(l, e)$, x, or $\texttt{ptradd}(x, e)$, but the result of $\mathit{eval}(\mathcal{Y}, p)$ can also have the form $\mathit{read}(\mathit{fields}, e_{\mathit{off}})$. This form appears when a pointer is read from the heap, since all heap accesses use the theory of arrays.

The difficulty is that there may be multiple possible values at e_{off}. For example, if *fields* is $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, x')$, then we

cannot evaluate this address without first resolving the symbolic pointers x and x'. Further, the values written by *write* can contain conditional expressions due to the conditional store performed by *heapPut*. So, in general, the fields array might include a chain of calls as in the following: $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, e'' \ ? \ x' : x'')$.

Our approach is to walk the call chain of *write*s to build *guarded expressions* that summarize the possible values at offset e_{off}. If the value stored by a *write* is a conditional expression, we also walk that conditional expression tree while computing the guarded expressions. This gives each guarded expression the form $e_{\mathit{grd}} \to p$, where each p has the form x, $\texttt{ptradd}(x, e)$, or $\texttt{ptr}(l, e)$. In the above example, we build guarded expressions $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge e'') \to x'$, and $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge \neg e'') \to x''$, and $(e_{\mathit{off}} \neq e'_{\mathit{off}} \wedge e_{\mathit{off}} = 1) \to x$, and so on down the call chain.

We then execute the memory operation on this set of guarded expressions. For stores, we evaluate each guarded expression independently: given $e_{\mathit{grd}} \to p$, we evaluate p using the rules in Figure 6, but we include e_{grd} in the condition passed to *heapPut*. For loads, we use the rules in Figure 6 to map each pair $e_{\mathit{grd}} \to p$ to a pair $e_{\mathit{grd}} \to e$, where e is the value loaded from pointer p. We then collect each $e_{\mathit{grd}} \to e$ into a conditional expression tree that represents the final value of the load. Continuing the above example, if the values at x, x', and x'' are v, v', and v'', respectively, then a load of the above example address would return the following conditional expression tree: $(e_{\mathit{off}} = e'_{\mathit{off}}) \ ? \ (e'' \ ? \ v' : v'') : (e_{\mathit{off}} = 1 \ ? \ x : \_)$.

Function Pointers. At indirect calls, we first use a sound static points-to analysis to enumerate a set of functions F that might be called, then we use *isSat* to prune functions from F that cannot be called given the current path constraint, and finally we fork for each of the remaining possibilities.

3.2 Initializing the Heap with Reaching Definitions

The initial symbolic state (S_{init}) actually contains an *empty* heap that is expanded lazily, as described above. As the heap graph expands, newly uncovered objects are initially *unconstrained*, as represented by the *fresh* symbolic array allocated for each primary object (recall Equation (1), above). This approach can be imprecise for the same reasons discussed in §2.3. We improve precision using reaching definitions, as follows.

We extend the reaching definition analysis from §2.3 to also compute a set of heap writes that *must-reach* the initial program context. Specifically, we compute a set of pairs $R_{\mathit{heap}} = \{(p_i, e_i)\}$, where the heap location referenced by p_i *must* have a value matching e_i in the initial state. We use standard flow functions to compute R_{heap} and we use a static points-to analysis to reason about aliasing.

Then, we modify *addPrimary* to exploit R_{heap}. Specifically, when adding a primary object l_x for symbolic pointer x, we append the following invariant to the current path constraint, C:

$$\bigwedge_{(\texttt{ptradd}(x, e_{\mathit{off}}), e_{\mathit{val}}) \in R_{\mathit{heap}}} \mathit{read}(\mathit{fresh}, x_{\mathit{off}} + e_{\mathit{off}}) = e_{\mathit{val}}$$

In the above, we enumerate all pairs $(p, e_{\mathit{val}}) \in R_{\mathit{heap}}$ where either $p = \texttt{ptradd}(x, e_{\mathit{off}})$ or $p = x$ (which we treat like $\texttt{ptradd}(x, 0)$). For each such pair, we emit a constraint on *fresh*, which is the symbolic array used to initialize l_x as shown in Equation (1).

4. Adding Threads and Synchronization

Figure 7 shows the syntax for *SimpThreads*, which adds shared-memory multithreading and synchronization to *SimpHeaps*.

Threads. *SimpThreads* supports cooperative thread scheduling with `yield()`, which nondeterministically selects another thread to run. Cooperative scheduling with `yield` is sufficient to model any data race free program. As with other memory errors (recall §3), data races have undefined behavior in C [17] and are runtime errors

$$\begin{aligned} s \in \mathit{Stmt} ::= \ & \ldots \mid \texttt{threadCreate}(e_f, e_{\mathit{arg}}) \mid \texttt{yield}() \\ & \mid \texttt{wait}(p) \mid \texttt{notifyOne}(p) \mid \texttt{notifyAll}(p) \\ & \underline{\textit{synchronization annotations}} \\ & \mid \texttt{acquire}(p) \mid \texttt{release}(p) \\ & \mid \texttt{barrierInit}(p, e) \mid \texttt{barrierArrive}(p) \end{aligned}$$

Figure 7. New statements for *SimpThreads*.

```
pthread_mutex_lock(mutex *m) {
  while (load ptradd(m, i_taken))    // while (m->taken)
    wait(ptradd(m, i_taken));        //   wait(&m->taken)
  store ptradd(m, i_taken) 1;        // m->taken = 1
  acquire(m);                        // acquire(m)
}
pthread_mutex_unlock(mutex *m) {
  store ptradd(m, i_taken) 0;        // m->taken = 0
  release(m);                        // release(m)
  notifyOne(ptradd(m, i_taken));     // notify(&m->taken)
  yield();                           // yield()
}
```

Figure 8. Pseudocode demonstrating how pthreads' mutexes might be implemented in *SimpThreads*.

in *SimpThreads*. Hence, cooperative scheduling is a valid model as we can assume that all *SimpThreads* programs are either data race free or will halt before the first race.

New threads are created by $\texttt{threadCreate}(e_f, e_{\mathit{arg}})$. This spawns a new thread that executes the function call $e_f(e_{\mathit{arg}})$, and the new thread will run until e_f returns. As *SimpThreads* uses cooperative scheduling, the new thread is *not* scheduled until another thread yields control.

Synchronization. We build higher-level synchronization objects such as barriers, condition variables, and queued locks using two primitive parts: cooperative scheduling with `yield`, which provides simple atomicity guarantees, and FIFO wait queues, which provide simple notify/wait operations that are common across a variety of synchronization patterns. Specifically, wait queues support three operations: `wait`, to yield control and move the current thread onto a wait queue; `notifyOne`, to wake the thread on the head of a wait queue; and `notifyAll`, to wake all threads on a wait queue.

We use these building blocks to implement standard threading and synchronization libraries such as POSIX threads (pthreads). To aid our symbolic semantics, we assume synchronization libraries have been instrumented with the *annotation functions* listed in Figure 7. Annotation functions are no-ops that do not actually perform synchronization—they merely provide higher-level information that we will exploit, as described later (§4.3, §4.4). The example in Figure 8 demonstrates how to annotate an implementation of pthreads' mutexes. We have written the example in a pseudocode that uses memory operations resembling those in *SimpThreads*.

Note that wait queues are named by pointers. There is an implicit wait queue associated with every memory address—no initialization is necessary. For example, Figure 8 uses the implicit wait queue associated with `&m->taken`. The `futex()` system call in Linux uses a similar design. The reason for naming wait queues by an address rather than an integer id will become clear in §4.3.

Challenges. The primary challenge introduced by *SimpThreads* is the need to reason about synchronization objects. Our approach includes a semantics for symbolic wait queues (§4.2) and a collection of synchronization-specific invariants (§4.3) that exploit facts learned from a context-specific dataflow analysis (§4.4).

