Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts

Kohei Suenaga[1] and Naoki Kobayashi[2]

[1] University of Tokyo
[2] Tohoku University

Abstract. The final goal of our research project is to establish a type-based method for verification of certain critical properties (such as deadlock- and race-freedom) of operating system kernels. As operating system kernels make heavy use of threads and interrupts, it is important that the method can properly deal with both of the two features. As a first step towards the goal, we formalize a concurrent calculus equipped with primitives for threads and interrupts handling. We also propose a type system that guarantees deadlock-freedom in the presence of interrupts. To our knowledge, ours is the first type system for deadlock-freedom that can deal with both thread and interrupt primitives.

1 Introduction

The goal of our research project is to establish a type-based method for verification of certain critical properties (such as deadlock- and race-freedom) of operating system kernels. As operating system kernels make heavy use of threads and interrupts, it is important that the method can properly deal with both of the two features. Though several calculi that deal with either interrupts [3, 14] or concurrency [12, 13] have been proposed, none of them deal with both.

Combination of those two features can actually cause errors which are very difficult to find manually. For example, consider the program in Figure 1. The example is taken from an implementation of a protocol stack used in an ongoing research project on cluster computing [11]. Though the original source code is written in C, the example is shown in an ML-style language. The function `flush_buffer` flushes the local buffer and sends pending packets to appropriate destinations. The function `receive_data` is called when a packet arrives. That function works as an interrupt handler (as specified in the main expression) and is asynchronously called whenever a packet arrives. Since `receive_data` calls `flush_buffer` in order for the local buffer to be flushed as soon as the function knows there is a room in the remote buffer (a similar mechanism called *congestion control* is used in TCP), the following control flow causes deadlock:

Call to `flush_buffer` → `lock(devlock)`
→ an interrupt (call to `receive_data`)
→ call to `flush_buffer` → `lock(devlock)`

```
let flush_buffer devlock =
  let data = dequeue () in
  while !data != NULL do
    (* Interrupts should be forbidden before this lock operation *)
    lock(devlock);
    ... (* send data to the device *) ...
    unlock(devlock);
    data := dequeue ()
  done

(* interrupt handler *)
let receive_data packettype data devlock =
  ...
  (* If there is room in the remote buffer, flush the local buffer *)
  if packettype = RoomInBuffer then
    flush_buffer devlock
  ...

(* main *)
let _ =
  (* set receive_data as an interrupt handler *)
  request_irq(receive_data);
  flush_buffer (get_devlock ())
```

Fig. 1. An example of program which cause deadlock.

Note that an interrupt handler does not voluntarily yield. To prevent the deadlock, `flush_buffer` has to forbid interrupts before it acquires the device lock as shown in Figure 2.

In order to statically detect such a deadlock, we propose (1) a calculus which is equipped with both interrupts and concurrency and (2) a type system for verifying deadlock-freedom. To our knowledge, ours is the first type system for deadlock-freedom that can deal with both thread and interrupt primitives.

Our type system associates a totally-ordered *lock level* to each lock and guarantees that locks are acquired in an increasing order of the levels even if interrupts occur. To achieve this, the type system tracks (1) a lower bound of the levels of locks acquired during evaluation and (2) an upper bound of the levels of locks acquired while interrupts are enabled. With our type system, the example in Figure 1 is rejected. On the other hand, if `flush_buffer` forbids interrupts before it acquires the device lock (as in Figure 2), our type system accepts the program.

The outline of this paper is as follows. Section 2 introduces the syntax and the semantics of our calculus. Section 3 shows our type system and states the type soundness theorem. After discussing related work in Section 4, we conclude in Section 5.

```
let flush_buffer devlock =
  let data = dequeue () in
  while !data != NULL do
    disable_interrupt(); lock(devlock);
    ... (* send data to the device *) ...
    unlock(devlock); enable_interrupt(); data := dequeue ()
  done
```

Fig. 2. A correct version of `flush_buffer`.

$$
\begin{array}{rcl}
x, y, z, f \ldots & \in & \mathit{Var} \\
\mathit{lck} & ::= & \textbf{acquired} \mid \textbf{released} \\
P & ::= & \widetilde{D} M \\
D & ::= & x(\widetilde{y}) = M \\
M & ::= & () \mid n \mid x \mid \textbf{true} \mid \textbf{false} \\
 & \mid & x(\widetilde{v}) \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \mid \textbf{if}\ v\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \\
 & \mid & \textbf{let}\ x = \textbf{ref}\ v\ \textbf{in}\ M \mid x := v \mid !v \\
 & \mid & (M_1 \mid M_2) \mid \textbf{let}\ x = \textbf{newlock}\ ()\ \textbf{in}\ M \\
 & \mid & \textbf{sync}\ x\ \textbf{in}\ M \mid \textbf{in\_sync}\ x\ \textbf{in}\ M \\
 & \mid & M_1 \triangleright M_2 \mid M_1 \blacktriangleleft_M M_2 \mid \textbf{disable\_int}\ M \mid \textbf{in\_disable\_int}\ M \\
v & ::= & () \mid \textbf{true} \mid \textbf{false} \mid n \mid x \\
E & ::= & [\,] \mid \textbf{let}\ x = E\ \textbf{in}\ M \\
 & \mid & (E \mid M) \mid (M \mid E) \\
 & \mid & \textbf{in\_sync}\ x\ \textbf{in}\ E \mid \textbf{in\_disable\_int}\ E \\
 & \mid & E \triangleright M \mid M_1 \blacktriangleleft_M E \\
I & ::= & \textbf{enabled} \mid \textbf{disabled}
\end{array}
$$

Fig. 3. The Syntax of Our Language.

2 Target Language

2.1 Syntax

The syntax of our target language is defined in Figure 3. Our language is an imperative language which is equipped with concurrency and interrupt handling.

A program P consists of a sequence of function definitions $\widetilde{D}$ and a main expression M. A function definition is constructed from a function name x, a sequence of formal arguments $\widetilde{y}$ and a function body. Function definitions can be mutually recursive. Note that a function name belongs to the class of variables, so that one can use a function name as a first-class value.

Expressions are ranged over by a meta-variable M. $\triangleright$ and $\blacktriangleleft$ are left-associative. For the sake of simplicity, we have only block-structured primitives (**sync** x **in** M and **disable_int** M) for acquiring/releasing locks and disabling/enabling interrupts. We explain intuition of several non-standard primitives below.

- **let** $x =$ **ref** v **in** M creates a fresh reference to v, binds x to the reference and evaluates M.

```
flush_buffer_iter(devlock, data) =
  if !data = Null then () else
    (sync devlock in ()); flush_buffer_iter(devlock, dequeue())
flush_buffer(devlock) = flush_buffer_iter(devlock, dequeue())
receive_data(packettype, data, devlock) =
  if packettype = Room then flush_buffer(devlock) else ()
(* Main expression *)
let devlock = newlock() in
let data = ref Null in
  flush_buffer(devlock) ▷ receive_data(Room, data, devlock)
```

Fig. 4. An Encoding of the Program in Figure 1

- $M_1 \mid M_2$ is concurrent evaluation of M_1 and M_2. Both of M_1 and M_2 should evaluate to ().
- **let** x = **newlock** () **in** M generates a new lock, binds x to the lock and evaluates M.
- **sync** x **in** M attempts to acquire the lock x and evaluates M after the lock is acquired. After M is evaluated to a value, the lock x is released.
- $M_1 \triangleright M_2$ installs an interrupt handler M_2 and evaluates M_1. Once an interrupt occurs, M_1 is suspended until M_2 evaluates to a value. When M_1 evaluates to a value v, $M_1 \triangleright M_2$ evaluates to v.
- **disable_int** M disables interrupts during an evaluation of M.

The following three primitives only occur during evaluation and should not be included in programs.

- **in_sync** x **in** M represents the state in which M is being evaluated with the lock x acquired. After M evaluates to a value, the lock x is released.
- $M_1 \blacktriangleleft_M M_2$ represents the state in which the interrupt handler M_2 is being evaluated. After M_2 evaluates to a value, the interrupted expression M_1 and the initial state of interrupt handler M are recovered.
- **in_disable_int** M represents the state in which M is being evaluated with interrupts disabled. After M evaluates to a value, interrupts are enabled.

