Pairwise Reachability Analysis for Higher Order Concurrent Programs by Higher-Order Model Checking

Kazuhide Yasukata, Naoki Kobayashi and Kazutaka Matsuda

The University of Tokyo

Abstract. We propose a sound, complete, and automatic method for pairwise reachability analysis of higher-order concurrent programs with recursion, nested locks, joins, and dynamic thread creation. The method is based on a reduction to higher-order model checking (i.e., model checking of trees generated by higher-order recursion schemes). It can be considered an extension of Gawlitz et al.'s work on the join-lock-sensitive reachability analysis for dynamic pushdown networks (DPN) to higher-order programs. To our knowledge, this is the first application of higher-order model checking to sound and complete verification of (reasonably expressive models of) concurrent programs.

1 Introduction

Verification of concurrent programs is important but fundamentally difficult, especially in the presence of recursion. Ramalingam [19] has shown that the reachability problem for two pushdown systems with rendezvous-style synchronization primitives is undecidable. There are two major approaches to cope with this limitation. One is to give up the soundness, and underapproximate the actual behavior of a concurrent program by bounding the number of context switches [18], etc. The other approach is to restrict the synchronization primitives. Kahlon et al. [7] have shown that the pairwise reachability problem ("Given two pushdown systems P_1 and P_2 and control locations ℓ_1 and ℓ_2, is there a reachable global state where P_1 is at ℓ_1 and P_2 is at ℓ_2?") is decidable if the two pushdown systems synchronize only via nested locking. Lammich et al. [14] later extended it to dynamic pushdown networks (DPN), where processes may be dynamically created. Gawlitza et al. [4] have further extended the result to allow synchronization via joins in addition to nested locking.

In the present paper, we follow the latter line of work and extend it to deal with *higher-order* concurrent programs with recursion, dynamic process creation, joins, and nested locking. We consider the pairwise reachability problem: "Given a program P and two control locations ℓ_1 and ℓ_2, may the program reach a state where one process is at ℓ_1 and another is at ℓ_2?". We show that this problem can be reduced to *higher-order model checking* [15, 9], hence it is decidable. The main idea is to transform a given program to a non-deterministic higher-order

recursion scheme (a kind of tree grammar where non-terminals may take higher-order functions as arguments) $\mathcal{G}$ that generates a tree language $\mathcal{L}(\mathcal{G})$ consisting of all the possible execution histories (called *action trees* [14, 4]) of the program ignoring the synchronization constraints imposed by joins and nested locking. Let L_1 be the set of action trees that respect the synchronization constraints imposed by joins and nested locking, and L_2 be the set of action trees that represent histories that end with a state where two of the processes are at ℓ_1 and ℓ_2. Then, ℓ_1 and ℓ_2 are pairwise reachable if and only if $\mathcal{L}(\mathcal{G}) \cap L_1 \cap L_2 \neq \emptyset$. Since both L_1 and L_2 are regular (where the regularity of L_1 is due to [4]), the latter condition can be decided by using higher-order model checking [15, 9]. We formalize the reduction and prove its correctness. We also report preliminary experimental results, which confirm that the approach is feasible at least for small programs, despite the extremely high worst-case complexity of higher-order model checking (k-EXPTIME complete for order-k higher-order recursion schemes [15, 12]).

To our knowledge, this is the first application of higher-order model checking to sound and *complete* verification of higher-order *concurrent* programs. The previous applications of higher-order model checking were mainly for higher-order *functional* programs [9, 13, 16]. For concurrent programs, the previous applications [10, 5] were based on the underapproximation approach. One may think that higher-order functions are exotic features for multi-threaded programs. As demonstrated in [9, 20], however, even if higher-order functions are not so often used explicitly in source programs, they are required for precisely modelling control/data structures such as exceptions and lists.

The rest of the paper is structured as follows. Section 2 introduces the target language and formally defines the pairwise reachability problem. After providing the necessary backgrounds (such as action trees [14, 4] and higher-order model checking [15, 9]) Section 3 provides the reduction of the pairwise reachability problem to higher-order model checking. Section 4 reports preliminary experiments. Section 5 discusses related work and Section 6 concludes the paper.

2 Target Language and Pairwise Reachability Problem

This section introduces a higher-order concurrent programming language and defines the pairwise reachability problem for it.

Definition 1. *A **program** is a finite set of* function definitions

$$\{F_1\,\tilde{x_1} = e_1,\ ...,\ F_n\,\tilde{x_n} = e_n\},$$

where F_i denotes a defined function symbol, and e ranges over the set Exp *of* expressions*, defined by:*

$$e ::= \$ \mid x \mid F \mid \mathbf{if}\_\, e_1\, e_2 \mid e_1\, e_2 \mid \mathbf{join};\, e \mid \mathbf{acq}_i;\, e \mid \mathbf{rel}_i;\, e \mid \mathbf{spawn}(e_c);\, e \mid e^{\ell}$$

Here, i ranges over a finite set Lock of (non-reentrant) locks, and ℓ ranges over a finite set Label of program point labels. Note that the arity of each function

may be 0. We require that the function symbols $F_1, ..., F_n$ are different from each other, and that any program p contains exactly one definition of a "main" function S, of the form $S = e$.

We explain the intuitive meaning of each expression; the formal operational semantics is given later. The expression \$ represents the termination of the current process. The expression $\mathbf{if}\_\, e_1\ e_2$ executes either e_1 or e_2 non-deterministically, and the expression $e_1 e_2$ applies the function e_1 to e_2. As defined later, function calls are based on the call-by-name semantics; call-by-value programs can be transformed to call-by-name programs by using the CPS transformation [17]. The expression $\mathbf{spawn}(e_c)$; e spawns a new child process that executes e_c, and continues to execute e without waiting for the child process. The expression $\mathbf{join}$; e waits for the termination of all the processes that the current process has created, and then executes e. The expression $\mathbf{acq}_i$; e waits to acquire the lock i, and then executes e. The expression $\mathbf{rel}_i$; e releases the lock i and executes e. The label ℓ in the expression e^ℓ is used for specifying the pairwise reachability problem, and does not affect the operational semantics.

In this paper, we consider only "well-typed" programs, as defined below.

Definition 2 (types). *The set of **types** is inductively defined by:*

$$\tau ::= \mathbf{unit} \mid \tau_1 \to \tau_2$$

Here, $\mathbf{unit}$ *is the type of the unit value* \$*, and* $\tau_1 \to \tau_2$ *is the type of functions from* τ_1 *to* τ_2*. The **order** and the **arity** of types are inductively defined by:*

$$\begin{array}{ll} \mathit{order}(\mathbf{unit}) = 0 & \mathit{order}(\tau_1 \to \tau_2) = \max(\mathit{order}(\tau_1) + 1, \mathit{order}(\tau_2)) \\ \mathit{arity}(\mathbf{unit}) = 0 & \mathit{arity}(\tau_1 \to \tau_2) = \mathit{arity}(\tau_2) + 1 \end{array}$$

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

$$\frac{}{\emptyset \vdash \$: \mathbf{unit}} \qquad \frac{}{\Gamma\{x \mapsto \tau\} \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathbf{unit} \qquad \Gamma \vdash e_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{if}\_\, e_1\, e_2 : \mathbf{unit}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \mathbf{unit}}{\Gamma \vdash \mathbf{join};\, e : \mathbf{unit}} \qquad \frac{\Gamma \vdash e : \mathbf{unit}}{\Gamma \vdash \mathbf{acq}_i;\, e : \mathbf{unit}}$$

$$\frac{\Gamma \vdash e : \mathbf{unit}}{\Gamma \vdash \mathbf{rel}_i;\, e : \mathbf{unit}} \qquad \frac{\Gamma \vdash e : \mathbf{unit} \qquad \Gamma \vdash e_c : \mathbf{unit}}{\Gamma \vdash \mathbf{spawn}(e_c);\, e : \mathbf{unit}} \qquad \frac{\Gamma \vdash e : \mathbf{unit}}{\Gamma \vdash e^\ell : \mathbf{unit}}$$

A program $p = \{F_1\, x_{11}, \ldots, x_{1k_1} = e_1, \ldots, F_n\, x_{n1}, \ldots, x_{nk_n} = e_n\}$ *is **well-typed** under* Γ *if* $\Gamma = \{\, F_i \mapsto \tau_{i1} \to ... \to \tau_{ik_i} \to \mathbf{unit} \mid i \in \{1, \ldots, n\}\,\}$ *and* $\Gamma \cup \{x_{j1} \mapsto \tau_{j1}, ..., x_{jk_j} \mapsto \tau_{jk_j}\} \vdash e_j : \mathbf{unit}$ *holds for each* $j \in \{1, \ldots, n\}$*. The **order** of* p *(well-typed under* Γ*) is* $\max(\{\, \mathit{order}(\Gamma(F)) \mid F \in \mathit{dom}(\Gamma)\,\})$*.*

Remark 1. In the language above, we have only the unit-value as a base value. As we have higher-order functions, however, we can encode booleans and conditionals by using the Church encoding. Values in infinite data domains (such as unbounded integers) can be dealt with (soundly but incompletely) by using predicate abstraction [9, 13].

