Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References

Kohei Suenaga

Tohoku University

Abstract. We present a type-based deadlock-freedom verification for concurrent programs with non-block-structured lock primitives and mutable references. Though those two features are frequently used, they are not dealt with in a sufficient manner by previous verification methods. Our type system uses a novel combination of *lock levels*, *obligations* and *ownerships*. Lock levels are used to guarantee that locks are acquired in a specific order. Obligations and ownerships guarantee that an acquired lock is released exactly once.

1 Introduction

Concurrent programs are getting important as multi-processor machines and clusters are getting popular. Many programs including operating systems and various network servers are written as concurrent programs.

A problem with a concurrent program is the possibility of a deadlock: a state in which every thread is waiting for a lock to be released by other threads. A deadlock is considered to be a serious problem since a deadlock causes unintentional halt of a system.

This paper presents a type-based method for deadlock-freedom verification. Our verification framework supports *non-block-structured lock primitives* and *mutable references to locks*. Those two features are heavily used in real-world software. For example, non-block-structured lock primitives, whose locking operations do not syntactically correspond to unlocking operations, are used in, for example, C programs with POSIX thread library.

Figure 1 shows a program with non-block-structured lock primitives and mutable references to locks, which suffers from a deadlock. The example is based on an actual bug found in nss_ldap-226-20.rpm [5]. In that example, a function `_nss_ldap_getgroups_dyn` first calls `_nss_ldap_enter` and then executes two branches. The first branch calls `_nss_ldap_leave` before executing `return` , while the second branch does not call `_nss_ldap_leave` . Because `_nss_ldap_enter` acquires a global lock `__lock` and returns without unlocking it, `__lock` is kept acquired if the second branch is executed after `_nss_ldap_getgroups_dyn` returns. This causes a deadlock if `_nss_ldap_getgroups_dyn` is called twice with an environment under which the second branch is executed.

```
static mutex_t __lock;
void _nss_ldap_enter() { ... mutex_lock(&__lock); ... }
void _nss_ldap_leave() { ... mutex_unlock(&__lock); ... }
char *_nss_ldap_getgroups_dyn(const char *user...) {
  ...
  _nss_ldap_enter ();
  if (...) { _nss_ldap_leave(); return NULL; }
  /* _nss_ldap_leave is not called in this branch. */
  if (...) { return NSS_STATUS_NOTFOUND; }
  ...
}
```

Fig. 1. A deadlock contained in nss_ldap-226-20.rpm.

```
static mutex_t lockA, lockB;
                                    void thread1() {
void accessA() {                      mutex_lock(&lockB);
  mutex_lock(&lockA);                 accessA();
  ...                                 mutex_unlock(&lockB);
  mutex_unlock(&lockA);             }
}                                   void thread2() {
void accessB() {                      mutex_lock(&lockA);
  mutex_lock(&lockB);                 accessB();
  ...                                 mutex_unlock(&lockA);
  mutex_unlock(&lockB);             }
}
```

Fig. 2. An example of a deadlock caused by circular dependency between locks.

Figure 2 shows another example of a deadlock, which is caused by circular dependency between locks. In the example, a function `accessA` acquires and releases a lock `lockA` , while `accessB` acquires and releases `lockB` . These two functions are called from two threads `thread1` and `thread2` . In those threads, `accessA` is called while `lockB` is acquired and `accessB` is called while `lockA` is acquired, so that the program may lead to a deadlock because of the lock order reversal between `lockA` and `lockB` .

The main idea of our type system is to guarantee that locks are acquired in a specific order, and that an acquired lock is released exactly once. The first property is guaranteed by *lock levels*, while the second property is guaranteed by *obligations* and *ownerships*.

So far, much effort has been paid for static deadlock-freedom verification [1, 2, 7, 8, 11, 12, 14]. However, non-block-structured lock primitives and mutable references to locks are not dealt with in a sufficient manner. For example, the analyses by Boyapati, Lee and Rinard [2] and by Flanagan and Abadi [8] consider only block-structured synchronization primitives (i.e., `synchronized` blocks in the Java language.) Kobayashi et al. [11, 12, 14] proposed a deadlock-freedom analy-

$$
\begin{array}{rcl}
x, y, z, f \ldots & \in & \mathit{Var} \\
\mathit{lck} & ::= & \mathbf{L} \mid \hat{\mathbf{L}} \\
P & ::= & \widetilde{D}s \\
D & ::= & x(\widetilde{y}) = s \\
v & ::= & \mathbf{true} \mid \mathbf{false} \\
s & ::= & \mathbf{skip} \mid x(\widetilde{y}) \mid (\mathbf{if}\ x\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) \mid \mathbf{putob}(x, y) \\
 & \mid & \mathbf{let}\ x = v\ \mathbf{in}\ s \mid \mathbf{let}\ x = \mathbf{ref}\ y\ \mathbf{in}\ s \mid \mathbf{let}\ x = !y\ \mathbf{in}\ s \mid x := y \\
 & \mid & \mathbf{spawn}\ s \mid \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s \mid \mathbf{lock}\ x \mid \mathbf{unlock}\ x \mid s_1; s_2 \\
E & ::= & [\,] \mid E; s
\end{array}
$$

Fig. 3. Syntax

$$
\frac{x' \text{ is fresh}}{(\widetilde{D}, \mathit{Env}, H, L, \{\mathbf{let}\ x = \mathbf{ref}\ y\ \mathbf{in}\ s\} \uplus S) \rightarrow (\widetilde{D}, \mathit{Env}, H[x' \mapsto y], L, \{[x'/x]s\} \uplus S)} \quad \text{(E-Ref)}
$$

$$
(\widetilde{D}, \mathit{Env}, H[x \mapsto y'], L, \{x := y\} \uplus S) \rightarrow (\widetilde{D}, \mathit{Env}, H[x \mapsto y], L, \{\mathbf{skip}\} \uplus S) \quad \text{(E-Assign)}
$$

$$
\begin{array}{c}
(\widetilde{D}, \mathit{Env}, H[y \mapsto z], L, \{\mathbf{let}\ x = !y\ \mathbf{in}\ s\} \uplus S) \rightarrow \\
(\widetilde{D}, \mathit{Env}, H[y \mapsto z], L, \{([z/x]s); \mathbf{putob}(z, y)\} \uplus S)
\end{array} \quad \text{(E-LetDeref)}
$$