We write $M_1; M_2$ for **let** $x = M_1$ **in** M_2 where x is not free in M_2.

Figure 4 shows how the example in Figure 1 is encoded in our language. Though that encoding does not strictly conform to the syntax of our language (e.g., *flush_buffer_iter* is applied to an expression *dequeue*(), not to a value), one can easily translate the program into one that respects our syntax.

Our interrupt calculus is very expressive and can model various interrupt mechanisms, as discussed in Examples 1–4 below.

Example 1. In various kinds of CPUs, there is a priority among interrupts. In such a situation, if an interrupt with a higher priority occurs, interrupts with lower priorities do not occur. We can express such priorities by connecting several expressions with $\triangleright$ as follows.

$$\mathit{do\_something}(\ldots) \triangleright \mathit{interrupt\_low}(\ldots) \triangleright \mathit{interrupt\_high}(\ldots)$$

If an interrupt occurs in $\mathit{do\_something}(\ldots) \triangleright \mathit{interrupt\_low}(\ldots)$ (note that $\triangleright$ is left-associative), the example above is reduced to

$$(\mathit{do\_something}(\ldots) \blacktriangleleft_{\mathit{interrupt\_low}(\ldots)} \mathit{interrupt\_low}(\ldots)) \triangleright \mathit{interrupt\_high}(\ldots).$$

That state represents that *interrupt_low* interrupted *do_something*. From that state, *interrupt_high* can still interrupt.

$$(\mathit{do\_something}(\ldots) \blacktriangleleft_{\mathit{interrupt\_low}(\ldots)} \mathit{interrupt\_low}(\ldots)) \\ \blacktriangleleft_{\mathit{interrupt\_high}(\ldots)} \mathit{interrupt\_high}(\ldots).$$

interrupt_high can interrupt also from the initial state.

$$(\mathit{do\_something}(\ldots) \triangleright \mathit{interrupt\_low}(\ldots)) \blacktriangleleft_{\mathit{interrupt\_high}(\ldots)} \mathit{interrupt\_high}(\ldots)$$

From the state above, *interrupt_low* cannot interrupt until $\mathit{interrupt\_high}(\ldots)$ evaluates to a value.

Example 2. In our calculus, we can locally install interrupt handlers. Thus, we can express a multi-threaded program in which an interrupt handler is installed on each thread.

$$(\mathit{thread1}(\ldots) \triangleright \mathit{handler1}(\ldots)) \mid (\mathit{thread2}(\ldots) \triangleright \mathit{handler2}(\ldots)) \ldots$$

This feature is useful for modeling a multi-CPU system in which even if an interrupt occurs in one CPU, the other CPUs continue to work in non-interrupt mode.

Example 3. In the example in Figure 4, we assume that no interrupt occur in the body of *receive_data*. One can express that an interrupt may occur during an execution of *receive_data* by re-installing an interrupt handler as follows.

$$\begin{array}{l} \mathit{receive\_data}(\mathit{packettype}, \mathit{data}, \mathit{devlock}) = \\ \quad (\textbf{if } \mathit{packettype} = \mathit{Room} \textbf{ then } \mathit{flush\_buffer}(\mathit{devlock}) \textbf{ else } ()) \triangleright \\ \hfill \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock}) \end{array}$$

Example 4. Since many operating system kernels are written in C, we make design decisions of our language based on that of C. For example, names of functions are first-class values in our language because C allows one to use a function name as a function pointer and because operating system kernels heavily use function pointers. With this feature, we can express a runtime change of interrupt handler as follows:

$$\textbf{let } x = \textbf{ref } f \textbf{ in } ((\ldots; x := g; \ldots) \triangleright (!x)())$$

Until g is assigned to the reference x, the installed interrupt handler is f. After the assignment, the interrupt handler is g. This characteristic is useful for modeling operating system kernels in which interrupt handlers are changed when, for example, device drivers are installed.

2.2 Operational Semantics

The semantics is defined as rewriting of a configuration $(\widetilde{D}, H, L, I, M)$. H is a heap, which is a map from variables to values. (Note that references are represented by variables.) L is a map from variables to $\{\textbf{acquired}, \textbf{released}\}$. I is an interrupt flag, which is either **enabled** or **disabled** [3].

Figure 5 shows the operational semantics of our language. We explain several important rules.

- In (E-Ref) and (E-LetNewLock), newly generated references and locks are represented by fresh variables.
- Reduction with the rule (E-Lock) succeeds only if the lock being acquired is not held. (E-Unlock) is similar.
- **disable_int** changes the interrupt flag only when the flag was **enabled** (rule (E-DisableInterrupt1)). Otherwise, **disable_int** does nothing (rule (E-DisableInterrupt2)).
- If the interrupt flag is **enabled**, then a handler M_2 can interrupt M_1 anytime with the rule (E-Interrupt). When the interrupt occurs, the initial expression of interrupt handler M_2 is saved. After the handler terminates, the saved expression is recovered with (E-ExitInterrupt).

The following example shows how the program in Figure 4 leads to a deadlocked state. We write L_u for $\{devlock' \mapsto \textbf{released}\}$ and L_l for $\{devlock' \mapsto \textbf{acquired}\}$. We omit $\widetilde{D}, H$ and I components of configurations.

$$\begin{array}{rl}
& (L_u, \mathit{flush_buffer}(devlock') \rhd \mathit{receive_data}(Room, data, devlock')) \\
\to^* & (L_u, \textbf{sync}\ devlock'\ \textbf{in}\ () \rhd \mathit{receive_data}(Room, data, devlock)) \\
\to & (L_l, \textbf{in_sync}\ devlock'\ \textbf{in}\ () \rhd \mathit{receive_data}(Room, data, devlock)) \\
\to & (L_l, \textbf{in_sync}\ devlock'\ \textbf{in}\ () \blacktriangleleft_{\mathit{receive_data}(\ldots)} \mathit{receive_data}(Room, data, devlock)) \\
\to^* & (L_l, \textbf{in_sync}\ devlock'\ \textbf{in}\ () \blacktriangleleft_{\mathit{receive_data}(\ldots)} \mathit{flush_buffer}(devlock')) \\
\to^* & (L_l, \textbf{in_sync}\ devlock'\ \textbf{in}\ () \blacktriangleleft_{\mathit{receive_data}(\ldots)} \textbf{sync}\ devlock'\ \textbf{in}\ ())
\end{array}$$

The last configuration is in a deadlock because the attempt to acquire $devlock'$, which is already acquired in L_l, never succeeds and because the interrupt handler **sync** $devlock'$ **in** () does not voluntarily yield.

3 Type System

3.1 Lock Levels

In our type system, every lock type is associated with a *lock level*, which is represented by a meta-variable *lev*. The set of lock levels is $\{-\infty, \infty\} \cup \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. We extend the standard partial order $\leq$ on $\mathbb{N}$ to that on $\{-\infty, \infty\} \cup \mathbb{N}$ by $\forall lev \in \{-\infty, \infty\} \cup \mathbb{N}. -\infty \leq lev \leq \infty$. We write $lev_1 < lev_2$ for $lev_1 \leq lev_2 \wedge lev_1 \neq lev_2$.

