# Verification of Higher-Order Concurrent Programs with Dynamic Resource Creation

Kazuhide Yasukata, Takeshi Tsukada, and Naoki Kobayashi

The University of Tokyo

**Abstract.** We propose a sound and complete static verification method for (higher-order) concurrent programs with *dynamic creation* of resources, such as locks and thread identifiers. To deal with (possibly infinite) resource creation, we prepare a finite set of abstract resource names and introduce the notion of *scope-safety* as a sufficient condition for avoiding the confusion of different concrete resources mapped to the same abstract name. We say that a program is *scope-safe* if no resource is used after the creation of another resource of the same abstract name. We prove that the pairwise-reachability problem is decidable for scope-safe programs with nested locking. We also propose a method for checking that a given program is scope-safe and with nested locking.

## 1 Introduction

Verification of concurrent programs is important but fundamentally difficult. Ramalingam [11] has proved that the reachability problem for two-thread programs is undecidable in the presence of rendezvous-style synchronization and recursive procedures. To deal with this limitation, several restricted models of concurrent computation have been studied. Kahlon et al. [2] have shown that the pairwise-reachability problem ("Given a program and their control locations, can the locations be reached simultaneously") for multi-threaded programs (without dynamic thread creation) is decidable if only nested locking with a finite number of locks is allowed as synchronization primitives. The result has later been extended to allow dynamic thread creation [7], joins (to wait for the termination of all the child threads) [1], and higher-order functions [13].

One of the important limitations in the programming models in the above line of work is that the dynamic creation of locks is not allowed; the number of locks must be finite and they must be statically allocated. In real programs, dynamic creation of locks is common; for example, in Java, every object may be used as a lock. Another, related limitation is that, although dynamic thread creation is supported [1, 7, 13], there is no way to refer to each thread, e.g., to specify the target of a join; the models in [1] and [13] support join operations, but they can only be used for synchronizing with *all* the child threads. Again, this deviates from real concurrent programming models.

To address the limitation above, we propose a method for checking the pairwise reachability of higher-order concurrent programs with primitives for dynamic creation of locks and thread identifiers. To keep the verification problem

decidable, however, we introduce the notion of *scope safety*. We consider a map from concrete resources (such as locks and thread identifiers) to *abstract* resources, and say that a program is scope-safe if, intuitively, at most one concrete resource per each abstract resource is accessible to each thread at each run-time state. For example, consider the following program:

```
main() {
  l = newlock();              /* create a new lock */
  spawn{acq(l);L:rel(l);};    /* spawn a child thread */
  acq(l);L:rel(l);            /* acquire and release l */
  main();   }                  /* repeat */
```

The main function repeatedly creates a new lock, spawns a child process, and synchronizes with it through the lock. Although infinitely many locks are created, only one lock is visible to each thread; thus, the program is scope-safe. We show that for the class of scope-safe programs with nested locking, the pairwise reachability problem is decidable. Our method is an extension of Yasukata et al.'s one [13], based on higher-order model checking. We also present a method for deciding whether a given program satisfies the required condition: scope safety and nested locking. The latter method is also based on a reduction to higher-order model checking. We have implemented our verification method and confirmed its effectiveness.

The rest of the paper is structured as follows. Section 2 introduces the target language with dynamic lock creation and gives the formal definitions of pairwise reachability and scope safety. Section 3 describes our method for checking the pairwise-reachability of a given program under the assumption that the program is scope-safe and has nested locking. Section 4 describes a method for checking whether a given program is scope-safe and has nested locking. Section 5 briefly describes how to extend our method for pairwise reachability to support join operations. Section 6 reports the experimental reports. Section 7 discusses related work and Section 8 concludes the paper.

Due to the space limitation, we omit some proofs and formal definitions, which can be found in the full version available at the last author's web page.

## 2 Pairwise Reachability Problem and Scope Safety

This section formally defines the problem that we address in this paper, namely, the pairwise reachability problem of higher-order concurrent programs with dynamic lock creation. We also introduce the notion of *scope safety*, which is a condition on programs that is crucial for the soundness and completeness of our verification method.

### 2.1 Language

The target of our analysis is a simply-typed, call-by-name, non-deterministic, higher-order language with primitives for nested locking and dynamic creation

of threads and locks. An extension with join primitives shall be introduced later in Section 5.

**Definition 1 (programs).** A *program* $p$ is a finite set of function definitions:

$$p = \{\, F_1\, \widetilde{x}_1 = e_1, \ldots, F_n\, \widetilde{x}_n = e_n \,\}\,.$$

Here, $F_i$ and $\tilde{x}_i$ denote a function symbol and a sequence of variables respectively. We allow more than one definition for each function symbol (so that a program has non-determinism), and assume that $F_i = S$ and $\widetilde{x}_i$ is an empty sequence for some $i$; $S$ serves as the "main" function. The meta-variable $e$ ranges over the set $\mathsf{Exp}$ of *expressions*, defined by:

$$e ::= (\,) \mid x \mid F \mid e_1\, e_2 \mid \mathbf{new}_\kappa\ e \mid \mathbf{acq}(e_1);\ e_2 \mid \mathbf{rel}(e_1);\ e_2 \mid \mathbf{spawn}(e_c);\ e_p \mid e^\ell.$$

Here, $\ell$ and $\kappa$ range over a finite set $\mathsf{Label}$ of program point labels and a finite set $\mathcal{K}$ of identifiers (called *abstract lock names*, or abstract locks).

The intuitive meaning of each expression is as follows; the formal operational semantics will be given later. The expression $(\,)$ denotes the unit value, and $e_1 e_2$ applies the function $e_1$ to $e_2$. The expression $\mathbf{new}_\kappa\ e$ creates a new (concrete) lock with abstract name $\kappa$, and passes it to the function $e$. The expression $\mathbf{acq}(e_1);\ e_2$ waits to acquire the lock $e_1$, and executes $e_2$ after the acquisition. The expression $\mathbf{rel}(e_1);\ e_2$ releases the lock $e_1$ and then executes $e_2$. The expression $\mathbf{spawn}(e_c);\ e_p$ spawns a child thread that executes $e_c$, and then the parent thread itself executes $e_p$. The labeled expression $e^\ell$ just behaves like $e$; $\ell$ is used to specify the pairwise reachability problem, and does not affect the operational semantics.

We require that all the programs must be simply-typed. The set of *types* is defined by:

$$\tau ::= \star \mid \mathbf{lock} \mid \tau_1 \to \tau_2,$$

where $\star$ and **lock** describe the unit value and locks respectively. The type $\tau_1 \to \tau_2$ describes functions from $\tau_1$ to $\tau_2$. The type system is deferred to Appendix A, as it is standard. The only point deserving attention is that in each function definition $F\, \widetilde{x} = e$, the type of $e$ must be $\star$; this condition can be ensured by applying CPS transformation [9].

*Remark 1.* The language has only the unit value and locks as primitive data. Boolean values can be expressed by using Church encoding. Infinite data domains such as integers and lists can also be handled by using predicate abstraction [6] (though the completeness of the analysis will be lost by the abstraction).

*Example 1.* Consider the following program $p$.

$$p = \left\{ \begin{array}{l} S = \mathbf{new}_\kappa\ F \\ F\ x = \mathbf{spawn}(\mathbf{acq}(x);\ (\mathbf{rel}(x);\ (\,))^\ell);\ (\mathbf{acq}(x);\ (\mathbf{rel}(x);\ S)^\ell) \end{array} \right\}.$$

This program is obtained by CPS-transforming the following C-like code:

```
main() {  x = newlock(); spawn{acq(x);L:rel(x);};
  acq(x);L:rel(x); main(); }
```

In every loop, the root thread and a created child thread enter the program point labeled $\ell$ by using a created lock $x$.

Now we define the operational semantics of the language. In the following semantics, we use a sequence $\iota$ of natural numbers as a thread identifier and a triple $(\kappa, \iota, m)$ as a lock identifier (also called a *concrete lock name*); that is just for the technical convenience in formalizing our method. Intuitively, $(\kappa, \iota, m)$ represents the $m$-th lock created by the thread $\iota$ at $\mathbf{new}_\kappa$. We write $\nu$ for a lock identifier. In presenting the operational semantics below, $e$ ranges over expressions extended with lock identifiers: $e ::= \cdots \mid (\kappa, \iota, m)$. A *thread state* is a quadruple $(e, L, s, \sigma)$ where $e$ is the (extended) expression to be executed by the thread, $L \in (\mathcal{K} \times \mathbb{N}^* \times \mathbb{N}_+)^*$ describes the lock acquisition history of the thread, $s \in \mathbb{N}$ is the number of children spawned by the thread so far, and $\sigma$ is a partial map from $\mathcal{K}$ to $\mathbb{N}^* \times \mathbb{N}_+$; intuitively, $\sigma(\kappa) = (\iota, m)$ means that the concrete lock with abstract name $\kappa$ created most recently by the thread or inherited from the parent thread is $(\kappa, \iota, m)$. A *configuration* is a partial map $c$ from a finite set consisting of sequences of natural numbers (where each sequence serves as a process identifier) to the set of thread states. A transition relation $c_1 \xrightarrow{\iota, a}_p c_2$ on configurations is the least relation closed under the rules given below. We write $\uplus$ for the disjoint union, $\emptyset$ for the empty map, and $f\,\{\, x \mapsto v \,\}$ for the map defined by: $f\,\{\, x \mapsto v \,\}\,(x) = v$ and $f\,\{\, x \mapsto v \,\}\,(y) = f(y)$ for $y \neq x$.

$$\frac{F\,\tilde{x} = e' \in p}{c \uplus \{\, \iota \mapsto (F\,\widetilde{e}, L, s, \sigma) \,\} \xrightarrow{\iota, \bullet}_p c \uplus \{\, \iota \mapsto ([\widetilde{e}/\tilde{x}]e', L, s, \sigma) \,\}}$$

$$\frac{\sigma(\kappa) = (\iota, m) \qquad \sigma' = \sigma\,\{\, \kappa \mapsto (\iota, m+1) \,\}}{c \uplus \{\, \iota \mapsto (\mathbf{new}_\kappa\, e, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{new}(\kappa, \iota, m+1)}_p c \uplus \{\, \iota \mapsto (e\ (\kappa, \iota, m+1), L, s, \sigma') \,\}}$$

$$\frac{\forall \iota', m.(\sigma(\kappa) = (\iota', m) \Rightarrow \iota \neq \iota') \qquad \sigma' = \sigma\,\{\, \kappa \mapsto (\iota, 1) \,\}}{c \uplus \{\, \iota \mapsto (\mathbf{new}_\kappa\, e, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{new}(\kappa, \iota, 1)}_p c \uplus \{\, \iota \mapsto (e\ (\kappa, \iota, 1), L, s, \sigma') \,\}}$$

$$\frac{(\kappa, \iota', m) \notin \mathbf{locked}(c \uplus \{\, \iota \mapsto (\mathbf{acq}(\kappa, \iota', m); e, L, s, \sigma) \,\})}{c \uplus \{\, \iota \mapsto (\mathbf{acq}(\kappa, \iota', m); e, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{acq}(\kappa, \iota', m)}_p c \uplus \{\, \iota \mapsto (e, L \cdot (\kappa, \iota', m), s, \sigma) \,\}}$$

$$\frac{L = L' \cdot (\kappa, \iota', m)}{c \uplus \{\, \iota \mapsto (\mathbf{rel}(\kappa, \iota', m); e, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{rel}(\kappa, \iota', m)}_p c \uplus \{\, \iota \mapsto (e, L', s, \sigma) \,\}}$$

$$\frac{}{c \uplus \{\, \iota \mapsto (\mathbf{spawn}(e_c); e_p, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{sp}(\iota \cdot s)}_p c \uplus \left\{ \begin{array}{ll} \iota & \mapsto (e_p, L, s+1, \sigma), \\ \iota \cdot s & \mapsto (e_c, \epsilon, 0, \sigma) \end{array} \right\}}$$

$$\frac{}{c \uplus \{\, \iota \mapsto (e^{\ell}, L, s, \sigma) \,\} \xrightarrow{\iota,\ell}_p c \uplus \{\, \iota \mapsto (e, L, s, \sigma) \,\}}$$

$$\frac{L = \epsilon}{c \uplus \{\, \iota \mapsto ((\,), L, s, \sigma) \,\} \xrightarrow{\iota,\$}_p c}$$

In the first rule, $[\widetilde{e}/\widetilde{x}]e'$ denotes the expression obtained from $e'$ by simultaneously substituting $\widetilde{e}$ for $\widetilde{x}$. The second and third rules are for lock creations. A lock identifier of the form $(\kappa, \iota, m')$ is allocated to a new lock, where $m'$ is the number of locks created so far (including the new one) by the thread $\iota$ at $\mathbf{new}_\kappa$. In the fourth rule, $\mathbf{locked}(c)$ denotes the set of locks acquired by some thread, i.e., the set $\{\, (\kappa, \iota, m) \mid \exists \iota'.\ c(\iota') = (e, L, s, \sigma) \wedge (\kappa, \iota, m) \in L \,\}$. The fourth and fifth rules ensure that locks are acquired/released in a nested manner, i.e., that each thread releases locks in the opposite order of acquisition; the execution of a thread violating this condition gets stuck. We write $c_0$ for the *initial configuration* $\{\, \epsilon \mapsto (S, \epsilon, 0, \emptyset) \,\}$. We sometimes omit transition labels and just write $c \to_p c'$ for $c \xrightarrow{\iota,a}_p c'$ for some $\iota$ and $a$.

Recall that a program has to acquire/release locks in the nested manner; otherwise a thread spawned by the program gets stuck at a release operation. We say that a program *has nested locking* if no release operations get stuck, that is, whenever the program reaches a configuration $c \uplus \{\, \iota \mapsto (\mathbf{rel}(\kappa, \iota', m); e, L, s, \sigma) \,\}$, $(\kappa, \iota', m)$ is the last lock acquired by the thread $\iota$, i.e., $L$ is of the form $L' \cdot (\kappa, \iota', m)$.

### 2.2 Pairwise Reachability

Now we define the goal of our analysis: pairwise reachability.

**Definition 2 (pairwise reachability).** Let $p$ be a program and $\ell_1, \ell_2$ be labels. We say that $(\ell_1, \ell_2)$ is *pairwise-reachable* by $p$, written $p \vDash \ell_1 \| \ell_2$, if

$$c_0 \to_p^* c \uplus \{\, \iota_1 \mapsto (e_1^{\ell_1}, L_1, s_1, \sigma_1), \iota_2 \mapsto (e_2^{\ell_2}, L_2, s_2, \sigma_2) \,\}$$

holds for some $c, \iota_1, \iota_2, L_1, L_2, s_1, s_2, \sigma_1, \sigma_2$ with $\iota_1 \neq \iota_2$. The *pairwise-reachability problem* is the problem of deciding whether $p \vDash \ell_1 \| \ell_2$ holds.