4.1 Symbolic Semantics

We now extend our symbolic execution algorithm for *SimpThreads*. As illustrated in Figure 9, we modify $\overline{\mathcal{Y}}$ and *CallCtx* to include one

$\overline{y}$	:	***ThreadId*** $\rightarrow$ Stack of (*Var* $\rightarrow$ *Expr*)	*(local variables)*
CallCtx	:	***ThreadId*** $\rightarrow$ Stack of *StmtLabel*	*(calling contexts)*
path	:	List of (***ThreadId***, *StmtLabel*)	*(execution trace)*
T^{Curr}	:	*ThreadId*	*(current thread)*
T^{E}	:	Set of *ThreadId*	*(enabled threads)*
WQ	:	List of (*Expr*, *ThreadId*)	*(global wait queue)*
$\mathtt{L}^{+}$	:	*ThreadId* $\rightarrow$ Set of *Expr*	*(acquired locksets)*
$\mathtt{B}^{\mathtt{cnts}}$	:	*Expr* $\rightarrow$ Set of *Expr*	*(barrier arrival cnts)*

Figure 9. Symbolic state for *SimpThreads*, with modifications to *SimpHeaps* **bolded**, above the line, and additions shown below.

call stack per thread, and we modify *path* to record a multithreaded trace. We add the following domains to the symbolic state:

- T^{Curr}, which is the id of the thread that is currently executing.
- T^{E}, which is the set of enabled threads, *i.e.*, the set of threads not blocked on synchronization. This includes T^{Curr}.
- *WQ*, which is a list that represents a global order of all waiting threads. Each entry of the list is a pair (p, t) signifying that thread t is blocked on the wait queue named by address p. The initial *WQ* can either be empty (all threads enabled) or non-empty (some threads blocked, as described in §4.2).
- $\mathtt{L}^{+}$, which describes a set of locks that *may* be held by each thread and is derived from `acquire` and `release` annotations.
- $\mathtt{B}^{\mathtt{cnts}}$, which describes a set of possible arrival counts for each barrier and is derived from `barrierInit` annotations.

$\mathtt{L}^{+}$ and $\mathtt{B}^{\mathtt{cnts}}$ are both *over-approximations*. They are initialized as described in §4.4 and they are used by invariants described in §4.3.

Symbolic Execution. Our first action during symbolic execution is to invoke $step(S_{init}, \mathtt{yield}())$, where `yield` forks execution once for each possible $T^{Curr} \in T^{E}$. This gives each thread a chance to run first. Note that context switches (updates to T^{Curr}) occur *only* either explicitly through `yield`, or implicitly when the current thread exits or is disabled through `wait`. Note also that execution has deadlocked when T^{E} is empty and *WQ* is non-empty.

4.2 Symbolic Wait Queues

We now give symbolic semantics for the three FIFO wait queue operations, `wait`, `notifyOne`, and `notifyAll`. When a thread t calls $\mathtt{wait}(p)$, we remove t from T^{E} and append the pair (p, t) to *WQ*. When t is notified, we remove it from *WQ* and add it to T^{E}. *Which* threads are notified is answered as follows:

notifyOne(p). Any thread in *WQ* with a matching queue address may be notified. Let (p_1, t_1) be the first pair in *WQ* and let (p_n, t_n) be the last pair. We walk this ordered list and fork execution up to $|WQ| + 1$ times. The possible execution forks are given by the following list of path constraints:

$$\begin{array}{rl} (1) & p_1 = p \\ (2) & p_1 \neq p \wedge p_2 = p \\ & \ldots \\ (n) & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n = p \\ (n{+}1) & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n \neq p \end{array}$$

In the first fork, we notify t_1, in the second, we notify t_2, and so on, until the nth fork, in which we notify t_n. In the final fork, no threads are notified. Only a subset of these forks may be feasible, so we use *isSat* to prune forked paths that have an infeasible path constraint. In particular, if there exists an i where $p_i{=}p$ *must* be true on the current path, then all forks from (i+1) onwards are infeasible and will be discarded. Further, as in §3, we increase precision by using a static points-to analysis to determine when it *cannot* be true that $p_i{=}p$. These semantics are simple but reveal a key design decision: by folding all concrete wait queues into a single global queue, *WQ*, we naturally allow each wait queue to be named by *symbolic* addresses.

notifyAll(p). Any subset of threads in *WQ* may be notified. We first compute the powerset of *WQ*, $\mathcal{P}(WQ)$, and then fork execution once for each set $S \in \mathcal{P}(WQ)$. Specifically, on the path that is forked for set S, we notify all threads in S and apply the following path constraint:

$$\bigwedge_{(p_i,t_i)\in WQ} \begin{cases} p_i = p & \mathtt{if}\ (p_i, t_i) \in S \\ p_i \neq p & \mathtt{otherwise} \end{cases}$$

This forks execution $2^{|WQ|}$ ways, though we expect that *isSat* and a points-to analysis will prune many of these in practice.

Initial Contexts with a Nonempty *WQ*. Suppose we want to analyze an initial program context in which some subset of threads begin in a *waiting* state, but we do not know the order in which the threads began waiting. One approach is to fork for each permutation of the wait order, but this is inefficient. Instead, our approach is to add *timestamp counters*. First, we tag each waiting thread with a timestamp derived from a global counter that is incremented on every call to `wait`, so that thread t_1 precedes thread t_2 in *WQ* if and only if t_1's timestamp is less than t_2's timestamp.

Then, we set up the program context so that each waiting thread begins with the call to `wait` it is waiting in. Before beginning normal symbolic execution, we execute these `wait` calls in any order, using the semantics for `wait` described above, but with one adjustment: we give each waiting thread t_i a symbolic timestamp, represented by the symbol x_i, and we bound each $x_i < 0$ so these waits occur before other calls to `wait` during normal execution. We say that $x_i < x_k$ is true in the concrete initial state when t_i and t_k are waiting on the same queue and t_i precedes t_k on that queue.

Next, we update the semantics of `notifyOne`. If there are n threads in *WQ* and w of those threads are *initial waiters*, meaning they have symbolic timestamps, then `notifyOne` uses the following sequence of path constraints, where $1 \leq i \leq w$:

$$\begin{array}{rl} (i) & p_i = p \wedge \left(\bigwedge_{1\leq k\leq w, k\neq i} (p_k = p) \Rightarrow (x_i < x_k) \right) \\ (w{+}1) & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_w \neq p \wedge p_{w+1} = p \\ & \ldots \\ (n) & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n = p \\ (n{+}1) & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n \neq p \end{array}$$

The first w constraints handle the cases where an initial waiter is notified. We can notify initial waiter t_i if it has a matching queue address, $p_i{=}p$, and it precedes all other initial waiters t_k with a matching address. The cases for w+1 and above are as before.

4.3 Synchronization Invariants

The semantics described above are sound, but the presence of unconstrained symbolic constants can cause our symbolic execution to explore infeasible paths. In an attempt to avoid infeasible paths, we augment the path constraint with higher-level program invariants. Specifically, this section proposes a particularly high-value set of *synchronization invariants*.

We cannot apply synchronization invariants without first identifying synchronization objects. Ideally we would locate such objects by scanning the heap, but our core language is untyped, so we cannot soundly determine the type of an object by looking at it. (This conservatively models our target language, C, where potentially unsafe type casts are prevalent.) Instead, we apply invariants when synchronization functions are called. For example, we instrument the implementation of `pthread_mutex_lock(m)` to apply invariants to `m` as the first step before locking the mutex. The rest of this section describes the invariants we have found most useful.

Locks. As illustrated in Figure 8, locks can be modeled by an object with a `taken` field that is non-zero when the lock is held and zero when the lock is released. Suppose a thread attempts to acquire a lock whose `taken` field is symbolic: execution must fork into two paths, one in which `taken`=0, so the lock can be acquired, and another in which `taken`≠0, so the thread must wait. One of these paths may be infeasible, as illustrated by Figure 1, so we need to further constrain lock objects to avoid such infeasible paths.

We use *locksets* to constrain the `taken` field of a lock object. Given a symbolic state with locksets $\mathtt{L}^+$ and a pointer p to some lock object, the lock's `taken` field can be non-zero only when there exists a thread T and an expression e, where $e \in \mathtt{L}^+(T)$, such that $e = p$. This invariant is expressed by the following constraint, where e_i ranges over all locks held by all threads:

$$(\mathtt{taken} = 0) \Leftrightarrow \left(\bigwedge_{e_i \in \mathtt{L}^+(*)} e_i \neq p \right)$$

Our dataflow analysis computes $\mathtt{L}^+$ for the initial symbolic state (§4.4). We keep $\mathtt{L}^+$ up-to-date during symbolic execution using the `acquire` and `release` annotations: on `acquire`(p) we add p to $\mathtt{L}^+(T^{Curr})$, and on `release`(p) we remove e from $\mathtt{L}^+(T^{Curr})$ where e must-equal p on the current path.

