# Secure Information Flow in ULM as a Safety Property

Zhengqin Luo
INRIA Sophia-Antipolis

April 4, 2009

**Abstract**

Following the line of work of [4], we study the secure information flow property as a standard safety property, namely that one should not store in a public location values elaborated using confidential information, under the context of *reactive programs* which manipulate broadcasting signals and synchronized threads. We show that in the reactive programming setting, it opens covert channels for leaking secret information by broadcasting signals and testing signals. From the point of programmers' view, the principles of safe programming are extended: one should never do something observable elaborated using confidential information. Particularly, we take the core ULM language [2], a ML-like multi-thread programming language extended with suspension and preemption constructs and non-deterministic scheduling mechanism, and show that any behavior violating those principles is an run-time error according to a security-minded semantics which dynamically check information flow. Moreover, we design a type and effect system to guarantee this safety property. Hopefully, the approach is able to be extended to different variants of reactive programming style.

## 1 Introduction

The issue of developing secure software has gained many focuses over last two decades. One of central and long-standing problem in this area is secure information flow problem, aimed at preventing programs from leaking confidential information (such as credit card number) to public in a language-based setting.

The information-flow security is a well established theory, providing static analysis technique to ensure the most famous property, *non-interference*, for programs in various languages. Informally, this property states that differences in secret inputs should not have effect on outputs of programs. The usual approach is to have a type system, and to show the soundness result, namely the typable programs satisfy non-interference. In some concurrent languages, this non-interference property is extend to some properties based on notion of bisimulation (such as "non-disclosure" in [5]).

However, it has been argued in [4] that, it is certainly not easy to find a counter-example to those programs which are not typable. Actually it is possible to have non-interferent programs which are not typable. For example the program below (informally we use footnote $L$ to represent a public value, and $H$ for a secret value), where we use ML jargon

$$v_L := (\mathsf{if}\ !u_H\ \mathsf{then}\ E\ \mathsf{else}\ E')$$

will be rejected by a standard type system, although it might be secure if $E$ and $E'$ always evaluate to same value. Nevertheless, the assignment to public location $v_L$ should be seen as a *programming error*, since the value stored contain secret information from dereferencing $u_H$. From a programmers' point view, one should not write secure programs in such a risk way. Then the approach is to give a security-mind semantics to the language, marking any attempt of assigning to public locations values elaborating using confidential information as *runtime error*, and the security property is a standard safety property, namely no runtime error will happen.

In this paper, we consider to extend this approach in a reactive programming (a.k.a. synchronous programming) setting , which is similar to concurrent programming. But it has mechanism to cooperatively synchronize between different threads without worrying about unpredictable behavior caused by non-deterministic thread scheduling, that is a thread will not yield the scheduler until it is suspended or terminated. Also a suspension can only be triggered by waiting a currently absent signal. It also provides a time-out mechanism (preemption) to control waiting time of a given thread. The main feature in this programming style is:

- *broadcast signal* Program components react according to the absence or presence of signals, by computing, and emitting signals which will be broadcast to each component of a given synchronous area.
- *suspension* Program components may be in a suspended state by waiting a signal which is absent at the moment.
- *preemption* There are ways to give up the execution of a program component (time-out), depend whether a given signal is present.
- *instants* Instant are successive periods of the execution of a program, where all the signals are reset to be absent at the start of the instant.

This reactive programming paradigm certainly opens new covert channels for leaking information flow. First let us consider the synchronous area of signal environment. One can imagine it as a set of boolean variables shared by threads. Thus, emitting a signal will correspond to updating the variable to true, testing a signal then actually is checking whether the value of given variable is true. Similar to the locations, we can distinguish between secret signals and public signals. Therefore, the meaning of "storing information in public locations" is extended, which includes also emitting public signals. The following program should be ruled out as insecure program as well in reactive setting, where emit construct will broadcast the signal to the signal environment:

$$\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{emit}\ s_L)\ \mathsf{else}\ (\mathsf{emit}\ s'_L)$$

because public view of the signal environment is dependent on the confidential information stored in $u_H$.