*Example 2.* Recall the program of Example 1. It has the following transitions:

$$
\begin{aligned}
&\{\,\epsilon \mapsto (S, \epsilon, 0, \emptyset)\,\} \longrightarrow \{\,\epsilon \mapsto (\mathbf{new}_\kappa\ F, \epsilon, 0, \emptyset)\,\} \\
\longrightarrow &\{\,\epsilon \mapsto (\mathbf{spawn}(\mathbf{acq}(\kappa, \epsilon, 1);\ (\mathbf{rel}(\kappa, \epsilon, 1);\ ())^\ell);\ \cdots, \epsilon, 0, \{\,\kappa \mapsto (\epsilon, 1)\,\})\,\} \\
\longrightarrow^* &\left\{\begin{array}{l}\epsilon \mapsto (\mathbf{new}_\kappa\ F, \epsilon, 1, \{\,\kappa \mapsto (\epsilon, 1)\,\}), \\ 0 \mapsto (\mathbf{acq}(\kappa, \epsilon, 1);\ (\mathbf{rel}(\kappa, \epsilon, 1);\ ())^\ell, \epsilon, 0, \{\,\kappa \mapsto (\epsilon, 1)\,\}\end{array}\right\} \\
\longrightarrow &\left\{\begin{array}{l}\epsilon \mapsto (\mathbf{spawn}(\mathbf{acq}(\kappa, \epsilon, 2);\ (\mathbf{rel}(\kappa, \epsilon, 2);\ ())^\ell);\ \cdots, \epsilon, 1, \{\,\kappa \mapsto (\epsilon, 2)\,\}), \\ 0 \mapsto (\mathbf{acq}(\kappa, \epsilon, 1);\ (\mathbf{rel}(\kappa, \epsilon, 1);\ ())^\ell, \epsilon, 0, \{\,\kappa \mapsto (\epsilon, 1)\,\})\end{array}\right\} \\
\longrightarrow^* &\left\{\begin{array}{l}\epsilon \mapsto (S, \epsilon, 2, \{\,\kappa \mapsto (\epsilon, 2)\,\}), \\ 0 \mapsto (\mathbf{acq}(\kappa, \epsilon, 1);\ (\mathbf{rel}(\kappa, \epsilon, 1);\ ())^\ell, \epsilon, 0, \{\,\kappa \mapsto (\epsilon, 1)\,\}), \\ 1 \mapsto (\mathbf{acq}(\kappa, \epsilon, 2);\ (\mathbf{rel}(\kappa, \epsilon, 2);\ ())^\ell, \epsilon, 0, \{\,\kappa \mapsto (\epsilon, 2)\,\})\end{array}\right\} \\
\longrightarrow^* &\left\{\begin{array}{l}\epsilon \mapsto (S, \epsilon, 2, \{\,\kappa \mapsto (\epsilon, 2)\,\}), 0 \mapsto ((\mathbf{rel}(\kappa, \epsilon, 1);\ ())^\ell, (\kappa, \epsilon, 1), 0, \{\,\kappa \mapsto (\epsilon, 1)\,\}), \\ 1 \mapsto ((\mathbf{rel}(\kappa, \epsilon, 2);\ ())^\ell, (\kappa, \epsilon, 2), 0, \{\,\kappa \mapsto (\epsilon, 2)\,\})\end{array}\right\}
\end{aligned}
$$

Thus, the program is pairwise-reachable to $(\ell, \ell)$. If the definition of $F$ is replaced by:

$$F\ x = \mathbf{spawn}(\mathbf{acq}(x);\ (\mathbf{rel}(x);\ ())^\ell);\ (\mathbf{acq}(x);\ (\mathbf{rel}(x);\ F\ x)^\ell),$$

then the resulting program is *not* pairwise-reachable to $(\ell, \ell)$, since all the program points $\ell$ are now guarded by the same (concrete) lock.

### 2.3 Scope Safety

Pairwise reachability is known to be decidable for the language *without* dynamic lock creations [13]. In the presence of dynamic lock creations, the decidability of pairwise reachability is open, to our knowledge. To make the problem tractable, we introduce the notion of *scope safety*: a thread of a scope-safe program can access only the newest lock in the scope for each abstract lock $\kappa$.

**Definition 3 (scope-safety).** A program $p$ is *scope-safe* if

$$c_0 \to_p^* c \uplus \{\,\iota \mapsto (op(\kappa, \iota', m); e,\ L, s, \sigma)\,\} \quad \Longrightarrow \quad \sigma(\kappa) = (\iota', m)$$

holds for every $c, \iota, (\kappa, \iota', m), e, L, s, \sigma$ and $op \in \{\,\mathbf{acq}, \mathbf{rel}\,\}$.

For a scope-safe program, the number of locks is locally bounded in the sense that, at each run-time state, the number of locks accessible to each thread is bounded. Note that the number of locks in a configuration is still unbounded.

*Example 3.* The program in Example 1 is scope-safe. Although infinitely many locks are created with the abstract lock name $\kappa$, every thread accesses only the lock that is most recently created by itself or the parent thread.

The following program is *not* scope-safe:

$$
\begin{aligned}
&S = \mathbf{new}_\kappa\ G \qquad G\ x = \mathbf{new}_\kappa(F\ x) \\
&F\ x\ y = \mathbf{spawn}(\mathbf{acq}(x);\ (\mathbf{rel}(x);\ ())^\ell);\ (\mathbf{acq}(y);\ (\mathbf{rel}(y);\ S)^\ell).
\end{aligned}
$$

$F\,x\,y$ accesses two locks $x$ and $y$ with the same abstract lock $\kappa$ simultaneously. If $\kappa$ in $G$ is renamed to $\kappa'$, however, the resulting program is scope-safe, since $x$ and $y$ now have different abstract lock names: $\kappa$ and $\kappa'$ respectively.

Now we can state the main result of this paper proved in the next section.

**Theorem 1.** *The pairwise reachability problem for scope-safe programs with nested locking is decidable.*

We think that the scope-safety is a natural assumption, and that there are many programs that create an unbounded number of locks but satisfy the scope safety. Like the program in Example 1, such a program typically spawns an unbounded number of threads, each of which creates a lock (thus; the number of locks is globally unbounded) and uses it locally for synchronizations with child threads.

## 3 Verification of Pairwise Reachability

This section gives a sound and complete verification method of the pairwise-reachability problem of scope-safe programs with nested locking. We reduce the problem to (a variant of) higher-order model checking, a decision problem whether a language of a given higher-order tree grammar is a subset of a given regular tree language.

1. We use (extended) action trees [1, 7], which represent transition sequences in a thread-wise manner. Let $\mathsf{ATrees}(p)$ be the set of all action trees representing possible transitions of the program $p$. Then $p \vDash \ell_1 \| \ell_2$ if and only if $\mathsf{ATrees}(p)$ contains an action tree with leaves labeled by $\ell_1$ and $\ell_2$, respectively. Writing $R_{\ell_1,\ell_2}$ for the set of action trees with leaves labeled by $\ell_1$ and $\ell_2$, the pairwise reachability problem is reduced to the emptiness problem of $\mathsf{ATrees}(p) \cap R_{\ell_1,\ell_2}$.

2. If we could represent $\mathsf{ATrees}(p)$ by a higher-order tree grammar, we would be done, since the emptiness problem $\mathsf{ATrees}(p) \cap R_{\ell_1,\ell_2} \stackrel{?}{=} \emptyset$ is equivalent to the higher-order model checking problem $\mathsf{ATrees}(p) \stackrel{?}{\subseteq} \overline{R_{\ell_1,\ell_2}}$; note that $\overline{R_{\ell_1,\ell_2}}$ is regular. Unfortunately, however, it is not easy to give a grammar to generate $\mathsf{ATrees}(p)$ because of synchronization by locks. Instead we consider a superset of $\mathsf{ATrees}(p)$, written $\mathsf{RelaxedATrees}(p)$, including action trees that are infeasible because of locks. In other words, an action tree in $\mathsf{RelaxedATrees}(p)$ represents a transition sequence in which the synchronization constraint on locks is ignored. It is easy to construct a grammar generating $\mathsf{RelaxedATrees}(p)$.

3. We give a way to check the feasibility of an action tree (i.e. whether an action tree conforms to the synchronization constraint on locks) by introducing an operational semantics of action trees. Let $\mathsf{LSATrees}$ be the set of feasible action trees. We show that $\mathsf{ATrees}(p) = \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$ provided that $p$ is a scope-safe program. Thus, the pairwise reachability is reduced to the (non)-emptiness of $\mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees} \cap R_{\ell_1,\ell_2}$.

4. We show that $\mathsf{LSATrees}$ is a regular tree language.

Now the pairwise reachability problem has been reduced to the emptiness problem $\mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees} \cap R_{\ell_1,\ell_2} \stackrel{?}{=} \emptyset$, which is equivalent to the instance of higher-order model checking problem:

$$\mathsf{RelaxedATrees}(p) \stackrel{?}{\subseteq} \overline{\mathsf{LSATrees} \cap R_{\ell_1,\ell_2}}$$

and thus decidable. In the rest of this section, we first review higher-order model checking [5, 8], in Section 3.1. We then explain each step in Sections 3.2–3.5.

### 3.1 Higher-order Model Checking

Higher-order model checking is concerned about properties of the trees generated by higher-order tree grammars called *higher-order recursion schemes* (HORS, in short). In the standard definition of higher-order model checking [5, 8], a HORS is treated as a generator of a *single*, possibly *infinite* tree. In the present paper, we consider a (non-deterministic) HORS as a generator of a *finite tree language* (i.e., a set of finite trees).

**Definition 4 (non-deterministic HORS).** Let $\Sigma$ be a finite set of symbols called *tree constructors*. We assume that each tree constructor is associated with a non-negative integer called an *arity*. A (non-deterministic) HORS is a set of function definitions: $\{ F_1\, \widetilde{x}_1 = t_1, \ldots, F_n\, \widetilde{x}_n = t_n \}$, where $t_i$ ranges over the set of terms given by: $t ::= a \mid x \mid F \mid t_1\, t_2$. Here, $a$ ranges over $\Sigma$. As in the language in Section 2, we allow more than one definition for each function symbol $F_i$, and require that $F_i = S$ and $\widetilde{x}_i$ is empty for some $i$. We also require that HORS be simply-typed; in each definition $F\, \widetilde{x} = t$, $t$ must have the tree type $\mathtt{o}$. Each constructor of arity $k$ is given type $\underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{k} \to \mathtt{o}$.

Given a HORS $\mathcal{G}$, the reduction relation $\longrightarrow_{\mathcal{G}}$ on terms is defined by: (i) $F\, t_1 \cdots t_k \longrightarrow_{\mathcal{G}} [t_1/x_1, \ldots, t_k/x_k]t$ if $F\, x_1 \cdots x_k = t \in \mathcal{G}$; and (ii) $a\, t_1 \cdots t_i \cdots t_k \longrightarrow_{\mathcal{G}} a\, t_1 \cdots t'_i \cdots t_k$ if $t_i \longrightarrow_{\mathcal{G}} t'_i$. We call a term $t$ a ($\Sigma$-labeled) *tree* if it consists of only tree constructors, and write **Tree** for the set of trees. The language generated by a HORS $\mathcal{G}$, written $\mathcal{L}(\mathcal{G})$, is $\{ t \in \mathbf{Tree} \mid S \longrightarrow^*_{\mathcal{G}} t \}$.

Compared with the language in the previous section, we have tree constructors in HORS instead of primitives on locks and treads. The following is an easy corollary of the decidability of (the standard version of) higher-order model checking [8].

**Theorem 2.** *Given a HORS $\mathcal{G}$ and a regular language $R$, it is decidable whether $\mathcal{L}(\mathcal{G}) \subseteq R$ holds.*

### 3.2 Action Trees

Action trees, first introduced by Lammich et al. [7], represent thread-wise action histories of a concurrent program. We extend them to deal with dynamic lock creation.

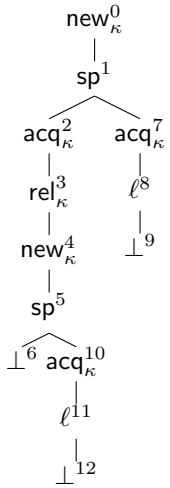
**Definition 5 (action trees).** The set $\mathcal{T}$ of *action trees*, ranged over by $\gamma$, is defined inductively by: $\gamma ::= \bot \mid \$ \mid \ell\,\gamma \mid \mathsf{new}_\kappa\gamma \mid \mathsf{acq}_\kappa\gamma \mid \mathsf{rel}_\kappa\gamma \mid \mathsf{sp}\,\gamma_p\,\gamma_c$.

Each inner node of $\gamma$ represents an action performed by a thread. The tree $\ell\,\gamma$ means that the thread has reached an expression labeled $\ell$ and then behaved like $\gamma$. The tree $\mathsf{new}_\kappa\gamma$ means that the thread has created a new lock of abstract name $\kappa$, and then behaved like $\gamma$. The tree $\mathsf{acq}_\kappa\gamma$ (resp. $\mathsf{rel}_\kappa\gamma$) means that the thread has acquired (resp. released) a lock of abstract name $\kappa$, and then behaved like $\gamma$. The tree $\mathsf{sp}\,\gamma_p\,\gamma_c$ means that the thread has spawned a child thread that behaved like $\gamma_c$, and the thread itself behaved like $\gamma_p$.

A leaf node represents the status of the thread: $\bot$ means that it is alive and \$ means that it has terminated.

*Example 4.* The figure on the righthand side below shows the action tree corresponding to the transition sequence in Example 2. The superscripts 0–12 are the node numbers added for the convenience of explanation; they reflect the order of actions in the transition sequence in Example 2.

The tree represents the computation in which (i) the root thread creates a new lock with abstract name $\kappa$ (as shown by node 0), spawns a new thread (node 1), acquires and releases the lock (nodes 2 and 3), creates another lock with the same abstract name $\kappa$ (node 4) and spawns another thread (node 5), and (ii) the two child threads acquire the locks (nodes 7 and 10) and reaches the program point $\ell$ (nodes 8 and 11). The leaves of the action tree show that all the threads are still alive (nodes 6, 9 and 12). Note that the locks acquired by the two child threads are different, although nodes 7 and 10 have the same label $\mathsf{acq}_\kappa$; based on the scope safety assumption, $\kappa$ refers to the lock created at the closest ancestor node labeled by $\mathsf{new}_\kappa$. Thus, nodes 7 and 10 refer to the locks created at nodes 0 and 4 respectively.




Note also that the action tree does not specify the order between actions of different threads. For example, the action at node 7 may occur before the one at node 4. Due to the synchronization constraint on locks, however, some order may be implicitly imposed; for example, since nodes 2, 3, and 7 refer to the same lock created at node 0, the action at 3 must precede the one at 7. □

For a sequence of events $(\iota_1, a_1)\cdot\cdots\cdot(\iota_n, a_n)$, we write $\mathsf{a}((\iota_1, a_1)\cdot\cdots\cdot(\iota_n, a_n))$ for the corresponding action tree; see Appendix B.2 for the formal definition. We write $\mathsf{ATrees}(p)$ for the set

$$\{\mathsf{a}((\iota_1, a_1)\cdot\cdots\cdot(\iota_n, a_n)) \mid c_0 \xrightarrow{\iota_1,a_1}_p c_1 \xrightarrow{\iota_2,a_2}_p \cdots \xrightarrow{\iota_n,a_n}_p c_n; c_0 \text{ is the initial configuration.}\}$$

of action trees of all the possible transition sequences of the program $p$.

We write $R_{\ell_1,\ell_2}$ for the set of action trees of the form $C[\ell_1\bot, \ell_2\bot]$, where $C$ is a tree context with two holes. Note that $R_{\ell_1,\ell_2}$ does not depend on the program. Clearly, the set $R_{\ell_1,\ell_2}$ is regular. By the definition of $\mathsf{ATrees}(p)$, the pairwise reachability is reduced to the non-emptiness of $\mathsf{ATrees}(p) \cap R_{\ell_1,\ell_2}$, as stated below.