$$
\frac{x' \text{ is fresh}}{\begin{array}{c}(\widetilde{D}, \mathit{Env}, H, L, \{\mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s\} \uplus S) \rightarrow \\ (\widetilde{D}, \mathit{Env}, H, L[x' \mapsto \hat{\mathbf{L}}], \{[x'/x]s\} \uplus S)\end{array}} \quad \text{(E-LetNewlock)}
$$

$$
(\widetilde{D}, \mathit{Env}, H, L[x \mapsto \hat{\mathbf{L}}], \{\mathbf{lock}\ x\} \uplus S) \rightarrow (\widetilde{D}, \mathit{Env}, H, L[x \mapsto \mathbf{L}], \{\mathbf{skip}\} \uplus S) \quad \text{(E-Lock)}
$$

$$
(\widetilde{D}, \mathit{Env}, H, L[x \mapsto \mathbf{L}], \{\mathbf{unlock}\ x\} \uplus S) \rightarrow (\widetilde{D}, \mathit{Env}, H, L[x \mapsto \hat{\mathbf{L}}], \{\mathbf{skip}\} \uplus S) \quad \text{(E-Unlock)}
$$

$$
\frac{(\widetilde{D}, \mathit{Env}, H, L, \{s\} \uplus S) \rightarrow (\widetilde{D}', \mathit{Env}', H', L', \{s'\} \uplus S')}{(\widetilde{D}, \mathit{Env}, H, L, \{E[\mathbf{spawn}\ s]\} \uplus S) \rightarrow (\widetilde{D}', \mathit{Env}', H', L', \{s, E[\mathbf{skip}]\} \uplus S')} \quad \text{(E-Spawn)}
$$

$$
(\widetilde{D}, \mathit{Env}, H, L, \mathbf{putob}(x, y)) \rightarrow (\widetilde{D}, \mathit{Env}, H, L, \mathbf{skip}) \quad \text{(E-Putob)}
$$

Fig. 4. Operational Semantics (excerpt)

sis for the π-calculus. Although their analysis can, in principle, handle references by encoding them into channels, the resulting analysis is too imprecise.

The rest of this paper is organized as follows. Section 2 defines our target language. Section 3 introduces a type system for deadlock-freedom analysis. Section 4 states soundness of our type system. After discussing related work in Section 5, we conclude in Section 6.

```
_nss_ldap_enter(__lock) = lock __lock
_nss_ldap_leave(__lock) = unlock __lock
_nss_ldap_getgroups_dyn(ret, cond, __lock) =
  _nss_ldap_enter(__lock);
  if cond then (_nss_ldap_leave(__lock); ret := 0)
  else ret := 0
```

Fig. 5. An encoding of the program in Figure 1 in our language.

2 Target Language

Figure 3 shows the syntax of our target language. A program P consists of mutually recursive function definitions $\widetilde{D}$ and a main statement s. A function definition D consists of the name of the function x, a sequence of arguments $\widetilde{y}$ and a function body s.

The meta-variable s ranges over the set of statements. The statement **skip** does nothing. The statement $x(\widetilde{y})$ is a function call. The conditional branch **if** x **then** s_1 **else** s_2 executes s_1 if x is **true** and s_2 otherwise. The statements **let** x = **ref** y **in** s, **let** x =!y **in** s and $x := y$ are for generating, dereferencing and assignment to a reference. We write **let** $x = y$ **in** s for **let** z = **ref** y **in let** x =!z **in** s if the variable z does not freely appear in s. The statement **spawn** s spawns a new thread that executes s. **putob**(x, y), which is used for type soundness proof and operationally equivalent to **skip**, represents the end of scope of x in **let** x =!y **in** s. **putob**(x, y) should not be included in a program. The statement **let** x = **newlock** () **in** s generates a fresh lock, binds x to the lock and executes s. The statements **lock** x and **unlock** x are for acquiring and releasing a lock x. The statement $s_1; s_2$ is a sequential composition of s_1 and s_2. Figure 5 and 6 show how the programs in Figure 1 and 2 are encoded in our language. We omit the main statement of the program in Figure 1.

The operational semantics of our calculus is defined as a transition relation between configurations. Figure 4 presents an excerpt of the transition rules. A configuration in our semantics is a tuple of a set of function definitions $\widetilde{D}$, an environment that maps a variable to a value Env, a heap H, a map from a lock variable to a state of the lock L and a multiset of running threads S. A state of a lock is either locked ($\mathbf{L}$) or unlocked ($\hat{\mathbf{L}}$.) In Figure 4, $S_1 \uplus S_2$ is the disjoint union of multisets S_1 and S_2.

3 Type System

3.1 Overview

We first present an overview of our type system. As mentioned in Section 1, our type system guarantees that locks are acquired in a specific order by using *lock*

```
$accessA(lockA) = \textbf{lock}\ lockA; \textbf{unlock}\ lockA$
$accessB(lockB) = \textbf{lock}\ lockB; \textbf{unlock}\ lockB$
$thread1(lockA, lockB) = \textbf{lock}\ lockB; accessA(lockA); \textbf{unlock}\ lockB$
$thread2(lockA, lockB) = \textbf{lock}\ lockA; accessB(lockB); \textbf{unlock}\ lockA$
$\textbf{let}\ lockA = \textbf{newlock}()\ \textbf{in let}\ lockB = \textbf{newlock}()\ \textbf{in}$
  $\textbf{spawn}\ (thread1(lockA, lockB)); \textbf{spawn}\ (thread2(lockA, lockB))$
```

Fig. 6. An encoding of the program in Figure 2 in our language.

```
$f(x, y) = \textbf{unlock}\ x; \textbf{unlock}\ y$
$main() = \textbf{let}\ x = \textbf{newlock}()\ \textbf{in}$
             $\textbf{lock}(x); (\textbf{let}\ y = \textbf{ref}\ x\ \textbf{in}\ f(x, !y))$
```

Fig. 7. Programs that contain aliasing to a lock occurs.

levels and that an acquired lock is released exactly once by using *obligations* and *ownerships*.

Lock levels Each lock type in our type system is associated with a natural number called lock level. The type system prevents deadlocks by guaranteeing that locks are acquired in a strict increasing order of lock levels. For example, the statement **spawn** (**lock** x; **lock** y; **unlock** y; **unlock** x);
(**lock** y; **lock** x; **unlock** x; **unlock** y) is rejected because the first thread requires the level of x to be less than that of y, while the second thread requires the level of y should be less than x.

Obligations In order to guarantee that an acquired lock is released exactly once, each lock type in our type system has information on *obligation* to release the lock. A lock type **lock**(lev, U) in our type system has a flow-sensitive component U called a usage, in addition to a lock level lev. A usage is either ob, which denotes an obligation to release the lock, or $\mathbf{1}$, which shows there is no such obligation.