Example 1. Here is an example of an order-2 program, well-typed under $\Gamma = \{S \mapsto \mathbf{unit}, F \mapsto (\mathbf{unit} \to \mathbf{unit}) \to \mathbf{unit} \to \mathbf{unit}, G \mapsto \mathbf{unit} \to \mathbf{unit}, H \mapsto (\mathbf{unit} \to \mathbf{unit}) \to \mathbf{unit} \to \mathbf{unit}\}$.

$$p = \left\{ \begin{array}{ll} S = F\,G\,\$ & F\,g\,t = \mathbf{if}\_(\mathbf{spawn}(H\,g\,\$);\ F\,g\,t)\,(\mathbf{join};\ t) \\ G\,t = t^{\ell} & H\,g\,t = \mathbf{acq}_1; (g\,(\mathbf{rel}_1; t)) \end{array} \right\}$$

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

```
main(){f(g);}
f(g){
  if(*){spawn{h(g);}; f(g);}
  else{join(); }
}
```

```
g(){
  L: <critical section>;
  return;
}
h(g){ acq(1); g(); rel(1);}
```

The root process spawns non-deterministically many child processes, and then waits for the child processes by the join operation. Each child acquires the lock 1 and then release 1 at the program point ℓ.

We now define the formal semantics of programs. A **configuration** c of a program is a map from a finite set consisting of sequences of natural numbers (where each sequence serves as a process identifier) to the set of triples (e, L, s) consisting of an expression e, a sequence L of locks, and a natural number s. Intuitively, $c(\pi) = (e, i_1 \cdots i_k, s)$ means that the process π is executing e, that it holds locks $i_1, \ldots, i_k$ that have been acquired in this order, and that it has created s child processes so far. The **reduction relation** $c \longrightarrow c'$ on configurations is defined by the following rules.

$$\frac{F\,x_1\ \cdots\ x_k \to e \in p}{c \uplus \{\,\pi \mapsto (F\,e_1\,...\,e_k, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto ([e_1/x_1,\ ...,\ e_k/x_k]e, L, s)\,\}}$$

$$c \uplus \{\,\pi \mapsto (\mathbf{if}\_\ e_1\ e_2, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e_i, L, s)\,\}\ \ (i \in \{\,1, 2\,\})$$

$$c \uplus \{\,\pi \mapsto (e^{\ell}, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e, L, s)\,\}$$

$$\frac{i \notin \mathbf{locked}(c \uplus \{\,\pi \mapsto (\mathbf{acq}_i; e, L, s)\,\})}{c \uplus \{\,\pi \mapsto (\mathbf{acq}_i; e, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e, L \cdot i, s)\,\}}$$

$$c \uplus \{\,\pi \mapsto (\mathbf{rel}_i; e, L \cdot i, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e, L, s)\,\}$$

$$c \uplus \{\,\pi \mapsto (\mathbf{spawn}(e_c);\ e, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e, L, s+1), \pi \cdot s \mapsto (e_c, \epsilon, 0)\,\}$$

$$\frac{\{\,j \mid \pi \cdot j \in \mathit{dom}(c)\,\} = \emptyset}{c \uplus \{\,\pi \mapsto (\mathbf{join}; e, L, s)\,\} \longrightarrow_p c \uplus \{\,\pi \mapsto (e, L, s)\,\}}$$

$$c \uplus \{\,\pi \mapsto (\$, \epsilon, s)\,\} \longrightarrow_p c$$

Here, $\mathbf{locked}(c)$ represents the set of all acquired locks, defined by:

$$\mathbf{locked}(c) = \bigcup_{c(\pi)=(e,i_1\cdots i_k,s)} \{\, i_1, \ldots, i_k \,\}.$$

Definition 3 (pairwise reachability). *Let p be a (well-typed) program and ℓ_1, ℓ_2 be labels. We say that (ℓ_1, ℓ_2) is **pairwise reachable** by p, written $p \models \ell_1 || \ell_2$, if there exist c, π_1, π_2 $(\pi_1 \neq \pi_2)$ such that $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow_p^* c$ with $c(\pi_1) = (e_1^{\ell_1}, L_1, s_1)$ and $c(\pi_2) = (e_2^{\ell_2}, L_2, s_2)$ for some $e_2, e_2, L_1, L_2, s_1, s_2$. The **pairwise reachability problem** is the decision problem of checking whether $p \models \ell_1 || \ell_2$ holds.*

Example 2. Recall the example program showed in Example 1. The verification problem "can the program point ℓ (i.e., the critical section) be reached by multiple processes simultaneously?" can be reduced to the pairwise reachability for the program p and the pair (ℓ, ℓ). In this case, the answer to the problem is "no".

3 From Pairwise Reachability to Higher-Order Model Checking

In this section, we show that the pairwise reachability problem can be reduced to higher-order model checking [15], hence it is decidable. The basic idea of this reduction is to transform a program to a grammar called a higher-order recursion scheme [15], which generates *action trees* [4, 14] that represent all the possible executions of the program. Since the set of action trees that represent valid executions (i.e., those that respect synchronization constraints on joins and nested locks) is regular, the pairwise reachability problem can be reduced to an inclusion problem between the tree language generated by the higher-order recursion scheme and the regular language, which can be further reduced to higher-order model checking. We first review action trees and higher-order model checking in Sections 3.1 and 3.2 respectively. We then present the reduction from the pairwise reachability analysis to higher-order model checking.

3.1 Action trees

An action tree [4, 14] is a finite tree that represents a history of executions of a program up to a certain state. It ignores how the executions of multiple processes are interleaved, and expresses only process-wise execution histories and the parent/child relationship between processes. Gawlitza et al. [4] originally introduced action trees to represent execution histories of dynamic pushdown networks, but the notion of action trees is independent of a particular computation model. Here we use them to represent execution histories of higher-order concurrent programs introduced in the previous section.

Definition 4 (action trees). *The set of* ***action trees****, ranged over by* γ*, is defined inductively by:*

$$\gamma ::= \bot \mid \langle \$ \rangle \mid \ell \mid \langle \mathsf{jo} \rangle\, \gamma \mid \langle \mathsf{sp} \rangle\, \gamma_1\, \gamma_2 \mid \langle \mathsf{Acq}_i \rangle\, \gamma \mid \langle \mathsf{Rel}_i \rangle\, \gamma.$$

Here, ℓ *ranges over (program) labels and* i *over locks.*

We have used the term representation of (labelled) trees above. Trees can also be considered as map from paths to labels, by: $(a\,\gamma_1 \cdots \gamma_n)^{\#} = \{\, \epsilon \mapsto a \,\} \cup \{\, i \cdot u \mapsto b \mid \gamma_i^{\#}(u) = b \,\}$.

Each non-leaf node represents an action of each process. The tree $\langle \mathsf{jo} \rangle\, \gamma$ ($\langle \mathsf{Acq}_i \rangle\, \gamma$ and $\langle \mathsf{Rel}_i \rangle\, \gamma$ respectively) means that the process performed join (acquires and releases the lock i, respectively) and then behaved like γ. The tree $\langle \mathsf{sp} \rangle\, \gamma_1\, \gamma_2$ means that the process spawned a child process that behaved like γ_2, and the process itself behaved like γ_1. Thus, the leftmost path from the root node in an action tree represents a sequence of actions performed by the root process, and each leftmost path from the second child of a $\langle \mathsf{sp} \rangle$-node represents a sequence of actions performed by the spawned process. Each leaf node of an action tree represents the current state of each process: $\langle \$ \rangle$ means that the process has terminated, ℓ means that the process is at the program point ℓ, and $\bot$ means that the process is at a program point not labeled by any element of Label. Figure 1 shows an example of an action tree. It represents an execution history where the root process spawns a child process, waits for the child, and terminates (represented by $\langle \$ \rangle$), and the child process acquires and releases the lock i, and then terminates. It corresponds to the following execution of the program in Example 1:

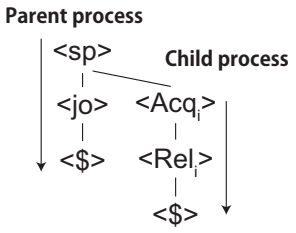

Fig. 1. Action tree.

$$\begin{aligned}
&\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow \{\, \epsilon \mapsto (F\,G\,\$, \epsilon, 0) \,\} \longrightarrow^* \{\, \epsilon \mapsto (\mathbf{spawn}(H\,G\,\$);\ F\,G\,\$, \epsilon, 0) \,\} \\
&\longrightarrow \{\, \epsilon \mapsto (F\,G\,\$, \epsilon, 1), 0 \mapsto (H\,G\,\$, \epsilon, 0) \,\} \\
&\longrightarrow^* \{\, \epsilon \mapsto (\mathbf{join}; \$, \epsilon, 1), 0 \mapsto (\mathbf{acq}_1; G(\mathbf{rel}_1; \$), \epsilon, 0) \,\} \\
&\longrightarrow^* \{\, \epsilon \mapsto (\mathbf{join}; \$, \epsilon, 1), 0 \mapsto (G(\mathbf{rel}_1; \$), 1, 0) \,\} \\
&\longrightarrow^* \{\, \epsilon \mapsto (\mathbf{join}; \$, \epsilon, 1), 0 \mapsto ((\mathbf{rel}_1; \$)^{\ell}, 1, 0) \,\} \\
&\longrightarrow^* \{\, \epsilon \mapsto (\mathbf{join}; \$, \epsilon, 1), 0 \mapsto (\$, \epsilon, 0) \,\} \\
&\longrightarrow \{\, \epsilon \mapsto (\mathbf{join}; \$, \epsilon, 1) \,\} \longrightarrow \{\, \epsilon \mapsto (\$, \epsilon, 1) \,\} \longrightarrow \emptyset
\end{aligned}$$

Note that not every action tree represents a valid execution history. For example, consider the action tree: $\langle \mathsf{sp} \rangle\, (\langle \mathsf{Acq}_1 \rangle\, \ell_1)\, (\langle \mathsf{Acq}_1 \rangle\, \ell_2)$. It represents a state where the parent and child processes are at program points ℓ_1 and ℓ_2 respectively, after *both* having acquired the lock 1 (and not released it yet), which is obviously impossible. In order to exclude action trees that do not respect synchronization constraints, we introduce a transition system on *abstract* configurations, obtained by removing expressions from configurations introduced in the previous section.