**Lemma 1.** *For every program $p$ and every pair of labels $(\ell_1, \ell_2)$, we have*

$$p \vDash \ell_1 \| \ell_2 \iff \mathsf{ATrees}(p) \cap R_{\ell_1,\ell_2} \neq \emptyset.$$

### 3.3 Relaxed Transition of Programs

The next step is to obtain a finitary representation of $\mathsf{ATrees}(p)$. If we were able to represent $\mathsf{ATrees}(p)$ as a HORS, then the (non-)emptiness problem $\mathsf{ATrees}(p) \cap R_{\ell_1,\ell_2}$ obtained in Lemma 1 can be solved by higher-order model checking. Unfortunately, a direct construction of such a HORS is difficult, due to the synchronization constraint on locks.

Instead, we consider an approximation $\mathsf{RelaxedATrees}(p)$ of $\mathsf{ATrees}(p)$, which are obtained by ignoring the synchronization constraint, and represent it as a HORS. The set $\mathsf{ATrees}(p)$ is then obtained as $\mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$, where $\mathsf{LSATrees}$ is the set of all the action trees that respect the synchronization constraint but are independent of the program $p$. In this subsection, we define $\mathsf{RelaxedATrees}(p)$ and provide its grammar representation; $\mathsf{LSATrees}$ shall be constructed and proved to be regular in Sections 3.4 and 3.5.

A *relaxed* transition relation $c_1 \overset{\iota,a}{\dashrightarrow}_p c_2$ on configurations is the least relation closed under the rules in Section 2 except that the conditions $(\kappa, \iota', m) \notin \mathbf{locked}(c \uplus \{\, \iota \mapsto (\mathbf{acq}(\kappa, \iota', m);\ e_2, L, \sigma) \,\})$ of the fourth rule (for lock acquisition), $L = L' \cdot (\kappa, \iota', m)$ of the fifth rule (for lock release) and $L = \epsilon$ of the last rule (for thread termination) are removed. Similarly to $\mathsf{ATrees}(p)$ we write $\mathsf{RelaxedATrees}(p)$ for the set

$$\{\mathsf{a}((\iota_1, a_1) \dots (\iota_n, a_n)) \mid c_0 \overset{\iota_1,a_1}{\dashrightarrow}_p \dots \overset{\iota_n,a_n}{\dashrightarrow}_p c_n\}$$

of action trees of all possible relaxed transition sequence of the program $p$. Obviously $\mathsf{RelaxedATrees}(p)$ is a superset of $\mathsf{ATrees}(p)$.

We can easily transform a given program $p$ into a HORS $\mathcal{G}_p$ whose language is $\mathsf{RelaxedATrees}(p)$. Each lock is replaced by a pair of tree constructors $\mathsf{acq}_\kappa$ and $\mathsf{rel}_\kappa$, and each action in the program is replaced by a construction of the corresponding tree node. For example, $\mathbf{spawn}(e_1);\ e_2$ and $\mathbf{new}_\kappa\ e_1$ are respectively transformed to $\mathsf{sp}\ e_1'\ e_2'$, and $\mathsf{new}_\kappa\ (e_1'\ (\mathsf{acq}_\kappa, \mathsf{rel}_\kappa))$, where $e_i'$ is obtained by recursively transforming $e_i$. (Here, for the sake of simplicity, we have used pairs as primitives; they can be represented as functions using the standard Church encoding.) In addition, since we are interested in intermediate states of a program instead of the final state, we prepare rules to abort reductions and generate leaves $\bot$. We illustrate these points through an example below; the formal definition of $\mathcal{G}_p$ is given in Appendix B.4.

*Example 5.* Recall Example 1. The set $\mathsf{RelaxedATrees}(p)$ is generated by the following HORS $\mathcal{G}_p$:

$$
\begin{array}{l}
S = \mathit{New}_\kappa\ F \\
F\ x = \mathit{Spawn}\ (\mathit{Acq}\ x\ (\mathit{Label}_\ell\ (\mathit{Rel}\ x\ \mathit{End})))\ (\mathit{Acq}\ x\ (\mathit{Label}_\ell\ (\mathit{Rel}\ x\ S))) \\
\mathit{New}_\kappa\ e = \mathsf{new}_\kappa\ (e\ (\mathsf{acq}_\kappa, \mathsf{rel}_\kappa)) \qquad \mathit{Acq}\ (x_a, x_r)\ e = x_a\ e \qquad \mathit{Rel}\ (x_a, x_r)\ e = x_r\ e \\
\mathit{Spawn}\ e_c\ e_p = \mathsf{sp}\ e_p\ e_c \qquad \mathit{Label}_\ell\ e = \ell\, e \qquad \mathit{End} = \$ \\
N\ \tilde{x} = \bot \qquad (\text{for each } N \in \{ S, F, \mathit{New}_\kappa, \mathit{Acq}, \mathit{Rel}, \mathit{Spawn}, \mathit{Label}_\ell, \mathit{End} \}.)
\end{array}
$$

The first two lines correspond to the function definitions in the original programs; we have just replaced each action with the corresponding function symbols. The next two lines define functions for generating a tree node corresponding to each action. The function $\mathit{New}_\kappa$ represents a new lock as a pair $(\mathsf{acq}_\kappa, \mathsf{rel}_\kappa)$ and passes it to $e$ as an argument. The function $\mathit{Acq}$ extracts the first component $\mathsf{acq}_\kappa$ of lock $x$, which is a tree constructor $\mathsf{acq}_\kappa$, and creates a node $\mathsf{acq}_\kappa$. Similarly the *Rel* rule creates a node $\mathsf{rel}_\kappa$. The last rule is used to stop the thread and generate the symbol $\bot$ meaning that the thread is alive. □

### 3.4 Lock Sensitivity of Action Trees

The set $\mathsf{RelaxedATrees}(p)$ may contain action trees for which there are no corresponding transition sequences that respect the synchronization constraint. To exclude them, we introduce a subset $\mathsf{LSATrees}$ of $\gamma$, which consists of only action trees that have corresponding transition sequences for *some* program (that satisfies scope safety and well-nested locking), so that the set $\mathsf{ATrees}(p)$ is represented by $\mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$. To this end, we introduce an *abstract* transition relation $\hat{c} \xrightarrow{\iota, a} \hat{c}'$, obtained by replacing expressions with action trees.

**Definition 6 (abstract configurations).** An *abstract thread state* is a quadruple $(\gamma, L, s, \sigma)$, obtained by replacing the first component of a thread state in Section 2 with an action tree $\gamma$. An *abstract configuration* $\hat{c}$ is a partial map from the set of thread identifiers to the set of abstract thread states. The transition relation on abstract configurations is defined by:

$$
\frac{}{\hat{c} \uplus \{\, \iota \mapsto (\gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \bullet} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L, s, \sigma) \,\}}
$$

$$
\frac{\sigma(\kappa) = (\iota, m) \qquad \sigma' = \sigma \{\, \kappa \mapsto (\iota, m+1) \,\}}{\hat{c} \uplus \{\, \iota \mapsto (\mathsf{new}_\kappa\, \gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{new}(\kappa, \iota, m+1)} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L, s, \sigma') \,\}}
$$

$$
\frac{\forall \iota', m. (\sigma(\kappa) = (\iota', m) \Rightarrow \iota \neq \iota') \qquad \sigma' = \sigma \{\, \kappa \mapsto (\iota, 1) \,\}}{\hat{c} \uplus \{\, \iota \mapsto (\mathsf{new}_\kappa\, \gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{new}(\kappa, \iota, 1)} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L, s, \sigma') \,\}}
$$

$$
\frac{\sigma(\kappa) = (\iota', m) \qquad (\kappa, \iota', m) \notin \mathbf{locked}(\hat{c} \uplus \{\, \iota \mapsto (\mathsf{acq}_\kappa\, \gamma, L, s, \sigma) \,\})}{\hat{c} \uplus \{\, \iota \mapsto (\mathsf{acq}_\kappa\, \gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{acq}(\kappa, \iota', m)} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L \cdot (\kappa, \iota', m), s, \sigma) \,\}}
$$

$$\frac{\sigma(\kappa) = (\iota', m) \qquad L = L' \cdot (\kappa, \iota', m)}{\hat{c} \uplus \{\, \iota \mapsto (\mathsf{rel}_\kappa\, \gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{rel}(\kappa, \iota', m)} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L', s, \sigma) \,\}}$$

$$\frac{}{\hat{c} \uplus \{\, \iota \mapsto (\mathsf{sp}\, \gamma_p\, \gamma_c, L, s, \sigma) \,\} \xrightarrow{\iota, \mathsf{sp}(\iota \cdot s)} \hat{c} \uplus \left\{ \begin{array}{ll} \iota & \mapsto (\gamma_p, L, s+1, \sigma), \\ \iota \cdot s & \mapsto (\gamma_c, \epsilon, 0, \sigma) \end{array} \right\}}$$

$$\frac{}{\hat{c} \uplus \{\, \iota \mapsto (\ell\, \gamma, L, s, \sigma) \,\} \xrightarrow{\iota, \ell} \hat{c} \uplus \{\, \iota \mapsto (\gamma, L, s, \sigma) \,\}}$$

$$\frac{L = \epsilon}{\hat{c} \uplus \{\, \iota \mapsto (\$, L, s, \sigma) \,\} \xrightarrow{\iota, \$} \hat{c}}$$

Here, $\mathbf{locked}(\hat{c})$ is defined similarly to that for (concrete) configurations, by

$$\mathbf{locked}(\hat{c}) = \{\, (\kappa, \iota', m) \mid \exists \iota.\ \hat{c}(\iota) = (\gamma, L, s, \sigma) \wedge (\kappa, \iota', m) \in L \,\}.$$

In the rules above for acquiring and releasing locks, we have added the condition $\sigma(\kappa) = (\iota', m)$, which captures the scope safety assumption.

Using the abstract transition relation, the set of lock sensitive action trees is defined as follows.

**Definition 7 (lock sensitivity).** An action tree $\gamma$ is *lock sensitive* if $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} \xrightarrow{*} \hat{\bot}$, where $\hat{\bot}$ is any abstract configuration such that $\hat{\bot}(\iota) = (\gamma, L, s, \sigma)$ implies $\gamma = \bot$. We write $\mathsf{LSATrees}$ for the set of lock-sensitive action trees.

**Theorem 3.** *Let $p$ be a scope-safe program. Then,*

$$\mathsf{ATrees}(p) = \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}.$$

Intuitively, the theorem above holds because the concrete transition system $c \xrightarrow{\iota, a}_p c'$ is obtained as the product of the relaxed transition system and the abstract transition system; see Appendix B.5 for a proof.

### 3.5 Regularity of LSATrees

We show that $\mathsf{LSATrees}$ is a regular tree language. To this end, we adapt the notion of an *acquisition structure* [1, 7] to deal with an unbounded number of locks. An acquisition structure is a summary of the usage of locks in an action tree.

Let us first review the idea behind acquisition structures. Let $\hat{c}_i = \{\, \iota_i \mapsto (\gamma_i, L_i, s_i, \sigma_i) \,\}$ $(i = 1, \ldots, n)$ be abstract configurations that are individually lock-sensitive, i.e. $\hat{c}_i \xrightarrow{*} \hat{\bot}_i$ for some bottom configuration $\hat{\bot}_i$ for each $i$. We would like to decide if the merged configuration is also lock-sensitive, i.e. whether $\biguplus_{i=1}^n \hat{c}_i \xrightarrow{*} \biguplus_{i=1}^n \hat{\bot}_i$. In some cases, it is obviously impossible.

– Let $\check{A^f}_i$ be the set of (concrete) locks that the final configuration $\hat{\perp}_i$ has. If $\check{A^f}_i \cap \check{A^f}_j \neq \emptyset$, the merged configuration is not lock-sensitive since $\biguplus_{i=1}^n \hat{\perp}_i$ violates the condition that each lock can be assigned to at most one thread.

– Let us call (an occurrence of) an acquire operation in the transition sequence $\pi : \hat{c}_i \xrightarrow{*} \hat{\perp}_i$ *final* if the lock is not released in the following subsequence. Let $G_\pi$ be the strict preorder[1] on concrete lock names defined by $(\nu, \nu') \in G_\pi$ just if an acquire operation of $\nu'$ appears after the final acquisition of $\nu$ in $\pi$. Let $\check{G}_i$ be the intersection of $G_\pi$ for all possible transition sequences $\pi : \hat{c}_i \xrightarrow{*} \hat{\perp}_i$. If $\bigcup_{i=1}^n \check{G}_i$ is cyclic, the merged configuration is not lock-sensitive. For example, if $(\nu, \nu'), (\nu', \nu) \in \bigcup_{i=1}^n \check{G}_i$ and $\pi : \biguplus_{i=1}^n \hat{c}_i \xrightarrow{*} \biguplus_{i=1}^n \hat{\perp}_i$, then the final acquisition of $\nu$ in $\pi$ must precede that of $\nu'$ and vice versa, a contradiction.

Conversely, provided that $L_i = \epsilon$ for all $i$, the above conditions are sufficient for the lock-sensitivity of the merged configuration. A transition sequence can be constructed by an eager scheduling as follows. If there is a thread whose next operation is not a final acquisition, run the thread. Furthermore if the thread acquires a lock, run it until the lock is released; then, by nested locking, the thread does not have any lock at that state. If all the threads reach $\perp$ or final acquisition operations, choose a thread acquiring a minimal lock with respect to $\bigcup_{i=1}^n \check{G}_i$. Since such a lock is ensured not to appear in the sequel, we can safely forget the lock and regard the thread as having no lock.

Unlike the previous work [1, 7], the number of locks is unbounded in our setting. However the above test is concerned only about the locks shared by $\hat{c}_i$ and $\hat{c}_j$ for some $i \neq j$. Thanks to the scope-safety of the program, the locks used by $\hat{c}_i$ are in $\{ (\kappa, \sigma(\kappa)) \mid \kappa \in \mathcal{K} \}$ or those that will be generated by $\hat{c}_i$ in the subsequent computation. Hence the restrictions of $\check{A^f}_i$ and $\check{G}_i$ to $\{ (\kappa, \sigma_i(\kappa)) \mid \kappa \in \mathcal{K} \}$, which is finite, is sufficient for the purpose. We represent those restrictions as sets and relations on abstract locks $\kappa \in \mathcal{K}$.

The formal definition of the acquisition structure of an action tree is as follows. It has additional fields: $A$ (the set of locks used in the action tree) is used to compute $G$ in an inductive way, and $R$ (the list of dangling release operations) and $T$ (the leaf node of this thread) are used to check if the locks are used in the expected manner (i.e. well-nested, no re-entrant and that a terminating thread have released all the locks). Given a relation $G$, let $G^+$ be its transitive closure and $G\restriction_A$ be the restriction $\{ (x, y) \in G \mid x, y \in A \}$. Given a set $A$, we write $A^{\circledast}$ for the set of all finite sequences on $A$ without repetition.