More precisely, the type system deals with obligations based on the following principles.

1. **lock** x can be executed if and only if (1) x does not have an obligation and (2) the level of every lock with an obligation is less than the level of x. x has an obligation after **lock** x is performed.
2. **unlock** x can be performed if and only if x has an obligation. x does not have the obligation after **unlock** x is performed.
3. An obligation is treated *linearly*, that is, if an alias to a lock with an obligation is generated, then exactly one of the lock or the alias inherits the obligation.

$$
\begin{array}{l}
\textbf{let } x = \textbf{newlock}() \textbf{ in let } y = \textbf{newlock}() \textbf{ in} \\
\textbf{let } z = \textbf{newlock}() \textbf{ in let } r = \textbf{ref } x \textbf{ in} \\
\quad \textbf{spawn } (\textbf{lock}(z); \textbf{lock}(!r); \textbf{unlock}(z); \textbf{lock}(z); \textbf{unlock}(!r); \textbf{unlock}(z)); \\
\quad (\textbf{lock}(z); r := y; \textbf{unlock}(z))
\end{array}
$$

Fig. 8. A program in which a lock contained in a reference is not correctly released.

For example, the type system rejects the program

$$
\begin{array}{l}
f(x) = \textbf{lock } x \\
\quad \textbf{let } x = \textbf{newlock}() \textbf{ in } f(x); \textbf{lock } x
\end{array}
$$

because x has an obligation after the function call $f(x)$ returns, which is followed by **lock** x, so that (1) in the first condition above is violated. The program in Figure 7 is also rejected because, from the third condition above, only one of x or y inherits the obligation generated by **lock** x after the reference y is generated, while both x and y are required to have an obligation just before $f(x, !y)$.

Note the difference between our obligations and flow-sensitive type qualifiers in CQual [9]. Flow-sensitive type qualifiers in CQual represent *the current state of values*, while our obligations represent *how variables should be used afterwards.* This difference matters when we consider a program with aliasing. For example, consider the program ($\textbf{lock } x; \textbf{let } y = x \textbf{ in } s$). In our type system, x has the type $\textbf{lock}(\mathit{lev}, \mathit{ob})$ just after **lock** x, which means the lock should be released *through x afterwards.* After the alias y of x is created, the type environment may be either $x : \textbf{lock}(\mathit{lev}, \mathit{ob}), y : \textbf{lock}(\mathit{lev}, \mathbf{1})$ or $x : \textbf{lock}(\mathit{lev}, \mathbf{1}), y : \textbf{lock}(\mathit{lev}, \mathit{ob})$ depending on how x and y are used in s. On the other hand, in CQual, if x is put in an abstract location ρ, then the flow-sensitive type qualifier assigned to ρ just after **lock** x is `locked`, which means that x is *currently* locked. After the alias y is created, x and y have the same type as they are bound to the same lock.

Ownership In order to guarantee deadlock-freedom of a program with thread creation and accesses to mutable references, obligations are still insufficient to guarantee that an acquired lock is released exactly once. For example, consider the program in Figure 8. That program consists of two threads. The first thread acquires and releases a lock contained in the reference r, while the second thread assigns another lock to the same reference. Then, the lock released by the first thread may be different from acquired one, so that the acquired lock may not be released.

The problem here can be described as follows: a write to a reference to a lock should not occur *while the lock is held.* Note that this property differs from race-freedom because race-freedom only guarantees that a write to a reference and another read or write to the reference do not occur at the same time. In fact, though the program in Figure 8 is race-free because each access to the reference r is guarded by a lock z, it still has a problem described above.

$$\begin{array}{rcl} lev & \in & \{0,1,\ldots\} \cup \{\infty\} \\ U & ::= & ob \mid \mathbf{1} \\ r & \in & [0,\infty) \\ \tau & ::= & \mathbf{bool} \mid \mathbf{lock}(lev, U) \mid \tau\ \mathbf{ref}^r \mid (\tau_1,\ldots,\tau_n)\overset{lev}{\rightarrow}(\tau'_1,\ldots,\tau'_n) \end{array}$$

Fig. 9. Syntax of types.

To solve such problem, our type system uses *ownerships*, a thread's capability to access a reference. As in Boyland [3], Terauchi [17] and Kikuchi and Kobayashi [10] do, we use rational-numbered ownerships. A well-typed program obeys the following rules on ownerships in manipulating references.

1. An ownership less than or equal to 1 is assigned to a reference to a lock when the reference is generated.
2. A thread is required to have an ownership greater than 0 on a reference in order to read a lock from the reference.
3. A thread is required to have an ownership 1 on a reference in order to write a lock to the reference.
4. When a thread is spawned, an ownership of each reference is divided and distributed to each thread.

Based on those rules, a thread has to have an ownership greater than 0 to acquire a lock through a reference, which prevents other threads from overwriting the reference while the lock is acquired. For example, the program in Figure 8 is rejected because the total ownership required on the reference r exceeds 1: the first thread requires an ownership more than 0 while the second thread requires 1.

3.2 Syntax

Figure 9 shows the syntax of types. The set of *lock levels*, ranged over by a meta-variable lev, is the set of natural numbers with ∞. We extend the standard partial order $\leq$ on the set of natural numbers to that on lock levels by $lev \leq \infty$ for any lev. We write $lev_1 < lev_2$ for $lev_1 \leq lev_2 \wedge lev_1 \neq lev_2$.

Usage, ranged over by a meta-variable U, represents whether there is an obligation to release a lock. A usage ob represents an obligation to release a lock, while a usage $\mathbf{1}$ represents that there is not such obligation.

The meta variable τ ranges over types. A lock type $\mathbf{lock}(lev, U)$ is for locks that should be used according to lev and U. For example, if a variable x has the type $\mathbf{lock}(1, \mathbf{1})$, then the lock can be acquired through x if locks whose levels are more than 1 are not already acquired. If a variable x has the type $\mathbf{lock}(1, ob)$ then the lock should be released exactly once through the variable x.