The other way of covert channel might be opened by testing a confidential signal with reactive constructs, due to that the suspension and preemption construct will change the order of the execution of threads, thus implicitly given some secret information. If one consider the following program, where informally the when construct will execute the code only if being waited signal is present, and "||" represents parallel composition of threads:

$$(\mathsf{when}\ s_H\ \mathsf{do}\ v_L := 1)||(\mathsf{emit}\ s_H); v_L := 2$$

If the secret signal $s_H$ is not initially present, then the thread one the left will get suspended, and the right one will emit the signal and assign $v_L$ with 2, then the left one will resume from suspension, assign $v_L$ with 1. In the other case, where $s_H$ is initially present, the final result for $v_L$ will be 1. So it has an insecure flow from $s_H$ to $v_L$.

In this paper, we investigate that to what extent the reactive construct will influence the flow of information in reactive programming paradigm, and we show how to define a run-time security error in this particularly setting, to prevent all the explicit and implicit insecure information flow. We consider here a core language which is a subset of ULM language that extends the imperative ML-like language with reactive primitive for suspension (when) and preemption (watch), together with a non-deterministic semantics for scheduling over threads. We implement the semantics as an abstract machine. At each step of the evaluation, we record a confidential level and a suspension level, respectively, for indicating the security level of reading and testing of signals might influence current thread so far. Then assignment to a location or emitting a signal which have level lower than currently recorded level will rise an run-time error. Especially, we separately record these two levels for each thread, to obtain the enough flexibility for different thread.

The rest of the paper is organized as follows. In Section 2 we introduce the language and its operational semantics, we will also show some examples of secure and insecure flow accounting for the semantics. Then in Section 3 we define the type system, and show some necessary properties of typed programs, and also the soundness result w.r.t the safety property.

$$
\begin{array}{rcl}
M, N & ::= & V \mid (\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N') \mid (MN) \mid (M; N) \\
& \mid & (\mathsf{ref}_{l,\theta} M) \mid (!M) \mid (M := N) \\
& \mid & \mathsf{sig}_l \mid (\mathsf{emit}\ M) \mid (\mathsf{thread}\ M) \\
& \mid & (\mathsf{when}\ M\ \mathsf{do}\ N) \mid (\mathsf{watch}\ M\ \ Do\ N) \\
\\
V, W & ::= & x \mid s_l \mid u_{l,\theta} \mid \lambda x M \mid tt \mid ff \mid ()
\end{array}
$$

Figure 1: Syntax of the language

# 2 The language

## 2.1 Syntax

The language in which we are interested here is an slight variant of the ULM, an ML-like imperative higher-order language extended with suspension and preemption constructs. Differing from the original ULM language, we confine us here to the local behavior of programs without considering the mobility of codes, and we also adopt a slightly different scheduling policy for local threads.

In the language, when a location or a signal is created, it will be assigned with a confidentiality level. We assume a security lattice $(\mathcal{L}, \preceq)$ for *confidentiality levels*. The least and greatest element in $\mathcal{L}$ is written $\bot$ and $\top$, respectively. And for any $l, l' \in \mathcal{L}$, we denote the meet (greatest lower bound) of them $l \curlywedge l'$ and the join (least upper bound) of them $l \curlyvee l'$. For ease of representation, we also use $H$ and $L$ to denote high level and low level, respectively, with $L \preceq H$

We assume two infinite and disjoint sets for signals and variables, $\mathcal{S}$ and $\mathcal{N}$. The syntax of the language is given in Figure 1.

In the grammar, $s_l$ is any signal name annotated with confidentiality level $l$, where $s \in \mathcal{S}$. $u_{l,\theta}$ is any reference annotated with security level $l$ and type $\theta$, where $u \in \mathcal{N}$. The definition of bound and free variables are as usual, and an expression is *closed* if it does not contain free variables. We also denote by $\{x \mapsto N\}M$ the capture-avoiding substitution of free variable $x$ in $M$ with $N$.