Definition 5. *An **abstract configuration** is a map from a finite set consisting of sequences of natural numbers (where each sequence serves as a process identifier) to the set of pairs (L, s) consisting of a sequence L of locks, and a natural number s. The transition relation on abstract configurations is defined by:*

$$\frac{i \notin \mathbf{locked}(c \uplus \{\, \pi \mapsto (L, s) \,\})}{c \uplus \{\, \pi \mapsto (L, s) \,\} \xrightarrow{\pi, \langle \mathsf{Acq}_i \rangle} c \uplus \{\, \pi \mapsto (L \cdot i, s) \,\}}$$

$$c \uplus \{\, \pi \mapsto (L \cdot i, s) \,\} \xrightarrow{\pi, \langle \mathsf{Rel}_i \rangle} c \uplus \{\, \pi \mapsto (L, s) \,\}$$

$$c \uplus \{\, \pi \mapsto (L, s) \,\} \xrightarrow{\pi, \langle \mathsf{sp} \rangle} c \uplus \{\, \pi \mapsto (L, s+1), \pi \cdot s \mapsto (\epsilon, 0) \,\}$$

$$\frac{\{\, k \mid \pi \cdot k \in \mathit{dom}(c) \,\} = \emptyset}{c \uplus \{\, \pi \mapsto (L, s) \,\} \xrightarrow{\pi, \langle \mathsf{jo} \rangle} c \uplus \{\, \pi \mapsto (L, s) \,\}}$$

$$c \uplus \{\, \pi \mapsto (\epsilon, s) \,\} \xrightarrow{\pi, \langle \$ \rangle} c$$

Here, $\mathbf{locked}(c)$ *is defined similarly to that for configurations, as*

$$\mathbf{locked}(c) = \bigcup_{c(\pi) = (i_1 \cdots i_k, s)} \{\, i_1, \ldots, i_k \,\}.$$

Each action tree can be mapped to an abstract configuration as follows.

$$\theta_{\pi,L,s}(\langle \$ \rangle) = \emptyset \qquad \theta_{\pi,L,s}(\langle \mathsf{sp} \rangle\, \gamma_1\, \gamma_2) = \theta_{\pi,L,s+1}(\gamma_1) \cup \theta_{\pi \cdot s, \epsilon, 0}(\gamma_2)$$
$$\theta_{\pi,L,s}(\langle \mathsf{jo} \rangle\, \gamma) = \theta_{\pi,L,s}(\gamma) \qquad \theta_{\pi,L,s}(\langle \mathsf{Acq}_i \rangle\, \gamma) = \theta_{\pi,L \cdot i,s}(\gamma)$$
$$\theta_{\pi,L \cdot i,s}(\langle \mathsf{Rel}_i \rangle\, \gamma) = \theta_{\pi,L,s}(\gamma) \qquad \theta_{\pi,L,s}(\gamma) = \{\, \pi \mapsto (L, s) \,\} \text{ (if } \gamma \in \{\, \bot \,\} \cup \mathsf{Label})$$

We write $\theta(t)$ for $\theta_{\epsilon,\epsilon,s}(t)$, and write $\gamma_1 \overset{u}{\rightsquigarrow} \gamma_2$ if $\gamma_1(u) = \bot$ and γ_2 is obtained from γ_1 by replacing $\bot$ at u with a tree of the form $a \bot \ldots \bot$ with $a = \langle \$ \rangle, \langle \mathsf{sp} \rangle, \langle \mathsf{jo} \rangle, \langle \mathsf{Acq}_i \rangle, \langle \mathsf{Rel}_i \rangle$.

We can now define "valid" action trees as follows.

Definition 6. *An action tree γ is **join-lock sensitive** if there is a sequence $\bot = \gamma_0 \overset{u_1}{\rightsquigarrow} \gamma_1 \overset{u_2}{\rightsquigarrow} \ldots \overset{u_n}{\rightsquigarrow} \gamma_n = \gamma'$ such that $\theta(\gamma_0) \xrightarrow{\mathsf{pn}(u_1,\gamma),\gamma_1(u_1)} \theta(\gamma_1) \xrightarrow{\mathsf{pn}(u_2,\gamma),\gamma_2(u_2)} \ldots \xrightarrow{\mathsf{pn}(u_n,\gamma),\gamma_n} \theta(\gamma_n)$, where γ' is the action tree obtained from γ by replacing all $\ell \in \mathsf{Label}$ with $\bot$. Here, $\mathsf{pn}(u, \gamma)$ represents the identifier of the process that executes the action of $\gamma(u)$. It is defined by:*

$$\begin{array}{lll}
\mathsf{pn}(\epsilon, \gamma) = \epsilon & \mathsf{cn}(\epsilon, \gamma) = 0 & \\
\mathsf{pn}(u \cdot 1, \gamma) = \mathsf{pn}(u, \gamma) & \mathsf{cn}(u \cdot 1, \gamma) = \mathsf{cn}(u, \gamma) & \text{if } \gamma(u) \neq \langle \mathsf{sp} \rangle \\
\mathsf{pn}(u \cdot 2, \gamma) = \mathsf{pn}(u, \gamma) \cdot \mathsf{cn}(u, \gamma) & \mathsf{cn}(u \cdot 1, \gamma) = \mathsf{cn}(u, \gamma) + 1 & \text{if } \gamma(u) = \langle \mathsf{sp} \rangle \\
 & \mathsf{cn}(u \cdot 2, \gamma) = 0 &
\end{array}$$

Note that $\mathsf{pn}(u_i, \gamma_i) = \mathsf{pn}(u_i, \gamma)$.

The following is the key property of action trees, which we use in our reduction from pairwise reachability analysis to higher-order model checking.

Lemma 1 (Gawlitza et al. 2011, Section 5 [4]). *The set $L_{\mathsf{sensitive}}$ of join-lock-sensitive action trees is a regular tree language.*

Remark 2. We have modified the original definition of join-lock sensitive (schedulable) action trees. Our notion of join-lock sensitive action trees corresponds to join-lock-well-formed, join-lock sensitive schedulable action trees [4].

3.2 Higher-Order Model Checking

In this subsection, we review *higher-order recursion schemes* and (a variation of) *higher-order model checking* [15].

The set of **sorts** is given by the grammar: $\kappa ::= \mathsf{o} \mid \kappa_1 \to \kappa_2$. Intuitively o describes trees, and $\kappa_1 \to \kappa_2$ describes functions from κ_1 to κ_2. The **order** of sorts is defined by: $\mathit{order}(\mathsf{o}) = 0$ and $\mathit{order}(\kappa_1 \to \kappa_2) = \max(\mathit{order}(\kappa_1) + 1, \mathit{order}(\kappa_2))$.

Definition 7 (Higher-Order Recursion Scheme). *A (non-deterministic) **higher-order recursion scheme** (HORS, for short) is a quadruple: $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where Σ is a ranked alphabet (i.e., a map from a finite set of symbols called **terminals** to their arities); $\mathcal{N}$ is a map from a finite set of symbols called **non-terminals** to sorts; $S \in \mathit{dom}(\mathcal{N})$ is the start symbol of sort o; and $\mathcal{R}$ is a finite set of **transition rules** of the form $A\, x_1 \cdots x_\ell \to t$, where t ranges over the set of applicative terms defined by $t ::= x \mid a \mid A \mid t_1 t_2$. Here, a ranges over $\mathit{dom}(\Sigma)$ and A ranges over $\mathit{dom}(\mathcal{N})$. If $A\, x_1 \cdots x_\ell \to t \in \mathcal{N}$, then $\mathcal{N}(A)$ must be of the form $\kappa_1 \to \cdots \to \kappa_\ell \to \mathsf{o}$ and $\mathcal{N} \cup \{ x_1 : \kappa_1, \ldots, x_\ell : \kappa_\ell \} \vdash_\Sigma t : \mathsf{o}$ must be derivable by using the following rules (where non-terminals are treated as variables).*

$$\mathcal{K} \vdash_\Sigma a : \underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{\Sigma(a)} \to \mathsf{o} \qquad\qquad \mathcal{K} \vdash_\Sigma x : \mathcal{K}(x)$$

$$\frac{\mathcal{K} \vdash_\Sigma t_1 : \kappa_1 \to \kappa_2 \qquad \mathcal{K} \vdash_\Sigma t_2 : \kappa_1}{\mathcal{K} \vdash_\Sigma t_1 t_2 : \kappa_2}$$

The order of a HORS $\mathcal{G}$, written $\mathit{order}(\mathcal{G})$ is $\max(\{ \mathit{order}(A) \mid A \in \mathit{dom}(\mathcal{N}) \})$.