The type $\tau\ \mathbf{ref}^r$ is for references, whose content should be used according to τ after it is read from the reference. The meta variable r, which is associated with a reference type, is a rational number in the set $[0,\infty)$ and represents a

$\boxed{U_1 \otimes U_2}$

$$\begin{aligned} ob \otimes ob &= \text{undefined} \\ ob \otimes \mathbf{1} &= ob \\ \mathbf{1} \otimes ob &= ob \\ \mathbf{1} \otimes \mathbf{1} &= \mathbf{1} \end{aligned}$$

$\boxed{\tau_1 \otimes \tau_2}$

$$\tau_1\ \mathbf{ref}^{r_1} \otimes \tau_2\ \mathbf{ref}^{r_2} = \tau_1 \otimes \tau_2\ \mathbf{ref}^{r_1+r_2}$$

$$\mathbf{lock}(lev, U_1) \otimes \mathbf{lock}(lev, U_2) = \mathbf{lock}(lev, U_1 \otimes U_2)$$

$$\tau \otimes \tau = \tau$$

(where τ is **int**, **bool**, or a function type.)

$\boxed{\Gamma_1 \otimes \Gamma_2}$

$$(\Gamma_1 \otimes \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & (\text{if } x \in \mathbf{Dom}(\Gamma_1)\backslash\mathbf{Dom}(\Gamma_2)) \\ \Gamma_2(x) & (\text{if } x \in \mathbf{Dom}(\Gamma_2)\backslash\mathbf{Dom}(\Gamma_1)) \\ \Gamma_1(x) \otimes \Gamma_2(x) & (\text{if } x \in \mathbf{Dom}(\Gamma_1) \cap \mathbf{Dom}(\Gamma_2)) \end{cases}$$

$\boxed{noob(U), noob(\tau)}$

$$noob(\mathbf{1}) \qquad \text{(NoOb-Unlocked)}$$

$$\frac{\tau \text{ is } \mathbf{bool} \text{ or a function type.}}{noob(\tau)} \qquad \text{(NoOb-Other)}$$

$$\frac{noob(U)}{noob(\mathbf{lock}(lev, U))} \qquad \text{(NoOb-Lock)}$$

$$\frac{noob(\tau)}{noob(\tau\ \mathbf{ref}^r)} \qquad \text{(NoOb-Ref)}$$

Fig. 10. Definition of auxiliary operators and predicates.

thread's capability to access the reference. An ownership being greater than 0 means that one can read a value through the reference. An ownership 1 means that one can write a value to the reference. A function type $\widetilde{\tau}\xrightarrow{lev}\widetilde{\tau}'$ consists of the following components.

- $\widetilde{\tau}$: the types of arguments before execution of the functions.
- $\widetilde{\tau}'$: the types of arguments after execution of the functions.
- lev: the minimum level of locks that may be acquired by the functions.

3.3 Type Judgment

The type judgment for statements is $\Gamma \vdash s \Rightarrow \Gamma' \ \&\ lev$. Type environments Γ and Γ' describe the types of free variables in s before and after execution of s. A lock level lev is an effect of s, which is a minimum level of locks that may be acquired during execution of s.

The type judgment intuitively means that (1) locks are acquired in an strict increasing order of their levels, (2) an acquired lock is released exactly once and, (3) the levels of acquired locks are greater than or equal to lev if s is executed under an environment described by Γ and with a continuation that respects types in Γ'.

The type judgment is defined as the least relation that satisfies the typing rules in Figure 11. In those rules, we use the operation $\otimes$ defined in Figure 10.

$$\emptyset \vdash \mathbf{skip} \Rightarrow \emptyset \mathrel{\&} \infty \qquad \text{(T-Nop)}$$

$$\frac{\Gamma_1 \vdash s_1 \Rightarrow \Gamma_2 \mathrel{\&} \mathit{lev} \qquad \Gamma_2 \vdash s_2 \Rightarrow \Gamma_3 \mathrel{\&} \mathit{lev}}{\Gamma_1 \vdash s_1; s_2 \Rightarrow \Gamma_3 \mathrel{\&} \mathit{lev}} \qquad \text{(T-Seq)}$$

$$\frac{x : \mathbf{bool}, \Gamma \vdash s \Rightarrow x : \mathbf{bool}, \Gamma' \mathrel{\&} \mathit{lev}}{\Gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ s \Rightarrow \Gamma' \mathrel{\&} \mathit{lev}} \qquad \text{(T-LetBool)}$$

$$\frac{x : \mathbf{lock}(\mathit{lev}', \mathbf{1}), \Gamma \vdash s \Rightarrow x : \mathbf{lock}(\mathit{lev}', \mathbf{1}), \Gamma' \mathrel{\&} \mathit{lev}}{\Gamma \vdash \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s \Rightarrow \Gamma' \mathrel{\&} \mathit{lev}} \qquad \text{(T-Newlock)}$$

$$x : \mathbf{lock}(\mathit{lev}, \mathbf{1}) \vdash \mathbf{lock}\ x \Rightarrow x : \mathbf{lock}(\mathit{lev}, \mathit{ob}) \mathrel{\&} \mathit{lev} \qquad \text{(T-Lock)}$$

$$x : \mathbf{lock}(\mathit{lev}, \mathit{ob}) \vdash \mathbf{unlock}\ x \Rightarrow x : \mathbf{lock}(\mathit{lev}, \mathbf{1}) \mathrel{\&} \infty \qquad \text{(T-Unlock)}$$

$$\frac{\tau = (\tau_1, \ldots, \tau_n) \xrightarrow{\mathit{lev}} (\tau_1', \ldots, \tau_n')}{y_1 : \tau_1 \otimes \cdots \otimes y_n : \tau_n \otimes x : \tau \vdash x(y_1, \ldots, y_n) \Rightarrow y_1 : \tau_1' \otimes \cdots \otimes y_n : \tau_n' \otimes x : \tau \mathrel{\&} \mathit{lev}} \qquad \text{(T-App)}$$

$$\frac{x : \mathbf{bool}, \Gamma \vdash s_1 \Rightarrow \Gamma' \mathrel{\&} \mathit{lev} \qquad x : \mathbf{bool}, \Gamma \vdash s_2 \Rightarrow \Gamma' \mathrel{\&} \mathit{lev}}{x : \mathbf{bool}, \Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \Rightarrow \Gamma' \mathrel{\&} \mathit{lev}} \qquad \text{(T-If)}$$

$$\frac{\begin{array}{c} x : \tau_1\ \mathbf{ref}^{r}, y : \tau_2, \Gamma \vdash s \Rightarrow x : \tau_1'\ \mathbf{ref}^{r'}, y : \tau_2', \Gamma' \mathrel{\&} \mathit{lev} \qquad \mathit{noob}(\tau_1') \\ r \leq 1 \text{ if } \neg \mathit{nolock}(\tau_1) \qquad \mathbf{wf}(\tau_1\ \mathbf{ref}^{r}) \qquad \mathbf{wf}(\tau_2) \end{array}}{y : \tau_1 \otimes \tau_2, \Gamma \vdash \mathbf{let}\ x = \mathbf{ref}\ y\ \mathbf{in}\ s \Rightarrow y : \tau_2', \Gamma' \mathrel{\&} \mathit{lev}} \qquad \text{(T-Ref)}$$