A reactive machine contains a collection of concurrent threads, organized in a multi-set denoted by $T$. All these threads share the same store $\mu$ and signal environment $\xi$. We precisely define the *configuration* of a machine as a quadruples of following form:

$$\mathcal{M} = [\mu, \xi, t, T]$$

where

- $\mu$ is the memory store shared by all threads, mapping each location in $\mathsf{dom}(\mu)$ to a value. We denote by $\mu[u_{l,\theta} := V]$ updating a reference;
- The signal environment $\xi \subseteq \mathcal{S}$ is the set of currently present signal;
- $t$ is the currently running thread, explained below in the semantics;
- $T$ is the multi-set of waiting threads, of the form $T = \{t_1, t_2, \cdots, t_n\}$

For the imperative constructs for the language is quite standard, details can be find in [?]. Let us briefly introduce the reactive construct now. Roughly speaking, an important feature of reactive programming is the ability of synchronizing by signals. By emit construct one can broadcast signals to signal environment $\xi$. The when construct will test whether a given signal is present or not, if it is present then the current thread will continue to execute, otherwise it will suspend to yield the scheduler to choose another non-suspended thread to execute. If current thread and all thread in the multi-set $T$ are suspended, the watch construct will take effect, killing its body if and only if the watching signal is present.

$$
\begin{array}{rcl}
S & ::= & \varepsilon \mid S \cdot \mathbf{F} \\
\mathbf{F} & ::= & (\mathsf{if}\ [\]\ N_0 N_1) \mid ([]\langle N\rangle^l) \mid (\langle V\rangle^l[]) \mid ([]; \langle N\rangle^l) \\
& \mid & (\langle \mathsf{ref}_{l,\theta}\rangle^{l'}[]) \mid ([] := \langle N\rangle^l) \mid (\langle V\rangle^l :=^{l'} []) \\
& \mid & (\langle \mathsf{emit}\rangle^l[]) \mid (\langle \mathsf{when}\ []\ \mathsf{do}\ \langle N\rangle^l)(\mathsf{when}\ s_l\ \mathsf{do}\ []) \\
& \mid & (\mathsf{watch}\ []\ \mathsf{do}\ \langle N\rangle^l) \mid (\mathsf{watch}\ \langle s_l\rangle^{l'}\ \mathsf{do}\ []) \\
& \mid & (\mathsf{watch}\ s_l\ \mathsf{do}\ [])
\end{array}
$$

Figure 2: Definition of stack frames

## 2.2 Operational semantics

The operational semantics are define by means of an abstract machine for each independent thread. It augment the standard machines by separately building a reading level and a suspension level while computing the value of an expression. We first define this abstract machine on single thread, and then lift the thread transitions to transitions over configuration.

### 2.2.1 Thread transition (abstract machine)

*Running threads* are organized as following form:

$$t = (\mathsf{pc}, \mathsf{cur}, S, M)$$

where

- pc records the upper bound of the *confidentiality* level of reading that may influence the final evaluation result of expression $M$ with the control stack $S$;
- cur records the *suspension* level, which is the upper bound of levels of reading references or testing signals that may influence suspensions during the evaluation to $M$ with the control stack $S$;
- $S$ is the control stack, which is a sequence of stack frame (given below);
- $M$ is the code to evaluate.

Notice that we separately record the level pc and cur for each threads, from which high level reading in one thread will not influence low level writing in another thread. The syntax of stacks and stack frames are given in Figure 2. Here $\varepsilon$ denotes the empty stack. Given a store $\mu$ and a signal environment $\xi$, the thread transitions has the form:

$$(\mu, \xi, t) \xrightarrow{t''} (\mu', \xi', t')$$

where $t'' = ()$ when there is no new thread spawned, otherwise $t''$ is the thread created in this step. The rules of thread transition are given below in Table 1.