**Definition 8 (acquisition structure).** The *acquisition structure* $\mathsf{as}(\gamma)$ of an action tree $\gamma$ is a tuple $(A, A^f, R, T, G) \in \mathcal{P}(\mathcal{K}) \times \mathcal{P}(\mathcal{K}) \times \mathcal{K}^{\circledast} \times \{ \$, \perp \} \times \mathcal{P}(\mathcal{K} \times \mathcal{K})$, inductively defined as follows (where we write $A_\gamma$ for the first component of $\mathsf{as}(\gamma)$ and so on, and $\mathsf{undef}$ means undefined):

$$\mathsf{as}(\perp) = (\emptyset, \emptyset, \epsilon, \perp, \emptyset) \qquad \mathsf{as}(\$) = (\emptyset, \emptyset, \epsilon, \$, \emptyset) \qquad \mathsf{as}(\ell\,\gamma) = \mathsf{as}(\gamma)$$

[1] A strict preorder, often written as $<$, is an irreflexive and transitive relation.

$$\mathsf{as}(\mathsf{acq}_\kappa\,\gamma) = \begin{cases} (A_\gamma \cup \{\kappa\}, A^f_\gamma, R', T_\gamma, G_\gamma) \\ \qquad\qquad (\text{if } R_\gamma = R' \cdot \kappa) \\ (A_\gamma \cup \{\kappa\}, A^f_\gamma \cup \{\kappa\}, \epsilon, T_\gamma, (G_\gamma \cup (\{\,\kappa\,\} \times A_\gamma))^+) \\ \qquad\qquad (\text{if } R_\gamma = \epsilon \text{ and } T_\gamma = \bot \text{ and } G_\gamma \cup (\{\,\kappa\,\} \times A_\gamma) \text{ is acyclic}) \\ \mathsf{undef} \qquad (\text{otherwise}) \end{cases}$$

$$\mathsf{as}(\mathsf{rel}_\kappa\,\gamma) = \begin{cases} (A_\gamma, A^f_\gamma, R_\gamma \cdot \kappa, T_\gamma, G_\gamma) & (\text{if } \kappa \notin R_\gamma) \\ \mathsf{undef} & (\text{if } \kappa \in R_\gamma) \end{cases}$$

$$\mathsf{as}(\mathsf{sp}\,\gamma_p\,\gamma_c) = \begin{cases} (A_{\gamma_p} \cup A_{\gamma_c}, A^f_{\gamma_p} \cup A^f_{\gamma_c}, R_{\gamma_p}, T_{\gamma_p}, (G_{\gamma_p} \cup G_{\gamma_c})^+) \\ \qquad\qquad (\text{if } R_{\gamma_c} = \epsilon \text{ and } A^f_{\gamma_p} \cap A^f_{\gamma_c} = \emptyset \text{ and } G_{\gamma_p} \cup G_{\gamma_c} \text{ is acyclic}) \\ \mathsf{undef} \qquad (\text{if } R_{\gamma_c} \neq \epsilon \text{ or } A^f_{\gamma_p} \cap A^f_{\gamma_c} \neq \emptyset \text{ or } G_{\gamma_p} \cup G_{\gamma_c} \text{ is cyclic}) \end{cases}$$

$$\mathsf{as}(\mathsf{new}_\kappa\,\gamma) = \begin{cases} (A_\gamma \backslash \{\kappa\}, A^f_\gamma \backslash \{\kappa\}, R_\gamma, T_\gamma, G_\gamma \restriction_{\mathcal{K} \backslash \{\kappa\}}) & (\text{if } \kappa \notin R) \\ \mathsf{undef} & (\text{if } \kappa \in R). \end{cases}$$

For every action tree $\gamma$, no element $\kappa \in \mathcal{K}$ appears twice in the sequence $R_\gamma \in \mathcal{K}^*$ (if defined). Hence the set of all acquisition structures can be seen as finite.

The set of lock-sensitive action trees is characterized by the acquisition structure; see Appendix B.6 for a proof.

**Theorem 4.** *An action tree $\gamma$ is lock sensitive iff* $\mathsf{as}(\gamma) = (\emptyset, \emptyset, \epsilon, T, \emptyset)$.

By the definition of the acquisition structure, it can be obviously computed by a bottom-up tree automaton. Hence:

**Theorem 5.** $\mathsf{LSATrees}$ *is a regular tree language.*

Now we can reduce the pairwise-reachability problem to a higher-order model checking problem as follows. Let $p$ be a scope-safe program and $(\ell_1, \ell_2)$ be a pair of labels. Then $p \vDash \ell_1 \| \ell_2$ if and only if $\mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees} \cap R_{\ell_1,\ell_2} \neq \emptyset$ by Lemma 1 and Theorem 3. This is equivalent to $\mathsf{RelaxedATrees}(p) \not\subseteq \overline{\mathsf{LSATrees} \cap R_{\ell_1,\ell_2}}$. Since $\mathsf{LSATrees}$ is regular (Theorem 5), the right-hand-side is regular. Hence the problem is an instance of higher-order model checking, and thus decidable. This completes the proof of the main theorem, Theorem 1.

## 4 Checking Scope-Safety and Well-Nested Locking

The verification method for the pairwise reachability in the previous section is sound and complete for the class of scope-safe programs with nested locking. For programs outside the class, our verification method is *unsound*.[2] Thus, it is desirable to have methods for checking that a given program satisfies the conditions of scope-safety and well-nested locking. Fortunately, higher-order model checking can also be used for that purpose, as described below.

[2] Since the pairwise reachability is usually considered an undesirable behavior (e.g. a race), we say that a pairwise reachability analysis is sound when it does not miss the possibility of pairwise reachability.

### 4.1 Strong Scope Safety

For ease of explanation, we first present a method for checking a stronger condition called *strong scope-safety*. We call a program *strongly scope-safe* if it satisfies the conditions of Definition 3 where the transition relation $\rightarrow_p$ is replaced with the relaxed transition relation $\dashrightarrow_p$.

It would be quite easy to find a violation of strong scope-safety if one could construct a "concrete action tree", in which lock operations are annotated by concrete lock names, e.g. $\mathsf{new}_\kappa^{(\iota,m)}$. In this setting, what we should do is to check whether the "concrete action tree" has a node $op_\kappa^{(\iota,m)}$ ($op \in \{\,\mathsf{acq}, \mathsf{rel}\,\}$) whose nearest ancestor $\mathsf{new}_\kappa$-node is annotated with a different concrete name $(\iota', m')(\neq (\iota, m))$. Although the naive application of this idea seems to require infinitely many names, we can do this by using only two names because it suffices to ensure that two chosen concrete lock names are indeed different.

Given a transition sequence, and a subset $\mathcal{X}$ of concrete lock names, its *semi-concrete action tree* is an action tree in which lock operations are annotated with $A$ or $B$, e.g. $\mathsf{new}_\kappa^A$, where $A$ means that the concrete lock name is in $\mathcal{X}$ and $B$ otherwise. A semi-concrete action tree *violates scope-safety* if there is a node $op_\kappa^B$ ($op \in \{\,\mathsf{acq}, \mathsf{rel}\,\}$) whose nearest ancestor $\mathsf{new}_\kappa$ node is labeled with $A$ (i.e. it is $\mathsf{new}_\kappa^A$). This is a regular tree property, which we write as $\mathsf{S}$.

A HORS $\mathcal{G}_p^{A,B}$ generating the set of semi-concrete action trees of a program $p$ can be constructed in the same way as in Section 3.3, except that the behaviour of $New_\kappa$ is now nondeterministic as follows:

$$New_\kappa\, e = \mathsf{new}_\kappa^A\, (e\, (\mathsf{ack}_\kappa^A, \mathsf{rel}_\kappa^A)) \qquad New_\kappa\, e = \mathsf{new}_\kappa^B\, (e\, (\mathsf{ack}_\kappa^B, \mathsf{rel}_\kappa^B)).$$

Intuitively $New$ nondeterministically chooses if the newly created concrete lock name should belong to $\mathcal{X}$ or not.

By the discussion above, it should be clear that a program is strongly scope-safe if and only if $\mathcal{L}(\mathcal{G}_p^{A,B}) \cap \mathsf{S} = \emptyset$. Thus, the problem to check whether a given program is strongly scope-safe is decidable.

### 4.2 Well-nested Locking

Here we give a method for conservatively checking whether a given *scope-safe* program has nested locking, ignoring inter-thread synchronization.

We give a sketch of the construction of a top-down tree automaton, which nondeterministically chooses a thread of the action tree and computes acquired locks, and accepts the tree if the chosen thread indeed violates well-nested locking. A state is either $\star$ (meaning that the automaton has not chosen a thread) or a sequence $R \in (\mathcal{K} \cup \{\,\sharp\,\})^*$ (meaning that the chosen thread has acquired (and not released) the locks in the order specified in $R$) such that each $\kappa \in \mathcal{K}$ appears at most once in $R$. The symbol $\sharp$ is used to express locks that have been shadowed (i.e., those that are no longer visible due to the creation of a lock of the same abstract name). For example, if the current node is $\mathsf{new}(\kappa)$ and the state is $R \cdot \kappa \cdot R'$, then the automaton changes its state to $R \cdot \sharp \cdot R'$ and moves

to the child node. If the node is $\mathsf{acq}_\kappa$ and the state is $R$, then the automaton checks if $\kappa$ appears in $R$; if so, the automaton rejects the tree since this thread gets stuck (note that locks are non-reentrant) and does not violate well-nested locking; otherwise, it moves to the child node with the state $R \cdot \kappa$. If the node is $\mathsf{rel}(\kappa)$ and the state is $R \cdot \kappa$, then the automaton goes to the child node with the state $R$. If the automaton sees $\mathsf{rel}(\kappa)$ at the state $R \cdot \xi$ ($\xi \in \mathcal{K} \cup \{\sharp\}$) with $\kappa \neq \xi$, then it accepts the tree because this release operation violates well-nested locking. In the construction above, $R$ may contain an unbounded number of $\sharp$, so the number of states is infinite. However, only the right-most occurrence of $\sharp$ is meaningful and one can safely forget the other occurrences of $\sharp$. Thus the number of states can be reduced to finite.

Let us write $\mathsf{N}$ for the tree language accepted by the above automaton. The following result is obvious.

**Lemma 2.** *Let $p$ be a scope-safe program. If $\mathcal{L}(\mathcal{G}_p^{A,B}) \cap \mathsf{N} = \emptyset$, then $p$ has nested locking.*

### 4.3 Scope-safety and well-nested locking

We have seen above that $\mathcal{L}(\mathcal{G}_p^{A,B}) \cap (\mathsf{N} \cup \mathsf{S}) = \emptyset$ implies $p$ is a scope-safe program with nested locking. The converse does not hold, however. This is because even if $\mathcal{L}(\mathcal{G}_p^{A,B}) \cap (\mathsf{N} \cup \mathsf{S}) \neq \emptyset$, $\gamma \in \mathcal{L}(\mathcal{G}_p^{A,B}) \cap (\mathsf{N} \cup \mathsf{S})$ may be infeasible because of the synchronization through locks.

We can obtain a complete method by taking into account the lock-sensitivity of action trees, in a manner similar to Section 3.4. We call a (semi-concrete) action tree $\gamma$ *almost lock-sensitive* if it is obtained by adding an action to a lock-sensitive action tree (i.e. there exists a pair of a one-hole tree context $C$ and an action tree $\gamma'$ such that $\gamma = C[\gamma']$, $C[\bot]$ is lock-sensitive and $\gamma'$ has exactly one node whose label is not $\bot$). Let $\mathsf{LSATrees}'$ be the set of semi-concrete action trees that are almost lock-sensitive.

**Lemma 3.** *$(\mathcal{L}(\mathcal{G}_p^{A,B}) \cap \mathsf{LSATrees}') \cap (\mathsf{N} \cup \mathsf{S}) = \emptyset$ if and only if $p$ is a scope-safe program with nested locking.*

As a corollary, we obtain:

**Theorem 6.** *The problem to check whether a given program is scope-safe and has nested locking is decidable.*

## 5 Extension with Join Operations

We briefly discuss our method for pairwise reachability (described in Section 3) to support first-class thread identifiers and join operations. The target language is extended as follows. We introduce a new base type **ID** for thread IDs. Each spawn expression $\mathbf{spawn}(e_c)$; $e_p$ is now annotated with an *abstract thread ID* $\theta$, which does not affect the transition but is used to define the notion of scope

safety. The expression spawns a new child thread $e_c$, and executes $e_p(\iota)$, where $\iota$ is the (unique) identifier (ID) of the new thread; thus $e_p$ has type $\mathbf{ID} \to \star$. We add a new expression $\mathbf{join}(e_1); e_2$, which waits for the termination of the thread with ID $e_1$, and then executes $e_2$. A program of the extended language is *scope-safe* if, in addition to the condition on scope-safety on locks, it satisfies the analogous condition on thread identifiers, that each join operation may refer to only the newest thread identifier in the scope for each abstract thread ID $\theta$.

*Example 6.* The following program is a variation of the program in Example 1.

$$p_2 = \left\{ \begin{array}{ll} S = \mathbf{new}_i\ F & F\ x = \mathbf{spawn}^\theta(\mathbf{acq}(x);\ (\mathbf{rel}(x);\ \$)^\ell); G\ x) \\ G\ x\ t = \mathbf{acq}(x);\ (\mathbf{rel}(x);\ \mathbf{join}(t);\ S). & \end{array} \right\}$$

The function $F$ takes a lock as an argument, spawns a new thread, and passes its identifier to $G\ x$. The program is scope-safe. Unlike the program in Example 1, $(\ell, \ell)$ is *not* pairwise reachable, because the root thread waits for the termination of a child thread before spawning another thread.

Our pairwise reachability verification method in Section 3 can be smoothly extended, except for the regularity of the set of lock-sensitive action trees, which we briefly discuss below. Let $\hat{c}_1 = \{\, \iota \mapsto (\gamma, s+1, L, \sigma) \,\}$ and $\hat{c}'_1 = \{\, \iota \cdot s \mapsto (\gamma', 0, \epsilon, \sigma) \,\}$ be abstract configurations and assume that each of them is schedulable alone, i.e. $\hat{c}_1 \xrightarrow{*}_p \hat{\perp}_1$ and $\hat{c}_2 \xrightarrow{*}_p \hat{\perp}_2$. The question is when $\hat{c}_1 \uplus \hat{c}'_1$ is schedulable. If $\hat{c}_1$ does not do $\mathbf{join}(\iota \cdot s)$, the schedulability of $\hat{c}_1 \uplus \hat{c}'_1$ can be checked in the same way as in Section 3.5. If $\hat{c}_1$ does a $\mathbf{join}(\iota \cdot s)$ action, an additional condition is required for schedulability of $\hat{c}_1 \uplus \hat{c}'_1$: if $\nu \in L$ and $\nu$ will not be released until the join operator, then $\hat{c}_2$ cannot use this lock. Hence the additionally required piece of data is the set of pairs $(\iota', \nu')$ of a thread ID and a lock name such that $\nu'$ is kept locked from the current state until a $\mathbf{join}(\iota')$ action. Since only the thread IDs and lock names in the scope are relevant, this information can be described in finite states.

## 6 Experiments

We have implemented a tool for checking the pairwise reachability and strong scope safety based on our methods. The tool uses HorSat2 [3] as the backend higher-order model checker. We have tested the tool on a machine with an Intel Core i5 CPU with 2.5GHz and 16GB memory.

| Program | Reachability | SS | PR |
|---|---|---|---|
| `example1` | YES | 0.002 | 0.385 |
| `example2` | NO | 0.002 | 29.5 |
| `datarace` | NO | 0.004 | 1.04 |

The table on the righthand side shows the result of preliminary experiments. The column "Reachability" shows the answers for the pairwise reachability problems. The columns "SS" and "PR" respectively show the times spent for checking (strong) scope-safety and pairwise-reachability, measured in seconds. indicates the elapsed time of scope-safety checking and pairwise-reachability checking. The programs `example1` and `example2` are those given in Examples 1 and 6 respectively. The benchmark program `datarace` models the following C-like code:

```
main() {  r = newref();  l = newlock();
  spawn{acq(l);write(r);rel(l);}; acq(l);write(r);rel(l); main(); }
```

where the dynamic creation of reference cells is handled in a manner similar to that of locks and thread identifiers. We checked whether two write commands for the *same* reference cell may be reached simultaneously. All the programs are strongly scope-safe. According to the experimental results, the strong scope safety can be checked instantly. The pairwise reachability checking is slower, but reasonably fast, considering the complexity of higher-order model checking [8]. The pairwise reachability checking for `example2` took much longer than for the other programs. We think this is due to the use of the join primitive, which probably blowed up the space exploited by the model checker. This suggests that a further improvement of the higher-order model checker is required for handling real-world programs; we leave it for future work.