[3] We do not assign an interrupt flag to each interrupt handler in order to keep the semantics simple. Even if we do so, the type system introduced in Section 3 can be used with only small changes.

$$\frac{x(\widetilde{y}) = M' \in \widetilde{D}}{(\widetilde{D}, H, L, I, E[x(\widetilde{v})]) \rightarrow (\widetilde{D}, H, L, I, E[[\widetilde{v}/\widetilde{y}]M'])} \quad \text{(E-App)}$$

$$(\widetilde{D}, H, L, I, E[\textbf{let } x = v \textbf{ in } M]) \rightarrow (\widetilde{D}, H, L, I, E[[v/x]M]) \quad \text{(E-Let)}$$

$$(\widetilde{D}, H, L, I, E[\textbf{if true then } M_1 \textbf{ else } M_2]) \rightarrow (\widetilde{D}, H, L, I, E[M_1]) \quad \text{(E-IfTrue)}$$

$$(\widetilde{D}, H, L, I, E[\textbf{if false then } M_1 \textbf{ else } M_2]) \rightarrow (\widetilde{D}, H, L, I, E[M_2]) \quad \text{(E-IfFalse)}$$

$$\frac{x' \text{ is fresh}}{(\widetilde{D}, H, L, I, E[\textbf{let } x = \textbf{ref } v \textbf{ in } M]) \rightarrow (\widetilde{D}, H[x' \mapsto v], L, I, E[[x'/x]M])} \quad \text{(E-Ref)}$$

$$(\widetilde{D}, H[x \mapsto v'], L, I, E[x := v]) \rightarrow (\widetilde{D}, H[x \mapsto v], L, I, E[()]) \quad \text{(E-Assign)}$$

$$(\widetilde{D}, H[x \mapsto v], L, I, E[!x]) \rightarrow (\widetilde{D}, H[x \mapsto v], L, I, E[v]) \quad \text{(E-Deref)}$$

$$\frac{x' \text{ is fresh}}{\begin{array}{l}(\widetilde{D}, H, L, I, E[\textbf{let } x = \textbf{newlock } () \textbf{ in } M]) \rightarrow \\ \qquad (\widetilde{D}, H, L[x' \mapsto \textbf{released}], I, E[[x'/x]M])\end{array}} \quad \text{(E-LetNewlock)}$$

$$(\widetilde{D}, H, L, I, E[() \mid ()]) \rightarrow (\widetilde{D}, H, L, I, E[()]) \quad \text{(E-ParEnd)}$$

$$\begin{array}{l}(\widetilde{D}, H, L[x \mapsto \textbf{released}], I, E[\textbf{sync } x \textbf{ in } M]) \rightarrow \\ \qquad (\widetilde{D}, H, L[x \mapsto \textbf{acquired}], I, E[\textbf{in\_sync } x \textbf{ in } M])\end{array} \quad \text{(E-Lock)}$$

$$(\widetilde{D}, H, L[x \mapsto \textbf{acquired}], I, E[\textbf{in\_sync } x \textbf{ in } v]) \rightarrow (\widetilde{D}, H, L[x \mapsto \textbf{released}], I, E[v]) \quad \text{(E-Unlock)}$$

$$(\widetilde{D}, H, L, \textbf{enabled}, E[M_1 \rhd M_2]) \rightarrow (\widetilde{D}, H, L, \textbf{enabled}, E[M_1 \blacktriangleleft_{M_2} M_2]) \quad \text{(E-Interrupt)}$$

$$(\widetilde{D}, H, L, I, E[M_1 \blacktriangleleft_{M_2} v]) \rightarrow (\widetilde{D}, H, L, I, E[M_1 \rhd M_2]) \quad \text{(E-ExitInterrupt)}$$

$$(\widetilde{D}, H, L, I, E[v \rhd M]) \rightarrow (\widetilde{D}, H, L, I, E[v]) \quad \text{(E-NoInterruptValue)}$$

$$(\widetilde{D}, H, L, \textbf{enabled}, E[\textbf{disable\_int } M]) \rightarrow (\widetilde{D}, H, L, \textbf{disabled}, E[\textbf{in\_disable\_int } M]) \quad \text{(E-DisableInterrupt1)}$$

$$(\widetilde{D}, H, L, \textbf{disabled}, E[\textbf{disable\_int } M]) \rightarrow (\widetilde{D}, H, L, \textbf{disabled}, E[M]) \quad \text{(E-DisableInterrupt2)}$$

$$(\widetilde{D}, H, L, I, E[\textbf{in\_disable\_int } v]) \rightarrow (\widetilde{D}, H, L, \textbf{enabled}, E[v]) \quad \text{(E-EnableInterrupt)}$$

Fig. 5. The Operational Semantics of Our Language.

3.2 Effects

Our type system guarantees that a program acquires locks in a strict increasing order of lock levels. To achieve this, we introduce *effects* which describe how a program acquires locks during evaluation.

An effect, represented by a meta-variable φ, is a pair of lock levels (lev_1, lev_2). The meaning of each component is as follows.

- lev_1 is a lower bound of the lock levels of locks that may be acquired.
- lev_2 is an upper bound of the lock levels of locks that may be acquired or have been acquired while interrupts are enabled.

$$\begin{aligned}\tau &::= \textbf{unit} \mid \textbf{int} \mid \textbf{bool} \mid \widetilde{\tau}_1 \xrightarrow{\varphi} \tau_2 \mid \tau\ \textbf{ref} \mid \textbf{lock}(lev)\\ lev &\in \{-\infty, \infty\} \cup \mathbb{N}\\ \varphi &::= (lev_1, lev_2)\end{aligned}$$

Fig. 6. Syntax of types.

For example, an effect $(0, -\infty)$ means that locks whose levels are more than or equal to 0 may be acquired and that no locks are acquired while interrupts are enabled. An effect $(0, 1)$ means that locks whose levels are more than or equal to 0 may be acquired and that a lock of level 1 may be acquired or has already been acquired while interrupts are enabled. We write $\emptyset$ for $(\infty, -\infty)$.

We define the subeffect relation and the join operator for effects as follows.

Definition 1 (Subeffect Relation). *$(lev_1, lev_2) \leq (lev'_1, lev'_2)$ holds if and only if $lev'_1 \leq lev_1$ and $lev_2 \leq lev'_2$.*

$(lev_1, lev_2) \leq (lev'_1, lev'_2)$ means that an expression that acquires locks according to the effect (lev_1, lev_2) can be seen as an expression with the effect (lev'_1, lev'_2). For example, $(1, 2) \leq (0, 3)$ holds. $\emptyset$ is the bottom of $\leq$.

Definition 2 (Join). $(lev_1, lev_2) \sqcup (lev'_1, lev'_2) = (min(lev_1, lev'_1), max(lev_2, lev'_2))$

For example, $(1, 2) \sqcup (0, 1) = (0, 2)$ and $(0, -\infty) \sqcup (1, 2) = (0, 2)$ hold. $\emptyset$ is an identity of $\sqcup$.

3.3 Syntax of Types

Figure 6 shows the syntax of types and effects. A type, represented by a metavariable τ, is either **unit**, **int**, **bool**, $\widetilde{\tau}_1 \xrightarrow{\varphi} \tau_2$, τ **ref** or **lock**(lev). We write $\widetilde{\tau}$ for a sequence of types. τ **ref** is the type of a reference to a value of type τ. $\widetilde{\tau}_1 \xrightarrow{\varphi} \tau_2$ is the type of functions which take a tuple of values of type $\widetilde{\tau}_1$ and return a value of type τ_2. φ is the latent effect of the functions.

3.4 Type Judgment

The type judgment form of our type system is $\Gamma \vdash M : \tau \ \&\ \varphi$ where Γ is a map from variables to types. The judgment means that the resulting value of the evaluation of M has type τ if an evaluation of M under an environment described by Γ terminates, and that locks are acquired in a strict increasing order of lock levels during the evaluation. The minimum and maximum lock levels acquired are constrained by φ. For example, $x : \textbf{lock}(0), y : \textbf{lock}(1) \vdash \textbf{sync}\ x\ \textbf{in}\ \textbf{sync}\ y\ \textbf{in}\ () : \textbf{unit}\ \&\ (0, 1)$ and $x : \textbf{lock}(0), y : \textbf{lock}(1) \vdash \textbf{sync}\ x\ \textbf{in}\ (\textbf{disable\_int}\ \textbf{sync}\ y\ \textbf{in}\ ()) : \textbf{unit}\ \&\ (0, 0)$ hold.

Definition 3. *The relation $\Gamma \vdash M : \tau \ \& \ \varphi$ is the smallest relation closed under the rules in Figures 7 and 8. The predicate noIntermediate(M) in Figure 8 holds if and only if M does not contain* **in_sync** *x* **in** *M',* **in_disable_int** *M' or $M_1 \blacktriangleleft_{M'} M_2$ as subterms.*

We explain several important rules.

- (T-Sync): If the level of x is lev, then M can acquire only locks whose levels are more than lev. That is guaranteed by the condition $lev < lev_1$ where lev_1 is a lower bound of the levels of locks that may be acquired by M.
- (T-DisableInterrupt): The second component of the effect of **disable_int** M is changed to $-\infty$ because no interrupt occurs in M, so that no locks are acquired by interrupt handlers.
- (T-InstHandler): The second component of the effect of M_1 should be less than the first component of the effect of M_2 because M_2 can interrupt M_1 at any time. This is why we need to include the maximum level in effects.
- (T-Fundef): The condition $\varphi' \leq \varphi_i$ guarantees that the latent effect of the type of the function being defined soundly approximates the runtime locking behavior.