$$\frac{\begin{array}{c} x : \tau_1, y : \tau_2\ \mathbf{ref}^{r}, \Gamma \vdash s \Rightarrow x : \tau_1', y : \tau_2'\ \mathbf{ref}^{r''}, \Gamma' \mathrel{\&} \mathit{lev} \qquad r' > 0 \\ \mathbf{wf}(\tau_1) \qquad \mathbf{wf}(\tau_2\ \mathbf{ref}^{r}) \end{array}}{y : \tau_1 \otimes \tau_2\ \mathbf{ref}^{r+r'}, \Gamma \vdash \mathbf{let}\ x = !y\ \mathbf{in}\ s \Rightarrow y : \tau_1' \otimes \tau_2'\ \mathbf{ref}^{r''}, \Gamma' \mathrel{\&} \mathit{lev}} \qquad \text{(T-Deref)}$$

$$\frac{\mathit{noob}(\tau_3) \qquad \mathbf{wf}(\tau_1) \qquad \mathbf{wf}(\tau_2)}{x : \tau_3\ \mathbf{ref}^{1}, y : \tau_1 \otimes \tau_2 \vdash x := y \Rightarrow x : \tau_1\ \mathbf{ref}^{1}, y : \tau_2 \mathrel{\&} \infty} \qquad \text{(T-Assign)}$$

$$x : \tau_1 \otimes \tau_2, y : \tau_3\ \mathbf{ref}^{r} \vdash \mathbf{putob}(x, y) \Rightarrow x : \tau_1, y : \tau_2 \otimes \tau_3\ \mathbf{ref}^{r} \mathrel{\&} \infty \qquad \text{(T-Putob)}$$

$$\frac{\Gamma_1 \vdash s \Rightarrow \Gamma_3 \mathrel{\&} \mathit{lev} \qquad \mathit{noob}(\Gamma_3) \qquad \mathbf{wf}(\Gamma_1) \qquad \mathbf{wf}(\Gamma_2)}{\Gamma_1 \otimes \Gamma_2 \vdash \mathbf{spawn}\ s \Rightarrow \Gamma_2 \mathrel{\&} \mathit{lev}} \qquad \text{(T-Spawn)}$$

$$\frac{\Gamma \vdash s \Rightarrow \Gamma' \mathrel{\&} \mathit{lev} \qquad \mathit{lev}' \leq \mathit{lev} \qquad \mathbf{max}(\mathit{level}_{ob}(\Gamma'')) < \mathit{lev}'}{\Gamma, \Gamma'' \vdash s \Rightarrow \Gamma', \Gamma'' \mathrel{\&} \mathit{lev}'} \qquad \text{(T-Weak)}$$

Fig. 11. Typing rules for statements.

$U_1 \otimes U_2$ gives the usage that means *both* obligations in U_1 and U_2 have to be fulfilled. $\mathit{ob} \otimes \mathit{ob}$ is undefined because releasing an acquired lock twice is prohibited. The operator $\otimes$ on usages are naturally extended to types.

We also use the following definitions in Figure 11.

$$\frac{\begin{array}{c}\widetilde{D} = \{f(x_{11},\dots,x_{1m_1}) = s_1,\dots,f(x_{n1},\dots,x_{nm_n}) = s_m\}\\ \Gamma = f_1 : (\tau_{1,1},\dots,\tau_{1,m_1}) \xrightarrow{lev_1} (\tau'_{1,1},\dots,\tau'_{1,m_1}),\dots,\\ f_n : (\tau_{n,1},\dots,\tau_{n,m_n}) \xrightarrow{lev_n} (\tau_{n,1},\dots,\tau_{n,m_n})\\ \Gamma, x_{i,1}:\tau_{i,1},\dots,x_{i,m_i}:\tau_{i,m_i} \vdash s_i \Rightarrow \Gamma, x_{i,1}:\tau'_{i,1},\dots,x_{i,m_i}:\tau'_{i,m_i} \;\&\; lev_i\end{array}}{\vdash_{Def} \widetilde{D} : \Gamma} \qquad \text{(T-}\textsc{Fundef})$$

$$\frac{\vdash_{Def} \widetilde{D} : \Gamma \qquad \Gamma \vdash s \Rightarrow \Gamma' \;\&\; lev \qquad noob(\Gamma')}{\vdash_{Prog} \widetilde{D}s} \qquad \text{(T-}\textsc{Prog})$$

Fig. 12. Typing rules for programs.

Definition 1 (No obligation). *$noob(\tau)$ is defined as the least predicate that satisfies the rules in Figure 10.*

The predicate $noob(\tau)$ asserts that τ does not have any obligation to fulfil.

Definition 2. *$level_U$, a function that takes a type and returns a set of lock levels, is defined as follows.*

$$\begin{array}{rl} level_U(\tau\ \mathbf{ref}^r) = level_U(\tau) & \\ level_U(\mathbf{lock}(lev, U')) = \{lev\} & \quad (\textit{where } U = U') \\ level_U(\tau) = \emptyset & \quad (\textit{otherwise}) \end{array}$$

$level_U(\Gamma)$ is defined as $\{lev \mid x : \tau \in \Gamma \wedge lev \in level_U(\tau)\}$.

The function $level_U$ collects levels of locks whose usages are equal to U.

Definition 3. *A predicate* $\mathbf{wf}$ *is defined as the least one that satisfies the following rules.*

$$\frac{\begin{array}{c}\tau \textit{ is } \mathbf{bool}, \mathbf{lock}(lev, U)\\ \textit{or a function type.}\end{array}}{\mathbf{wf}(\tau)} \qquad \frac{\mathbf{wf}(\tau) \quad \neg noob(\tau) \Rightarrow r > 0}{\mathbf{wf}(\tau\ \mathbf{ref}^r)} \qquad \frac{\forall x : \tau \in \Gamma. \mathbf{wf}(\tau)}{\mathbf{wf}(\Gamma)}$$

The predicate $\mathbf{wf}(\Gamma)$ asserts that ownerships of each reference type in Γ are consistent with its content type. Note that $\mathbf{wf}(\tau\ \mathbf{ref}^r)$ requires $r > 0$ if τ has an obligation to release a lock because one has to read the reference to release the lock.