Barriers. A pthreads barrier can be modeled by two fields, `expected` and `arrived`, and a wait queue, where `arrived` is the number of threads that have arrived at the barrier, the barrier triggers when `arrived=expected`, and the wait queue is used to release threads when the barrier triggers.

Suppose a program has N threads spin in a loop, where each loop iteration includes a barrier with `expected`=N. Now suppose we analyze the program from an initial context where the barrier is unconstrained. When the first thread arrives at the barrier, execution forks at the condition `arrived=expected`. In the true branch we set `arrived=0` and notify the queue, and in the false branch we increment `arrived` and wait. This repeats for the other threads, and an execution tree unfolds in which we explore $O(2^N)$ paths through a code fragment that has exactly one feasible path.

We compute invariants for both of these fields. Bounds for `arrived` can be determined by examining *WQ*: the number of threads that have arrived at a barrier is exactly the number of threads that are waiting on the barrier's wait queue. Let q be the wait queue address used by the barrier and let *C* be the current path constraint. We compute conservative lower- and upper-bounds for `arrived`. The lower-bound L is the number of pairs $(p, t) \in$ *WQ* for which *mustBeTrue*$(C, p{=}q)$, and the upper-bound H is the number of pairs for which *mayBeTrue*$(C, p{=}q)$. Given these bounds, the invariant is $L \leq \mathtt{arrived} \leq H$.

A barrier's `expected` count is specified during barrier initialization, *i.e.*, when `pthread_barrier_init` is called. Each symbolic state contains a $\mathtt{B}^{\mathtt{cnts}}$ that maps barrier pointers p to a set of expressions that describes the set of possible `expected` counts for all barriers pointed-to by p. So, we can use $\mathtt{B}^{\mathtt{cnts}}$ directly to construct an invariant for `expected`:

$$\bigwedge_{p' \in \mathtt{B}^{\mathtt{cnts}}} \left(\bigvee_{e \in \mathtt{B}^{\mathtt{cnts}}(p')} (p' = p) \Rightarrow (\mathtt{expected} = e) \right)$$

$\mathtt{B}^{\mathtt{cnts}}$ is computed for the initial state (§4.4) and does not change during symbolic execution—when `pthread_barrier_init` is called during symbolic execution we write to the barrier's `expected` field directly, making $\mathtt{B}^{\mathtt{cnts}}$ irrelevant in this case.

Other Types of Synchronization. The invariant described above for a barrier's `arrived` field is more generally stated as an invariant on the size of a given wait queue, making it applicable to other data structures that use wait queues, such as condition variables and queued locks.

Why Wait Queues are Named by Address. For standard synchronization objects such as barriers, condition variables, and queued locks, different objects do not share the same wait queue. For example, notifying the queue of lock L should not notify threads waiting at any other lock. By using the address of L to name L's wait queue, we state this invariant implicitly.

For contrast, suppose we instead named wait queues by an integer id. We would be forced to add a `queueId` field to each lock, then state the following invariant: $\forall p_1, p_2 : (p_1 = p_2) \Leftrightarrow (id_1 = id_2)$, where p_1 and p_2 range over the set of pointers to locks, and where id_1 and id_2 are the `queueId` fields in p_1 and p_2, respectively. Stating this as an axiom would require enumerating the complete set of pointers to locks, which can be extremely inefficient.

4.4 Approximating the Initial State of Synchronization

We update the context-specific dataflow framework introduced in §2.3 to support multiple threads. Specifically, we apply the dataflow framework as described in §2.3 to each thread, separately, and then combine the per-thread results to produce a multithreaded analysis. We perform the following analyses for *SimpThreads*:

Reaching Definitions. We update the reaching definitions analysis described in §3.2 to support multiple threads. Importantly, since we analyze each thread in isolation, we must reason about cross-thread interference. Our approach is to label memory locations in R_{heap} as either *conflict-free* or *shared*. A location is conflict-free if it is provably thread-local (via an escape analysis) or if all writes to the location must-occur before the first call to `threadCreate`—the second case captures a common idiom where the main thread initializes global data that is kept read-only during parallel execution. Shared locations may have conflicts—we reason about these conflicts using *interference-free regions* [8].

Locksets. We use a lockset analysis to compute $\mathtt{L}^+(T)$, the set of locks that *may* be held at thread T's initial program counter. Our analysis uses relative locksets as in RELAY [26]: each function summary includes two sets, $\mathtt{L}_f^+$ and $\mathtt{L}_f^-$, where $\mathtt{L}_f^+$ is the set of locks that function f *may acquire* without releasing, and $\mathtt{L}_f^-$ is the set of locks that f *always releases* without first acquiring.

The key difference between our implementation and RELAY's is that we compute may-be-held sets while RELAY computes must-be-held sets. This reflects differing motivations: as a static race detector, RELAY wants to know which locks *must* be held to determine if accesses are properly guarded, but we want to know which locks *may* be held to determine when two `lock()` calls may need to be serialized (as motivated by Figure 1). Hence, our $\mathtt{L}^+$ and $\mathtt{L}^-$ are may-acquire and must-release, while those used by RELAY are must-acquire and may-release.

Barrier Expected Arrivals. To compute $\mathtt{B}^{\mathtt{cnts}}$, we simply enumerate all calls to `barrierInit`(p, e) that might be performed on some path from program entry up to the initial context, and for each such call, we add e to the set $\mathtt{B}^{\mathtt{cnts}}(p)$. This can be viewed as *may-reach* analysis applied to each barrier's `expected` field.

Barrier Matching. A large class of data-parallel algorithms use barriers to execute threads in lock-step. For example, a program might execute the following loop in N different threads, where each iteration happens in lock-step:

```
for (i=0; i < Z; ++i) { barrierArrive(b); ... }
```

Suppose we are given an initial program context in which each thread begins inside this loop. In this case, since the loop runs in lock-step, we know that all threads must start from the same dynamic loop iteration, so we can add a constraint that equates the loop induction variable, `i`, across all threads. This constraint is included in the initial path constraint, $S_{init}.C$.

This is the *barrier matching* problem: given two threads, must they pass the same sequence of barriers from program entry up to

the initial context? Solutions have been proposed—we adapt [28], which builds *barrier expressions* to describe the possible sequence of barriers each thread might pass through. Two threads are barrier-synchronized if their barrier expressions are compatible.

The algorithm in [28] does not support our use case directly because it cannot reason about loops with input-dependent trip counts. So, we extend that algorithm by computing a symbolic trip count for each loop node in a barrier expression. Two loops match if their symbolic trips counts *must* be equal. We compute trip counts using a standard algorithm, but we discard trip counts that depend on *shared* memory locations (recall the definition of *shared*, from above). To determine if the trip count can be kept, we compute a backwards slice of the trip count expression and ensure that slice does not depend on any *shared* locations.

5. Soundness and Completeness

Our symbolic execution algorithm is sound, and it is complete except when the SMT solver uses concretization to make progress through an unsolvable query (recall §2.1). Our theorem relies on a notion of correspondence between concrete and symbolic states—because the heap is expanded lazily in the symbolic semantics, this notion relies on *partial equivalence* and is somewhat technical. We give the full concrete semantics and a proof of the theorem in a supplementary appendix.

Definition 1 (Correspondence of concrete and symbolic states)**.** *We say that symbolic state S_S* models *concrete state S_K* under *constraint C if there exists an assignment Σ that assigns all symbolic constants in S_S to values such that (a) Σ is a valid assignment under the constraint C, and (b) the application of Σ to S_S produces a state that is* partially-equivalent *to S_K (as defined in the appendix).*

Theorem 1 (Soundness and completeness of symbolic execution)**.** *Consider an initial program context, an initial concrete state S_K for that context, and an initial symbolic state S_S:*

- ***Soundness:*** *If symbolic execution from S_S outputs a pair (p, C), then for all S_K such that S_S models S_K under C, concrete execution from S_K must follow path p as long as context switches happen exactly as specified by path p.*
- ***Completeness:*** *If concrete execution from S_K follows path p, then for all S_S such that S_S models S_K under S_S.C, symbolic execution from S_S will either (a) output a pair (p, C), for some C, or (b) encounter a query that the SMT solver cannot solve.*

6. Implementation and Evaluation

We implemented the above algorithms on top of Cloud9 [4], which symbolically executes C programs that use pthreads and are compiled to LLVM bytecode (Cloud9 operates directly on LLVM bytecode). Where a points-to analysis is needed, we use DSA [20].