Note that unlike *deterministic* HORS, there may be an arbitrary number of rewriting rules for each non-terminal. We omit the adjective 'non-deterministic' in the rest of this paper.

To define the rewriting relation, we define the notion of the reduction context.

Definition 8. *The set of **reduction contexts** is defined by:*

$$C ::= [\,] \mid a\, t_1 \ldots t_{i-1}\, C\, t_{i+1} \ldots t_n$$

For a reduction context C, we write $C[t]$ for the term obtained from C by replacing $[\,]$ with t.

Then, the rewriting relation $\longrightarrow_{\mathcal{G}}$ on terms is defined by:

$$C[A\,t_1\,\ldots\,t_m] \longrightarrow_{\mathcal{G}} C[\,[t_1/x_1,\ldots,t_m/x_m]t\,] \quad (\text{if } A\,x_1\,\ldots\,x_m \to t)$$

In this paper, we consider a HORS as a generator of a language of finite trees, rather than that of an infinite tree [15].

Definition 9 (Tree Languages of Recursion Schemes). *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a HORS. The language generated by $\mathcal{G}$ is defined by:*

$$\mathcal{L}(\mathcal{G}) = \{t \mid t \text{ is a ranked } \Sigma\text{-labeled tree and } S \longrightarrow_{\mathcal{G}}^{*} t\}$$

Example 3. Consider an order-2 HORS $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ such that

$$\begin{aligned}
\Sigma &= \{\, a \mapsto 2, b \mapsto 1, c \mapsto 0 \,\} \\
\mathcal{N} &= \{\, S \mapsto \mathtt{o}, F \mapsto (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}, G \mapsto \mathtt{o} \to \mathtt{o}, T \mapsto (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o} \,\} \\
\mathcal{R} &= \{S \to F\,G\,c,\ F\,g\,x \to a\,(g\,x)\,(F\,(T\,g)\,x),\ F\,g\,x \to g\,x, \\
&\qquad G\,x \to b\,x,\ T\,g\,x \to g\,(g\,x)\}
\end{aligned}$$

Here is an example of reduction from S to $a\,(b\,c)\,(b^2\,c)$.

$$\begin{aligned}
&S \longrightarrow_{\mathcal{G}} F\,Gc \longrightarrow_{\mathcal{G}} a\,(G\,c)\,(F\,(T\,G)c) \longrightarrow_{\mathcal{G}} a\,(b\,c)(F\,(T\,G)\,c) \longrightarrow_{\mathcal{G}} a\,(b\,c)(T\,G\,c) \\
&\longrightarrow_{\mathcal{G}} a\,(b\,c)(G\,(G\,c)) \longrightarrow_{\mathcal{G}}^{*} a\,(b\,c)(b\,(b\,c))
\end{aligned}$$

The language $\mathcal{L}(\mathcal{G})$ is $\{\, a\,(b\,c)(a\,(b^2\,c)(a \ldots (a\,(b^{2^{n-1}}\,c)(b^{2^n}\,c)) \ldots)) \mid n \in \mathbb{N}_+ \,\}$.

The following theorem is an easy corollary of Ong's result on the model checking of (deterministic) HORS [15].

Theorem 1. *Given a HORS $\mathcal{G}$ and a regular tree language L, it is decidable whether $\mathcal{L}(\mathcal{G}) \subseteq L$.*

In the present paper, we call the inclusion problem above a higher-order model checking problem. The standard model checking problem for HORS [15] is the problem of deciding whether the infinite tree generated by a deterministic HORS satisfies a given property.

3.3 Reduction from Pairwise Reachability to Higher-Order Model Checking

Now we show how to reduce pairwise reachability to higher-order model checking. First, we define a transformation from a concurrent program to a HORS that generates action trees of the which join-lock sensitive subset represent all and only the possible reachable configurations of the program.

Definition 10. *Let $p = \{F_1\,x_{11}, \ldots, x_{1k_1} = e_1, \ldots, F_n\,x_{n1}, \ldots, x_{nk_n} = e_n\}$ be a well-typed higher-order concurrent program under Γ. A (non-deterministic)*

HORS $\mathcal{G}_p$ is defined by:

$$
\begin{aligned}
\Sigma &= \{\, \langle \mathsf{sp}\rangle \mapsto 2, \langle \mathsf{jo}\rangle \mapsto 1, \langle \$\rangle \mapsto 0, \bot \mapsto 0 \,\} \\
&\quad \cup \{\, \langle \mathsf{Acq}_i\rangle, \langle \mathsf{Rel}_i\rangle \mid i \in \mathsf{Lock} \,\} \cup \{\, \ell \mapsto 0 \mid \ell \in \mathsf{Label} \,\} \\
\mathcal{G}_p &= (\Sigma, \mathcal{N}_\Gamma \cup \mathcal{N}_0, \mathcal{R}_\Gamma \cup \mathcal{R}_0, S) \\
\mathcal{N}_\Gamma &= \{F_1 \mapsto (\Gamma(F_1))^\sharp, \ldots, F_n \mapsto (\Gamma(F_n))^\sharp\} \\
\mathcal{N}_0 &= \{E_\$ \mapsto \mathsf{o},\ E_{\mathbf{if}\_} \mapsto (\mathsf{o} \to \mathsf{o} \to \mathsf{o}),\ E_{\mathbf{join}} \mapsto (\mathsf{o} \to \mathsf{o}),\ E_{\mathbf{spawn}} \mapsto (\mathsf{o} \to \mathsf{o} \to \mathsf{o})\} \\
&\quad \cup \{E_{\mathbf{acq}_i} \mapsto (\mathsf{o} \to \mathsf{o}) \mid i \in \mathsf{Lock}\} \cup \{E_{\mathbf{rel}_i} \mapsto (\mathsf{o} \to \mathsf{o}) \mid i \in \mathsf{Lock}\} \\
&\quad \cup \{E_\ell \mapsto (\mathsf{o} \to \mathsf{o}) \mid \ell \in \mathsf{Label}\} \\
\mathcal{R}_\Gamma &= \{F_1 \tilde{x}_1 \to \mathcal{E}(e_1),\ \ldots,\ F_n \tilde{x}_n \to \mathcal{E}(e_n)\} \\
\mathcal{R}_0 &= \{E_{\mathbf{if}\_}\, x\, y \to x,\ E_{\mathbf{if}\_}\, x\, y \to y,\ E_{\mathbf{join}}\, x \to \langle \mathsf{jo}\rangle\, x,\ E_{\mathbf{spawn}}\, x\, y \to \langle \mathsf{sp}\rangle\, x\, y\} \\
&\quad \cup \{E_{\mathbf{acq}_i}\, x \to \langle \mathsf{Acq}_i\rangle\, x \mid i \in \mathsf{Lock}\} \cup \{E_{\mathbf{rel}_i}\, x \to \langle \mathsf{Rel}_i\rangle\, x \mid i \in \mathsf{Lock}\} \\
&\quad \cup \{E_\ell\, x \to \ell \mid \ell \in \mathsf{Label}\} \cup \{E_\ell\, x \to x \mid \ell \in \mathsf{Label}\} \cup \{\, E_\$ \to \langle \$\rangle \,\} \\
&\quad \cup \{E\, \tilde{x} \to \bot \mid E \in \mathit{dom}(\mathcal{N}_\Gamma \cup \mathcal{N}_0) \setminus \{E_\ell \mid \ell \in \mathsf{Label}\}\}
\end{aligned}
$$

Here, $(\cdot)^\sharp$ is a transformation from types of expressions to sorts, defined by: $\mathbf{unit}^\sharp = \mathsf{o}$ and $(\tau_1 \to \tau_2)^\sharp = \tau_1^\sharp \to \tau_2^\sharp$. The function $\mathcal{E}$ transforms an expression to an applicative term, defined inductively by:

$$
\begin{array}{llllll}
\mathcal{E}(\$) = E_\$ & \mathcal{E}(x) = x & \mathcal{E}(F) = F & \mathcal{E}(\mathbf{if}\_\, e_1\, e_2) = E_{\mathbf{if}\_}\, \mathcal{E}(e_1)\, \mathcal{E}(e_2) \\
\mathcal{E}(e_1\, e_2) = \mathcal{E}(e_1)\, \mathcal{E}(e_2) & \mathcal{E}(\mathbf{join};\, e) = E_{\mathbf{join}}\, \mathcal{E}(e) & \mathcal{E}(\mathbf{acq}_i;\, e) = E_{\mathbf{acq}_i}\, \mathcal{E}(e) \\
\mathcal{E}(\mathbf{rel}_i;\, e) = E_{\mathbf{rel}_i}\, \mathcal{E}(e) & \mathcal{E}(\mathbf{spawn}(e_c);\, e) = E_{\mathbf{spawn}}\, \mathcal{E}(e)\, \mathcal{E}(e_c) & \mathcal{E}(e^\ell) = E_\ell\, \mathcal{E}(e)
\end{array}
$$

The idea of the transformation above is quite simple: just replace each synchronization primitive **op** with a non-terminal $E_{\mathbf{op}}$, which will generate a tree node $\langle \mathsf{op}\rangle$ indicating that the operation **op** has been performed. Additionally, in order to generate all the intermediate states of an execution, we allow each non-terminal to be reduced to $\bot$ or $\ell \in \mathsf{Label}$. Note that $\mathit{order}(\Gamma(F)) = \mathit{order}(\mathcal{N}(F))$ holds for every function symbol F of p. Thus, $\mathit{order}(\mathcal{G}_p) = \max(\mathit{order}(p), 1)$.