We explain important rules in Figure 11. In the rule (T-Newlock), $noob(U_1)$ means that the newly generated lock has no obligation. $noob(U_2)$ means that all the obligations in the type of x should be fulfilled at the end of s because x cannot be accessed after execution of s.

(T-Lock) guarantees that there is no obligation before execution of **lock** x. After execution of **lock** x, x has an obligation to release the lock.

In the rule (T-Ref), we use a predicate $nolock(\tau)$. This predicate holds if and only if τ does not contain lock types as its component. The rule (T-Ref)

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathit{\Gamma}_3 \vdash \textbf{unlock } z \Rightarrow \mathit{\Gamma}_4}}{\mathit{\Gamma}_1 \vdash \textbf{let } z = !x \textbf{ in unlock } z \Rightarrow \mathit{\Gamma}_2} \quad \mathit{noob}(\textbf{lock}(1, \mathbf{1}))}{y : \textbf{lock}(1, \mathit{ob}) \vdash \textbf{let } x = \textbf{ref } y \textbf{ in let } z = !x \textbf{ in unlock } z \Rightarrow y : \textbf{lock}(1, \mathbf{1})}$$

Fig. 13. A derivation tree of **let** x = **ref** y **in unlock** x under the assumption y : **lock**$(1, \mathit{ob})$.

states that the ownership assigned to the new reference x in typing s is less than or equal to 1 if the type of x contains lock types as its component. At the end of s, x should not have any obligation because x cannot be accessed after execution of s.

In the rule (T-Deref), if y has an obligation to release a lock, only one of x and y inherits that obligation during execution of s. The rule (T-Deref) also states that the ownership assigned to the type of y should be greater than 0.

A derivation of a statement **let** x = **ref** y **in let** $z = !x$ **in unlock** z under the type environment y : **lock**$(1, \mathit{ob})$ in Figure 13 shows how (T-Ref) and (T-Deref) work. The type environments $\mathit{\Gamma}_1, \dots, \mathit{\Gamma}_4$ in that figure are defined as follows.

$$\begin{aligned}
\mathit{\Gamma}_1 &= x : \textbf{lock}(1, \mathit{ob})\ \textbf{ref}^1, y : \textbf{lock}(1, \mathbf{1}) \\
\mathit{\Gamma}_2 &= x : \textbf{lock}(1, \mathbf{1})\ \textbf{ref}^1, y : \textbf{lock}(1, \mathbf{1}) \\
\mathit{\Gamma}_3 &= x : \textbf{lock}(1, \mathbf{1})\ \textbf{ref}^{1-r}, y : \textbf{lock}(1, \mathbf{1}), z : \textbf{lock}(1, \mathit{ob}) \\
\mathit{\Gamma}_4 &= x : \textbf{lock}(1, \mathbf{1})\ \textbf{ref}^{1-r}, y : \textbf{lock}(1, \mathbf{1}), z : \textbf{lock}(1, \mathbf{1}).
\end{aligned}$$

Here, r is an arbitrary rational number in the set $(0, 1)$. Note that the obligation of y is passed to the newly generated reference x, delegated to z and fulfilled through z.

The rule (T-Assign) guarantees that there is no obligation that must be fulfilled through the reference x because x is being overwritten. If y has an obligation, then either x or y inherits that obligation after execution of $x := y$. For example, both

$$\begin{aligned}
x : \textbf{lock}(0, \mathbf{1})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathit{ob}) \vdash x := y \Rightarrow x : \textbf{lock}(0, \mathbf{1})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathit{ob}) \\
x : \textbf{lock}(0, \mathbf{1})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathit{ob}) \vdash x := y \Rightarrow x : \textbf{lock}(0, \mathit{ob})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathbf{1})
\end{aligned}$$

hold. After the assignment, the obligation originally owned by y should be fulfilled through y in the first case, while it should be fulfilled through the reference x in the second case. However,

$$x : \textbf{lock}(0, \mathbf{1})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathit{ob}) \vdash x := y \Rightarrow x : \textbf{lock}(0, \mathit{ob})\ \textbf{ref}^1, y : \textbf{lock}(0, \mathit{ob})$$

does not hold.

In the rule (T-Spawn), the pre type environment of the conclusion part is split into $\mathit{\Gamma}_1$ and $\mathit{\Gamma}_2$. The environment $\mathit{\Gamma}_1$ is for the newly generated thread s, while $\mathit{\Gamma}_2$ is for the continuation of **spawn** s. The condition $noob(\mathit{\Gamma}_3)$ imposes that all the obligations in $\mathit{\Gamma}_1$ should be fulfilled in the newly generated thread s.

The rule (T-Weak) is for adding redundant variables to type environments. In that rule, the condition $\mathbf{max}(level_{ob}(\mathit{\Gamma}''))<lev'$ guarantees that if newly added lock-typed variables have obligations, then the levels of those lock types ($level_{ob}(\mathit{\Gamma}'')$) should be less than the level of locks that may be acquired in s (lev'). With this condition, we can guarantee that locks are acquired in a strict increasing order of lock levels.

The type judgment for programs $\vdash_{Prog} \widetilde{D}s$ is defined as the least relation that satisfies the rules in Figure 12. The rule (T-Prog) states that a program $\widetilde{D}s$ is well-typed if (1) the defined functions have the types described in a type environment $\mathit{\Gamma}$ and (2) the main statement s is well-typed under $\mathit{\Gamma}$ and (3) all the obligations generated during execution of the program are fulfilled after execution of s. The rule (T-Fundef), which is a rule for function definitions, guarantees that each function has the type described in $\mathit{\Gamma}$.

Example In the program in Figure 5, the function *_nss_ldap_leave* has type $\mathbf{lock}(lev, ob)\xrightarrow{lev}\mathbf{lock}(lev,\mathbf{1})$ where lev is an arbitrary natural number. Thus, *__lock* in the body of *_nss_ldap_getgroups_dyn* has type $\mathbf{lock}(lev,\mathbf{1})$ at the end of the first branch and $\mathbf{lock}(lev, ob)$ at the end of the second branch, which violates the condition of (T-If) that type environments at the end of two branches have to agree. In the example in Figure 6, the condition $\mathbf{max}(level_{ob}(\mathit{\Gamma}''))<lev'$ in (T-Weak) imposes that the level of *lockB* has to be less than that of *lockA* in the body of *thread1*. For the same reason, the level of *lockA* has to be less than that of *lockB* in the body of *thread2*, so that the program is ill-typed.

3.4 Type Inference

We informally describe a type inference algorithm in this section. Our algorithm is a standard constraint-based one; the algorithm takes a program as input, generates a constraint set based on the typing rules in Figure 11 and reduces those constraints.