The C language allows casts between pointers and integers. This is not modeled in our semantics but is partially supported by our implementation. Our approach is to represent each pointer expression p like any other integer expression. Then, at each memory access, we analyze p to extract (*base*, *offset*) components. For example, our implementation represents `int *p = &a[x*3]` as $p = a + 4 \cdot (x \cdot 3)$, and to access p we transform it to $\texttt{ptr}(a, 12 \cdot x)$. We determine that a is the *base* address by exploiting LLVM's simple type system to learn which terms are used as pointers.

The precise semantics of integer-to-pointer conversions in C are implementation-defined (§6.3.2.3 of [17]). Our implementation does not support programs that use integer arithmetic to jump between two separately-allocated objects, such as via the classic "XOR" trick for doubly-linked lists. Such programs are not amenable to garbage collection for analogous reasons [2], even though they are supported by some C implementations.

Evaluation: Infeasible Paths. Recall (§1.1) that our approach lies on a spectrum between a *naïve* approach, which approximates the initial state very conservatively by leaving all memory locations unconstrained, and a *fully precise* approach, which constructs a perfectly precise initial state using an intractably expensive analysis.

We first compare the *naïve* approach with our approach: *how many fewer infeasible paths do we explore?* We answer this question for a given program context C by exhaustively enumerating all paths reachable from C up to a bounded depth. Any path that is enumerated by the *naïve* approach, but not by our approach, *must* be an infeasible path that our approach has avoided. We use a bounded depth to make exhaustive exploration feasible.

Table 1 summarizes our results. Each row summarizes experiments for a unique program context. We selected applications from standard benchmark suites to cover a range of parallelism styles, including fork-join parallelism, barrier-synchronized parallelism, task parallelism, and pipeline parallelism. For each application, we manually selected one or two program contexts in which at least two threads begin execution from the middle of a core loop. Column 2 shows the number of threads used in each initial context, and Column 3 shows the maximum number of conditional branches executed on each path during bounded-depth exploration.

Columns 4 and 7 show the number of paths explored by our fully optimized approach (*Full*) and the *naïve* approach, respectively. To further characterize our approach, we also ran our approach with optimizations disabled: *-RD* disables reaching definitions (§2.3, §3.2, §4.4) and *-SI* disables synchronization invariants (§4.3, §4.4). Our approach explores significantly fewer infeasible paths compared to the *naïve* approach, and a comparison across Columns 4–7 shows that each optimization is essential.

It is difficult to compare our approach with the *fully precise* approach, as the *fully precise* approach is intractable. For `lu` and `streamcluster`, we have manually inspected the paths explored by our approach (Column 4) and estimated, through our best understanding of the code, how many of those paths are infeasible (Column 18). Sources of infeasible paths include the following: Both programs assign each thread a unique *id* parameter (*e.g.*, by incrementing a global counter), but we are unable prove that these *id*s are unique across threads. Further, they performs calls of the form `pthread_join(t[i])`—we are unable to prove that each `t[i]` is a valid thread id, so we must fork for (infeasible) error cases.

Evaluation: Performance. Columns 8–12 show the average number of LLVM instructions executed per second (IPS), and Columns 13–17 show the percentage of total execution time devoted to *isSat*. The two metrics are correlated, as slower *isSat* times lead to lower IPS. *Full* uses more precise constraints than the *naïve* approach, but this does not necessarily lead to higher IPS for *Full*. Namely, precise and simple constraints such as $x = 5$ lead to high IPS, but precise and complex constraints can lead to low IPS—the latter effect has been observed previously [16, 19].

To further understand the overheads of our approach, we symbolically executed multiple *whole program* paths that each begin at program entry and pass through the initial context (*WP* in Columns 8 and 13). Although *Full* can be an order-of-magnitude slower than *WP*, many paths explored by *WP* visit 100s of branches before reaching the initial context, suggesting that exhaustive summarization of all paths from program entry is infeasible—approximating the initial context is *necessary*. Further profiling shows that much of our overhead comes from resolving symbolic pointers: LLVM's `load` and `store` instructions typically comprised 15% to 50% of total execution time in *Full*, but $< 5\%$ in *WP*.

Lastly, we tried disabling our use of a points-to analysis to restrict aliasing (§3.1, §4.2). With this optimization disabled, each symbolic pointer was assigned 100s of aliases, leading to large heap-update expressions and poor solver performance—so slow

Program Context			Num Paths				Avg IPS					Exec Time in *isSat*					inf.
	thr	br	*Full*	*-RD*	*-SI*	*N*	*WP*	*Full*	*-RD*	*-SI*	*N*	*WP*	*Full*	*-RD*	*-SI*	*N*	pths
blackscholes	4	20	763	1087	765	1087	927	176	1171	178	1206	75%	93%	65%	93%	65%	–
dedup-1	5	10	103	122	863	971	4731	72	49	67	64	3%	30%	62%	36%	51%	–
dedup-2	5	12	458	550	1811	1904	4692	45	26	39	32	5%	35%	64%	30%	59%	–
lu-1	4	22	681	1026	1133	1864	3997	93	170	64	107	2%	55%	16%	75%	57%	625
lu-2	4	18	554	1400	1290	4680	3860	80	136	105	162	32%	57%	23%	56%	26%	380
pfscan	3	18	246	246	3785	3785	6250	5368	5650	5254	5503	17%	28%	25%	15%	13%	–
streamcluster	3	11	60	617	229	1004	5382	161	59	7	19	15%	9%	35%	74%	31%	48

Table 1. Results for manually-selected program contexts. *Full* is our fully optimized approach, and *N* is the *naïve* approach.

that on most benchmarks, throughput decreased to well under 5 IPS. Hence, we consider this optimization so vital that we left it enabled in all experiments.

Evaluation: Input-Covering Schedules. We evaluate how well our techniques support an algorithm for finding *input-covering schedules* [1]. The algorithm partitions execution into *epochs* of bounded length and uses symbolic execution to enumerate a set of input-covering schedules for each epoch. The beginning of each epoch is defined by a multithreaded program context with fully specified call stacks, so our techniques are directly applicable.

We ran the algorithm (from [1]) on the programs shown in the table below, using our approach at various optimization levels, along with the *naïve* approach. We report the number of schedules enumerated by that algorithm and the algorithm's total runtime. Similarly to the infeasible paths evaluation above, if a schedule is enumerated with the *naïve* approach, but not our approach, then it must be an *infeasible* schedule. The results show, again, that our techniques are essential: the *naïve* approach suffers from slower algorithm runtimes and more infeasible schedules.

Program		NumSchedules / RunningTime			
	thr	*Full*	*-RD*	*-SI*	*N*
fft	2	2/ 9s	2/12s	2/ 10s	2/ 11s
lu	4	3/ 6s	23/14s	1550/396s	1976/202s
pfscan	2	455/24s	455/28s	2245/ 78s	2273/ 80s

7. Related Work

We have cited related work throughout the paper. We stress that prior approaches to path explosion, such as summarization [12, 15, 23], heuristics [5, 21], path merging [16, 19], and partial order reductions [10], are completely orthogonal to our symbolic execution semantics and could be profitably incorporated along with our ideas. We believe our integration of dataflow analysis with symbolic execution is novel, and §6 shows that our choice of analysis represents an essential sweet spot in our context. However, we do not claim that our dataflow analysis is *powerful* in a novel way.

Some prior work has investigated symbolic pointers. Our approach is most similar to *bbr*'s [7], but *bbr* cannot reason about interior pointers or memory allocations of a symbolic size. SAGE [9] handles interior pointers, as in our semantics, but does not support symbolic object sizes or symbolic pointer inputs. CUTE [24] supports symbolic pointer inputs in C unit tests, but can reason about aliasing relationships only when they are explicitly stated in program branches, such as via `if(p1!=p2)`. Pex [25] exploits static types (not available to us) and cannot reason soundly when type casts are involved (our approach is sound on an untyped language). We originally implemented the algorithm from Khurshid *et al.* [18], which explores each possible memory graph via aggressive forking, but this suffered from unacceptably extreme path explosion.

Program verifiers often represent memory with a global symbolic array [3, 6]. This can result in large chains of writes to the global array, which can burden symbolic executors, like Cloud9, which frequently invoke *isSat* to resolve local properties (*e.g.*, to resolve a branch). Our semantics assigns each heap object its own symbolic array to reduce such solver pressure and to simplify caching optimizations such as those implemented by Klee [5].