We remark here that the thread transitions can only occur when a thread is not suspended, that is it is not waiting the presence of a signal by when construct. We define the suspension predicate $(\xi, S)\dagger$ , which is formalized by the following rule:

$$\frac{S = S' \cdot (\mathsf{when}\ s_l\ \mathsf{do}\ []) \cdot S'' \quad s_l \notin \xi}{(\xi, S)\dagger}$$

We also write $(\xi, S)\downarrow$ for $\neg(\xi, S)\dagger$.

| *code* | *transition* |
|---|---|
| (if $M$ then $N_0$ else $N_1$) | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\text{if } M \text{ then } N_0 \text{ else } N_1))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\text{if } [] \ N_0 N_1), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\text{if } [] \ N_0 N_1), tt)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, N_0))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\text{if } [] \ N_0 N_1), ff)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, N_1))$ |
| $(MN)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (MN))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([]\langle N\rangle^{\mathsf{pc}}), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([]\langle N\rangle^{l}), V)) \xrightarrow{()} (\mu, \xi, (l, \mathsf{cur}, S \cdot (\langle V\rangle^{\mathsf{pc}}[]), N))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \lambda x M\rangle^{l}[]), V)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc} \curlyvee l, \mathsf{cur}, S, \{x \mapsto V\}M))$ |
| $(M; N)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (M; N))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([]; \langle N\rangle^{\mathsf{pc}}), M))$<br>$(\mathsf{pc}, \mathsf{cur}, S \cdot ([]; \langle N\rangle^{l}), V)) \xrightarrow{()} (\mu, \xi, (l, \mathsf{cur}, S, N))$ |
| $(\mathsf{ref}_{l,\theta}\ M)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{ref}_{l,\theta}\ M))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{ref}_{l,\theta}\rangle^{\mathsf{pc}}[]), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{ref}_{l,\theta}\rangle^{l'}[]), V)) \xrightarrow{()} (\mu \cup \{u_{l,\theta} \mapsto V\}, \xi, (l', \mathsf{cur}, S, u_{l,\theta}))$<br>where $u_{l,\theta}$ is fresh and $\boxed{(\mathsf{pc} \curlyvee \mathsf{cur}) \preceq l}$ |
| $(!M)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (!\ M))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (![]), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (![]), u_{l,\theta})) \xrightarrow{()} (\mu, \xi, (\mathsf{pc} \curlyvee l, \mathsf{cur}, S, V))$<br>where $\mu(u_{l',\theta}) = V$ |
| $(M := N)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (M := N))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([] := \langle N\rangle^{\mathsf{pc}}), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([] := \langle N\rangle^{l}), V)) \xrightarrow{()} (\mu, \xi, (l, \mathsf{cur}, S \cdot (\langle V\rangle^{\mathsf{pc}} :=^{l} []), N))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle u_{l,\theta}\rangle^{l_0} :=^{l_1} []), V)) \xrightarrow{()} (\mu[u_{l,\theta} \mapsto V], \xi, (l_1, \mathsf{cur}, S, ()))$<br>where $\boxed{(\mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l_0) \preceq l}$ |
| $(\mathsf{sig}_l)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{sig}_l))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, s_l))$<br>where $s_l$ is fresh |
| $(\mathsf{emit}\ M)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{emit}\ M))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{emit}\rangle^{\mathsf{pc}}[]), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{emit}\rangle^{l_0}[]), s_l)) \xrightarrow{()} (\mu, \xi \cup \{s_l\}, (l_0, \mathsf{cur}, S, ()))$<br>where $\boxed{\mathsf{pc} \curlyvee \mathsf{cur} \preceq l}$ |
| $(\mathsf{thread}\ M)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{thread}\ M))) \xrightarrow{t} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, ()))$<br>where $t = (\mathsf{pc}, \mathsf{cur}, S^{\mathsf{w}}, M)$ |
| $(\mathsf{when}\ M\ \mathsf{do}\ N)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{when}\ M\ \mathsf{do}\ N))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{when}\ []\ \mathsf{do}\ \langle N\rangle^{\mathsf{pc}}), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{when}\ []\ \mathsf{do}\ \langle N\rangle^{l}), s_{l'})) \xrightarrow{()} (\mu, \xi, (l, \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l', S \cdot (\mathsf{when}\ s_{l'}\ \mathsf{do}\ []), N))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{when}\ s_{l'}\ \mathsf{do}\ []), V)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, V))$ |
| $(\mathsf{watch}\ M\ \mathsf{do}\ N)$ | $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{watch}\ M\ \mathsf{do}\ N))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{watch}\ []\ \mathsf{do}\ \langle N\rangle^{\mathsf{pc}}), M))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{watch}\ []\ \mathsf{do}\ \langle N\rangle^{l}), s_{l'})) \xrightarrow{()} (\mu, \xi, (l, \mathsf{cur}, S \cdot (\mathsf{watch}\ \langle s_{l'}\rangle^{\mathsf{pc}}\ \mathsf{do}\ []), N))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{watch}\ \langle s_{l'}\rangle^{l}\ \mathsf{do}\ []), V)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, V))$<br>$(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{watch}\ s_l\ \mathsf{do}\ []), V)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, V))$ |