## 7 Related Work

There have been several studies on the decidability of pairwise reachability of concurrent programs with nested locking [1, 2, 7, 13]. To our knowledge, however, our result is the first one that allows dynamic creation of an unbounded number of locks, albeit under the condition of scope safety. The notion of scope safety is also new. The idea of reducing pairwise reachability to higher-order model checking has been first proposed by Yasukata et al. [13]; our method described in Section 3 is an extension of their method to deal with an unbounded number of locks. There are many other methods for analyzing concurrent programs with dynamic resource creation [4, 10, 12], but they are either incomplete or unsound (due to over- or under-approximation of reachable states).

The idea of non-deterministically tracking the usage of locks used in Section 4 (for checking scope safety and well-nested locking) has been inspired from Kobayashi's work for applying higher-order model checking to resource usage analysis [5]. His method is for sequential (functional) programs, however.

## 8 Conclusion

We have presented a method for deciding the pairwise reachability of concurrent programs with dynamic creation of resources and thread identifiers. We have introduced the notion of scope safety, and proved that our method is sound and complete for scope-safe programs with nested locking. We have also presented methods for checking whether a given program satisfies the conditions of scope safety and well-nested locking.

### Acknowledgment

We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP16K16004.

## References


1. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: Proceedings of VMCAI 2011. LNCS, vol. 6538, pp. 199–213 (2011)
2. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV. LNCS, vol. 3576, pp. 505–518. Springer (2005)
3. Kobayashi, N.: HorSat2: A saturation-based higher-order model checker. `http://www-kb.is.s.u-tokyo.ac.jp/~koba/horsat2/`
4. Kobayashi, N.: Type systems for concurrent programs. In: Proceedings of UNU/IIST 20th Anniversary Colloquium. LNCS, vol. 2757, pp. 439–453. Springer (2003)
5. Kobayashi, N.: Model checking higher-order programs. J. ACM 60(3), 20 (2013)
6. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI 2011. pp. 222–233 (2011)
7. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 525–539. Springer (2009)
8. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS. pp. 81–90 (2006)
9. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
10. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Practical static race detection for C. ACM Transactions on Programming Languages and Systems (TOPLAS) 33(1), Article 3 (Jan 2011)
11. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
12. Terauchi, T.: Checking race freedom via linear programming. In: Proceedings of PLDI '08. pp. 1–10. ACM, New York, NY, USA (2008)
13. Yasukata, K., Kobayashi, N., Matsuda, K.: Pairwise reachability analysis for higher order concurrent programs by higher-order model checking. In: CONCUR 2014, LNCS, vol. 8704, pp. 312–326 (2014)

# Appendix

## A Type System of the Target Language

This section defines a simple type system for the target language.

A *type environment* $\Gamma$ is a map from variables to types. A *type judgment relation* $\Gamma \vdash e : \tau$ for expressions is the least relation closed under the following rules:

$$\Gamma \vdash () : \star \qquad \text{(Term)}$$

$$\Gamma \cup \{\, x : \tau \,\} \vdash x : \tau \qquad \text{(Var)}$$

$$\Gamma \cup \{\, F : \tau \,\} \vdash F : \tau \qquad \text{(FVar)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(App)}$$

$$\frac{\Gamma \vdash e : \mathbf{lock} \to \star}{\Gamma \vdash \mathbf{new}_i\, e : \star} \qquad \text{(New)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{lock} \qquad \Gamma \vdash e_2 : \star}{\Gamma \vdash \mathbf{acq}(e_1);\ e_2 : \star} \qquad \text{(Acq)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{lock} \qquad \Gamma \vdash e_2 : \star}{\Gamma \vdash \mathbf{rel}(e_1);\ e_2 : \star} \qquad \text{(Rel)}$$

$$\frac{\Gamma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : \star}{\Gamma \vdash \mathbf{spawn}(e_c);\ e_p : \star} \qquad \text{(Spawn)}$$

$$\frac{\Gamma \vdash e : \star}{\Gamma \vdash e^{\ell} : \star} \qquad \text{(Label)}$$

A program $p = \{\, F_1\, \tilde{x}_1 = e_1, \ldots, F_n\, \tilde{x}_n = e_n \,\}$ is *well-typed* under $\Gamma$ if

$$\Gamma = \{\, F_i \mapsto \tau_{i1} \to \cdots \to \tau_{ik_i} \to \star \mid i = 1, \ldots, n \,\}$$

and $\Gamma \cup \{\, x_{i1} \mapsto \tau_{i1}, \ldots, x_{ik_i} \mapsto \tau_{ik_i} \,\} \vdash e_i : \star$ holds for every $i \in \{\, 1, \ldots, n \,\}$.

## B Supplementary Material for Section 3

Let $p$ be a program, fixed in this section.

### B.1 Sequences of actions

This section formally defines the notion of actions and introduces an important notion of independence, which we shall use in the proofs.

Recall that a thread ID $\iota$ is a sequence of natural numbers. We write $\iota \sqsubseteq \iota'$ if $\iota$ is a prefix of $\iota'$. Hence $\iota \sqsubseteq \iota'$ means the thread $\iota$ is an ancestor of the thread $\iota'$.

We formally define the notion of *actions*, which has been given implicitly as the set of labels of transitions. The set of *actions* is given by the grammar $a ::= \bullet \mid \$ \mid \ell \mid \mathsf{new}(\kappa, \iota, m) \mid \mathsf{acq}(\kappa, \iota, m) \mid \mathsf{rel}(\kappa, \iota, m) \mid \mathsf{sp}(\iota)$. We use $a$ as a metavariable of actions. An *action with the thread ID* is a pair $(\iota, a)$ of a thread ID and an action of the form

$$(\iota, \bullet), \quad (\iota, \$), \quad (\iota, \ell),$$
$$(\iota, \mathsf{new}(\kappa, \iota', m)), \quad (\iota, \mathsf{acq}(\kappa, \iota', m)), \quad (\iota, \mathsf{rel}(\kappa, \iota', m)), \quad (\iota, \mathsf{sp}(\iota \cdot s))$$

where $s$ is a natural number and $\iota'$ is an arbitrary thread ID. We use $\xi$ and $\zeta$ as metavariables of actions with the thread IDs. An action with the thread ID is often simply called an action.

Let $\boldsymbol{\xi}$ be a sequence of actions. We write $\boldsymbol{\xi}\restriction_{\iota \cdot s}$ for the subsequence of $\boldsymbol{\xi}$ consisting actions $(\iota', a)$ of children of the thread $\iota \cdot s$ (i.e. $\iota \cdot s \sqsubseteq \iota'$). The sequence $\boldsymbol{\xi}\restriction_{-(\iota \cdot s)}$ is defined as the subsequence of $\boldsymbol{\xi}$ obtained by removing actions of children of $\iota \cdot s$. We say $\boldsymbol{\xi}$ *$\bullet$-free* if it does not contain an action of the form $(\iota, \bullet)$. We define $\boldsymbol{\xi}\restriction_{-\bullet}$ as the subsequence of $\boldsymbol{\xi}$ obtained by removing $\bullet$-actions.

**Definition 9 (Independent action).** Let $\xi = (\iota, a)$ and $\xi' = (\iota', a')$. We say that $\xi$ and $\xi'$ *dependent* if either (1) $\iota = \iota'$, (2) $a = \mathsf{sp}(\iota')$, or (3) $a' = \mathsf{sp}(\iota)$. They are *independent* if they are not dependent. If $\xi$ and $\xi'$ are independent actions, we write

$$\boldsymbol{\zeta} \cdot \xi \cdot \xi' \cdot \boldsymbol{\zeta}' \sim_1 \boldsymbol{\zeta} \cdot \xi' \cdot \xi \cdot \boldsymbol{\zeta}'$$

for every sequences $\boldsymbol{\zeta}$ and $\boldsymbol{\zeta}'$ of actions. We define $\sim$ as the symmetric and transitive closure of $\sim_1$, which is an equivalence relation. Then $\boldsymbol{\zeta} \sim \boldsymbol{\zeta}'$ means that $\boldsymbol{\zeta}'$ is obtained from $\boldsymbol{\zeta}$ by swapping consecutive independent actions several times.

The next lemma says that a swapping of actions in a $\bullet$-free subsequence can be lifted to the original sequence.

**Lemma 4.** *Let $\boldsymbol{\xi}'$ and $\boldsymbol{\zeta}'$ be $\bullet$-free sequences of actions and $\boldsymbol{\xi}$ be a sequence of actions such that $\boldsymbol{\xi}\restriction_{-\bullet} = \boldsymbol{\xi}'$. If $\boldsymbol{\xi}' \sim \boldsymbol{\zeta}'$, then there exists $\boldsymbol{\zeta}$ such that $\boldsymbol{\xi} \sim \boldsymbol{\zeta}$ and $\boldsymbol{\zeta}\restriction_{-\bullet} = \boldsymbol{\zeta}'$.*

*Proof.* It suffices to prove the result when $\boldsymbol{\xi}' \sim_1 \boldsymbol{\zeta}'$. Then

$$\boldsymbol{\xi}' = \xi'_1 \cdot \dots \cdot \xi'_i \cdot \xi'_{i+1} \cdot \dots \cdot \xi'_{n'}$$
$$\boldsymbol{\zeta}' = \xi'_1 \cdot \dots \cdot \xi'_{i+1} \cdot \xi'_i \cdot \dots \cdot \xi'_{n'}$$

where $\xi'_i$ and $\xi'_{i+1}$ are independent actions.

Let

$$\boldsymbol{\xi} = \xi_1 \cdot \cdots \cdot \xi_j \cdot \xi_{j+1} \cdot \cdots \cdot \xi_{j+k-1} \cdot \xi_{j+k} \cdot \cdots \cdot \xi_n$$

where $\xi_j$ is the action corresponding to $\xi'_i$ and $\xi_{j+k}$ to $\xi'_{i+1}$. Since $\boldsymbol{\xi}\restriction_{-\bullet} = \boldsymbol{\xi}'$, we know that $\xi_{j+1} \cdot \cdots \cdot \xi_{j+k-1}$ is a sequence of $\bullet$-actions. Let $\iota$ be the thread ID of action $\xi_{j+k} = \xi'_{i+1}$. Let $\boldsymbol{\xi}''$ be a sequence obtained by moving every occurrence of $(\iota, \bullet)$ in $\xi_{j+1} \cdot \cdots \cdot \xi_{j+k-1}$ to a position before $\xi_j$ and every occurrence of other $\bullet$-actions to a position after $\xi_{j+k}$. It is easy to show that $\boldsymbol{\xi} \sim \boldsymbol{\xi}''$ and $\boldsymbol{\xi}''\restriction_{-\bullet} = \boldsymbol{\xi}'$.

Now we can assume without loss of generality that the actions corresponding to $\xi'_i \cdot \xi'_{i+1}$ are also consecutive in $\boldsymbol{\xi}$. We obtain a desired sequence $\zeta$ by commuting them. □

### B.2 Action tree representation of a transition sequence

This subsection gives a formal definition of the action tree of a given transition sequence and discuss its properties.

Given a sequence $\boldsymbol{\xi}$ of actions, its *action tree* $\mathsf{a}(\boldsymbol{\xi})$ is defined by induction on $\boldsymbol{\xi}$ as follows.

$$\begin{aligned}
\mathsf{a}(\epsilon) &:= \bot \\
\mathsf{a}((\iota, \bullet) \cdot \boldsymbol{\xi}) &:= \mathsf{a}(\boldsymbol{\xi}) \\
\mathsf{a}((\iota, \$) \cdot \boldsymbol{\xi}) &:= \$ && (\text{if } \boldsymbol{\xi} = \epsilon) \\
\mathsf{a}((\iota, \ell) \cdot \boldsymbol{\xi}) &:= \ell\,(\mathsf{a}(\boldsymbol{\xi})) \\
\mathsf{a}((\iota, \mathsf{new}(\kappa, \iota', m)) \cdot \boldsymbol{\xi}) &:= \mathsf{new}_\kappa\,(\mathsf{a}(\boldsymbol{\xi})) \\
\mathsf{a}((\iota, \mathsf{acq}(\kappa, \iota', m)) \cdot \boldsymbol{\xi}) &:= \mathsf{acq}_\kappa\,(\mathsf{a}(\boldsymbol{\xi})) \\
\mathsf{a}((\iota, \mathsf{rel}(\kappa, \iota', m)) \cdot \boldsymbol{\xi}) &:= \mathsf{rel}_\kappa\,(\mathsf{a}(\boldsymbol{\xi})) \\
\mathsf{a}((\iota, \mathsf{sp}(\iota \cdot s)) \cdot \boldsymbol{\xi}) &:= \mathsf{sp}\ \left(\mathsf{a}(\boldsymbol{\xi}\restriction_{-(\iota \cdot s)})\right)\ (\mathsf{a}(\boldsymbol{\xi}\restriction_{\iota \cdot s}))\,.
\end{aligned}$$

The function $\mathsf{a}$ is a partial function. So we have to ensure that it is defined on sequences of actions of transition sequences. We gives a sufficient condition for well-definedness of $\mathsf{a}(\boldsymbol{\xi})$.

**Definition 10.** Let $\boldsymbol{\xi}$ be a sequence of actions and $\psi$ be a partial function from thread IDs to natural numbers. We define the relation $\psi \rhd \boldsymbol{\xi}$ by induction on $\boldsymbol{\xi}$ as follows.

$$\begin{aligned}
\psi \rhd (\iota, \bullet) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \$) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi\{\iota \mapsto \text{undef}\} \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \ell) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \mathsf{new}(\kappa)) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \mathsf{acq}(\kappa)) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \mathsf{rel}(\kappa)) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } \psi \rhd \boldsymbol{\xi} \\
\psi \rhd (\iota, \mathsf{sp}(\iota \cdot s)) \cdot \boldsymbol{\xi} &\text{ iff } \iota \in \mathit{dom}(\psi) \text{ and } s = \psi\iota \text{ and } \psi\{\iota \mapsto s+1, \iota \cdot s \mapsto 0\} \rhd \boldsymbol{\xi}
\end{aligned}$$

(We define $\iota \notin \mathit{dom}(\psi\{\iota \mapsto \text{undef}\})$ for every $\psi$.)

Basically this tracks the live thread IDs and checks if every operation is made by a live thread. It is clear that every sequence of actions obtained from a transition sequence satisfies this relation for the appropriate $\psi$ describing the live threads in the first configuration.