We omit an explanation on the constraint generation phase which is done in a standard manner. A generated constraint is either (1) $lexp_1 \leq lexp_2$, (2) $\rho = ob \Rightarrow lexp_1 \leq lexp_2$, (3) $\rho = Uexp_1 \otimes \cdots \otimes Uexp_n$ or (4) a linear inequality on ownerships. Here, $lexp$ and $Uexp$ are defined by the following syntax.

$$\begin{aligned} lexp &::= \phi \text{ (lock level variables) } \mid \infty \mid lexp + 1 \\ Uexp &::= \rho \text{ (usage variables) } \mid ob \mid \mathbf{1}. \end{aligned}$$

Generated constraints are reduced as follows. First, linear inequalities on ownerships are solved using an external solver. Then, constraints of the form $\rho = Uexp_1 \otimes \cdots \otimes Uexp_n$ are reduced to a substitution on usage variables. This is done by applying a standard constraint reduction algorithm for linear type systems (e.g., one presented in [13].) By applying the obtained substitution to

$$\frac{\vdash_{Def} \widetilde{D} : \Gamma_{fun} \qquad \Gamma_{fun}(x) = \tau}{\vdash (\widetilde{D} \uplus \{x(\widetilde{y}) = s\}, \mathit{Env}, H, L, x) : \tau, \emptyset, \emptyset} \quad \text{(TC-FUNDEF)}$$

$$\vdash (\widetilde{D}, \mathit{Env} \uplus \{x \mapsto v\}, H, L, x) : \mathbf{bool}, \emptyset, \emptyset \quad \text{(TC-ENV)}$$

$$\frac{\vdash (\widetilde{D}, \mathit{Env}, H \uplus \{x \mapsto x'\}, L, x') : \tau, P, O}{\vdash (\widetilde{D}, \mathit{Env}, H \uplus \{x \mapsto x'\}, L, x) : \tau\ \mathbf{ref}^{r}, P, O \otimes \{x \mapsto r\}} \quad \text{(TC-HEAP)}$$

$$\vdash (\widetilde{D}, \mathit{Env}, H, L \uplus \{x \mapsto \mathit{lck}\}, x) : \mathbf{lock}(\mathit{lev}, U), \{x \mapsto U\}, \emptyset \quad \text{(TC-LOCKSTATE)}$$

$$\frac{\begin{array}{c} \forall x \in \mathbf{Dom}(\widetilde{D} \cup \mathit{Env} \cup H \cup L).\ \vdash (\widetilde{D}, \mathit{Env}, H, L, x) : \Gamma(x), P_x, O_x \\ P = \bigotimes_{x \in \mathbf{Dom}(\widetilde{D} \cup \mathit{Env} \cup H \cup L)} P_x \qquad O = \bigotimes_{x \in \mathbf{Dom}(\widetilde{D} \cup \mathit{Env} \cup H \cup L)} O_x \\ \forall x \in \mathbf{Dom}(L).\ \text{if } L(x) = \hat{\mathbf{L}} \text{ then } P(x) = \mathbf{1} \text{ else } P(x) = \mathit{ob} \\ \forall y \in \mathbf{Dom}(H). \neg \mathit{nolock}(\Gamma(y)) \Longrightarrow O(y) \leq 1 \end{array}}{\vdash_E (\widetilde{D}, \mathit{Env}, H, L) : \Gamma} \quad \text{(T-ENV)}$$

$$\frac{\begin{array}{c} \vdash_E (\widetilde{D}, \mathit{Env}, H, L) : (\Gamma_1 \otimes \cdots \otimes \Gamma_n) \qquad \Gamma_i \vdash s_i \Rightarrow \Gamma_i' \ \&\ \mathit{lev}_i \\ \mathbf{wf}(\Gamma_i) \qquad \mathit{noob}(\Gamma_i') \quad (s_i \in S) \end{array}}{\vdash_{\mathit{Conf}} (\widetilde{D}, \mathit{Env}, H, L, S)} \quad \text{(T-CONFIG)}$$

Fig. 14. Typing rules for configurations.

constraints of the form $\rho = \mathit{ob} \Rightarrow \mathit{lexp}_1 \leq \mathit{lexp}_2$, we obtain a constraint set of the form $\{\mathit{lexp}_1 \leq \mathit{lexp}_1', \ldots, \mathit{lexp}_1 \leq \mathit{lexp}_n'\}$. This constraint set on lock levels can be solved in the same way as Kobayashi's deadlock-freedom analysis [11].

4 Type Soundness

This section states soundness of the type system introduced in the previous section. The proof of the soundness statement will appear in the full version of the current paper.

Because a deadlock is expressed as a stuck state in our language, soundness of the type system introduced in the previous section is stated as follows.

Theorem 1 (Type soundness). *If* $\vdash_{Prog} \widetilde{D}s$ *and* $(\widetilde{D}, \emptyset, \emptyset, \emptyset, \{s\}) \rightarrow^* (\widetilde{D}', \mathit{Env}', H', L', S')$, *then* $S = \emptyset$ *or there exists a configuration* $(\widetilde{D}'', \mathit{Env}'', H'', L'', S'')$ *that satisfies* $(\widetilde{D}', \mathit{Env}', H', L', S') \rightarrow (\widetilde{D}'', \mathit{Env}'', H'', L'', S'')$.

To state lemmas that are used in the proof of the theorem above, we first introduce a type judgment for configurations. Type judgments $\vdash_{\mathit{Conf}} (\widetilde{D}, \mathit{Env}, H, L, S)$, $\vdash_E (\widetilde{D}, \mathit{Env}, H, L) : \Gamma$ and $\vdash (\widetilde{D}, \mathit{Env}, H, L, x) : \tau, P$ are defined as the least relation that satisfies the rules in Figure 14. Here, the meta-variable P represents a map from lock-typed variables to usages and is used to describe which variable has an obligation to release each lock. The meta-variable O is a map from reference-typed variables to ownerships and used for calculating the sum of ownerships assigned to each reference. Operators $P_1 \otimes P_2$ and $O_1 \otimes O_2$ are defined as follows.