References

[1] T. Bergan, L. Ceze, and D. Grossman. Input-Covering Schedules for Multithreaded Programs. In *OOPSLA*, 2013.

[2] H.-J. Boehm. Simple Garbage-Collector-Safety. In *PLDI*, 1996.

[3] S. Böhme and M. Moskal. Heaps and Data Structures: A Challenge for Automated Provers. In *Conf. on Automated Deduction*, 2011.

[4] S. Bucur, V. Ureche, G. Candea, et al. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[5] C. Cadar et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[6] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A Reachability Predicate for Analyzing Low-Level Software. In *TACAS*, 2007.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Partial Replay of Long-Running Applications. In *FSE*, 2011.

[8] L. Effinger-Dean, H.-J. Boehm, et al. Extended Sequential Reasoning for Data-Race-Free Programs. In *MSPC*, 2011.

[9] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *ISSTA*, 2009.

[10] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[11] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.

[12] P. Godefroid. Compositional Dynamic Test Generation. In *POPL*, '07.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[14] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[15] P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *ISSTA*, 2011.

[16] T. Hansen, P. Schachte, and H. Sondergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *RV*, 2009.

[17] ISO. *C Language Standard, ISO/IEC 9899:2011*. 2011.

[18] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, 2003.

[19] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[20] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., UIUC, Urbana, IL, May 2005.

[21] Y. Li, Z. Su, L. Wang, and X. Li. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA*, 2013.

[22] C. S. Pasareanu, N. Rungta, and W. Visser. Symbolic Execution with Mixed Concrete-Symbolic Solving. In *ISSTA*, 2011.

[23] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing Procedures in Concurrent Programs. In *POPL*, 2004.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *FSE*, 2005.

[25] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Tests and Proofs (TAP)*, 2008.

[26] J. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *FSE*, 2007.

[27] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated Debugging for Arbitrarily Long Executions. In *HotOS*, 2013.

[28] Y. Zhang and E. Duesterwald. Barrier Matching for Programs With Textually Unaligned Barriers. In *PPoPP*, 2007.

Appendix A. Concrete Semantics of *SimpThreads*

Full syntax for *SimpThreads*:

$$
\begin{array}{rll}
r \in \mathit{Var} & & \textit{(local variables)} \\
x, y \in \mathit{SymbolicConst} & & \textit{(symbolic constants)} \\
f \in \mathit{FName} & & \textit{(function names)} \\
l \in \mathit{Loc} & & \textit{(heap locations)} \\
i \in \mathbb{Z} & & \textit{(integers)}
\end{array}
$$

$$
\begin{array}{rcl}
v \in \mathit{Value} & ::= & f \mid i \mid \texttt{null} \mid \texttt{ptr}(l, i) \\
e \in \mathit{Expr} & ::= & v \mid r \mid x \mid e \wedge e \mid e \vee e \mid e < e \mid \ldots \\
& & \mid \texttt{ptradd}(p, e)
\end{array}
$$

$$
\begin{array}{rcl}
\gamma \in \mathit{StmtLabel} & & \\
s \in \mathit{Stmt} & ::= & \texttt{return}\ e \\
& & \mid \texttt{br}\ e,\ \gamma_t,\ \gamma_f \\
& & \mid r \leftarrow e(e^*) \\
& & \mid r \leftarrow \texttt{load}\ p \mid \texttt{store}\ p,\ e \\
& & \mid r \leftarrow \texttt{malloc}(e) \mid \texttt{free}(p) \\
& & \mid \texttt{threadCreate}(e_f, e_{arg}) \mid \texttt{yield}() \\
& & \mid \texttt{wait}(p) \mid \texttt{notifyOne}(p) \mid \texttt{notifyAll}(p) \\
& & \underline{\textit{synchronization annotations}} \\
& & \mid \texttt{acquire}(p) \mid \texttt{release}(p) \\
& & \mid \texttt{barrierInit}(p, e) \mid \texttt{barrierArrive}(p) \\
\mathit{Func} & ::= & \texttt{func}\ f(r^*)\{\ (\gamma : s;)^*\ \}
\end{array}
$$

Program state in the concrete semantics:

$$
\begin{array}{rcll}
\mathcal{H} & : & \mathit{Loc} \to \{\text{fields} : (\mathbb{Z} \to \mathit{Value})\} & \textit{(heap)} \\
\overline{\mathcal{Y}} & : & \mathit{ThreadId} \to \text{Stack of } (\mathit{Var} \to \mathit{Value}) & \textit{(local variables)} \\
\mathit{CallCtx} & : & \mathit{ThreadId} \to \text{Stack of } \mathit{StmtLabel} & \textit{(calling contexts)} \\
T^{Curr} & : & \mathit{ThreadId} & \textit{(current thread)} \\
T^{E} & : & \text{Set of } \mathit{ThreadId} & \textit{(enabled threads)} \\
\mathit{WQ} & : & \text{List of } (\mathit{Value}, \mathit{ThreadId}) & \textit{(global wait queue)} \\
\mathtt{L}^{+} & : & \mathit{ThreadId} \to \text{Set of } \mathit{Value} & \textit{(acquired locksets)}
\end{array}
$$

Auxiliary Functions:

- $\mathit{eval} : ((\mathit{Var} \to \mathit{Value}) \times \mathit{Expr}) \to \mathit{Value}$
  This is just as in the symbolic semantics (§2.1), except that all expressions can be reduced to values as symbolic constants do not appear in the concrete language.
- $\mathit{wqGetOne} : (\mathit{WaitQueue}, \mathit{Value}) \to (\mathit{ThreadId}, \mathit{WaitQueue})$
  Given $\mathit{wqGetOne}(\mathit{WQ}, v)$, we walk the global queue WQ and return the first thread (t) waiting on the queue at address v. If found, we return a pair (t, WQ'), where WQ' is WQ with t dequeued. If the queue at address v is empty, we return ϵ.
- $\mathit{wqGetAll} : (\mathit{WaitQueue}, \mathit{Value}) \to (\text{Set of } \mathit{ThreadId}, \mathit{WaitQueue})$
  As above, except we dequeue all threads waiting on the queue at address v. If that queue is empty, we return (ϵ, WQ).
- $\mathit{wqAppend} : (\mathit{WaitQueue}, \mathit{Value}, \mathit{ThreadId}) \to \mathit{WaitQueue}$
  Given $\mathit{wqGetOne}(\mathit{WQ}, v, t)$, we append the pair (v, t) to WQ.

Statement evaluation rules. The $\Longrightarrow$ relation, defined below, is equivalent to step in the symbolic semantics except that $\Longrightarrow$ never forks the concrete state. We do not include semantics for the annotations `barrierInit` and `barrierArrive`, as these are no-ops during symbolic execution. We include a few shorthands for brevity: We elide a domain from a rule if the domain is not used or updated by the rule. We use $\mathcal{Y}$ (without the overline) to refer to the current stack frame (namely, the youngest stack frame of $\overline{\mathcal{Y}}(T^{Curr})$). We write $M[k \mapsto v]$ to assign $k = v$ in map M, and we write M/k to remove $M(k)$ from map M. We also elide rules that check for memory errors such as out-of-bounds accesses and data races, as the details of these checkers are orthogonal to this paper.