**Lemma 5.** *If* $\{\iota \mapsto s\} \rhd \boldsymbol{\xi}$*, then* $\mathsf{a}(\boldsymbol{\xi})$ *is defined*

*Proof.* Easy induction on the length of $\boldsymbol{\xi}$, using the fact that $\{\iota \mapsto s, \iota' \mapsto s'\} \rhd \boldsymbol{\xi}$ implies $\{\iota \mapsto s\} \rhd \boldsymbol{\xi}\restriction_{-\iota'}$ and $\{\iota' \mapsto s'\} \rhd \boldsymbol{\xi}\restriction_{\iota'}$. □

We write $\xi \overset{\text{ab}}{=} \xi'$ if they are equivalent except for the concrete lock names. Formally $\xi \overset{\text{ab}}{=} \xi'$ is defined by the following rules:

$$\xi \overset{\text{ab}}{=} \xi$$
$$(\iota, \mathsf{new}(\kappa, \iota', m')) \overset{\text{ab}}{=} (\iota, \mathsf{new}(\kappa, \iota'', m''))$$
$$(\iota, \mathsf{acq}(\kappa, \iota', m')) \overset{\text{ab}}{=} (\iota, \mathsf{acq}(\kappa, \iota'', m''))$$
$$(\iota, \mathsf{rel}(\kappa, \iota', m')) \overset{\text{ab}}{=} (\iota, \mathsf{rel}(\kappa, \iota'', m''))$$

This relation can be extended to sequences in the obvious way. We write $\sim\!\!\overset{\text{ab}}{=}$ for the composite of $\sim$ and $\overset{\text{ab}}{=}$ (i.e. $\boldsymbol{\xi} \sim\!\!\overset{\text{ab}}{=} \boldsymbol{\xi}'$ if and only if $\exists \boldsymbol{\zeta}.\ \boldsymbol{\xi} \sim \boldsymbol{\zeta} \overset{\text{ab}}{=} \boldsymbol{\xi}'$).

**Lemma 6.** $\sim\!\!\overset{\text{ab}}{=}$ *is an equivalence relation. Furthermore* $\boldsymbol{\xi}_i \sim\!\!\overset{\text{ab}}{=} \boldsymbol{\xi}'_i$ *for* $i = 1, 2$ *implies* $\boldsymbol{\xi}_1 \cdot \boldsymbol{\xi}_2 \sim\!\!\overset{\text{ab}}{=} \boldsymbol{\xi}'_1 \cdot \boldsymbol{\xi}'_2$.

*Proof.* It is an easy consequence of the fact that $\xi_1 \cdot \dots \cdot \xi_n \overset{\text{ab}}{=} \xi'_1 \cdot \dots \cdot \xi'_n$ and $\xi_i$ and $\xi_{i+1}$ are independent implies $\xi'_i$ and $\xi'_{i+1}$ are independent. □

Roughly speaking, an action tree corresponds to an equivalence class of $\sim\!\!\overset{\text{ab}}{=}$. We formally state and prove this observation.

**Lemma 7.** *Let* $\boldsymbol{\xi}$ *and* $\boldsymbol{\zeta}$ *be sequences of actions such that* $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\xi}$ *and* $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\zeta}$*. Suppose that* $\boldsymbol{\xi}$ *and* $\boldsymbol{\zeta}$ *are* $\bullet$*-free. Then* $\mathsf{a}(\boldsymbol{\xi}) = \mathsf{a}(\boldsymbol{\zeta})$ *if and only if* $\boldsymbol{\xi} \sim\!\!\overset{\text{ab}}{=} \boldsymbol{\zeta}$.

*Proof.* Given an action tree $\gamma$ and a pair of a thread ID $\iota$ and a natural number $s$, we define the *canonical action sequence* $\mathsf{c}_{\iota,s}(\gamma)$ by induction on $\gamma$ as follows:

$$\begin{aligned}
\mathsf{c}_{\iota,s}(\bot) &= \epsilon \\
\mathsf{c}_{\iota,s}(\$) &= (\iota, \$) \\
\mathsf{c}_{\iota,s}(\ell\, \gamma) &= (\iota, \ell) \cdot \mathsf{c}_{\iota,s}(\gamma) \\
\mathsf{c}_{\iota,s}(\mathsf{new}_\kappa\, \gamma) &= (\iota, \mathsf{new}(\kappa, \epsilon, 1)) \cdot \mathsf{c}_{\iota,s}(\gamma) \\
\mathsf{c}_{\iota,s}(\mathsf{acq}_\kappa\, \gamma) &= (\iota, \mathsf{acq}(\kappa, \epsilon, 1)) \cdot \mathsf{c}_{\iota,s}(\gamma) \\
\mathsf{c}_{\iota,s}(\mathsf{rel}_\kappa\, \gamma) &= (\iota, \mathsf{rel}(\kappa, \epsilon, 1)) \cdot \mathsf{c}_{\iota,s}(\gamma) \\
\mathsf{c}_{\iota,s}(\mathsf{sp}\, \gamma_p\, \gamma_c) &= (\iota, \mathsf{sp}(\iota \cdot s)) \cdot \mathsf{c}_{\iota,s+1}(\gamma_p) \cdot \mathsf{c}_{\iota \cdot s,0}(\gamma_c).
\end{aligned}$$

Here $(\kappa, \epsilon, 1)$ is a dummy concrete lock name.

We prove that $\boldsymbol{\xi} \sim^{\mathrm{ab}}_{=} \mathsf{c}_{\iota,s}(\mathsf{a}(\boldsymbol{\xi}))$ by induction on the length of $\xi$. The only non-trivial case is that $\boldsymbol{\xi} = (\iota, \mathsf{sp}(\iota')) \cdot \boldsymbol{\xi}''$. Since $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\xi}$, we know that $\iota' = \iota \cdot s$. In this case,

$$\mathsf{a}(\boldsymbol{\xi}) = \mathsf{sp}\,(\mathsf{a}(\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}))\,(\mathsf{a}(\boldsymbol{\xi}''\restriction_{\iota\cdot s})).$$

Since $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\xi}$, the action $(\iota, \mathsf{sp}(\iota \cdot s))$ has at most one occurrence in $\boldsymbol{\xi}$ and thus $\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}$ does not have this action. Hence every action in $\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}$ is independent of every action in $\boldsymbol{\xi}''\restriction_{\iota\cdot s}$. As a result

$$\boldsymbol{\xi} \;\sim\; (\iota, \mathsf{sp}(\iota \cdot s)) \cdot (\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}) \cdot (\boldsymbol{\xi}''\restriction_{\iota\cdot s}).$$

By the induction hypothesis, we have $\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)} \;\sim^{\mathrm{ab}}_{=}\; \mathsf{c}_{\iota,s+1}(\mathsf{a}(\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}))$ and $\boldsymbol{\xi}''\restriction_{\iota\cdot s} \sim^{\mathrm{ab}}_{=} \mathsf{c}_{\iota\cdot s,0}(\mathsf{a}(\boldsymbol{\xi}''\restriction_{\iota\cdot s}))$. By Lemma 6, we have

$$\begin{aligned}
\boldsymbol{\xi} &\sim (\iota, \mathsf{sp}(\iota \cdot s)) \cdot (\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}) \cdot (\boldsymbol{\xi}''\restriction_{\iota\cdot s}) \\
&\sim^{\mathrm{ab}}_{=} (\iota, \mathsf{sp}(\iota \cdot s)) \cdot \mathsf{c}_{\iota,s+1}(\mathsf{a}(\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)})) \cdot \mathsf{c}_{\iota\cdot s,0}(\mathsf{a}(\boldsymbol{\xi}''\restriction_{\iota\cdot s})) \\
&= \mathsf{c}_{\iota,s}(\mathsf{sp}\,(\mathsf{a}(\boldsymbol{\xi}''\restriction_{-(\iota\cdot s)}))\,(\mathsf{a}(\boldsymbol{\xi}''\restriction_{\iota\cdot s}))) \\
&= \mathsf{c}_{\iota,s}(\mathsf{a}((\iota, \mathsf{sp}(\iota \cdot s)) \cdot \boldsymbol{\xi}'')),
\end{aligned}$$

which implies $\boldsymbol{\xi} \sim \mathsf{c}_{\iota,s}(\mathsf{a}(\boldsymbol{\xi}))$.

The claim of the lemma follows from

$$\boldsymbol{\xi} \sim^{\mathrm{ab}}_{=} \mathsf{c}_{\iota,s}(\mathsf{a}(\boldsymbol{\xi})) \sim^{\mathrm{ab}}_{=} \mathsf{c}_{\iota,s}(\mathsf{a}(\boldsymbol{\zeta})) \sim^{\mathrm{ab}}_{=} \boldsymbol{\zeta}$$

because $\sim^{\mathrm{ab}}_{=}$ is an equivalence relation (Lemma 6). □

Because $\mathsf{a}(\boldsymbol{\xi}) = \mathsf{a}(\boldsymbol{\xi}\restriction_{-\bullet})$, we obtain the following result as a corollary.

**Corollary 1.** *Let $\boldsymbol{\xi}$ and $\boldsymbol{\zeta}$ be sequences of actions $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\xi}$ and $\{\, \iota \mapsto s \,\} \rhd \boldsymbol{\zeta}$. Then $\mathsf{a}(\boldsymbol{\xi}) = \mathsf{a}(\boldsymbol{\zeta})$ if and only if $\boldsymbol{\xi}\restriction_{-\bullet} \sim^{\mathrm{ab}}_{=} \boldsymbol{\zeta}\restriction_{-\bullet}$.*

### B.3 Relaxed semantics

This subsection defines the relaxed semantics. A *state in the relaxed semantics* (or simply a *state*) is a tuple of an expression, a natural number and a partial function from $\mathcal{K}$ to $\mathbb{N}^* \times \mathbb{N}_+$. A *configuration in the relaxed semantics* (or simply a *configuration*) is a finite partial map from thread IDs to states (in the relaxed semantics). We use $d$ for configurations in the relaxed semantics. The transition rules are listed as follows.

$$\frac{F\,\tilde{x} = e' \in p}{d \uplus \{\, \iota \mapsto (F\,\widetilde{e}, s, \sigma) \,\} \xrightarrow{\iota,\bullet}_p d \uplus \{\, \iota \mapsto ([\widetilde{e}/\tilde{x}]e', s, \sigma) \,\}}$$

$$\frac{\sigma(\kappa) = (\iota, m) \qquad \sigma' = \sigma\,\{\,\kappa \mapsto (\iota, m+1)\,\}}{d \uplus \{\,\iota \mapsto (\mathbf{new}_\kappa\, e, s, \sigma)\,\} \overset{\iota,\mathsf{new}(\kappa,\iota,m+1)}{\dashrightarrow_p} d \uplus \{\,\iota \mapsto (e\ (\kappa, \iota, m+1), s, \sigma')\,\}}$$

$$\frac{\forall \iota', m.(\sigma(\kappa) = (\iota', m) \Rightarrow \iota \neq \iota') \qquad \sigma' = \sigma\,\{\,\kappa \mapsto (\iota, 1)\,\}}{d \uplus \{\,\iota \mapsto (\mathbf{new}_\kappa\, e, s, \sigma)\,\} \overset{\iota,\mathsf{new}(\kappa,\iota,1)}{\dashrightarrow_p} d \uplus \{\,\iota \mapsto (e\ (\kappa, \iota, 1), s, \sigma')\,\}}$$

$$\frac{}{d \uplus \{\,\iota \mapsto (\mathbf{acq}(\kappa, \iota', m); e, s, \sigma)\,\} \overset{\iota,\mathsf{acq}(\kappa,\iota',m)}{\dashrightarrow_p} d \uplus \{\,\iota \mapsto (e, s, \sigma)\,\}}$$

$$\frac{}{d \uplus \{\,\iota \mapsto (\mathbf{rel}(\kappa, \iota', m); e, s, \sigma)\,\} \overset{\iota,\mathsf{rel}(\kappa,\iota',m)}{\dashrightarrow_p} c \uplus \{\,\iota \mapsto (e, s, \sigma)\,\}}$$

$$\frac{}{d \uplus \{\,\iota \mapsto (\mathbf{spawn}(e_c); e_p, s, \sigma)\,\} \overset{\iota,\mathsf{sp}(\iota\cdot s)}{\dashrightarrow_p} d \uplus \left\{ \begin{array}{ll} \iota & \mapsto (e_p, s+1, \sigma), \\ \iota \cdot s & \mapsto (e_c, 0, \sigma) \end{array} \right\}}$$

$$\frac{}{d \uplus \{\,\iota \mapsto (e^\ell, s, \sigma)\,\} \overset{\iota,\ell}{\dashrightarrow_p} d \uplus \{\,\iota \mapsto (e, s, \sigma)\,\}}$$

$$\frac{}{d \uplus \{\,\iota \mapsto ((\,), s, \sigma)\,\} \overset{\iota,\$}{\dashrightarrow_p} d}$$

An important property of the relaxed semantics is that the set of sequences of actions are closed under $\sim$.

**Lemma 8.** *Let $d$ be any configuration and assume that $d \overset{\boldsymbol{\xi}}{\dashrightarrow_p} d'$. Then, for every $\boldsymbol{\xi}'$ such that $\boldsymbol{\xi} \sim \boldsymbol{\xi}'$, one has $d \overset{\boldsymbol{\xi}'}{\dashrightarrow_p} d'$.*

*Proof.* It suffices to prove the following claim:

> Let $d_1$ be a configuration and $\xi_1$ and $\xi_2$ be independent actions. If $d_1 \overset{\xi_1}{\dashrightarrow_p} d_2 \overset{\xi_2}{\dashrightarrow_p} d_3$, then $d_1 \overset{\xi_2}{\dashrightarrow_p} d_2' \overset{\xi_1}{\dashrightarrow_p} d_3$.

Let $(\iota_i, a_i) = \xi_i$ $(i = 1, 2)$. Then $\iota_1 \neq \iota_2$ and those threads are in $d_1$ by independence of $\xi_1$ and $\xi_2$. Since the relaxed semantics does not synchronize two threads at all, we can commute those actions resulting in the required transition sequence. □

### B.4 Recursion scheme generating RelaxedATrees($p$)

We define a non-deterministic HORS $\mathcal{G}_p = (\Sigma, \mathcal{N}, \mathcal{R}, S)$. Although the HORS constructed in the main text has pairs as a primitive, here we give a HORS without pairs; the pair constructs are removed by using the standard encoding.

The alphabet is the same as the labels for nodes of an action trees, given by:

$$\begin{aligned}\Sigma \;=\; & \{\, \mathsf{new}_\kappa \mapsto 1, \mathsf{acq}_\kappa \mapsto 1, \mathsf{rel}_\kappa \mapsto 1 \mid \kappa \in \mathcal{K} \,\} \\ & \cup\, \{\, \bot \mapsto 0, \$ \mapsto 0, \mathsf{sp} \mapsto 2 \,\} \\ & \cup\, \{\, \ell \mapsto 1 \mid \ell \in \mathsf{Label} \,\}\,.\end{aligned}$$

Given a type $\tau$ of the language, the type $\tau^\dagger$ for the recursion schemes is defined as follows:

$$\begin{aligned}\star^\dagger &= \mathsf{o} \\ \mathbf{lock}^\dagger &= (\mathsf{o} \to \mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o} \\ (\tau_1 \to \tau_2)^\dagger &= \tau_1^\dagger \to \tau_2^\dagger.\end{aligned}$$

The type $\mathbf{lock}^\dagger$ takes (the Church encoding of) boolean $\mathsf{o} \to \mathsf{o} \to \mathsf{o}$ (of which typical inhabitants are $\lambda xy.x$ and $\lambda xy.y$) and returns an operation $\mathsf{o} \to \mathsf{o}$, which is typically either $\mathsf{acq}_\kappa$ or $\mathsf{rel}_\kappa$ in our translation.