We show how the program in Figure 4 is rejected in our type system. From the derivation tree in Figure 9, *flush_buffer_iter* has a type $(\mathbf{lock}(1), \tau_d\ \mathbf{ref}) \stackrel{(1,1)}{\rightarrow}$ **unit**, where τ_d is the type of the contents of the reference *data*. Thus, *flush_buffer* has a type $\mathbf{lock}(1) \stackrel{(1,1)}{\rightarrow}$ **unit** and *receive_data* has a type $(\tau_p, \tau_d, \mathbf{lock}(1)) \stackrel{(1,1)}{\rightarrow}$ **unit**, where τ_p is the type of *packettype*.

Consider the main expression of the example. Let Γ be *devlock* : $\mathbf{lock}(1)$, *data* : τ_d **ref**. Then, we have

- $\Gamma \vdash$ *flush_buffer*(*devlock*) : **unit** & $(1, 1)$ and
- $\Gamma \vdash$ *receive_data*(*Room*, *data*, *devlock*) : **unit** & $(1, 1)$.

However, the condition $lev_2 < lev'_1$ of the rule (T-InstHandler) prevents the main expression to be well-typed ($1 < 1$ does not hold).

Suppose that **sync** *devlock* **in** () in the body of *flush_buffer_iter* is replaced by **disable_int sync** *devlock* **in** (). Then, *flush_buffer_iter* has a type $(\mathbf{lock}(1), \tau_d\ \mathbf{ref}) \stackrel{(1,-\infty)}{\rightarrow}$ **unit** Thus, because $\Gamma \vdash$ *flush_buffer*(*devlock*) : **unit** & $(1, -\infty)$ and $-\infty < 1$ hold, the program is well-typed.

3.5 Type Soundness

We prove the soundness of our type system. Here, type soundness means that a well-typed program does not get deadlocked if one begins an evaluation of the program under an initial configuration (i.e., under an empty heap, an empty lock environment and enabled interrupt flag).

We first define deadlock. The predicate *deadlocked*(L, M) defined below means that M is in a deadlocked state under L.

$$\Gamma \vdash () : \mathbf{unit}\ \&\ \emptyset \quad \text{(T-Unit)} \qquad \Gamma \vdash n : \mathbf{int}\ \&\ \emptyset \quad \text{(T-Int)}$$

$$\Gamma \vdash \mathbf{true} : \mathbf{bool}\ \&\ \emptyset \quad \text{(T-True)} \qquad \Gamma \vdash \mathbf{false} : \mathbf{bool}\ \&\ \emptyset \quad \text{(T-False)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau\ \&\ \emptyset} \quad \text{(T-Var)} \qquad \frac{x : (\tau_1, \ldots, \tau_n) \xrightarrow{\varphi'} \tau \in \Gamma \quad \Gamma \vdash v_i : \tau_i\ \&\ \emptyset \quad (i = 1, \ldots, n)}{\Gamma \vdash x(v_1, \ldots, v_n) : \tau\ \&\ \varphi'} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1\ \&\ \varphi_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau\ \&\ \varphi_2}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau\ \&\ \varphi_1 \sqcup \varphi_2} \quad \text{(T-Let)}$$

$$\frac{\Gamma \vdash v : \mathbf{bool}\ \&\ \emptyset \quad \Gamma \vdash M_1 : \tau\ \&\ \varphi_1 \quad \Gamma \vdash M_2 : \tau\ \&\ \varphi_2}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \tau\ \&\ \varphi_1 \sqcup \varphi_2} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash v : \tau\ \&\ \emptyset \quad x : \tau\ \mathbf{ref}, \Gamma \vdash M : \tau'\ \&\ \varphi}{\Gamma \vdash \mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in}\ M : \tau'\ \&\ \varphi} \quad \text{(T-Ref)}$$

$$\frac{x : \tau\ \mathbf{ref} \in \Gamma \quad \Gamma \vdash v : \tau\ \&\ \emptyset}{\Gamma \vdash x := v : \mathbf{unit}\ \&\ \emptyset} \quad \text{(T-Assign)}$$

$$x : \tau\ \mathbf{ref}, \Gamma \vdash !x : \tau\ \&\ \emptyset \quad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash M_1 : \mathbf{unit}\ \&\ \varphi_1 \quad \Gamma \vdash M_2 : \mathbf{unit}\ \&\ \varphi_2}{\Gamma \vdash M_1 \mid M_2 : \mathbf{unit}\ \&\ \varphi_1 \sqcup \varphi_2} \quad \text{(T-Par)}$$

$$\frac{x : \mathbf{lock}(lev), \Gamma \vdash M : \tau\ \&\ (lev_1, lev_2)}{\Gamma \vdash \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ M : \tau\ \&\ (lev_1, lev_2)} \quad \text{(T-Newlock)}$$

$$\frac{x : \mathbf{lock}(lev) \in \Gamma \quad \Gamma \vdash M : \tau\ \&\ (lev_1, lev_2) \quad lev < lev_1 \quad \varphi = (lev, lev) \sqcup (lev_1, lev_2)}{\Gamma \vdash \mathbf{sync}\ x\ \mathbf{in}\ M : \tau\ \&\ \varphi} \quad \text{(T-Sync)}$$

$$\frac{x : \mathbf{lock}(lev) \in \Gamma \quad \Gamma \vdash M : \tau\ \&\ (lev_1, lev_2) \quad lev < lev_1 \quad \varphi = (\infty, lev) \sqcup (lev_1, lev_2)}{\Gamma \vdash \mathbf{in\_sync}\ x\ \mathbf{in}\ M : \tau\ \&\ \varphi} \quad \text{(T-Insync)}$$

$$\frac{\Gamma \vdash M : \tau\ \&\ (lev_1, lev_2)}{\Gamma \vdash \mathbf{disable\_int}\ M : \tau\ \&\ (lev_1, -\infty)} \quad \text{(T-DisableInterrupt)}$$

$$\frac{\Gamma \vdash M : \tau\ \&\ (lev_1, lev_2)}{\Gamma \vdash \mathbf{in\_disable\_int}\ M : \tau\ \&\ (lev_1, -\infty)} \quad \text{(T-InDisableInterrupt)}$$

$$\frac{\Gamma \vdash M_1 : \tau\ \&\ (lev_1, lev_2) \quad \Gamma \vdash M_2 : \mathbf{unit}\ \&\ (lev'_1, lev'_2) \quad lev_2 < lev'_1 \quad \varphi = (lev_1, lev_2) \sqcup (lev'_1, lev'_2)}{\Gamma \vdash M_1 \rhd M_2 : \tau\ \&\ \varphi} \quad \text{(T-InstHandler)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \tau\ \&\ (lev_1, lev_2) \quad \Gamma \vdash M_2 : \mathbf{unit}\ \&\ (lev'_1, lev'_2) \quad \Gamma \vdash M : \mathbf{unit}\ \&\ (lev''_1, lev''_2) \\ lev_2 < lev'_1 \quad lev_2 < lev''_1 \quad \varphi' = \varphi \sqcup (lev_1, lev_2) \sqcup (lev'_1, lev'_2)\end{array}}{\Gamma \vdash M_1 \blacktriangleleft_M M_2 : \tau\ \&\ \varphi'} \quad \text{(T-InInterrupt)}$$