$$\boxed{\mathcal{H}; \overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathit{WQ}; \mathtt{L}^{+}; \mathtt{B}^{\mathtt{cnts}} | \mathit{Stmt} \Longrightarrow \mathcal{H}'; \overline{\mathcal{Y}}'; \mathit{CallCtx}'; T^{Curr}_{new}; T^{E}_{new}; \mathit{WQ}'; \mathtt{L}^{+}_{new}; \mathtt{B}^{\mathtt{cnts}}_{new}}$$

$$\frac{\begin{array}{c}\mathit{size}(\mathit{CallCtx}(T^{Curr})) > 1 \qquad \overline{\mathcal{Y}}' = \overline{\mathcal{Y}}[T^{Curr} \mapsto \mathit{pop}(\overline{\mathcal{Y}}(T^{Curr}))] \qquad \mathit{CallCtx}' = \mathit{CallCtx}[T^{Curr} \mapsto \mathit{pop}(\mathit{CallCtx}(T^{Curr}))] \\ \gamma' = \mathit{top}(\mathit{CallCtx}'(T^{Curr})) \qquad \mathit{Stmts}(\gamma') = r \leftarrow e_f(e_a^*) \qquad \mathit{eval}(\mathcal{Y}, e) = v \qquad \mathcal{Y}' = \mathit{top}(\overline{\mathcal{Y}}'(T^{Curr}))[r \mapsto v]\end{array}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E} | \texttt{return}\ e \Longrightarrow \overline{\mathcal{Y}}'[T^{Curr} \mapsto \mathit{replaceTop}(\overline{\mathcal{Y}}'(T^{Curr}), \mathcal{Y}')]; \mathit{CallCtx}'; T^{Curr}; T^{E}}$$

$$\frac{\mathit{size}(\mathit{CallCtx}(T^{Curr})) = 1 \qquad T^{Curr}_{new} \in T^{E} \qquad T^{Curr}_{new} \neq T^{Curr}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathtt{L}^{+} | \texttt{return}\ e \Longrightarrow \overline{\mathcal{Y}}[T^{Curr} \to \epsilon]; \mathit{CallCtx}[T^{Curr} \to \epsilon]; T^{Curr}_{new}; T^{E}/T^{Curr}; \mathtt{L}^{+}[T^{Curr} \to \epsilon]}$$

$$\frac{\mathit{size}(\mathit{CallCtx}(T^{Curr})) = 1 \qquad T^{E} = \{T^{Curr}\}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathtt{L}^{+} | \texttt{return}\ e \Longrightarrow \epsilon; \epsilon; \epsilon; \epsilon; \epsilon} \qquad \frac{\mathit{eval}(\mathcal{Y}, e) = i \qquad \gamma = ((i \neq 0)\ ?\ \gamma_t\ :\ \gamma_f)}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr} | \texttt{br}\ e,\ \gamma_t,\ \gamma_f \Longrightarrow \overline{\mathcal{Y}}; \mathit{goto}(\mathit{CallCtx}, T^{Curr}, \gamma); T^{Curr}}$$

$$\frac{\begin{array}{c}\mathit{eval}(\mathcal{Y}, e_f) = f \qquad \mathit{eval}(\mathcal{Y}, e_i) = v_i \qquad \mathit{Funcs}(f) = \texttt{func}\ f(r_i*)\{\gamma_0 : \ldots\} \\ \overline{\mathcal{Y}}' = \overline{\mathcal{Y}}[T^{Curr} \mapsto \mathit{push}(\overline{\mathcal{Y}}(T^{Curr}), \{r_i \to v_i\})] \qquad \mathit{CallCtx}' = \mathit{CallCtx}[T^{Curr} \mapsto \mathit{push}(\mathit{CallCtx}(T^{Curr}), \gamma_0)]; T^{Curr}\end{array}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr} | r \leftarrow e_f(e_i^*) \Longrightarrow \overline{\mathcal{Y}}'; \mathit{CallCtx}'; T^{Curr}}$$

$$\frac{\mathit{eval}(\mathcal{Y}, p) = \texttt{ptr}(l, i) \qquad (l, \{\text{fields}\}) \in \mathcal{H}}{\mathcal{H}; \mathcal{Y} | r \leftarrow \texttt{load}\ p \Longrightarrow \mathcal{H}; \mathcal{Y}[r \mapsto \text{fields}(i)]} \qquad \frac{\mathit{eval}(\mathcal{Y}, p) = \texttt{ptr}(l, i) \qquad \mathit{eval}(\mathcal{Y}, e) = e' \qquad (l, \{\text{fields}\}) \in \mathcal{H}}{\mathcal{H}; \mathcal{Y} | \texttt{store}\ p,\ e \Longrightarrow \mathcal{H}[l \mapsto \{\text{fields}[i \mapsto e']\}]; \mathcal{Y}}$$

$$\frac{l = \mathit{fresh\ loc}}{\mathcal{H}; \mathcal{Y} | r \leftarrow \texttt{malloc}(e) \Longrightarrow \mathcal{H}[l \mapsto \{\lambda i.\texttt{undef}\}]; \mathcal{Y}[r \mapsto \texttt{ptr}(l, 0)]} \qquad \frac{\mathit{true}}{\mathcal{H}; \mathcal{Y} | \texttt{free}(p) \Longrightarrow \mathcal{H}; \mathcal{Y}}$$

$$\frac{\mathit{eval}(\mathcal{Y}, e_f) = f \qquad \mathit{eval}(\mathcal{Y}, e_{arg}) = v \qquad \mathit{Funcs}(f) = \texttt{func}\ f(r)\{\gamma_0 : \ldots\} \qquad T_{new} = \mathit{fresh\ id}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{E} | \texttt{threadCreate}(e_f, e_{arg}) \Longrightarrow \overline{\mathcal{Y}}[T_{new} \mapsto \{\{r \to v\}\}]; \mathit{CallCtx}[T_{new} \mapsto \gamma_0]; T^{E} \cup \{T_{new}\}}$$

$$\frac{T_{new}^{Curr} \in T^E}{T^{Curr}; T^E | \texttt{yield}() \Longrightarrow T_{new}^{Curr}; T^E} \qquad \frac{eval(\mathcal{Y}, p) = v \qquad wqAppend(WQ, v, T^{Curr}) = WQ' \qquad T^E = \{T^{Curr}\}}{\overline{\mathcal{Y}}; T^{Curr}; T^E; WQ | \texttt{wait}(p) \Longrightarrow \overline{\mathcal{Y}}; \epsilon; \{\}; WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqAppend(WQ, v, T^{Curr}) = WQ' \qquad T_{new}^{Curr} \in T^E \qquad T_{new}^{Curr} \neq T^{Curr}}{\overline{\mathcal{Y}}; T^{Curr}; T^E; WQ | \texttt{wait}(p) \Longrightarrow \overline{\mathcal{Y}}; T_{new}^{Curr}; T^E / \{T^{Curr}\}; WQ'} \qquad \frac{eval(\mathcal{Y}, p) = v \qquad wqGetOne(WQ, v) = \epsilon}{\overline{\mathcal{Y}}; T^E; WQ | \texttt{notifyOne}(p) \Longrightarrow \overline{\mathcal{Y}}; T^E; WQ}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqGetOne(WQ, v) = (hd, WQ')}{\overline{\mathcal{Y}}; T^E; WQ | \texttt{notifyOne}(p) \Longrightarrow \overline{\mathcal{Y}}; T^E \cup \{hd\}; WQ'} \qquad \frac{eval(\mathcal{Y}, p) = v \qquad wqGetAll(WQ, v) = \{T_{woke}, WQ'\}}{\overline{\mathcal{Y}}; T^E; WQ | \texttt{notifyAll}(p) \Longrightarrow \overline{\mathcal{Y}}; T^E \cup \{T_{woke}\}; WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v}{\overline{\mathcal{Y}}; T^{Curr} | \texttt{acquire}(p) \Longrightarrow \overline{\mathcal{Y}}; T^{Curr}; \mathtt{L}^+[T^{Curr} \mapsto \mathtt{L}^+(T^{Curr}) \cup \{v\}]} \qquad \frac{eval(\mathcal{Y}, p) = v}{\overline{\mathcal{Y}}; T^{Curr}; \mathtt{L}^+ | \texttt{release}(p) \Longrightarrow \overline{\mathcal{Y}}; T^{Curr}; \mathtt{L}^+[T^{Curr} \mapsto \mathtt{L}^+(T^{Curr}) / \{v\}]}$$

Appendix B. Soundness and Completeness

We restate and expand on Definition 1 and then restate and give a proof for Theorem 1. These were first stated in §5.

Definition 1 (Correspondence of concrete and symbolic states)**.** *We say that symbolic state S_S* models *concrete state S_K* under *constraint C if there exists an assignment Σ that assigns all symbolic constants in S_S to values such that (a) Σ is a valid assignment under the constraint C, and (b) the application of Σ to S_S produces a state S'_S that is* partially-equivalent *to S_K (as defined below).*

The above definition relies on a notion of *partial* equivalence between S'_S and S_K, rather than true equivalence, because we expand the symbolic memory graph lazily (recall §3.1). Thus, the symbolic heap may contain a subset of the objects contained in the concrete heap. Our notion of partial equivalence considers only this overlapping subset of S'_S and S_K. Hence, to determine whether S'_S and S_K are partially equivalent, we must construct a mapping, λ, that maps locations l_S in S'_S to isomorphic locations l_K in S_K. This is actually a many-to-one mapping as multiple locations in the symbolic heap can alias a single location in the concrete heap, due to our representation of aliasing in $S_S.\mathcal{H}$ (again, recall §3.1).