A function symbol of the recursion scheme represents a function symbol of the program, a lock operation or an operation about the pair constructs.

$$\begin{aligned}\mathcal{N} \;=\; & \{\, F \;\mapsto\; \tau^\dagger \mid F \text{ is a function symbol of } p \text{ typed with } \tau \,\} \\ & \cup\, \{\, \mathit{New}_\kappa \;\mapsto\; (\mathbf{lock}^\dagger \to \mathsf{o}) \to \mathsf{o} \mid \kappa \in \mathcal{K} \,\} \\ & \cup\, \{\, \mathit{Acq} \;\mapsto\; \mathbf{lock}^\dagger \to \mathsf{o} \to \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{Rel} \;\mapsto\; \mathbf{lock}^\dagger \to \mathsf{o} \to \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{Spawn} \;\mapsto\; \mathsf{o} \to \mathsf{o} \to \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{End} \;\mapsto\; \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{Label}_\ell \;\mapsto\; \mathsf{o} \to \mathsf{o} \mid \ell \in \mathsf{Label} \,\} \\ & \cup\, \{\, \mathit{Pair} \;\mapsto\; (\mathsf{o} \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{Fst} \;\mapsto\; \mathsf{o} \to \mathsf{o} \to \mathsf{o} \,\} \\ & \cup\, \{\, \mathit{Snd} \;\mapsto\; \mathsf{o} \to \mathsf{o} \to \mathsf{o} \,\}\end{aligned}$$

By using these function symbols, we can transform an expression of the language into a term of the recursion scheme. The transformation is given by:

$$\begin{aligned}(\,)^\dagger &= \mathit{End} \\ x^\dagger &= x \\ F^\dagger &= F \\ (e_1\, e_2)^\dagger &= e_1^\dagger\, e_2^\dagger\end{aligned}$$

$$
\begin{aligned}
(\mathbf{new}_\kappa\, e)^\dagger &= \mathit{New}\, e^\dagger \\
(\mathbf{acq}(e_1);\ e_2)^\dagger &= \mathit{Acq}\, e_1^\dagger\, e_2^\dagger \\
(\mathbf{rel}(e_1);\ e_2)^\dagger &= \mathit{Rel}\, e_1^\dagger\, e_2^\dagger \\
(\mathbf{spawn}(e_c);\ e_p)^\dagger &= \mathit{Spawn}\, e_p^\dagger\, e_c^\dagger \\
(e^\ell)^\dagger &= \mathit{Label}_\ell\, e^\dagger \\
((\kappa, \iota, m))^\dagger &= \mathit{Pair}\, \mathsf{acq}_\kappa\, \mathsf{rel}_\kappa.
\end{aligned}
$$

The rewriting rules are given as follows:

$$
\begin{aligned}
\mathcal{R} \;=\; & \left\{ \begin{array}{l} F\,\tilde{x} = e^\dagger \\ F\,\tilde{x} = \bot \end{array} \;\middle|\; (F\,\tilde{x} = e) \in p \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{New}_\kappa\, x = \mathsf{new}_\kappa\, (x\, (\mathit{Pair}\, \mathsf{acq}_\kappa\, \mathsf{rel}_\kappa)) \\ \mathit{New}_\kappa\, x = \bot \end{array} \;\middle|\; \kappa \in \mathcal{K} \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{Acq}\, l\, x = l\, \mathit{Fst}\, x \\ \mathit{Acq}\, l\, x = \bot \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{Rel}\, l\, x = l\, \mathit{Snd}\, x \\ \mathit{Rel}\, l\, x = \bot \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{Spawn}\, x_p\, x_c = \mathsf{sp}\, x_p\, x_c \\ \mathit{Spawn}\, x_p\, x_c = \bot \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{End} = \$ \\ \mathit{End} = \bot \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \mathit{Label}_\ell\, x = \ell\, x \\ \mathit{Label}_\ell\, x = \bot \end{array} \;\middle|\; \ell \in \mathsf{Label} \right\} \\
& \cup \{\, \mathit{Pair}\ \mathit{acq}\ \mathit{rel}\ f\ x = f\, (\mathit{acq}\, x)\, (\mathit{rel}\, x) \,\} \\
& \cup \{\, \mathit{Fst}\, x\, y = x \,\} \\
& \cup \{\, \mathit{Snd}\, x\, y = y \,\}
\end{aligned}
$$

Given a term $t$ of type $\mathsf{o}$, we define $\mathcal{L}(t, \mathcal{G}_p)$ as the language generated by $\mathcal{G}_p$ using $t$ as the initial term.

**Lemma 9.** *If* $\{\, \iota \mapsto (e, s, \sigma) \,\} \xrightarrow{\boldsymbol{\xi}}_p c$ *for some* $\iota$, $s$, $\sigma$ *and* $c$, *then* $\mathsf{a}(\boldsymbol{\xi}) \in \mathcal{L}(e^\dagger, \mathcal{G}_p)$.

*Proof.* By easy induction on the length of $\xi$.

Consider the case that $\boldsymbol{\xi} = (\iota, \mathsf{sp}(\iota \cdot s)) \cdot \boldsymbol{\xi}'$. Then $e = \mathbf{spawn}(e_c);\ e_p$ and $e^\dagger = \mathit{Spawn}\, e_p^\dagger\, e_c^\dagger$. Then $\mathsf{a}(\boldsymbol{\xi}) = \mathsf{sp}\, (\mathsf{a}(\boldsymbol{\xi}' \restriction_{-(\iota \cdot s)}))\, (\mathsf{a}(\boldsymbol{\xi}' \restriction_{\iota \cdot s}))$ by the definition of the action tree. Since $\boldsymbol{\xi}' \sim (\boldsymbol{\xi}' \restriction_{-(\iota \cdot s)}) \cdot (\boldsymbol{\xi}' \restriction_{\iota \cdot s})$, we have

$$
\{\, \iota \mapsto (e_p, s+1, \sigma),\ \iota \cdot s \mapsto (e_c, 0, \sigma) \,\} \xrightarrow{(\boldsymbol{\xi}' \restriction_{-(\iota \cdot s)}) \cdot (\boldsymbol{\xi}' \restriction_{\iota \cdot s})}_p c
$$

by Lemma 8. This can be decomposed into

$$
\{\, \iota \mapsto (e_p, s+1, \sigma) \,\} \xrightarrow{\boldsymbol{\xi}' \restriction_{-(\iota \cdot s)}}_p c_p
$$

and

$$\{\, \iota \mapsto (e_c, 0, \sigma) \,\} \overset{\boldsymbol{\xi}'\restriction_{\iota\cdot s}}{\dashrightarrow}_p c_c$$

with $c = c_p \uplus c_c$. By the induction hypothesis, $\mathsf{a}(\boldsymbol{\xi}'\restriction_{-(\iota\cdot s)}) \in \mathcal{L}(e_p^\dagger, \mathcal{G}_p)$ and $\mathsf{a}(\boldsymbol{\xi}'\restriction_{\iota\cdot s}) \in \mathcal{L}(e_c^\dagger, \mathcal{G}_p)$. Now we have

$$e^\dagger \;=\; \mathit{Spawn}\, e_p^\dagger\, e_c^\dagger \;\longrightarrow\; \mathsf{sp}\, e_p^\dagger\, e_c^\dagger \;\longrightarrow^*\; \mathsf{sp}\,(\mathsf{a}(\boldsymbol{\xi}'\restriction_{-(\iota\cdot s)}))\,(\mathsf{a}(\boldsymbol{\xi}'\restriction_{\iota\cdot s}))$$

as required.

Other cases can be proved similarly. □

**Lemma 10.** *If $\gamma \in \mathcal{L}(e^\dagger, \mathcal{G}_p)$, for every $\iota$ and $s$, there exists a transition sequence $\{\, \iota \mapsto (e, s) \,\} \overset{\boldsymbol{\xi}}{\dashrightarrow}_p c$ such that $\gamma = \mathsf{a}(\boldsymbol{\xi})$.*

*Proof.* By induction on the structure of the expression $e$. □

### B.5 Proof of Theorem 3

We prove that $\mathsf{ATrees}(p) = \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$ under the assumption that $p$ is scope-safe.

It is easy to see that $\gamma \in \mathsf{ATrees}(p)$ implies $\gamma \in \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$. Suppose that $c_0 \xrightarrow{\xi_1}_p c_1 \xrightarrow{\xi_2}_p \ldots \xrightarrow{\xi_n}_p c_n$ and $\gamma = \mathsf{a}(\boldsymbol{\xi})$. Then $\gamma \in \mathsf{RelaxedATrees}(p)$ is witnessed by the reduction sequence $d_0 \overset{\xi_1}{\dashrightarrow}_p d_1 \overset{\xi_2}{\dashrightarrow}_p \ldots \overset{\xi_n}{\dashrightarrow}_p d_n$ is obtained by removing lock information from $c_i$. Similarly the transition sequence $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} = \hat{c}_0 \xrightarrow{\xi_1} \hat{c}_1 \xrightarrow{\xi_2} \ldots \xrightarrow{\xi_n} \hat{c}_n$ is constructed from it by replacing each expression in $c_i$ to an appropriate action tree. In this step, we use the assumption that the program is scope-safe. Since action trees do not have any information of the concrete lock name, it has to be recovered from the abstract lock name and the current scope as in the rules of the abstract transition system. Scope-safety of the program ensures that recovered concrete names are correct.

To prove the converse, we use the following lemma.

**Lemma 11.** *Let $p$ be a program that is not necessarily scope-safe. Let $\gamma \in \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$. Then there exist sequences $\boldsymbol{\xi}$ and $\boldsymbol{\zeta}$ of actions such that*

- $\{\, 0 \mapsto (S, 0) \,\} = d_0 \overset{\boldsymbol{\xi}}{\dashrightarrow}_p d_n$,
- $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} \xrightarrow{\boldsymbol{\zeta}} \hat{\bot}$, *and*
- $\boldsymbol{\xi} \overset{\mathrm{ab}}{=} \boldsymbol{\zeta}$.

*Proof.* Let $\gamma \in \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$.

By the definition of $\mathsf{LSATrees}$, we have $\boldsymbol{\zeta}'$ that satisfies the second condition and $\gamma = \mathsf{a}(\boldsymbol{\zeta}')$. We can assume without loss of generality that $\boldsymbol{\zeta}'$ is $\bullet$-free. Similarly, since $\gamma \in \mathsf{RelaxedATrees}(p)$, there exists $\boldsymbol{\xi}'$ such that $d_0 \overset{\boldsymbol{\xi}'}{\dashrightarrow}_p d_n$ for some configuration $d_n$ and $\gamma = \mathsf{a}(\boldsymbol{\xi}')$.

By Corollary 1, we have $\boldsymbol{\xi}'\restriction_{-\bullet} \sim^{\mathrm{ab}} \boldsymbol{\zeta}'\restriction_{-\bullet} = \boldsymbol{\zeta}'$ (since $\boldsymbol{\zeta}'$ is $\bullet$-free). By Lemma 4, there exists $\boldsymbol{\xi}$ such that $\boldsymbol{\xi}\restriction_{-\bullet} \overset{\mathrm{ab}}{=} \boldsymbol{\zeta}'$ and $\boldsymbol{\xi} \sim \boldsymbol{\xi}'$. By Lemma 8, we have $d_0 \overset{\boldsymbol{\xi}}{\dashrightarrow}_p d_n$.

Now the difference of $\boldsymbol{\xi}$ and $\boldsymbol{\xi}'$ is the concrete lock names and $\bullet$-actions. Since $\hat{c} \overset{(\iota,\bullet)}{\dashrightarrow}_p \hat{c}$ for every abstract configuration $\hat{c}$ (provided that $\iota \in dom(\hat{c})$), by inserting $\bullet$-actions appropriately, we obtain $\boldsymbol{\zeta}$ such that $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} \xrightarrow{\boldsymbol{\zeta}} \hat{\bot}$ and $\boldsymbol{\xi} \overset{\mathrm{ab}}{=} \boldsymbol{\zeta}$. □

Assume that $\gamma \in \mathsf{RelaxedATrees}(p) \cap \mathsf{LSATrees}$ and that $p$ is scope-safe. Let $\boldsymbol{\xi} = \xi_1 \dots \xi_n$ and $\boldsymbol{\zeta} = \zeta_1 \dots \zeta_n$ be sequences of actions of Lemma 11 and $d_0 \overset{\xi_1}{\dashrightarrow}_p d_1 \overset{\xi_2}{\dashrightarrow}_p \dots \overset{\xi_n}{\dashrightarrow}_p d_n$ and $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} = \hat{c}_0 \xrightarrow{\zeta_1} \hat{c}_1 \xrightarrow{\zeta_2} \dots \xrightarrow{\zeta_n} \hat{c}_n$ be the associated transition sequences. Although Lemma 11 ensures only $\boldsymbol{\xi} \overset{\mathrm{ab}}{=} \boldsymbol{\zeta}$, under the assumption that $p$ is scope-safe, one can prove that $\boldsymbol{\xi} = \boldsymbol{\zeta}$. This is proved by constructing configurations $c_i$ by merging $d_i$ and $\hat{c}_i$, as well as "merged", reduction sequences as follows.

It is easy to prove by induction on $i$ that

- $dom(d_i) = dom(\hat{c}_i)$ and
- for every $\iota \in dom(d_i)$, if $d_i(\iota) = (e, s_1, \sigma_1)$ and $\hat{c}_i(\iota) = (\gamma, L, s_2, \sigma_2)$, then $\sigma_1 = \sigma_2$ and $s_1 = s_2$.

Now the configuration $c_i$ is defined by

$$c_i(\iota) = \begin{cases} (e, L, s, \sigma) & (\text{if } d_i(\iota) = (e, s, \sigma) \text{ and } \hat{c}_i(\iota) = (\gamma, L, s, \sigma)) \\ \text{undefined} & (\text{if } \iota \notin dom(d_i) = dom(\hat{c}_i(\iota))). \end{cases}$$

It is easy to show that $\xi_i = \zeta_i$ and

$$c_0 \xrightarrow{\xi_1}_p c_1 \xrightarrow{\xi_2}_p \dots \xrightarrow{\xi_n}_p c_n$$

by using scope-safety of the program.

### B.6 Proof of Theorem 4

The goal of this subsection is to prove that $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} \xrightarrow{*} \hat{\bot}$ if and only if $\mathsf{as}(\gamma) = (\emptyset, \emptyset, \epsilon, T, \emptyset)$. It is not difficult to prove the left-to-right direction. To prove the converse, we prove a stronger claim by induction on the size of the action trees.

Let $\iota$ be a thread ID and $\hat{c}$ be an abstract configuration. If $\hat{c}(\iota) = (\gamma, L, s, \sigma)$, then we define $\gamma[\hat{c}, \iota] = \gamma$, $L[\hat{c}, \iota] = L$, $s[\hat{c}, \iota] = s$ and $\sigma[\hat{c}, \iota] = \sigma$. They are undefined if $\hat{c}(\iota)$ is undefined. If $\hat{c}$ is clear from the context, we write $\gamma_\iota$ for $\gamma[\hat{c}, \iota]$ and so on.

**Definition 11.** Let $\hat{c}$ be an abstract configuration. It is *consistent* if it satisfies the following conditions:

– If $\iota \cdot s \in dom(\hat{c})$, then either $\hat{c}(\iota)$ is undefined or $s_\iota > s$.
– If $\iota_1, \iota_2 \in dom(\hat{c})$ and $\iota_1 \neq \iota_2$, then $L_{\iota_1}$ and $L_{\iota_2}$ do not contain a common element.
– If $\sigma_\iota(\kappa) = (\iota, m)$, then $(\kappa, \iota, m+1)$ is fresh, i.e. there is no $\iota'$ such that $\sigma_{\iota'}(\kappa) = (\iota, m')$ or $L_{\iota'}$ contains $(\kappa, \iota, m')$ for some $m' > m$.
– If $\sigma_\iota(\kappa) = (\iota', m)$ with $\iota \neq \iota'$, then $(\kappa, \iota, 1)$ is fresh.

This is a property that holds for all abstract configurations reachable from the initial one $\{\, \epsilon \mapsto (\gamma, \epsilon, 0, \emptyset) \,\}$.