Fig. 7. Typing rules

$$\frac{\begin{array}{c}\Gamma \supseteq f_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \stackrel{\varphi_1}{\to} \tau_1, \ldots, f_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \stackrel{\varphi_n}{\to} \tau_n \\ \Gamma, x_{i,1} : \tau_{i,1}, \ldots, x_{i,m_i} : \tau_{i,m_i} \vdash M_i : \tau_i \ \&\ \varphi' \\ \varphi' \leq \varphi_i \qquad \mathit{noIntermediate}(M_i)\end{array}}{\Gamma \vdash_D f_i(x_{i,1}, \ldots, x_{i,m_i}) = M_i : (\tau_{i,1}, \ldots, \tau_{i,m_i}) \stackrel{\varphi_i}{\to} \tau_i} \quad \text{(T-Fundef)}$$

$$\frac{\begin{array}{c}\{f_1, \ldots, f_n\} \text{ is the set of names of functions declared in } \widetilde{D} \\ \Gamma \supseteq \{f_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \stackrel{\varphi_1}{\to} \tau_1, \ldots, f_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \stackrel{\varphi_n}{\to} \tau_n\} \\ \Gamma \vdash_D D_i : (\tau_{i,1}, \ldots, \tau_{i,m_i}) \stackrel{\varphi_i}{\to} \tau_i \quad (1 \leq i \leq n) \\ \Gamma \vdash M : \mathbf{unit} \ \&\ \varphi \qquad \mathit{noIntermediate}(M)\end{array}}{\vdash_P \widetilde{D}M} \quad \text{(T-Prog)}$$

$$\frac{\begin{array}{c}\widetilde{D} = \{f_1(x_{1,1}, \ldots, x_{1,m_1}) = M_1, \ldots, f_l(x_{l,1}, \ldots, x_{l,m_l}) = M_l\} \\ H = \{y_1 \mapsto v_1, \ldots, y_k \mapsto v_k\} \\ L = \{z_1 \mapsto \mathit{lck}_1, \ldots, z_n \mapsto \mathit{lck}_n\} \\ \Gamma \vdash_D (f_i(x_{i,1}, \ldots, x_{i,m_i}) = M_i) : (\tau_{i,1}, \ldots, \tau_{i,m_i}) \stackrel{\varphi_i}{\to} \tau_i \quad (1 \leq i \leq l) \\ \Gamma \vdash v_i : \tau_i' \ \&\ \emptyset \quad (1 \leq i \leq k) \\ \Gamma = f_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \stackrel{\varphi_1}{\to} \tau_1, \ldots, f_l : (\tau_{l,1}, \ldots, \tau_{l,m_l}) \stackrel{\varphi_l}{\to} \tau_l, \\ y_1 : \tau_1' \ \mathbf{ref}, \ldots, y_k : \tau_k' \ \mathbf{ref}, \\ z_1 : \mathbf{lock}(\mathit{lev}_1), \ldots, z_n : \mathbf{lock}(\mathit{lev}_n)\end{array}}{\vdash_{\mathit{Env}} (\widetilde{D}, H, L) : \Gamma} \quad \text{(T-Env)}$$

$$\frac{\vdash_{\mathit{Env}} (\widetilde{D}, H, L) : \Gamma \qquad \Gamma \vdash M : \tau \ \&\ (\mathit{lev}_1, \mathit{lev}_2)}{\vdash_C (\widetilde{D}, H, L, I, M) : \tau} \quad \text{(T-Config)}$$

Fig. 8. Typing Rules for Program and Configuration.

Definition 4 (Deadlock). *The predicate* $\mathit{deadlocked}(L, M)$ *holds if and only if for all* E *and* i, $M = E[i]$ *implies that there exist* x *and* M' *such that* $i = \mathbf{sync}\ x\ \mathbf{in}\ M' \wedge L(x) = \mathbf{acquired}$. *Here,* i *is defined by the following syntax.*

$$\begin{array}{rl} i ::= & x(\widetilde{v}) \mid \mathbf{let}\ x = v\ \mathbf{in}\ M \\ \mid & \mathbf{if\ true\ then}\ M_1\ \mathbf{else}\ M_2 \mid \mathbf{if\ false\ then}\ M_1\ \mathbf{else}\ M_2 \\ \mid & \mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in}\ M \mid x := v \mid !x \\ \mid & \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ M \mid (()|()) \mid \mathbf{sync}\ x\ \mathbf{in}\ M \mid \mathbf{in\_sync}\ x\ \mathbf{in}\ v \\ \mid & M_1 \blacktriangleleft_{M_2} v \mid v \rhd M \mid \mathbf{disable\_int}\ M \mid \mathbf{in\_disable\_int}\ v \end{array}$$

In the definition above, i is a term that can be reduced by the rules in Figure 5. Thus, $\mathit{deadlocked}(L, M)$ means that every reducible subterm in M is a blocked lock-acquiring instruction. For example,

$$\mathit{deadlocked}(L, (\mathbf{in\_sync}\ x\ \mathbf{in}\ (\mathbf{sync}\ y\ \mathbf{in}\ 0)) \mid (\mathbf{in\_sync}\ y\ \mathbf{in}\ (\mathbf{sync}\ x\ \mathbf{in}\ 0)))$$

holds where $L = \{x \mapsto \mathbf{acquired}, y \mapsto \mathbf{acquired}\}$.

Note that $M_1 \rhd M_2$ is not included in the definition of i because, in the real world, whether $M_1 \rhd M_2$ is reducible or not depends on the external environ-

$$\dfrac{\vdots \quad \mathcal{T}_1 \quad \mathcal{T}_2}{\varGamma \vdash \textbf{if} \ldots \textbf{then}\ () \ \textbf{else}\ (\textbf{sync}\ \mathit{devlock}\ \textbf{in}\ ()); \mathit{flush\_buffer\_iter}(\ldots) : \textbf{unit}\ \&\ (1,1)}$$

$$\vdots$$

where

$$\mathcal{T}_1 = \dfrac{\varGamma \vdash () : \textbf{unit}\ \&\ \emptyset \quad 1 < \infty}{\varGamma \vdash \textbf{sync}\ \mathit{devlock}\ \textbf{in}\ () : \textbf{unit}\ \&\ (1,1)}$$

$$\mathcal{T}_2 = \dfrac{\varGamma \vdash \mathit{flush\_buffer\_iter} : (\textbf{lock}(1), \tau_d) \stackrel{(1,1)}{\to} \textbf{unit}\ \&\ \emptyset \quad \vdots}{\varGamma \vdash \mathit{flush\_buffer\_iter}(\mathit{devlock}, \mathit{dequeue}()) : \textbf{unit}\ \&\ (1,1)}$$

Fig. 9. Derivation Tree of the body of *flush_buffer_iter*. $\varGamma = \mathit{flush\_buffer\_iter} : (\textbf{lock}(1), \tau_d\ \textbf{ref}) \stackrel{(1,1)}{\to} \textbf{unit}, \mathit{flush\_buffer} : \textbf{lock}(1) \stackrel{(1,1)}{\to} \textbf{unit}, \mathit{receive\_data} : (\tau_p, \tau_d\ \textbf{ref}, \textbf{lock}(1)) \stackrel{(1,1)}{\to} \textbf{unit}, \mathit{devlock} : \textbf{lock}(1), \mathit{data} : \tau_d$.

ment which is not modeled in our calculus. For example, $(\textbf{sync}\ x\ \textbf{in}\ ()) \rhd ()$ is deadlocked under the environment in which x is acquired.

Theorem 1 (Type Soundness). *If* $\vdash_P \widetilde{D}M$ *and* $(\widetilde{D}, \emptyset, \emptyset, \textbf{enabled}, M) \to^* (\widetilde{D}', H', L', I', M')$*, then* $\neg \mathit{deadlocked}(L', M')$*.*

The theorem above follows from Lemmas 1–4 below. In those lemmas, we use a predicate $\mathit{wellformed}(L, I, M)$ which means that L, I and the shape of M are consistent.