Example 4. The program p of Example 1 is transformed to the recursion scheme $\mathcal{G}_p = (\Sigma, \mathcal{N}_\Gamma \cup \mathcal{N}_0, \mathcal{R}_\Gamma \cup \mathcal{R}_0, S)$ where

$$
\mathcal{N}_\Gamma = \{S \mapsto \mathsf{o},\ F \mapsto (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o},\ G \mapsto \mathsf{o} \to \mathsf{o},\ H \mapsto (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}\}
$$

$$
\mathcal{R}_\Gamma = \left\{ \begin{array}{l}
S \to F\, G\, E_\$ \\
F\, g\, t \to E_{\mathbf{if}\_}\, (E_{\mathbf{spawn}}\, (F\, g\, t)\, (H\, g\, E_\$))\, (E_{\mathbf{join}}\, t) \\
G\, t \to E_\ell\, t \\
H\, g\, t \to E_{\mathbf{acq}_i}\, (g\, (E_{\mathbf{rel}_i}\, t))
\end{array} \right\}.
$$

The action tree in Figure 1 is generated by the following reduction sequence:

$$
\begin{aligned}
&S \longrightarrow F\, G\, E_\$ \longrightarrow E_{\mathbf{if}\_}\, (E_{\mathbf{spawn}}\, (F\, G\, E_\$)\, (H\, G\, E_\$))\, (E_{\mathbf{join}}\, E_\$) \\
&\longrightarrow E_{\mathbf{spawn}}\, (F\, G\, E_\$)\, (H\, G\, E_\$)) \longrightarrow \langle \mathsf{sp}\rangle\, (F\, G\, E_\$)\, (H\, G\, E_\$)) \\
&\longrightarrow^* \langle \mathsf{sp}\rangle\, (E_{\mathbf{if}\_}\, (E_{\mathbf{spawn}}\, (F\, G\, E_\$)\, (H\, G\, E_\$))\, (E_{\mathbf{join}}\, E_\$))\, (E_{\mathbf{acq}_i}\, (G\, (E_{\mathbf{rel}_i}\, E_\$))) \\
&\longrightarrow^* \langle \mathsf{sp}\rangle\, (E_{\mathbf{join}}\, E_\$)\, (\langle \mathsf{Acq}_i\rangle\, (G\, (E_{\mathbf{rel}_i}\, E_\$))) \longrightarrow^* \langle \mathsf{sp}\rangle\, (\langle \mathsf{jo}\rangle\, E_\$)\, (\langle \mathsf{Acq}_i\rangle\, (E_{\mathbf{rel}_i}\, E_\$)) \\
&\longrightarrow^* \langle \mathsf{sp}\rangle\, (\langle \mathsf{jo}\rangle\, \langle \$\rangle)\, (\langle \mathsf{Acq}_i\rangle\, (\langle \mathsf{Rel}_i\rangle\, E_\$)) \longrightarrow \langle \mathsf{sp}\rangle\, (\langle \mathsf{jo}\rangle\, \langle \$\rangle)\, (\langle \mathsf{Acq}_i\rangle\, (\langle \mathsf{Rel}_i\rangle\, \langle \$\rangle))
\end{aligned}
$$

Now we show that the grammar generates all the action trees that corresponds to the reachable configurations of a program. We first prepare some definitions. To clarify the relationship between an applicative term (consisting of terminals and non-terminals of a HORS) and a configuration, we first define the terms that can appear at "run-time" and thus have unique corresponding configurations.

Definition 11. *An applicative term t is called a **run-time term** if (1) t has sort o, (2) t contains no labels nor $\bot$, and (3) no terminals occur in the arguments of non-terminals in t.*

Definition 12. *Let t be a run-time term of sort o. The action tree $t^\bot$ is defined by:*

$$(E_\ell\, t)^\bot = \ell \qquad (A\, t_1\, \ldots\, t_n)^\bot = \bot \ (\text{if } A \notin \{\, E_\ell \mid \ell \in \mathsf{Label} \,\})$$
$$(a\, t_1\, \ldots\, t_n)^\bot = a\, t_1^\bot\, \ldots\, t_n^\bot$$

We extend the map $\mathcal{X}_{\pi,L,S}(\cdot)$ to that on run-time trees, by:

$$\mathcal{X}_{\pi,L,s}(\langle \mathsf{sp}\rangle\, t_1\, t_2) = \mathcal{X}_{\pi,L,s+1}(t_1) \cup \mathcal{X}_{\pi\cdot s,\epsilon,0}(t_2) \quad \mathcal{X}_{\pi,L,s}(\langle \mathsf{jo}\rangle\, t) = \mathcal{X}_{\pi,L,s}(t)$$
$$\mathcal{X}_{\pi,L,s}(\langle \$\rangle) = \emptyset \quad \mathcal{X}_{\pi,L,s}(\langle \mathsf{Acq}_i\rangle\, t) = \mathcal{X}_{\pi,L\cdot i,s}(t) \quad \mathcal{X}_{\pi,L\cdot i,s}(\langle \mathsf{Rel}_i\rangle\, t) = \mathcal{X}_{\pi,L,s}(t)$$
$$\mathcal{X}_{\pi,L,s}(t) = \{\, \pi \mapsto (\mathcal{E}^{-1}(t), L, s) \,\} \ \text{(for the other cases)}$$

Here, $\mathcal{E}^{-1}$ is the inverse of $\mathcal{E}$. We write $\mathcal{X}(t)$ for $\mathcal{X}_{\epsilon,\epsilon,0}(t)$.

The following lemmas establish the correspondence between p and $\mathcal{G}_p$. Proofs are given in Appendix A.

Lemma 2. *Suppose $S \longrightarrow_{\mathcal{G}_p}^* t$ where t is a run-time term and $t^\bot$ is join-lock sensitive. Then, $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow_p^* \mathcal{X}(t)$ holds.*

Lemma 3. *If $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow_p^* c$, then there exists a run-time term t such that $S \longrightarrow_{\mathcal{G}_p}^* t$, $\mathcal{X}(t) = c$, and $t^\bot$ is join-lock-sensitive.*

By the above lemmas, the pairwise reachability problem on p is reduced to higher-order model checking on $\mathcal{G}_p$.

Theorem 2. *Let p be a program and (ℓ_1, ℓ_2) be a pair of labels. Let L_{ℓ_1,ℓ_2} be the set $\{\, \gamma \mid \exists u_1, u_2.\gamma(u_1) = \ell_1 \wedge \gamma(u_2) = \ell_2 \wedge u_1 \neq u_2 \,\}$ of action trees. Then, $p \models \ell_1 || \ell_2$ if and only if $\mathcal{L}(\mathcal{G}_p) \not\subseteq \overline{L_{\mathsf{sensitive}}} \cup \overline{L_{\ell_1,\ell_2}}$ holds.*

Proof. Suppose $p \models \ell_1 || \ell_2$. Then there exists c such that $\{\, (S, \epsilon, 0) \,\} \longrightarrow_p^* c$ with $c(\pi_1) = (e_1^{\ell_1}, L_1, s_1)$, $c(\pi_2) = (e_2^{\ell_2}, L_2, s_2)$, and $\pi_1 \neq \pi_2$. By Lemma 3, there exists a run-time term t such that $S \longrightarrow_{\mathcal{G}_p}^* t$, $\mathcal{X}(t) = c$, and $t^\bot$ is join-lock-sensitive. By the conditions $\mathcal{X}(t) = c$ and $t^\bot$, $t^\bot \in L_{\ell_1,\ell_2}$. Since $S \longrightarrow_{\mathcal{G}_p}^* t \longrightarrow_{\mathcal{G}_p}^* t^\bot$, we have $t^\bot \in \mathcal{L}(\mathcal{G}_p) \cap L_{\mathsf{sensitive}} \cap L_{\ell_1,\ell_2}$, i.e., $\mathcal{L}(\mathcal{G}_p) \not\subseteq \overline{L_{\mathsf{sensitive}}} \cup \overline{L_{\ell_1,\ell_2}}$.

Conversely, suppose $\mathcal{L}(\mathcal{G}_p) \not\subseteq \overline{L_{\mathsf{sensitive}}} \cup \overline{L_{\ell_1,\ell_2}}$, i.e., $\gamma \in \mathcal{L}(\mathcal{G}_p) \cap L_{\mathsf{sensitive}} \cap L_{\ell_1,\ell_2}$ for some action tree γ. Then there exists a run-time term t such that $t^\bot = \gamma$ and $S \longrightarrow_{\mathcal{G}_p}^* t$. By Lemma 2, we have $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow_p^* \mathcal{X}(t)$. By the conditions $t^\bot = \gamma$ and $\gamma \in L_{\ell_1,\ell_2}$, t has two distinct sub-terms (at redex-positions) of the

form $E_{\ell_1}\, t_1$ and $E_{\ell_2}\, t_2$. Thus, there exist π_1, π_2 such that $\mathcal{X}(t)(\pi_1) = (e_1^{\ell_1}, L_1, s_1)$ and $\mathcal{X}(t)(\pi_2) = (e_2^{\ell_2}, L_2, s_2)$ with $\pi_1 \neq \pi_2$. Therefore, we have $p \models \ell_1||\ell_2$ as required. □

Because $\overline{L_{\mathsf{sensitive}}} \cup \overline{L_{\ell_1,\ell_2}}$ is a regular tree language, by Theorems 1 and 2, the pairwise reachability $p \models \ell_1||\ell_2$ is decidable.