Table 1: The Abstract Machine

### 2.2.2 Configuration transitions

To define the transitions over configurations, we first lift the thread transitions of the currently active thread to the configuration transitions. When a thread spawn a new thread $t'$, then new thread $t'$ will be added to the multi-set of waiting thread.

$$\frac{(\mu,\xi,t) \xrightarrow{()} (\mu',\xi',t')}{[\mu,\xi,t,T] \to [\mu',\xi',t',T]} \qquad \frac{(\mu,\xi,t) \xrightarrow{t'} (\mu',\xi',t'')}{[\mu,\xi,t,T] \to [\mu',\xi',t'',T \cup t']}$$

Then, when current thread is suspended or terminated, we should non-deterministic choose a non-suspended thread from the multi-set $T$ of waiting threads. Abusively, we use $(\xi,T)\downarrow$ to denote that there exists some threads in $T$ which is not suspended, formalized by following rules:

$$\frac{\neg(\xi,S)\dagger \quad M \notin \mathcal{V}al \text{ or } S \neq \varepsilon}{(\xi,t)\downarrow} \; t = (\mathsf{pc},\mathsf{cur},S,M) \qquad \frac{(\xi,T)\downarrow}{(\xi,T \cup T')\downarrow} \qquad \frac{(\xi,T)\downarrow}{(\xi,T' \cup T)\downarrow}$$

Now the rule of scheduling transition is as follows:

$$\frac{\neg(\xi,t)\downarrow \quad (\xi,T)\downarrow \quad t' \in T \quad (\xi,t')\downarrow}{[\mu,\xi,t,T] \longmapsto [\mu,\xi,t',(T \backslash \{t'\}) \cup \{t\}]}$$

When all thread are suspended, the instant transition will occur. During the instant transition, any suspended watch construct should be eliminated when the being waited signal is present. We also update the suspension level $l_s(\mathsf{cur})$ for each thread according to the watch constructs in their control stacks. The signal environment will be reset to empty set after the transition.

We define $\kappa^\xi$ to do such a transformation on a single thread, where $\xi$ is the current present signal environment. We also extend $\kappa^\xi$ to the multi-set $T$ of waiting threads:

$$\begin{aligned}
\kappa^\xi(\mathsf{pc},\mathsf{cur},\varepsilon,M) &= (\mathsf{pc},\mathsf{cur},\varepsilon,M) \\
\kappa^\xi(\mathsf{pc},\mathsf{cur},(\mathsf{when}\ s_l\ \mathsf{do}\ [])\cdot S,M) &= \begin{cases} (\mathsf{pc},\mathsf{cur},(\mathsf{when}\ s_l\ \mathsf{do}\ [])\cdot S,M) & \text{if } s_l \notin \xi \\ (\mathsf{pc}',\mathsf{cur}',(\mathsf{when}\ s_l\ \mathsf{do}\ [])\cdot S',M') & \text{otherwise} \end{cases} \\
\kappa^\xi(\mathsf{pc},\mathsf{cur},(\mathsf{watch}\ \langle s_l\rangle^{l'}\ \mathsf{do}\ [])\cdot S,M) &= \begin{cases} (\mathsf{pc},\mathsf{cur} \curlyvee l \curlyvee l',\varepsilon,()) & \text{if } s_l \in \xi \\ (\mathsf{pc}',\mathsf{cur}' \curlyvee l \curlyvee l',(\mathsf{watch}\ s_l\ \mathsf{do}\ [])\cdot S',M') & \text{otherwise} \end{cases} \\
\kappa^\xi(\mathsf{pc},\mathsf{cur},(\mathsf{watch}\ s_l\ \mathsf{do}\ [])\cdot S,M) &= \begin{cases} (\mathsf{pc},\mathsf{cur},\varepsilon,()) & \text{if } s_l \in \xi \\ (\mathsf{pc}',\mathsf{cur}',(\mathsf{watch}\ s_l\ \mathsf{do}\ [])\cdot S',M') & \text{otherwise} \end{cases} \\
\kappa^\xi(\mathsf{pc},\mathsf{cur},\mathbf{F}\cdot S,M) &= (\mathsf{pc}',\mathsf{cur}',\mathbf{F}\cdot S',M') \\
& \qquad \text{where } (\mathsf{pc}',\mathsf{cur}',S',M') = \kappa^\xi(\mathsf{pc},\mathsf{cur},S,M)
\end{aligned}$$

$$\kappa^\xi(T) = \{\kappa^\xi(t_1),\ldots,\kappa^\xi(t_n)\} \text{ where } T = \{t_1,\ldots,t_n\}$$

Finally, the rules of instant transition is

$$\frac{\neg(\xi,t)\downarrow \quad \neg(\xi,T)\downarrow}{[\mu,\xi,t,T] \hookrightarrow [\mu,\emptyset,\kappa^\xi(t),\kappa^\xi(T)]}$$

## 2.3 Secure and Insecure programs

Before going into the semantics, let us first see some example of insecure programs, that we want to mark as run-time error. We will abusively use "||" again as the informal parallel composition of thread.

First, we sure want to rule out explicit flow as usual, thus in the semantics we should increase the reading level when we execute the dereferencing.

$$(u_L := v_H) \tag{1}$$

$$(\mathsf{ref}_L(!u_H)) \tag{2}$$

The usual implicit flow must be also be ruled out, for doing this we should transmit the reading level from evaluating the guard

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ v_L := tt\ \mathsf{else}\ v_L := ff) \tag{3}$$

However we don't want the semantics to be too conservative. For example in the sequential composition of programs $(M; N)$, the reading level concerning $M$ won't influence the result of computing $N$, that is the following programs should be considered as secure:

$$!u_H; (v_L := tt)$$

Thus, the reading level should not be propagated along sequential composition.

In the application construct, the following should be considered as insecure flow,

$$(\lambda x(v_L := x)(!u_H)) \tag{4}$$

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ \lambda x(v_L := tt)\ \mathsf{else}\ \lambda x(v_L := ff))() \tag{5}$$

Basically, we also have to take into consideration the security level accumulated for evaluating the function and the argument separately.

Extended to the reactive setting, the definition of insecure flow is widened. When emitting a signal, the following implicit flow is a run-time error.

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{emit}\ s_L)\ \mathsf{else}\ ()) \tag{6}$$

Implicit flow can also occur when spawning a new threads. To prevent this we should keep the current recorded level from parent thread to child thread.