Definition 5. *$\mathit{wellformed}(L, I, M)$ holds if and only if*

- $L(x) = \textbf{released}$ *or* $x \notin \textbf{Dom}(L)$ *implies that* M *does not contain* $\textbf{in\_sync}\ x$*,*
- $L(x) = \textbf{acquired}$ *implies* $\mathit{AckIn}(x, M)$*,*
- $I = \textbf{enabled}$ *implies that* M *does not contain* $\textbf{in\_disable\_int}$*.*
- $I = \textbf{disabled}$ *implies that there exist* E *and* M' *such that* $M = E[\textbf{in\_disable\_int}\ M']$ *and both* E *and* M' *do not contain* $\textbf{in\_disable\_int}$*.*

Here, $\mathit{AckIn}(x, M)$ *is the least predicate that satisfies the following rules.*

$$\dfrac{E \text{ and } M' \text{ do not contain } \textbf{in\_sync}\ x\ \textbf{in}}{\mathit{AckIn}(x, E[\textbf{in\_sync}\ x\ \textbf{in}\ M'])} \ (\textsc{AckIn-Base})$$

$$\dfrac{\mathit{AckIn}(x, M_1) \quad E, M' \text{ and } M_2 \text{ do not contain } \textbf{in\_sync}\ x\ \textbf{in}}{\mathit{AckIn}(x, E[M_1 \blacktriangleleft_{M'} M_2])} \ (\textsc{AckIn-Interrupt})$$

Lemma 1. *If* $\vdash_P \widetilde{D}M$*, then* $\mathit{wellformed}(\emptyset, \textbf{enabled}, M)$ *and* $\vdash_C (D, \emptyset, \emptyset, \textbf{enabled}, M)$*.*

Lemma 2. *If* $\mathit{wellformed}(L, I, M)$ *and* $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$*, then* $\mathit{wellformed}(L', I', M')$*.*

Lemma 3 (Preservation). *If* $\vdash_C (\widetilde{D}, H, L, I, M) : \tau$ *and* $(\widetilde{D}, H, L, I, M) \rightarrow (\widetilde{D}', H', L', I', M')$*, then* $\vdash_C (\widetilde{D}', H', L', I', M') : \tau$.

Lemma 4 (Deadlock-Freedom). *If* $\vdash_C (\widetilde{D}, H, L, I, M) : \tau$ *and* $\mathit{wellformed}(L, I, M)$*, then* $\neg \mathit{deadlocked}(L, M)$.

Proofs of those lemmas are in Appendix A and B.

3.6 Type Inference

We can construct a standard constraint-based type inference algorithm as follows. The algorithm takes a program as an input, prepares variables for unknown types and lock levels, and extracts constraints on them based on the typing rules. By the standard unification algorithm and the definition of the subeffect relation, the extracted constraints can then be reduced to a set of constraints of the form $\{\rho_1 \geq \xi_1, \ldots, \rho_n \geq \xi_n\}$ where the grammar for $\xi_1, \ldots, \xi_n$ is given by

$$\begin{aligned} \xi ::= & \ \rho \text{ (lock level variables)} \\ | & \ -\infty \mid \infty \mid \mathit{min}(\xi_1, \xi_2) \mid \mathit{max}(\xi_1, \xi_2) \mid \xi + 1. \end{aligned}$$

Note that $\mathit{lev} < \mathit{lev}_1$ in (T-SYNC) can be replaced by $\mathit{lev} + 1 \leq \mathit{lev}_1$. The constraints above can be solved as in Kobayashi's type-based deadlock analysis for the π-calculus [7]. We will formalize the algorithm in the full version of the current paper.

4 Related Work

Chatterjee et al. have proposed a calculus that is equipped with interrupts [3, 14]. They also proposed a static analysis of stack boundedness (i.e., interrupt chains cannot be infinite) of programs. The main differences between our calculus and their calculus are as follows. (1) Their calculus is not equipped with concurrency primitives. (2) Each handler has its own interrupt flag in their calculus. (3) Our calculus can express an install, a change and a detach of interrupt handlers. Due to (1), we cannot use their calculus to discuss deadlock-freedom analysis. As for (2), their calculus has an interrupt mask register (imr) to control which handlers are allowed to interrupt and which are not. This feature is indispensable in the verification of operating system kernels. We can extend our calculus to incorporate this feature by adding a tag to each interrupt handler ($M \rhd \{t_1 : M_1, \ldots, t_n : M_n\}$) and by specifying a tag on interrupt disabling primitives (**disable_int** t **in** M). A handler with tag t cannot interrupt inside **disable_int** t **in**. We also extend effects like $(\mathit{lev}, \mathit{taglevel})$, where $\mathit{taglevel}$ is a map from tags to lock levels. $\mathit{taglevel}(t)$ is an upper bound of the lock levels of locks that may be acquired or have been acquired while interrupts specified by t is enabled. Typing rules need to be modified accordingly. Concerning (3), our calculus can express a change of interrupt handlers as shown in Section 2.

Much work [2, 7–9] on deadlock-freedom analysis of concurrent programs has been done. However, none of them deal with interrupts. Kobayashi et al. [7–9] have proposed type systems for deadlock-freedom of π-calculus processes. Their idea is (1) to express how each channel is used as a *usage expression* and (2) to add *capability levels* and *obligation levels* to the inferred usage expression in order to detect circular dependency among input/output operations to channels. Their capability/obligation levels correspond to our lock levels. Their usage expressions are unnecessary in the present framework because our synchronization primitive is block-structured. That notion would be useful if we allow non-block-structured lock primitives. Flanagan and Abadi [1, 4] have proposed a type-based deadlock-freedom and race-freedom analysis for a Java-like language. Though their type system also uses lock levels, they need to track only a lower bound of acquired level as an effect because they do not deal with interrupts. In our type system, we need to track lower and upper bounds of levels as an effect in order to guarantee deadlock-freedom in the presence of interrupts.

Asynchronous exceptions [5, 10] in Java and Haskell are similar to interrupts in that both cause an asynchronous jump to an exception/interrupt handler. Asynchronous exceptions are the exceptions that may be unexpectedly thrown during an execution of a program as a result of some events such as timeouts or stack overflows. Marlow et al. [10] extended Concurrent Haskell [6] with support for handling asynchronous exceptions. However, an asynchronous exception does not require the context in which the exception is thrown to be resumed after an exception handler returns, while an interrupt requires the context to be resumed.

5 Conclusion

We have proposed a calculus which is equipped with concurrency and interrupts. We have also proposed a type system for verification of deadlock-freedom for the calculus.

There remain much work to be done to make our framework applicable to verification of real operating system kernels. Since many operating system kernels are written in C, we need to include records, arrays and pointer arithmetics in our calculus. For those extensions, we may also need to refine the type system. In the current lock-level-based approach, a lock level is statically assigned to each *syntactic* occurence of a lock, so that the same lock level may be assigned to different locks. To prevent that problem, we may need to introduce lock-level polymorphism and run-time ordering of lock levels as proposed in [2].

We also plan to develop type systems for verifying other crucial safety properties such as race-freedom and atomicity.

Acknowledgement

We are grateful to Eijiro Sumii, Hiroya Matsuba, Toshiyuki Maeda and Yutaka Ishikawa for the comment on this research. We are also grateful to the anonymous reviewers for their fruitful comments.

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. 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.
4. 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.
5. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Professional, June 2005.
6. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996)*, pages 295–308, January 1996.
7. Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.
8. 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.
9. Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503, August 2000.
10. Simon Marlow, Simon Peyton Jones, and Andrew Moran. Asynchronous exceptions in haskell. In *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001)*, June 2001.
11. Hiroya Matsuba and Yutaka Ishikawa. Single IP address cluster for internet servers. In *Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS2007)*, March 2007.
12. Robin Milner. *Communication and Concurrency.* Prentice Hall, September 1995.
13. Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.
14. 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.

Appendix

A Proofs of Lemma 2 and 3

Lemma 5 (Weakening). *If $\Gamma \vdash M : \tau \;\&\; \varphi$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash M : \tau \;\&\; \varphi$.*

Proof. By induction on the derivation of $\Gamma \vdash M : \tau \;\&\; \varphi$.

Lemma 6. *Suppose $\Gamma \vdash v : \tau \;\&\; \emptyset$, then*

- *if $\tau = \mathbf{bool}$, then $v = \mathbf{true}$ or $v = \mathbf{false}$ or $v = x$ for some x*
- *if $\tau = \mathbf{unit}$, then $v = ()$ or $v = x$ for some x*
- *if $\tau = \widetilde{\tau}' \xrightarrow{\varphi} \tau'$ or $\tau = \tau'$ $\mathbf{ref}$ or $\tau = \mathbf{lock}(lev)$, then v is a variable.*

Proof. Case analysis on the rule use for deriving $\Gamma \vdash v : \tau \;\&\; \emptyset$.

Lemma 7 (Substitution). *If $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \;\&\; \varphi$ and $\Gamma \vdash v_i : \tau_i \;\&\; \emptyset$ for $i = 1, \ldots, n$, then $\Gamma \vdash [\widetilde{v}/\widetilde{x}]M : \tau \;\&\; \varphi$.*

Proof. By induction on the derivation of $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \;\&\; \varphi$. We perform case analysis on the last rule used for deriving $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \;\&\; \varphi$. We show only important cases. Other cases are similar.