Complexity. Recall that the order of $\mathcal{G}_p$ is $\max(\mathit{order}(p), 1)$ and the model checking of order-k HORS is k-EXPTIME [15]. Therefore, the pairwise reachability analysis for order-k programs is k-EXPTIME for $k \geq 1$. As for the lower-bound, the problem of checking whether $\ell \in \mathcal{L}(\mathcal{G})$ (where ℓ is a singleton tree consisting of the leaf ℓ) is already $(k-1)$-EXPTIME-hard [12]. Since it can be easily reduced to a pairwise reachability analysis problem, the pairwise reachability is $(k-1)$-EXPTIME-hard. It should be noted, however, that if a regular tree language L is fixed, and also if both the largest arity and order of symbols are fixed, then $\mathcal{L}(\mathcal{G}) \subseteq L$ can be decided in time linear in the size of $\mathcal{G}$ [11]. In our method, the regular tree language $\overline{L_{\mathsf{sensitive}}} \cup \overline{L_{\ell_1,\ell_2}}$ is determined by the sets Label and Lock. We can fix Label as $\{\, \ell_1, \ell_2 \,\}$ by omitting the other labels from the input program. Therefore, if we fix (i) the largest arity and order of functions in a program and (ii) the number of locks used in the program (i.e., $|\mathsf{Lock}|$), then the pairwise reachability can also be decided in time linear in the size of the program.

4 Preliminary Experiments

We carried out preliminary experiments to check the feasibility of our verification method. As the underlying model checker, we used HorSat [1]. At the time of writing this paper, we have not yet fully automated the translation from programs to HORS, but doing so is not difficult.

Table 1 shows the experimental results. The column "Order" and "# of functions" indicate the order and the number of function definitions of each program. The rightmost column shows the times spent for higher-order model checking (excluding those for translations, which can be performed instantly once automated). Note that the Reachability column shows the answers for pairwise reachability, not for the original verification problems. The benchmark program `example` has been taken from Example 1. The program `example_wrong` is a variation of `example`, obtained by omitting **join** operation of F. The other benchmark programs have been obtained by encoding exceptions, Java-style "synchronized" constructs (but with non-reentrant locks), and lists. The encoding uses higher-order functions, so it demonstrates an advantage of being able to deal with higher-order programs. For example, `exception` models the following OCaml-like program:

```
let rec read_and_update x =
  let n = read_int(x) (* may raise an Eof exception *) in
```

Table 1. The experimental results

Program	Order	# of functions	Checked pair	Reachability	Elapsed time [sec.]
`example`	2	4	(ℓ, ℓ)	no	0.16
`example_wrong`	2	4	(ℓ, ℓ)	yes	0.15
`exception`	3	7	(ℓ, ℓ)	no	0.38
`exception_wrong`	3	7	(ℓ, ℓ)	yes	0.04
`synchronized`	3	7	(ℓ_1, ℓ_1)	no	1.14
			(ℓ_1, ℓ_2)	yes	1.69
`list`	4	8	(ℓ_1, ℓ_1)	no	0.55
			(ℓ_1, ℓ_2)	no	0.73

```
    lock g; c := c+n; (* may raise an Overflow exception *)
    unlock g; read_and_update x;;
let rec f file =
  let x = open_in file in
  try read_and_update(x) with
    Eof -> close x | Overflow -> unlock g;
spawn(f("foo"));spawn(f("bar"));join();print c
```

Here, the goal of verification is to check that no race occurs on the shared variable `c`. The above program is encoded into the following order-3 program of the language in Section 2;

$$R\ h\ k = \mathbf{if}\_(h\ True)\ (\mathbf{acq}_g; (\mathbf{if}\_\ (\mathbf{rel}_g; R\ h\ k)\ (h\ False))^\ell)$$
$$F\ k = R\ H\ k \qquad H\ b = b\ \$\ (\mathbf{rel}_g; \$) \qquad P\ k = k^\ell$$
$$S = \mathbf{spawn}(F);\ (\mathbf{spawn}(F);\ (\mathbf{join}; (P\ \$))) \qquad True\ x\ y = x \qquad False\ x\ y = y$$

Here, the function R corresponds to `read_and_update`; we have abstracted away `x` and instead added an exception handler `h` and a continuation parameter `k` in order to precisely model exception primitives. The program `exception_wrong` is a variation of `exception` obtained by omitting $\mathbf{acq}_g$ and $\mathbf{rel}_g$ operations. The other benchmark programs are explained in Appendix B.

All the benchmark programs have been verified within a few seconds. Although the programs are very small, this is encouraging, considering the worst-case complexity of higher-order model checking. As discussed at the end of the previous section, the pairwise reachability can be decided in time linear *with respect to the size of HORS* under a certain assumption; thus, the results indicate that our method may scale for larger programs.

5 Related Work

As already mentioned in Section 1, the present work is based on the series of work on verification of pushdown systems with nested locking [7, 14, 4], and extends

it to deal with higher-order programs. The target language of our verification is more expressive than dynamic pushdown networks and corresponds to an extension of collapsible pushdown systems [6] (which are higher-order pushdown systems extended with collapse operations) with concurrency primitives. Gawlitza et al. [4] used a clever encoding of a configuration of a dynamic pushdown network, so that the (forward) reachable set of configurations can be represented as a regular tree language and the reachability problem can be reduced to the inclusion between regular tree languages. One of our insights was that thanks to the decidability of higher-order model checking, actually we need not represent the reachable set as a regular language; a higher-order tree language (generated by a HORS) suffices. This has enabled not only the higher-order extension, but also a conceptual simplification of the verification method in our opinion.

Higher-order model checking has recently been applied to program verification, but most of them have been for sequential programs [9, 13, 16]. Kobayashi and Igarashi [10] have shown that the reachability problem for higher-order concurrent programs with a bounded number of context switches can be reduced to higher-order model checking. (They have also shown a reduction from verification of higher-order concurrent programs to an extension of higher-order model checking, but the latter is undecidable.) Hague [5] has shown the decidability of reachability of ordered, phase-bounded and scope-bounded concurrent collapsible pushdown systems. These methods underapproximate the reachable set of ordinary higher-order concurrent programs (without the "bound" conditions). To our knowledge, there is no realistic implementation of those methods. There are also overapproximation approaches to static analysis or verification of higher-order concurrent programs [2, 3, 8].

6 Conclusion

We have shown the decidability of pairwise reachability of higher-order concurrent programs with recursion, dynamic process creation, joins, and nested locking. To our knowledge, this is the first realistic application of higher-order model checking to verification of concurrent programs. Despite the extremely high worst-case complexity of higher-order model checking, preliminary experiments show that our approach is feasible at least for small programs.

Acknowledgement We would like to thank Markus Müller-Olm for the discussion on the subject and information about dynamic pushdown networks, and anonymous referees for useful comments. We would also like to thank Taku Terao for providing his higher-order model checker for the experiments. This work was supported by JSPS Kakenhi 23220001.

References

1. C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In S. R. D. Rocca, editor, *CSL*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

2. E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. In *Proceedings of the 20th Static Analysis Symposium*, SAS'13. Springer-Verlag, 2013.
3. J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1), 2005.
4. T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2011.
5. M. Hague. Saturation of concurrent collapsible pushdown systems. In *Proceedings of FSTTCS 2013*, volume 24 of *LIPIcs*, pages 313–325, 2013.
6. M. Hague, A. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 452–461. IEEE Computer Society, 2008.
7. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 505–518. Springer, 2005.
8. N. Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 20th Anniversary Colloquium*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2003.
9. N. Kobayashi. Model checking higher-order programs. *J. ACM*, 60(3):20, 2013.
10. N. Kobayashi and A. Igarashi. Model checking higher-order programs with recursive types. In *Proceedings of ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 392–411. Springer, 2013.
11. N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*, pages 179–188, 2009.
12. N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011.
13. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of PLDI 2011*, pages 222–233, 2011.
14. P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2009.
15. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006.
16. C.-H. L. Ong and S. Ramsay. Verifying higher-order programs with pattern-matching algebraic data types. In *Proceedings of POPL 2011*, pages 587–598, 2011.
17. G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
18. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
19. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
20. R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *Proceedings of PEPM 2013*, pages 53–62, 2013.

Appendix

A Proofs for Section 3

In this section, we will prove Lemmas 2 and 3.

An applicative term can be viewed as a map from paths to actions or terms.