Let $\sigma$ be a scope function, i.e. a partial function from abstract lock names $\kappa$ to pairs $(\iota, m)$ (such that $(\kappa, \iota, m)$ is the concrete lock name assigned to $\kappa$). When $(\iota, m) = \sigma(\kappa)$, we write $(\kappa, \sigma(\kappa))$ to mean $(\kappa, \iota, m)$. For a sequence $R = \kappa_1 \dots \kappa_n$ of abstract lock names, we write $\sigma(R)$ for $(\kappa_1, \sigma(\kappa_1)) \dots (\kappa_n, \sigma(\kappa_n))$. For a set $A \subseteq \mathcal{K}$, we write $\sigma(A)$ for $\{\, (\kappa, \sigma(\kappa)) \mid \kappa \in A \,\}$. For a relation $G \subseteq \mathcal{K} \times \mathcal{K}$, we write $\sigma(G)$ for $\{\, ((\kappa, \sigma(\kappa)), (\kappa', \sigma(\kappa'))) \mid (\kappa, \kappa') \in G \,\}$.

Given an abstract configuration $\hat{c}$, let $(A[\hat{c}, \iota], A^f[\hat{c}, \iota], R[\hat{c}, \iota], T[\hat{c}, \iota], G[\hat{c}, \iota]) := \mathsf{as}(\gamma[\hat{c}, \iota])$ be the acquisition structure of the action tree of thread ID $\iota$ for every $\iota \in dom(\hat{c})$. If $\hat{c}$ is clear from the context, we simply write as $A_\iota$ and so on.

Before the proof of the key lemma, we formally define the notion of *final acquisition*, which is used in the proof.

**Definition 12.** Let $\gamma = \mathsf{acq}_\kappa\, \gamma'$ be an action tree starting from the acquisition operation and assume $(A, A^f, R, T, G) = \mathsf{as}(\mathsf{acq}_\kappa\, \gamma')$ and $(A', A^{f\prime}, R', T', G') = \mathsf{as}(\gamma')$. By the definition of the acquisition structure, there are two cases:

– $R' = R \cdot \kappa$.
– $R' = R = \epsilon$ and $\{\, \kappa \,\} = A^f \backslash A^{f\prime}$.

In the latter case, this acquisition operation is called *final*.

**Lemma 12.** *Let $\hat{c}$ be a consistent configuration. Assume the following conditions:*

– *$\sigma_\iota(R_\iota)$ is a postfix of $L_\iota$ for every $\iota \in dom(\hat{c})$. We write $F_\iota$ for the set of concrete lock names that appears in $L_\iota$ but not in $\sigma_\iota(R_\iota)$.*
– *If $T_\iota = \$$, then $F_\iota = \emptyset$.*
– *There is at most one thread $\iota$ such that $R_\iota \neq \epsilon$.*
– *$(F_\iota \cup \sigma_\iota(A^f_\iota)) \cap (F_{\iota'} \cup \sigma_{\iota'}(A^f_{\iota'})) = \emptyset$ if $\iota \neq \iota'$.*
– *$F_\iota \cap \sigma_{\iota'}(A_{\iota'}) = \emptyset$ for every $\iota, \iota' \in dom(\hat{c})$ (including the case that $\iota = \iota'$).*
– *$G_{\hat{c}} := \bigcup_{\iota \in dom(\hat{c})} \sigma_\iota(G_\iota)$ is acyclic.*

*Then $\hat{c} \xrightarrow{*} \hat{\bot}$.*

*Proof.* We prove the claim by induction on the size $\sum_{\iota \in dom(\hat{c})} \mathrm{size}(\gamma[\hat{c}, \iota])$ of the abstract configuration $\hat{c}$, where the size of an action tree is a number of its nodes with labels other than $\bot$. If the size of the abstract configuration is 0, then $\gamma[\hat{c}, \iota] = \bot$ for every $\iota \in dom(\hat{c})$, which itself is one of $\hat{\bot}$ configurations. Assume that the size of configurations is not zero.

There are two cases:

1. There exists $\iota$ such that (1) the root of $\gamma[\hat{c}, \iota]$ is not a final acquisition nor $\bot$ and (2) $R_{\iota'} = \epsilon$ for every $\iota' \in dom(\hat{c})$ with $\iota' \neq \iota$.
2. $R_\iota = \epsilon$ and the root of $\gamma[\hat{c}, \iota]$ is a final acquisition or $\bot$ for every $\iota \in dom(\hat{c})$.

Note that $R_\iota \neq \epsilon$ implies that the root of $\gamma[\hat{c}, \iota]$ is not a final acquisition. Hence every abstract configuration that satisfies the condition of the lemma is either (1) or (2). In case (1), we execute the thread $\iota$ in the next step. In case (2), there exists a minimal element in $\{ (\kappa, \sigma[\hat{c}, \iota](\kappa)) \mid \iota \in dom(\hat{c}), \gamma[\hat{c}, \iota] = \mathsf{acq}_\kappa\, \gamma' \}$ with respect to $G^+_{\hat{c}}$ since this is a finite set and $G_{\hat{c}}$ is acyclic. Then we execute in the next step the thread that acquires such a minimal lock.

Case (1) and $\gamma[\hat{c}, \iota] = \$$: Then $T_\iota = \$$ and $R_\iota = \epsilon$. By the condition, $F_\iota = \emptyset$, which means that $L[\hat{c}, \iota] = \epsilon$. Hence $\hat{c} \xrightarrow{(\iota, \$)} \hat{c}\{\iota \mapsto \mathrm{undef}\}$. It is easy to see that $\hat{c}\{\iota \mapsto \mathrm{undef}\}$ satisfies all the conditions (since we just remove a thread).

Case (1) and $\gamma[\hat{c}, \iota] = \ell\, \gamma'$: Since $(\iota, \ell)$ cannot be blocked, we have $\hat{c} \xrightarrow{(\iota, \ell)} \hat{c}'$ for some $\hat{c}'$. The abstract configuration $\hat{c}'$ satisfies the conditions because all the data of $\hat{c}'$ are the same as $\hat{c}$ except for $\gamma[\hat{c}', \iota]$.

Case (1) and $\gamma[\hat{c}, \iota] = \mathsf{new}_\kappa\, \gamma'$: Since the action $(\iota, \mathsf{new}(\kappa))$ cannot be blocked, we have $\hat{c} \xrightarrow{(\iota, \mathsf{new}(\kappa))} \hat{c}'$ for some $\hat{c}'$. The abstract configuration $\hat{c}'$ satisfies the conditions because the lock $(\kappa, \iota, m)$ generated by this operation is fresh (because the abstract configuration $\hat{c}$ is consistent), and thus it can only appear in $\sigma[\hat{c}', \iota](A[\hat{c}', \iota])$ and $\sigma[\hat{c}', \iota](A^f[\hat{c}', \iota])$.

Case (1) and $\gamma[\hat{c}, \iota] = \mathsf{acq}_\kappa\, \gamma'$: We first show that $(\kappa, \sigma_\iota(\kappa)) \notin \mathbf{locked}(\hat{c})$. This means that $(\kappa, \sigma_\iota(\kappa)) \notin \bigcup_{\iota' \in dom(\hat{c})} (F_{\iota'} \cup \sigma_{\iota'}(R_{\iota'}))$ (here by abuse of notation, $R_{\iota'}$ means the set of abstract lock names appearing in the sequence $R_{\iota'}$). For $\iota' \neq \iota$, since $R_{\iota'} = \epsilon$, the set of locks in $L_{\iota'}$ is $F_{\iota'}$. By the definition of the acquisition structure, we have $\kappa \in A_\iota$. By the conditions, $\sigma_\iota(A_\iota) \cap F_{\iota'} = \emptyset$. Now it remains to prove that $\kappa \notin R_\iota$. By the definition of the acquisition structure and since this acquisition is not final, by writing $(A', A^{f\prime}, R', T', G') = \mathsf{as}(\gamma')$, we have $R' = R_\iota \cdot \kappa$. Since $R'$ cannot have two occurrences of $\kappa$, this means that $R_\iota$ does not contain $\kappa$. Hence $\hat{c} \xrightarrow{(\iota, \mathsf{acq}(\kappa))} \hat{c}'$ for some $\hat{c}'$. It is easy to see that $\hat{c}'$ satisfies the conditions since $A[\hat{c}', \iota] \subseteq A[\hat{c}, \iota]$, $R[\hat{c}', \iota] = R[\hat{c}, \iota] \cdot \kappa$, $\gamma[\hat{c}', \iota] = \gamma'$, and all other components are the same as $\hat{c}$.

Case (1) and $\gamma[\hat{c}, \iota] = \mathsf{rel}_\kappa\, \gamma'$: By the definition of the acquisition structure, we have $R_\iota = R' \cdot \kappa$. Since $\sigma[\hat{c}, \iota](R_\iota)$ is a postfix of $L[\hat{c}, \iota]$, the last element of $L[\hat{c}, \iota]$ is $(\kappa, \sigma(\kappa))$. Hence $\hat{c} \xrightarrow{(\iota, \mathsf{rel}(\kappa))} \hat{c}'$ for some $\hat{c}'$. It is easy to see that $\hat{c}'$ satisfies all the conditions.

Case (1) and $\gamma[\hat{c}, \iota] = \mathsf{sp}\, \gamma_p\, \gamma_c$: Since $(\iota, \mathsf{sp}(\iota \cdot s_\iota))$ cannot be blocked, we have $\hat{c} \xrightarrow{(\iota, \mathsf{sp}(()\iota \cdot s_\iota))} \hat{c}'$ for some $\hat{c}'$. The abstract configuration $\hat{c}'$ satisfies the condition because each component datum of $\hat{c}'$ is the same as $\hat{c}$ or obtained by dividing data of the thread ID $\iota$ into two.

Case (2): Let $\iota$ be the ID of the thread that acquires the (chosen) minimal concrete lock with respect to $G^+_{\hat{c}}$. Let $\kappa$ be the abstract lock name that the thread $\iota$ acquires.

We first show that $(\kappa, \sigma_\iota(\kappa)) \notin \mathbf{locked}(\hat{c})$ and thus $\hat{c} \xrightarrow{(\iota, \mathsf{acq}(\kappa))} \hat{c}'$ for some $\hat{c}'$. Since $R_{\iota'} = \epsilon$ for every $\iota \in dom(\hat{c})$, it suffices to see that $(\kappa, \sigma_\iota(\kappa)) \notin \bigcup_{\iota' \in dom(\hat{c})} F_{\iota'}$. By the definition of the acquisition structure, we have $\kappa \in A_\iota$. By the conditions, for every $\iota' \in dom(\hat{c})$, we have $A_\iota \cap F_{\iota'} = \emptyset$ and thus $(\kappa, \sigma_\iota(\kappa)) \notin F_{\iota'}$.

It suffices to prove that $\hat{c}'$ satisfies the conditions. What is difficult is to show that $F[\hat{c}', \iota] \cap \sigma[\hat{c}', \iota'](A[\hat{c}', \iota']) = \emptyset$ for every $\iota' \in dom(\hat{c}')$. It suffices to show that $(\kappa, \sigma_\iota(\kappa)) \notin \sigma[\hat{c}', \iota'](A[\hat{c}', \iota'])$ for every $\iota' \in dom(\hat{c})$.

- Suppose that $\iota = \iota'$. Since this acquisition is final, $G[\hat{c}, \iota] \supseteq \{\kappa\} \times A[\hat{c}', \iota]$. By the conditions, $G[\hat{c}, \iota]$ is acyclic. This means that $\kappa \notin A[\hat{c}', \iota]$.
- Suppose that $\iota \neq \iota'$ and $\gamma[\hat{c}, \iota'] = \gamma[\hat{c}', \iota'] = \bot$. Then $A[\hat{c}', \iota'] = \emptyset$.
- Suppose otherwise. Then $\iota \neq \iota'$ and $\gamma[\hat{c}, \iota'] = \gamma[\hat{c}', \iota'] = \mathsf{acq}_{\kappa'}\, \gamma'_{\iota'}$, where $\iota'$ is an final acquisition. Hence $G[\hat{c}, \iota'] = G[\hat{c}', \iota'] \supseteq \{\kappa'\} \times (A[\hat{c}', \iota'] \backslash \{\kappa'\})$. By the construction, $(\kappa, \sigma_\iota(\kappa))$ is minimal with respect to $G_{\hat{c}} \supseteq \sigma[\hat{c}, \iota'](G[\hat{c}, \iota'])$. This means that $(\kappa, \sigma_\iota(\kappa)) \notin \sigma_{\iota'}(A[\hat{c}', \iota'] \backslash \{\kappa'\})$. What remains is to show that $(\kappa, \sigma_\iota(\kappa)) \neq (\kappa', \sigma_{\iota'}(\kappa'))$. Since both acquisitions are final, $\kappa \in A^f[\hat{c}, \iota]$ and $\kappa' \in A^f[\hat{c}, \iota']$. By the conditions, $\sigma_\iota(A^f[\hat{c}, \iota]) \cap \sigma_{\iota'}(A^f[\hat{c}, \iota']) = \emptyset$, and thus $(\kappa, \sigma_\iota(\kappa)) \neq (\kappa', \sigma_{\iota'}(\kappa'))$.

□

Hence $\mathsf{as}(\gamma) = (\emptyset, \emptyset, \epsilon, T, \emptyset)$ implies $\{\, 0 \mapsto (\gamma, \epsilon, 0, \emptyset) \,\} \xrightarrow{*} \hat{\bot}$.

## C Proof of Lemma 3

($\Rightarrow$) We prove the contraposition. Assume that $p$ is not scope-safe or not with nested locking. Let $c_0 \xrightarrow{\iota_1, a_1}_p \cdots \xrightarrow{\iota_n, a_n}_p c_n \xrightarrow{\iota_{n+1}, a_{n+1}}_p c_{n+1}$ be a transition sequence in which only the last step violates the requirements. Let $\mathcal{X}$ be the set of concrete lock names consisting of all but the one used in the last step. Let $\gamma$ be the labeled action tree generated by the pair of the above transition sequence and $\mathcal{X}$. Obviously $\gamma \in \mathcal{L}(\mathcal{G}_p^{AB})$. Since $p$ behaves as if it were a scope-safe program with nested locking for the first $n$-steps, the corresponding action tree is lock-sensitive, i.e. $\gamma \in \mathsf{LSATrees}'$. Since the last step violates scope-safety or well-nestedness of locking, $\gamma \in \mathsf{N} \cup \mathsf{S}$. Hence $\gamma \in (\mathcal{G}_p^{AB} \cap \mathsf{LSATrees}') \cap (\mathsf{N} \cup \mathsf{S})$ and thus it is not empty.

($\Leftarrow$) Assume that $p$ is a scope-safe program with nested locking and let $\gamma \in (\mathcal{L}(\mathcal{G}_p^{AB}) \cap \mathsf{LSATrees}') \cap (\mathsf{N} \cup \mathsf{S})$. By definition of $\mathsf{LSATrees}'$, we have a one-hole tree context $C$ and an action tree $\gamma_0$ with exactly one non-$\bot$ node such that $\gamma = C[\gamma_0]$ and $C[\bot] \in \mathsf{LSATrees}$. By Theorem 4, there is a transition sequence $c_0 \xrightarrow{\iota_1, a_1}_p \cdots \xrightarrow{\iota_n, a_n}_p c_n$ such that $C[\bot] = \mathsf{a}((\iota_1, a_1) \ldots (\iota_n, a_n))$. Since $\gamma \in \mathsf{N} \cup \mathsf{S}$, a step in this transition sequence or the next step from $c_n$ described by $\gamma_0$ violates scope-safety or well-nestedness of locking. This means that $p$ violates these conditions. A contradiction.