- Case (T-Var): In this case, $M = y$ and $y : \tau \in \{\Gamma, \widetilde{x} : \widetilde{\tau}\}$. If $y \in \mathbf{Dom}(\Gamma)$, then $[\widetilde{v}/\widetilde{x}]M = y$. From (T-Var), $\Gamma \vdash y : \tau \;\&\; \varphi$ follows as required.
  Suppose that $y : x_i$ for some i. In this case, $[\widetilde{v}/\widetilde{x}]M = v_i$ and $\tau = \tau_i$. Thus, $\Gamma \vdash [\widetilde{v}/\widetilde{x}]M : \tau \;\&\; \varphi$ follows from $\Gamma \vdash v_i : \tau_i \;\&\; \emptyset$.
- Case (T-App): In this case, $M = x(v'_1, \ldots, v'_m)$. Thus, $x : (\tau'_1, \ldots, \tau'_m) \xrightarrow{\varphi} \tau \in \{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n\}$ and $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash v'_i : \tau'_i \;\&\; \emptyset$ for $1 \leq i \leq m$. From I.H., we have $\Gamma \vdash [\widetilde{v}/\widetilde{x}]v'_i : \tau'_i \;\&\; \emptyset$. If $x \in \mathbf{Dom}(\Gamma)$, then $\Gamma \vdash [\widetilde{v}/\widetilde{x}]M : \tau \;\&\; \varphi$ follows immediately. If $x = x_i$ for some i, then $\Gamma \vdash v_i : (\tau'_1, \ldots, \tau'_m) \xrightarrow{\varphi} \tau \;\&\; \emptyset$. From Lemma 6, $v_i = y$ for some y. From $\Gamma \vdash v_i : \tau_i \;\&\; \emptyset$, we have $y \in \mathbf{Dom}(\Gamma)$. Thus, $\Gamma \vdash [\widetilde{v}/\widetilde{x}]M : \tau \;\&\; \varphi$ as required.

Proof of Lemma 2. From $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$, $M = E[M_1]$ and $M' = E[M'_1]$ for some E, M_1 and M'_1. We perform case analysis on the rule used for deriving $(\widetilde{D}, H, L, I, E[M_1]) \to (\widetilde{D}', H', L', I', E[M'_1])$. We examine only important cases.

- (E-Lock): $L' = L[x \mapsto \mathbf{acquired}]$ and $M' = E[\mathbf{in\_sync}\ x\ \mathbf{in}\ M''_1]$. Thus, it is sufficient to show $AckIn(x, M')$, which immediately follows from the definition of $AckIn$ and $wellformed(L, I, M)$. Note that the latter implies that $\mathbf{in\_sync}\ x$ does not occur in E or M''_1.
- (E-Unlock): $L' = L[x \mapsto \mathbf{released}]$ and $M' = E[v]$. Thus, it is sufficient to show that E does not contain $\mathbf{in\_sync}\ x$, which immediately follows from $wellformed(L, I, E[M_1])$ and $L(x) = \mathbf{acquired}$.

– (E-DisableInterrupt1): $M' = E[\textbf{in\_disable\_int}\ M'']$ and $I' = \textbf{disabled}$. Thus, it is sufficient to show that E and M'' do not contain **in_disable_int**, which immediately follows from $\mathit{wellformed}(L, I, E[M_1])$.
– (E-DisableInterrupt2): $M = E[\textbf{disable\_int}\ M'']$ and $I = \textbf{disabled}$. From $\mathit{wellformed}(L, \textbf{disabled}, M)$, there exists E' and E'' such that $E = E'[\textbf{in\_disable\_int}\ E'']$ and neither E' nor $E''[\textbf{disable\_int}\ M'']$ contain **in_disable_int**. Thus, from $I' = \textbf{disabled}$ and $M' = E[M''] = E'[\textbf{in\_disable\_int}\ E''[M'']]$, $\mathit{wellformed}(L', I', M')$ holds as required.
– (E-EnableInterrupt): In this case $M = E[\textbf{in\_disable\_int}\ v]$ and $M' = E[v]$. From $I = \textbf{disabled}$, E does not contain **in_disable_int**. Thus, $\mathit{wellformed}(L', I', M')$ holds as required.

Proof of Lemma 3. We prove the following proposition which is stronger than Lemma 3.

Proposition 1. *If* $\vdash_{Env} (\widetilde{D}, H, L) : \Gamma$ *and* $\Gamma \vdash M : \tau \ \&\ \varphi$ *and* $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$*, then* $\vdash_{Env} (\widetilde{D}', H', L') : \Gamma'$ *and* $\Gamma' \vdash M' : \tau \ \&\ \varphi'$ *and* $\Gamma' \supseteq \Gamma$ *and* $\varphi' \leq \varphi$

From $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$, there exist E and M_1 such that $M = E[M_1]$. We perform induction on the structure of E.

– Case $E = [\,]$: We perform case analysis on the rule used for deriving $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$. We show only interesting cases. Other cases are similar.
  • Case (E-App):
    In this case,
    ∗ (1) $M_1 = x(v_1, \ldots, v_n)$ and
    ∗ (2) $x(\widetilde{y}) = M_1' \in \widetilde{D}$.
    $\Gamma \vdash M : \tau \ \&\ \varphi$ must have been derived using (T-App). Thus, we have
    ∗ (3) $x : (\tau_1, \ldots, \tau_n) \xrightarrow{\varphi} \tau \in \Gamma$ and
    ∗ (4) $\Gamma \vdash v_i : \tau_i \ \&\ \emptyset \quad (i = 1, \ldots, n)$.
    From (2), (3), (T-Env) and (T-Fundef), we have
    ∗ (5) $\Gamma, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash M_1' : \tau \ \&\ \varphi'$ and
    ∗ (6) $\varphi' \leq \varphi$.
    Thus, we have $\Gamma \vdash [\widetilde{v}/\widetilde{y}]M_1' : \tau \ \&\ \varphi'$ and $\varphi' \leq \varphi$ as required. from (4), (5), (6) and Lemma 7.
  • Case (E-Interrupt): In this case,
    ∗ (1) $M_1 = M_1' \rhd M_2'$ and
    ∗ (2) $I = \textbf{enabled}$
    for some M_1' and M_2'. $\Gamma \vdash M : \tau \ \&\ \varphi$ must have been derived using (T-InstHandler). Thus, we have
    ∗ (3) $\Gamma \vdash M_1' : \tau \ \&\ (lev_1, lev_1')$,
    ∗ (4) $\Gamma \vdash M_2' : \textbf{unit} \ \&\ (lev_2, lev_2')$
    ∗ (5) $lev_1' < lev_2$
    ∗ (6) $\varphi = (lev_1, lev_1') \sqcup (lev_2, lev_2')$

for some lev'_1, lev''_1, lev'_2 and lev''_2. Thus, from (3), (4), (5), (6) and (T-InInterrupt), we have $\Gamma \vdash M_1 \blacktriangleleft_{M_2} M_2 : \tau \ \& \ \varphi$ as required.

– $E = M_2 \blacktriangleleft_{M_3} E'$: In this case, (1) $M = M_2 \blacktriangleleft_{M_3} E'[M_1]$ and (2) $(\widetilde{D}, H, L, I, E'[M_1]) \rightarrow (\widetilde{D}', H', L', I', M'_1)$. $\Gamma \vdash M : \tau \ \& \ \varphi$ must have been derived by (T-InInterrupt). Thus, we have
  - (3) $\Gamma \vdash M_2 : \tau \ \& \ (lev_1, lev'_1)$
  - (4) $\Gamma \vdash E'[M_1] : \mathbf{unit} \ \& \ (lev_2, lev'_2)$
  - (5) $\Gamma \vdash M_3 : \mathbf{unit} \ \& \ (lev_3, lev'_3)$
  - (6) $lev'_1 < lev_2$
  - (7) $lev'_3 < lev_2$ and
  - (8) $\varphi = (lev_1, lev'_1) \sqcup (lev_2, lev'_2) \sqcup (lev_3, lev'_3)$

  for some lock levels. From (2), (4) and I.H., we have
  - (9) $\Gamma' \vdash M'_1 : \mathbf{unit} \ \& \ (lev_4, lev'_4)$
  - (10) $\vdash_{Env} (\widetilde{D}', H', L') : \Gamma'$
  - (11) $\Gamma' \supseteq \Gamma$ and
  - (12) $(lev_4, lev'_4) \leq (lev_2, lev'_2)$.