Definition 13. *Each applicative term t on Σ of the sort o can be regarded as a map from $\mathbb{N}_+^*$ to $dom(\Sigma)$ or applicative terms, inductively defined as follows.*

$$\begin{aligned}(a\,t_1\,\cdots\,t_n)(\epsilon) &= a\\ (A\,t_1\,\cdots\,t_n)(\epsilon) &= A\,t_1\,\cdots\,t_n\\ (a\,t_1\,\cdots\,t_i\,\cdots\,t_n)(i\cdot\pi) &= t_i(\pi)\end{aligned}$$

Let us write $\lfloor t\rfloor^\perp$ for the action tree obtained from a run-time term t by replacing all terms of the form $A\,\tilde{t'}$ by $\perp$. Note that $t^\perp$ and $\lfloor t\rfloor^\perp$ only differ in the positions u with $t(u) = E_\ell\,t'$. By definition, $\lfloor t\rfloor^\perp$ is join-lock sensitive if and only if $t^\perp$ is.

For a reduction context C, let us define the **hole position** of C by the position u such that $C(u) = [\,]$. Then, we have the following lemma.

Lemma 4. *Let C be a reduction context. Then, there exist c, π, L, s such that $\mathcal{X}(C[t]) = c \uplus \mathcal{X}_{\pi,L,s}(t)$ holds for all run-time terms t, and $\pi = \mathsf{pn}(u, \lfloor C[t]\rfloor^\perp)$ where u is the hole position of C.*

Above, we implicitly use the fact that $C[t]$ is a run-time term if t is.

Given a configuration c, we write $\mathsf{erase}(c)$ for the abstract configuration obtained from the configuration c by removing expression parts. The following lemma relates $\mathcal{X}(\cdot)$ and $\theta(\cdot)$.

Lemma 5. *For a run-time term t, $\mathsf{erase}(\mathcal{X}(t)) = \theta(\lfloor t\rfloor^\perp)$ holds.*

The following lemma allows us to re-order the reduction sequence.

Lemma 6. *Let t_0, t_1, t_2 be run-time terms such that $t_0 \longrightarrow_{\mathcal{G}_p} t_1 \longrightarrow_{\mathcal{G}_p} t_2$ holds. Assume that the first reduction occurs at the position u_1 and the second reduction occurs at the position u_2. If u_1 is not a prefix of u_2, then there exists a run-time term t_1' such that $t_0 \longrightarrow_{\mathcal{G}_p} t_1' \longrightarrow_{\mathcal{G}_p} t_2$ holds and that the first reduction occurs at u_2 and the second reduction occurs at u_1.*

Proof. By the assumption, t_0 must be of the form $C[s_1, s_2]$ where C is a context with two holes at u_1 and u_2, with $t_1 = C[s_1', s_2]$, $t_2 = C[s_1', s_2']$, and $s_i \longrightarrow_{\mathcal{G}_p} s_i'$ (for $i = 1, 2$). Thus, the required result holds for $t_1' = C[s_1, s_2']$. □

The following lemma plays an important role to prove Lemma 2. Roughly speaking, this lemma says that, if we can derive a run-time term t such that $t^\perp$ is join-lock sensitive, then we have a reduction sequence that leads to t according to a valid execution history specified by $t^\perp$.

Lemma 7. *Suppose $S \longrightarrow^*_{\mathcal{G}_p} t$ where t is a run-time term and $t^\perp$ is join-lock sensitive. Then, there exists a sequence of run-time terms $S = t_0 \longrightarrow_{\mathcal{G}_p} t_1 \longrightarrow_{\mathcal{G}_p} \ldots \longrightarrow_{\mathcal{G}_p} t_n = t$ such that, for all $1 \leq i \leq n$,*

- *$\lfloor t_{i-1} \rfloor^\perp = \lfloor t_i \rfloor^\perp$ or $\lfloor t_{i-1} \rfloor^\perp \overset{u}{\rightsquigarrow} \lfloor t_i \rfloor^\perp$, and*
- *if $\lfloor t_{i-1} \rfloor^\perp \overset{u}{\rightsquigarrow} \lfloor t_i \rfloor^\perp$, then $\theta(\lfloor t_{i-1} \rfloor^\perp) \xrightarrow{\mathsf{pn}(u, \lfloor t \rfloor^\perp), \lfloor t \rfloor^\perp(u)} \theta(\lfloor t_i \rfloor^\perp)$ holds.*

Proof. Since $t^\perp$ is join-lock sensitive, there is a sequence $\perp = \gamma_0 \overset{u_1}{\rightsquigarrow} \gamma_1 \overset{u_2}{\rightsquigarrow} \ldots \overset{u_m}{\rightsquigarrow} \gamma_m = \lfloor t \rfloor^\perp$ such that $\theta(\gamma_0) \xrightarrow{\mathsf{pn}(u_1, \lfloor t \rfloor^\perp(u_1)), \lfloor t \rfloor^\perp(u_1)} \theta(\gamma_1) \xrightarrow{\mathsf{pn}(u_2, \lfloor t \rfloor^\perp(u_2)), \lfloor t \rfloor^\perp(u_2)} \ldots \xrightarrow{\mathsf{pn}(u_m, \lfloor t \rfloor^\perp(u_m)), \lfloor t \rfloor^\perp(u_m)} \theta(\gamma_m)$. By Lemma 6, we have

$$S = t_0 = t'_0 \longrightarrow^*_{\mathcal{G}_p} t'_1 \longrightarrow^*_{\mathcal{G}_p} t'_2 \longrightarrow^*_{\mathcal{G}_p} \ldots \longrightarrow^*_{\mathcal{G}_p} t'_m \longrightarrow^*_{\mathcal{G}_p} t \qquad (*)$$

where all the reductions in the subsequence $t'_i \longrightarrow^*_{\mathcal{G}_p} t'_{i+1}$ occur at u_{i+1} and the reductions in the subsequence $t'_m \longrightarrow^*_{\mathcal{G}_p} t$ occur at leaf positions. By construction, all the terms t' that occur in $t'_i \longrightarrow^*_{\mathcal{G}_p} t'_{i+1}$ except t'_{i+1} satisfy $\lfloor t' \rfloor^\perp = \gamma_i$. Thus, gathering all the terms that occur in the reduction sequence (*), we obtain the sequence satisfying the required conditions. □

Now, we are ready to prove Lemma 2.

Proof of Lemma 2. By Lemma 7, we have a sequence of run-time terms $S = t_0 \longrightarrow_{\mathcal{G}_p} t_1 \longrightarrow_{\mathcal{G}_p} \ldots \longrightarrow_{\mathcal{G}_p} t_n = t$ such that, for all $1 \leq k \leq n$,

- $\lfloor t_{k-1} \rfloor^\perp = \lfloor t_i \rfloor^\perp$ or $\lfloor t_{k-1} \rfloor^\perp \overset{u}{\rightsquigarrow} \lfloor t_k \rfloor^\perp$, and
- if $\lfloor t_{k-1} \rfloor^\perp \overset{u}{\rightsquigarrow} \lfloor t_k \rfloor^\perp$, then $\theta(\lfloor t_{k-1} \rfloor^\perp) \xrightarrow{\mathsf{pn}(u, \lfloor t \rfloor^\perp), \lfloor t \rfloor^\perp(u)} \theta(\lfloor t_k \rfloor^\perp)$ holds.

Then, we prove $\mathcal{X}(t_0) \longrightarrow_p \ldots \longrightarrow_p \mathcal{X}(t_k)$ for any k ($0 \leq k \leq n$) by the induction on k.

For the base case where $k = 0$, the statement trivially holds.

Let us consider the step case where $k > 0$. We only discuss the two representative cases: (1) $t_{k-1} = C[F\,\widetilde{t'}]$ and $t_k = C[\,[\widetilde{t'}/\tilde{x}]\mathcal{E}(e)\,]$ for a rule $F\,\tilde{x} \to e$, and (2) $t_{k-1} = C[E_{\mathbf{acq}_i}\,t']$ and $t_k = C[\langle \mathsf{Acq}_i \rangle\,t']$. The other cases can be proved similarly to either of the two cases.

Case (1). By Lemma 4, we have $\mathcal{X}(t_{k-1}) = c \uplus \{\, \pi \mapsto (F\,\mathcal{E}^{-1}(\widetilde{t'}), L, s) \,\}$, and $\mathcal{X}(t_k) = c \uplus \{\, \pi \mapsto ([\mathcal{E}^{-1}(\widetilde{t'})/\tilde{x}]e, L, s) \,\}$. Thus, we conclude that $\mathcal{X}(t_{k-1}) \longrightarrow_p \mathcal{X}(t_k)$. Note that, in this case, we have $\lfloor t_k \rfloor^\perp = \lfloor t_{k-1} \rfloor^\perp$ and $\theta(\lfloor t_k \rfloor^\perp) = \theta(\lfloor t_{k-1} \rfloor^\perp)$. A Similar discussion can be applicable to the cases where $t_{k-1} = C[E_{\mathbf{if}\_}\,t'_1\,t'_2]$ and $t_{k-1} = C[E_\ell\,t'_1]$, and thus we shall omit the proofs.