We give a complete definition of partial equivalence below:

Definition 2 (Partial equivalence of states)**.** *Given a symbolic state S_S, an assignment Σ, and a concrete state S_K, let S'_S be the state produced when Σ is applied to S_S. We assume, without loss of generality, that the set of locations used by values in S'_S is disjoint from the set of locations used by values in S_K.*

We check if the memory graph in S'_S is isomorphic to a subset of the memory graph in S_K, where each memory graph is defined relative to the pointer roots in $\overline{\mathcal{Y}}$. If no such isomorphism exists, then S'_S and S_K are not partially equivalent.

Specifically, we must find a correspondence $\lambda(l_S) = l_K$ between heap locations l_S from S'_S and l_K from S_K such that λ satisfies the following conditions. We say that S'_S and S_K are partially equivalent *if and only if such a λ exits.*

- *$\forall l_S \in S'_S.l_S \in \lambda$. That is, if location l_S is used by any value in S'_S (not just in $S'_S.\mathcal{H}$), then it must have a mapping in λ.*
- *$\forall l_K \in S_K.\mathcal{H}$, if there does not exist an l_S such that $\lambda(l_S) = l_K$ and $l_S \in S'_S.\mathcal{H}$, then it should be possible to "expand" some symbolic pointer in S'_S to reach such an l_S. Specifically, there should exist an x such that $\Sigma(x) = \texttt{ptr}(l'_S, i)$ and $\lambda(l'_S) = l'_K$ (but $l'_S \notin S'_S.\mathcal{H}$, as x should be unexpanded), where either (a) $l'_K = l_K$, or (b) $l'_K \neq l_K$, but l_K is reachable from l'_K in $S_K.\mathcal{H}$. In case (a), we expand x directly to a heap object l_S (where $\lambda(l_S) = l_K$), and in case (b), we expand x to some heap object l'_S from which object l_S is transitively reachable.*
- *$\forall l_S \in S'_S.\mathcal{H}$, the heap object at $S'_S.\mathcal{H}(l_S)$ matches the heap object at $S_K.\mathcal{H}(\lambda(l_S))$. We say that two values v_S and v_K "match" if either $v_S = v_K$ or if $v_S = \texttt{ptr}(l_S, i_S)$ and $v_K = \texttt{ptr}(l_K, i_K)$ where $\lambda(l_S) = l_K$ and $i_S = i_K$.*
- *$T \in S'_S.\overline{\mathcal{Y}}$ if and only if $T \in S_K.\overline{\mathcal{Y}}$, and further, $\forall T \in S'_S.\overline{\mathcal{Y}}$, the stack $S'_S.\overline{\mathcal{Y}}(T)$ matches the youngest stack frames in $S_K.\overline{\mathcal{Y}}(T)$. This definition allows S_K to have deeper stack frames than S'_S, as the call stacks in S'_S may be underspecified (recall §2.2).*
- *$T \in S'_S.CallCtx$ if and only if $T \in S_K.CallCtx$, and further, $\forall T \in S'_S.CallCtx$, the stack $S'_S.CallCtx(T)$ matches the youngest stack frames in $S_K.CallCtx(T)$. As above, this definition allows S_K to have deeper stack frames than S'_S.*
- *T^{Curr} and T^E match exactly in S'_S and S_K.*
- *WQ matches exactly in S'_S and S_K. If the symbolic WQ uses initial waiter timestamps $\{x_0, x_1, \cdots\}$ (recall §4.2), then those initial entries of $S'_S.WQ$ are ordered by the concrete values of their respective timestamps as assigned by Σ.*
- *$T \in S'_S.\mathtt{L}^+$ if and only if $T \in S_K.\mathtt{L}^+$, and further, $\forall T \in S'_S.\mathtt{L}^+$, $S_K.\mathtt{L}^+(T) \subseteq S'_S.\mathtt{L}^+(T)$. This definition allows $S'_S.\mathtt{L}^+$ to be a conservative over-approximation of $S_K.\mathtt{L}^+$.*

Theorem 1 (Soundness and completeness of symbolic execution)**.** *Consider an initial program context, an initial concrete state S_K for that context, and an initial symbolic state S_S:*

- ***Soundness:** If symbolic execution from S_S outputs a pair (p, C), then for all S_K such that S_S models S_K under C, concrete execution from S_K must follow path p as long as context switches happen exactly as specified by path p.*
- ***Completeness:** If concrete execution from S_K follows path p, then for all S_S such that S_S models S_K under $S_S.C$, symbolic execution from S_S will either (a) output a pair (p, C), for some C, or (b) encounter a query that the SMT solver cannot solve.*

Proof. First, we state our assumptions. We assume the underlying points-to analysis is sound. We assume that *isSat* is sound, though not necessarily complete. We assume that synchronization libraries are correctly implemented, as otherwise, the invariants described in §4.3 would be incorrect. Note that the initial symbolic state S_S is a given, thus our theorem implicitly assumes correctness of the dataflow analyses used to construct that initial symbolic state. Finally, our completeness proof is valid only for paths p in which no thread continues executing after returning from its initial stack frame (recall from §2.2 that symbolic execution does not continue beyond this point). We now prove the theorem:

Soundness. The proof proceeds by induction over the length of an execution trace, with symbolic and concrete executions proceeding in lockstep. The base case is given. In the inductive cases, sym-

bolic execution from S_S and concrete execution from S_K take the next action on path p, resulting in states S'_S and S'_K, respectively, and we show that the inductive hypothesis is maintained (*i.e.*, that S'_S models S'_K under C).

Importantly, our proof must demonstrate that, whenever the symbolic execution makes a choice (such as at branch statements), the path constraint $S'_S.C$ and execution trace $S'_S.path$ must be updated in such a way that the concrete execution is *forced* to make the same choice. That is, the concrete execution cannot make a different choice unless it is *not* true that S_S models S_K under C.

In this proof, as a shorthand, we say that an expression in the symbolic state is *consistent* with a value in the concrete state if there exists a valid assignment Σ such that, if Σ is applied to the symbolic expression, then the resulting value matches the concrete value. (This is merely an application of Definition 1.)

We give one case for each possible program statement:

- **return stmt** *"return e"*: The symbolic and concrete executions pop call stacks in the same mechanical way, hence their updates will be consistent. The consistency of return value e in the symbolic and concrete executions follows from the inductive hypothesis. If the current thread T is returning from its final stack frame in $S_S.CallCtx$, then there are three special cases: (1) T is the only thread in T^E, in which case symbolic execution halts and the path completes; otherwise (2) $|S_K.CallCtx| = 1$ and T exits from both the symbolic and concrete executions; or (3) $|S_K.CallCtx| > 1$, in which case the call stacks are underspecified and symbolic execution will not schedule T again (recall §2.2). In case (3), on a technical point, to maintain the inductive hypothesis, we must ensure that $T \in S_S.\overline{\mathcal{Y}}$ if and only if $T \in S_K.\overline{\mathcal{Y}}$ (recall Definition 2)—we do this by not removing T from $\overline{\mathcal{Y}}$ or *CallCtx* when T exits (*e.g.*, see the second rule for `return` in the concrete semantics).
- **call stmt** *"$r \leftarrow e_f(e^*)$"*: The symbolic and concrete executions push call stacks in the same mechanical way, hence their updates will be consistent. The consistency of values e^* in the symbolic and concrete executions follows from the inductive hypothesis. At indirect calls, if the symbolic execution invokes function f, it updates the path constraint to $C' = (C \wedge e_f = f)$. By the inductive hypothesis, the concrete execution must derefence a function pointer e_f that is consistent with C' in the symbolic execution. Thus, the concrete execution must invoke the same function.
- **branch stmt** *"br e, γ_t, γ_f"*: When symbolic execution takes the branch to γ_t, it updates the path constraint to $C' = C \wedge e$ (similarly, to $C' = C \wedge \neg e$ for γ_f). This constraint is sufficiently narrow to force the concrete execution to take the same branch.
- **the first access of symbolic pointer x**: Suppose the symbolic execution is about to access pointer x for the first time. In this case, there does not yet exist a record $\{x, l_x, n_x\} \in S_S.\mathcal{A}$. Such a record will be appended and a new primary object l_x will be allocated. We must show that this update is performed in such a way that the resulting symbolic state will be consistent with the concrete state. Specifically, consider all pairs (l_S, i) such that $\Sigma(x) = \texttt{ptr}(l_S, i)$, where Σ is a valid assignment as in Definition 1. We first show that, for all such pairs (l_S, i), $x = \texttt{ptr}(l_S, i) \implies S_S.\mathcal{H}(l_S).fields = S_S.\mathcal{H}(l_x).fields$. This is ensured by Equation (1), which ensures that l_x has an initial state that matches all possible aliases. (Note that this argument implicitly assumes that the set of aliases is soundly chosen—this follows from our assumption that our underlying static points-to analysis is sound.) We next show that, for any l_K in the concrete heap that corresponds to one possible l_S, the objects at l_K and l_S are consistent. This follows directly from the inductive hypothesis when $l_S \neq l_x$, and otherwise, it follows from the fact that we assign each new symbolic heap object a fresh symbolic array (recall Equation (1)).
- **memory access stmt** *"$r \leftarrow load(p)$ or $store(p, e)$"*: First we assume that the access does not have a memory error. By the inductive hypothesis, the pointer p and value e are consistent in the symbolic and concrete executions. We consider three cases:

  (1) p has the form $\texttt{ptr}(l, e_{off})$ in the symbolic state. Note that this case can arise only if l was allocated by a call to `malloc` during symbolic execution. That is, l cannot alias any symbolic pointer x in S_S, and we do not need to consider the correctness of aliases. Thus, for this case, loads and stores perform the same mechanical action in both semantics, and we conclude that the resulting states are consistent.

  (2) p has the form x or $\texttt{ptradd}(x, e_{off})$ in the symbolic state and the action is a load. Suppose that $\Sigma(x) = \texttt{ptr}(l_S, i)$, where Σ satisfies Definition 1. We must show that the symbolic and concrete executions read consistent values. By the inductive hypothesis combined with the above case for the "first access of x," location l_S in the symbolic heap must be consistent with some corresponding location l_K in the concrete heap. Thus, we conclude that the concrete and symbolic loads will return consistent values.

  (3) p has the form x or $\texttt{ptradd}(x, e_{off})$ in the symbolic state and the action is a store. Suppose again that $\Sigma(x) = \texttt{ptr}(l_S, i)$. By the inductive hypothesis combined with the above case for the "first access of x," location l_S in the symbolic heap must be consistent with some corresponding location l_K in the concrete heap. We must show that l_S and l_K are updated in a consistent manner. This follows from, first, the fact that symbolic execution updates all $l \in S_S.lookupAliases(x)$, and second, from the assumption that our static points-to analysis is sound, which implies that the set of aliases in $S_S.\mathcal{A}$ soundly covers all possible aliases in the symbolic heap. Thus, we conclude that the concrete and symbolic heaps are updated in a consistent manner.

  As we do not give detailed semantics for memory errors in this paper, our proof will not discuss memory errors in detail. Briefly, at each access of p, the symbolic execution forks into two states—one with a memory error and one without—and updates the path constraint C in each state to describe each case. For soundness, the constraints must be narrow enough so that, if the symbolic execution does (or does not) follow a path with a memory error, then the concrete execution must (or must not) follow a path with the same memory error.
- **allocator stmt** *"malloc(e) or free(p)"*: Again, as we do not discuss memory errors in this paper in detail, these cases are trivial as the concrete and symbolic executions both perform the same mechanical action.
- **threadCreate stmt:** The consistency of values e_f and e_{arg} in the symbolic and concrete executions follows from the inductive hypothesis. Hence, as this statement has the same mechanical action in both semantics, the resulting states are consistent.
- **yield stmt:** By the inductive hypothesis, $S_S.T^E = S_K.T^E$. Hence, whichever next T^{Curr} is selected in the symbolic execution can also be selected by the concrete execution. Further, we conclude that the *same* next T^{Curr} will be selected by the concrete execution, as our inductive hypothesis assumes that context switches in the concrete execution are dictated precisely by the *path* output by the symbolic execution.
- **wait stmt** *"wait(p)"*: The consistency of value p in the symbolic and concrete executions follows from the inductive hypothesis. Hence, *WQ* is updated the same way in both executions. Further, as with *yield*, the next T^{Curr} selected by the symbolic execution will also be selected by the concrete execution.

- **notify stmt** *"notifyOne(p) or notifyAll(p)"*: By the inductive hypothesis, the wait queues *WQ* are consistent in both the symbolic and concrete states. Suppose the symbolic execution wakes a set of threads T^{woke} (which includes at most one thread for `notifyOne`). We submit that the constraints described in §4.2 are sufficiently narrow to force the concrete execution to wake the exact same set of threads, T^{woke}.
- **annotation** *"acquire(p) or release(p)"*: By the inductive hypothesis, $\texttt{L}^+$ is consistent in both the symbolic and concrete states. On `acquire`, $\texttt{L}^+$ is updated in mechanically the same way in both the symbolic and concrete semantics (recall §4.3). On `release`, the symbolic semantics removes p from $\texttt{L}^+(T^{Curr})$ only if there exists a lock in $\texttt{L}^+(T^{Curr})$ that must-equal p given the current path constraint. This makes the symbolic $\texttt{L}^+$ an over-approximation of the concrete $\texttt{L}^+$, which is allowed by Definition 2.

Completeness. The theorem trivially holds when *isSat* encounters an unsolvable query. Thus, in the remainder of this proof, we assume that *isSat* will soundly resolve any query it is given.

As above, the proof proceeds by induction over the length of an execution trace, with symbolic and concrete executions proceeding in lockstep. The base case is given. In the inductive cases, symbolic execution from S_S and concrete execution from S_K take the next action on path p, resulting in states S'_S and S'_K, respectively, and we show that the inductive hypothesis is maintained (*i.e.*, that S'_S models S'_K under $S'_S.C$). While the soundness proof required constraints to be sufficiently *narrow* to force concrete execution down a specific path, here we require constraints to be sufficiently *wide* so that the symbolic execution can cover all paths that might be followed during the concrete execution.

We again have one case for each possible program statement:

- **call stmt** *"$r \leftarrow e_f(e^*)$"*: As before, we argue that the symbolic and concrete executions push call stacks in the same mechanical way, hence their updates will be consistent. At indirect calls, if the concrete execution invokes function f, then the symbolic execution must cover a path that invokes function f. This follows, first, from the inductive hypothesis (the expression e_f in the symbolic state is consistent with f), and second, from the assumed soundness of our static points-to analysis that selects the set of possible target functions for this call site.
- **branch stmt** *"br e, γ_t, γ_f"*: When symbolic execution takes the branch to γ_t, it updates the path constraint to $C' = C \wedge e$ (similarly, to $C' = C \wedge \neg e$ for γ_f). These two constraints are sufficiently wide to cover all possible paths that the concrete execution may take.
- **memory access stmt or allocator stmt:** In the absence of memory errors, we can reuse the same argument from the soundness proof to argue that the resulting states are consistent. In the case of memory errors, recall again that at each access of p, the symbolic execution forks into two states—one with a memory error (S'_S) and one without (S''_S). For completeness, if the concrete execution encounters a memory error, then the updated path constraints in $S'_S.C$ must be wide enough so that, as execution proceeds from S'_S, the symbolic execution will encounter the same memory error.
- **yield stmt:** By the inductive hypothesis, $S_S.T^E = S_K.T^E$. Hence, whichever next T^{Curr} is selected in the concrete execution can also be selected by the symbolic execution.
- **notify stmt:** By the inductive hypothesis, the wait queues *WQ* are consistent in both the symbolic and concrete states. Suppose the concrete execution wakes a set of threads T^{woke} (which includes at most one thread for `notifyOne`). We submit that the constraints described in §4.2 are sufficiently wide so that, in at least one forked state, the symbolic execution will explore a path in which the exact same set of threads, T^{woke}, is woken.
- **return stmt, threadCreate stmt, wait stmt, or annotation:** In these cases, the concrete and symbolic semantics perform essentially the same mechanical updates. Hence, similarly to the proof cases stated above under *soundness*, the theorem holds for these statements. (Further, for `wait`(p), we observe that, as for `yield`(), any next T^{Curr} chosen by the concrete execution can also be chosen in the symbolic execution.)

□