  From (11), (3), (5) and Lemma 5, we have
  - (13) $\Gamma' \vdash M_2 : \tau \ \& \ (lev_1, lev'_1)$
  - (14) $\Gamma' \vdash M_3 : \mathbf{unit} \ \& \ (lev_3, lev'_3)$

  From (6), (7) and (12), we have $lev'_1 < lev_4$ and $lev'_3 < lev_4$. Thus, we have $\Gamma' \vdash M' : \tau \ \& \ (lev_1, lev'_1) \sqcup (lev_4, lev'_4) \sqcup (lev_3, lev'_3)$. From (12), $\varphi' \leq \varphi$ as required.

□

B Proof of Lemma 4

This section proves Lemma 4.

Lemma 8. *If $\Gamma \vdash E[M] : \tau \ \& \ (lev_1, lev_2)$, then $\Gamma \vdash M : \tau' \ \& \ (lev'_1, lev'_2)$ and $lev_1 \leq lev'_1$ for some τ', lev'_1 and lev'_2.*

Proof Sketch. $\Gamma \vdash E[M] : \tau' \ \& \ (lev'_1, lev'_2)$ follows from induction on the structure of E. To prove $lev_1 \leq lev'_1$, note that in every typing rule, first components of effect parts of type judgments are monotonic. □

Lemma 9. *If $\Gamma \vdash M : \tau \ \& \ \varphi$ and Γ contains only reference types, function types and lock types, then $M = v$ for some v or $M = E[i]$ for some E and i.*

Proof Sketch. Induction on the derivation of $\Gamma \vdash M : \tau \ \& \ \varphi$. □

Lemma 10. *If $wellformed(L, I, E[M_1 \blacktriangleleft_{M'} M_2])$, then M_1 does not contain* **in_disable_int**.

Proof. If $I =$ **enabled**, M_1 does not contain **in_disable_int** because $wellformed(L, I, E[M_1 \blacktriangleleft_{M'} M_2])$. Suppose $I =$ **disabled**. In this case $E[M_1 \blacktriangleleft_{M'} M_2] = E''[\mathbf{in\_disable\_int}\ M''']$ for some E'' and M''' and neither E'' nor M''' contain **in_disable_int**. From the definition of evaluation contexts, M_1 is contained in E'' or M'''. Thus, M_1 does not contain **in_disable_int** as required. □

Lemma 11. *If $\Gamma, x : \mathbf{lock}(lev) \vdash M_1 : \tau \ \&\ (lev_1, lev_2)$ and $AckIn(x, M_1)$ and $wellformed(L, I, M_1 \blacktriangleleft_M M_2)$ hold, then $lev \leq lev_2$.*

Proof Sketch. Induction on the derivation of $AckIn(x, M_1)$. From $AckIn(x, M_1)$, we have $M_1 = E[\mathbf{in\_sync}\ x\ \mathbf{in}\ M']$ or $M_1 = E[M'_1 \blacktriangleleft_{M'} M'_2]$ for some E, M'_1, M' and M'_2. Here, E does not contain $\mathbf{in\_disable\_int}\ E'$ from Lemma 10. Thus, by noting that the second components of effects are monotonic in the typing rules other than (T-DisableInterrupt) and (T-InDisableInterrupt), we have $lev \leq lev_2$. □

Proof of Lemma 4. From $\vdash_C (\widetilde{D}, H, L, I, M) : \tau$, we have $\vdash_{Env} (\widetilde{D}, H, L) : \Gamma$ and $\Gamma \vdash M : \tau \ \&\ (lev_1, lev_2)$. Let L_a be the set $\{y \in \mathbf{Dom}(L) | L(y) = \mathbf{acquired}\}$. and let $y \in L_a$ be a lock whose level lev in Γ is maximum among the levels of the locks in L_a. From $L(y) = \mathbf{acquired}$ and $wellformed(L, I, M)$, we have $AckIn(y, M)$. We perform case-analysis on the last rule that derives $AckIn(y, M)$.

- Case (AckIn-Base): We have $M = E[\mathbf{in\_sync}\ y\ \mathbf{in}\ M']$ for some E and M'. Thus, from Lemma 8, we have $\Gamma \vdash \mathbf{in\_sync}\ y\ \mathbf{in}\ M' : \tau' \ \&\ (lev'_1, lev'_2)$. That judgment must have been derived by (T-Insync). Thus, we have (1) $\Gamma \vdash M' : \tau' \ \&\ (lev''_1, lev''_2)$ and (2) $lev < lev''_1$. From Lemma 9, we have $M' = v$ for some v or $M' = E'[i]$ for some E' and i. (Note that Γ contains only reference types, function types and lock types because $\vdash_{Env} (\widetilde{D}, H, L) : \Gamma$ is derived by (T-Env).)
  - If $M' = v$, then $\neg deadlocked(L, M)$ because $M = E[\mathbf{in\_sync}\ y\ \mathbf{in}\ v]$.
  - Suppose that $M' = E'[i]$.
    - If $i \neq \mathbf{sync}\ y'\ \mathbf{in}\ M''$, then $\neg deadlocked(L, M)$ because $M = E_1[i]$ where $E_1 = E[\mathbf{in\_sync}\ y'\ \mathbf{in}\ E']$ and $i \neq \mathbf{sync}\ y'\ \mathbf{in}\ M''$.
    - Suppose that $i = \mathbf{sync}\ y'\ \mathbf{in}\ M''$ for some y' and M''. From Lemma 8 and (1), we have (3) $\Gamma \vdash \mathbf{sync}\ y'\ \mathbf{in}\ M'' : \tau'' \ \&\ (lev'''_1, lev'''_2)$ and (4) $lev''_1 \leq lev'''_1$. (3) must have been derived by (T-Sync). Thus, we have (5) $\Gamma(y') = \mathbf{lock}(lev')$ and (6) $lev' = lev'''_1$. From (2), (4) and (6), we have $lev < lev'$. Since lev is the maximum level of acquired locks, y' is not acquired. Thus, $\neg deadlocked(L, M)$ holds because $M = E[\mathbf{in\_sync}\ y\ \mathbf{in}\ E'[\mathbf{sync}\ y'\ \mathbf{in}\ M'']]$ where $L(y') = \mathbf{released}$.
- Case (AckIn-Interrupt): $M = E[M_1 \blacktriangleleft_{M'} M_2]$ and $AckIn(y, M_1)$ for some E, M_1, M' and M_2. Thus, from Lemma 8, we have $\Gamma \vdash M_1 \blacktriangleleft_{M'} M_2 : \tau' \ \&\ \varphi$. Since that judgment must have been derived by (T-InInterrupt), we have (1) $\Gamma \vdash M_1 : \tau' \ \&\ (lev'_1, lev'_2)$ and (2) $\Gamma \vdash M_2 : \mathbf{unit} \ \&\ (lev''_1, lev''_2)$ and (3) $lev'_2 < lev''_1$. From Lemma 9, $M_2 = v$ for some v or $M_2 = E'[i]$ for some E' and i.
  - Case $M_2 = v$: $\neg deadlocked(L, M)$ because $M = E[M_1 \blacktriangleleft_{M'} v]$.
  - Case $M_2 = E'[i]$: If $i \neq \mathbf{sync}\ y'\ \mathbf{in}\ M''$, then $\neg deadlocked(L, M)$. Consider the case $i = \mathbf{sync}\ y'\ \mathbf{in}\ M''$. From (2) and Lemma 8, we have (4) $y' : \mathbf{lock}(lev') \in \Gamma$ and (5) $\Gamma \vdash \mathbf{sync}\ y'\ \mathbf{in}\ M'' : \tau' \ \&\ (lev'_1, lev'''_2)$ and (6) $lev''_1 \leq lev'$. From $wellformed(L, I, M)$ and $AckIn(y, M_1)$ and (1), we have (7) $lev \leq lev'_2$. From (3), (6) and (7), we have $lev < lev'$. Since lev is the maximum level among acquired locks, y' is not acquired. Thus, we have $\neg deadlocked(L, M)$.

□