Case (2). By Lemma 4, we have $\mathcal{X}(t_{k-1}) = c \uplus \{\, \pi \mapsto (\mathbf{acq}_i; \mathcal{E}^{-1}(t'), L, s) \,\}$ and $\mathcal{X}(t_k) = c \uplus \{\, \pi \mapsto (\mathcal{E}^{-1}(t'), L \cdot i, s) \,\}$. Here, $\lfloor t_{k-1} \rfloor^\perp \overset{u}{\rightsquigarrow} \lfloor t_k \rfloor^\perp$ where u is the hole position of C. By Lemma 4, we have $\pi = \mathsf{pn}(u, \lfloor t_{k-1} \rfloor^\perp) = \mathsf{pn}(u, \lfloor t \rfloor^\perp)$. Thus, we have $\theta(\lfloor t_{k-1} \rfloor^\perp) \xrightarrow{\pi, \langle \mathsf{Acq}_i \rangle} \theta(\lfloor t_k \rfloor^\perp)$. By Lemma 5, we have $\mathsf{erase}(\mathcal{X}(t_{k-1})) \xrightarrow{\pi, \langle \mathsf{Acq}_i \rangle} \mathsf{erase}(\mathcal{X}(t_k))$. Thus, we have $\mathcal{X}(t_{k-1}) \longrightarrow_p \mathcal{X}(t_k)$. A Similar discussion can be

applicable to the cases where $t_{k-1} = C[E_{\mathbf{rel}_i}\, t']$, $t_{k-1} = C[E_{\mathbf{spawn}}\, t'_1\, t'_2]$, $t_{k-1} = C[E_{\mathbf{join}}\, t']$ and $t_{k-1} = C[E_{\$}]$, and thus we shall omit the proofs. □

Then, we switch to prove Lemma 3. The following lemma is a key to prove Lemma 3.

Lemma 8. *Suppose that $c \longrightarrow_p c'$ and $\mathcal{X}(t) = c$ for some run-time term t such that $\lfloor t \rfloor^{\perp}$ is join-lock sensitive. Then, we have run-time term t' such that $t \longrightarrow_{\mathcal{G}_p} t'$, $\mathcal{X}(t') = c'$ and $\lfloor t' \rfloor^{\perp}$ is join-lock sensitive.*

Proof. We prove the lemma by the case analysis on $\longrightarrow_p$.

Consider the case where $c = c'' \uplus \{\, \pi \mapsto (F\, e_1, \ldots, e_k, L, s) \,\}$ and $c' = c'' \uplus \{\, \pi \mapsto ([e_1/x_1, \ldots, e_k/x_k]e, L, s) \,\}$ where p has the rule $F\, x_1\, \ldots\, x_k = e$. Since $\mathcal{X}(t) = c$, there are $t_1, \ldots, t_k$ such that $t = C[F\, t_1\, \ldots\, t_k]$ and $\mathcal{E}^{-1}(t_i) = e_i$ for any i. From the definition of $\mathcal{G}_p$, we can reduce the term t to $t' = C[\,[t_1/x_1, \ldots, t_k/x_k]\mathcal{E}(e)\,]$. Note that $\mathcal{E}(e)$ does not contain terminals by its definition, and, since t is a run-time term, $t_1, \ldots, t_n$ do not contain terminals either. Thus, we have $\mathcal{X}(t') = c'' \uplus \{\, \pi \mapsto \mathcal{E}^{-1}([t_1/x_1, \ldots, t_k/x_k]\mathcal{E}(e)) \,\}$. Since $\mathcal{E}$ does only renaming and $\mathcal{E}^{-1}$ is the inverse of $\mathcal{E}$, we have $\mathcal{E}^{-1}([t_1/x_1, \ldots, t_k/x_k]\mathcal{E}(e)) = e[\mathcal{E}^{-1}(t_1)/x_1, \ldots, \mathcal{E}^{-1}(t_k)/x_k]$, and thus $\mathcal{X}(t') = c'$. Note that we have $\lfloor t \rfloor^{\perp} = \lfloor t' \rfloor^{\perp}$ and $\mathsf{erase}(c) = \mathsf{erase}(c')$ in this case. The cases where $c = c'' \uplus \{\, \pi \mapsto (\mathbf{if}\_\, e_1\, e_2, L, s) \,\}$, and where $c = c'' \uplus \{\, \pi \mapsto (e^{\ell}, L, s) \,\}$ can be proved similarly, and thus we shall omit proofs for these cases.

Consider the case where $c = c'' \uplus \{\, \pi \mapsto (\mathbf{acq}_i; e, L, s) \,\}$ and $c' = c'' \uplus \{\, \pi \mapsto (e, L \cdot i, s) \,\}$. In this case, since $\mathcal{X}(t) = c$, t must be of the form $C[E_{\mathbf{acq}_i}\, t_1]$. Let t' be $C[\langle \mathsf{Acq}_i \rangle\, t_1]$. Then, it satisfies $\mathcal{X}(t') = c'$ and $t \longrightarrow_{\mathcal{G}_p} t'$. Now, we discuss the join-lock sensitivity of $\lfloor t' \rfloor^{\perp}$. We have $\lfloor t \rfloor^{\perp} \overset{u}{\rightsquigarrow} \lfloor t' \rfloor^{\perp}$, where u is the hole position of C. By Lemma 4, we have $\pi = \mathsf{pn}(u, \lfloor t' \rfloor^{\perp})$. Since we have $c \longrightarrow_p c'$, we have $\mathsf{erase}(c) \xrightarrow{\pi, \langle \mathsf{Acq}_i \rangle} \mathsf{erase}(c')$. By Lemma 5, we have $\theta(\lfloor t \rfloor^{\perp}) \xrightarrow{\pi, \langle \mathsf{Acq}_i \rangle} \theta(\lfloor t' \rfloor^{\perp})$. Thus, we conclude that $\lfloor t' \rfloor^{\perp}$ is join-lock sensitive. The cases where $c = c'' \uplus \{\, \pi \mapsto (\mathbf{rel}_i; e, L, s) \,\}$, $c = c'' \uplus \{\, \pi \mapsto (\mathbf{spawn}(e_2); e_1, L, s) \,\}$, $c = c'' \uplus \{\, \pi \mapsto (\mathbf{join}; e_1, L, s) \,\}$ and $c = c'' \uplus \{\, \pi \mapsto (\$, L, s) \,\}$ can be proved similarly and thus we shall omit proofs for these cases. □

Now, we are ready to prove Lemma 3.

Proof of Lemma 3. Since we have $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} \longrightarrow_p^* c$, we have a sequence $\{\, \epsilon \mapsto (S, \epsilon, 0) \,\} = c_0 \longrightarrow_p c_1 \longrightarrow_p \ldots \longrightarrow_p c_n = c$. We have $\mathcal{X}(S) = c_0$, and $\lfloor S \rfloor^{\perp} = \perp$ is clearly join-lock sensitive. Then, applying Lemma 8 inductively on the derivation length, we obtain a run-time term t such that $\mathcal{X}(t) = c$ and $\lfloor t \rfloor^{\perp}$ is join-lock sensitive. Thus, by definition, $t^{\perp}$ is join-lock sensitive. □

B Benchmark Programs

Here we explain the other benchmark programs used in the experiments described in Section 4.

The program `synchronized` is:

$$
\begin{aligned}
S &= \mathbf{spawn}(F\,\$);\ \mathbf{spawn}(F\,\$);\ \mathbf{join};\ \$ \\
G_1\,f\,k &= \mathbf{acq}_1; (f(\mathbf{rel}_1; k)) \\
G_2\,f\,k &= \mathbf{acq}_2; (f(\mathbf{rel}_2; k)) \\
O_1\,k &= k^{\ell_1} \\
O_2\,k &= k^{\ell_2} \\
P\ o\ g\ k &= g\ o\ k \\
F\,k &= \mathbf{if}\_\,(P\,O_1\,G_1\,k)\,(P\,O_2\,G_2\,k)
\end{aligned}
$$

It models the following Java-like program:

```
Object g1,g2;
put_message(msg, out, g) = {
  synchronize(g){
    out.println(msg);
  }
}
f() = {
  if (<some_condition>) {
    put_message <result_message> System.out g1;
  }else{
    put_message <error_message> System.err g2;
  }
}
main() = spawn(f());spawn(f());join;
```

In the former program, the higher-order function G_i models the construct `synchronize`, which acquires and releases lock i respectively before and after executing f. The goal of the verification is to check

1. whether no race occurs on the same same output functions O_i and
2. whether O_1 and O_2 can be executed concurrently.

Note that these conditions can be checked independently. The answers to them are yes.

The program `list` is:

$$
\begin{aligned}
S &= F\ (Cons\ P_1\ (Cons\ P_1\ (Cons\ P_2\ (Cons\ P_2\ Nil)))) \ \$ \\
U\ p\ k &= \mathbf{acq}_i;\ (p\ (p\ (\mathbf{rel}_i;\ k))) \\
F\ l\ k &= l\ k\ G \\
G\ p\ k &= (\mathbf{spawn}(U\ p\ \$);\ k)) \\
P_1\ k &= k^{\ell_1} \\
P_2\ k &= k^{\ell_2} \\
Nil\ n\ c &= n \\
Cons\ x\ xs\ n\ c &= c\ x\ (xs\ n\ c)
\end{aligned}
$$

It models the following O'Caml like program, using higher-order functions to encode list operations.

```
let p1 = ref 1 in let p2 = ref 2 in
let update_and_print p =
    (lock g;
    p := !p+n;
    print !p;
    unlock g);;
let rec f ps =
  match ps with
    []->()
    | (p:ps) -> (spawn(update_and_print p); f ps);;
main = f [p1;p1;p2;p2];
```

In the former program, the higher-order function P_i models the variable accesses to p_i in `update_and_print`. The goal of the verification is to check

1. whether no race occurs on the same variable access P_i and
2. whether P_1 and P_2 can be executed concurrently.

The answer is yes for the former question but no for the latter.