$$(P_1 \otimes P_2)(x) = \begin{cases} P_1(x) \quad (x \in \mathbf{Dom}(P_1) \backslash \mathbf{Dom}(P_2)) \\ P_2(x) \quad (x \in \mathbf{Dom}(P_2) \backslash \mathbf{Dom}(P_1)) \\ P_1(x) \otimes P_2(x) \quad (x \in \mathbf{Dom}(P_1) \cap \mathbf{Dom}(P_2)) \end{cases}$$

$$(O_1 \otimes O_2)(x) = \begin{cases} O_1(x) \quad (x \in \mathbf{Dom}(O_1) \backslash \mathbf{Dom}(O_2)) \\ O_2(x) \quad (x \in \mathbf{Dom}(O_2) \backslash \mathbf{Dom}(O_1)) \\ O_1(x) + O_2(x) \quad (x \in \mathbf{Dom}(O_1) \cap \mathbf{Dom}(O_2)) \end{cases}$$

The judgment $\vdash (\widetilde{D}, Env, H, L, x) : \tau, P, O$ means that (1) x has a type τ under $\widetilde{D}, Env, H$ and L, (2) x or a value reachable from x through H has obligations to release a lock y if $P(y) = ob$ and (3) references reachable from x are assigned ownerships as in O. By using this judgment, the rule (T-Env) guarantees that, for each held lock, there exists exactly one variable that is reachable to the lock and that has the obligation to release the lock. The rule (T-Config) guarantees that each obligation is fulfilled by exactly one thread.

The theorem above is proved using the following three lemmas.

Lemma 1. *$\vdash_{Prog} \widetilde{D}s$ implies $\vdash_{Conf} (\widetilde{D}, \emptyset, \emptyset, \emptyset, \{s\})$.*

Lemma 2 (Preservation). *If $\vdash_{Conf} (\widetilde{D}, Env, H, L, S)$ and $(\widetilde{D}, Env, H, L, S) \rightarrow (\widetilde{D}', Env', H', L', S')$, then $\vdash_{Conf} (\widetilde{D}', Env', H', L', S')$.*

Lemma 3 (Progress). *If $\vdash_{Conf} (\widetilde{D}, Env, H, L, S)$ then $S = \emptyset$ or there exists a configuration $(\widetilde{D}', Env', H', L', S')$ such that $(\widetilde{D}, Env, H, L, S) \rightarrow (\widetilde{D}', Env', H', L', S')$.*

5 Related Work

Kobayashi et al. [11, 12, 14] proposed type systems for deadlock-freedom of π-calculus processes. Their idea is (1) to express how each channel is used by a *usage expression* and (2) to add *capability levels* and *obligation levels* to the inferred usage expressions in order to detect circular dependency among input/output operations to channels. Our usages can be seen as a simplified form of their usage expressions; following their encoding [11], $\mathbf{lock}(lev, ob)$ corresponds to $()/ * I^{\infty}_{lev}.O^{lev}_{\infty}$ and $\mathbf{lock}(lev, \mathbf{1})$ to $()/O^{lev}_{\infty} | * I^{\infty}_{lev}.O^{lev}_{\infty}$. Their verification method is applicable to programs which use various synchronization primitives other than mutexes because they use π-calculus as their target language. However, their framework does not have references as primitives and cannot deal with references encoded using channels accurately.

Boyapati, Lee and Rinard [2] proposed a type-based deadlock- and race-freedom verification of Java programs. In our previous work [16], we have proposed a type-based deadlock-freedom analysis for concurrent programs with block-structured lock primitives, references and interrupts. The main difference between those type systems and our type system is that our type system deals with non-block-structured lock primitives, while their type system only deals with block-structured lock primitives.

Foster, Terauchi and Aiken [9] proposed a type system with flow-sensitive type qualifiers [9] and applied their type system to an analysis which checks locks are not doubly acquired nor released. Their type system adds a flow-sensitive type qualifier (`locked` or `unlocked` in their lock usage analysis) to each abstract memory location which contains locks, and checks whether qualifiers are in an expected state. They check that each locking operation is followed by an unlocking operation but do not guarantee deadlock-freedom. They do not deal with concurrency, either. As discussed in Section 3.1, the meaning of our obligations differs from that of their flow-sensitive type qualifiers.

6 Conclusion

We have proposed a type-based deadlock-freedom verification method for concurrent programs with non-block-structured lock primitives and references. Our type system verifies deadlock-freedom by guaranteeing that locks are acquired in a specific order by using lock levels and that an acquired lock is released exactly once by using obligations and ownerships.

Future work includes conducting deadlock-freedom verification experiments of practical software. We have implemented a prototype of a verifier based on our framework and have successfully verified deadlock-freedom of a network device driver. We are trying to apply our verifier to larger software such as network servers.

Another future work is to extend our framework with several practical features such as interrupts, recursive types and synchronization primitives other than mutexes. We are especially interested in dealing with interrupts which are essential in verifying low-level software such as operating system kernels as pointed out in several papers [4, 6, 15, 16]. We consider extending usages with information on whether the lock may be held while interrupts are enabled.

Acknowledgement We thank Naoki Kobayashi for his fruitful comments on our research. We also thank the members of Kobayashi-Sumii group in Tohoku University and anonymous reviewers for their comments. This research is partially supported by Grant-in-Aid for JSPS Fellows (19·1504) and JSPS Research Fellowships for Young Scientists.

References

1. Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, March 2006.

2. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA 2002)*, volume 37 of *SIGPLAN Notices*, pages 211–230, November 2002.
3. John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium (SAS 2003)*, pages 55–72, 2003.
4. Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. *Information and Computation*, 194(2):144–174, 2004.
5. Nalin Dahyabhai. *Bugzilla Bug 439215: dbus-daemon-1 hangs when using the option nss_initgroups_ignoreusers in /etc/ldap.conf with the user root.* Red Hat, Inc., March 2008. `https://bugzilla.redhat.com/show_bug.cgi?id=439215` (accessed on June 19th, 2008).
6. Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Programming Language Design and Implementation (PLDI)*, June 2008.
7. Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.
8. Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of 8the European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, March 1999.
9. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2002.
10. Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Proceedings of the Fifth ASIAN Symposium on Programming Languages and Systems*, November 2007.
11. Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.
12. Naoki Kobayashi. A new type system for deadlock-free processes. In *Proceedings of the 17th International Conference on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247, August 2006.
13. Naoki Kobayashi. Substructural type systems for program analysis. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, volume 4989, page 14, April 2008. Slides available from `http://www.kb.ecei.tohoku.ac.jp/~koba/slides/FLOPS2008.pdf`.
14. Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, August 2000.
15. Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proceedings of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 291–310, September 2002.
16. Kohei Suenaga and Naoki Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Proceedings of 16th European Symposium on Programming (ESOP 2007)*, pages 490–504, March 2006.
17. Tachio Terauchi. *Types for Deterministic Concurrency.* PhD thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, August 2006.