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{thread}\ (v_L := tt))\ \mathsf{else}\ ()) \tag{7}$$

Since our language adopts cooperative scheduling, the only way for switching threads is by termination or suspension. Thus those suspension which depend on high level information may influence the order of "low level commands". For example

$$\begin{array}{c}((\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{when}\ s_L\ \mathsf{do}\ ())\ \mathsf{else}\ ()); v_L = tt; (\mathsf{emit}\ s_L))\ ||\\ ((\mathsf{if}\ !u_H\ \mathsf{then}\ ()\ \mathsf{else}\ (\mathsf{when}\ s_L\ \mathsf{do}\ ())); v_L = ff; (\mathsf{emit}\ s_L))\end{array} \tag{8}$$

show be note as insecure. Dissimilar to the reading level, as the suspension will affect all the up-coming computation, there is no way to recover the suspension level as we do for sequential computation. The principle should be that there is no low level behavior after testing a high level signal. Also we mark following programs as insecure:

$$(\mathsf{when}\ s_H\ \mathsf{do}\ (u_L = tt))\ ||\ (\mathsf{if}\ !v_H\ \mathsf{then}\ (\mathsf{emit}\ s_H)) \tag{9}$$

$$((\mathsf{when}\ s_H\ \mathsf{do}\ ()); u_L = tt)\ ||\ (\mathsf{if}\ !v_H\ \mathsf{then}\ (\mathsf{emit}\ s_H)) \tag{10}$$

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ ()\ \mathsf{else}\ (\mathsf{when}\ s_L\ \mathsf{do}\ ())); v_L = ff\ ||\ (\mathsf{emit}\ s_L); v_L = tt \tag{11}$$

$$\begin{array}{c}(\mathsf{if}\ !u_H\ \mathsf{then}\ u_H := (\lambda x(\mathsf{when}\ s_L\ \mathsf{do}\ ()))\\ \mathsf{else}\ u_H := (\lambda x()); (!u_H()); v_l := tt\ ||\\ (\mathsf{emit}\ s_L); v_L := ff\end{array} \tag{12}$$

Although the preemption does not cause suspension, it causes the recovery of the suspension. So the tested signal might leak some information. The following examples show some insecure flow.

$$\begin{array}{l}(\mathsf{watch}\ s_H\ \mathsf{do}\ (\mathsf{when}\ s_L\ \mathsf{do}\ (v_L := tt)))\ ||\\ (\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{emit}\ s_H)\ \mathsf{else}\ ()); \mathsf{pause}; (\mathsf{emit}\ s_L)\end{array} \tag{13}$$

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ (\mathsf{emit}\ s_H)\ \mathsf{else}\ ());\mathsf{pause};(\mathsf{emit}\ s_L; v_L := \mathit{ff})\ ||\\ (\mathsf{watch}\ s_H\ \mathsf{do}\ (\mathsf{when}\ s_L\ \mathsf{do}\ ()));v_L := \mathit{tt} \tag{14}$$

where pause can be coded by only using when and watch for some fresh $s_L$ and $s'_L$:

$$\mathsf{pause} := (\mathsf{watch}\ s_L\ \mathsf{do}\ ((\mathsf{emit}\ s_L);\mathsf{when}\ s'_L\ \mathsf{do}()))$$

Note that preemption only occurs at the end of each instant, so actually the testing of the signal occurs at the end of instants. To be flexible, we should only increase the suspension level caused by preemption at the end of instants. For example the following program is secure:

$$(\mathsf{watch} s_H \mathsf{do}(u_L := \mathit{tt}));v_L := \mathit{ff}$$

## 2.4 Explanations

Now let us explain how the information flow is controlled in the semantics. First, one should observe that if we abandon security levels recorded in the semantics, this abstract machine is just a standard stack machine. For example, the rules for application are:

$$\begin{array}{rcl}(\mu,\xi,(S,(MN))) & \xrightarrow{()} & (\mu,\xi,(S\cdot([]N),M))\\ (\mu,\xi,(S\cdot([]N),V)) & \xrightarrow{()} & (\mu,\xi,(S\cdot(V[]),N))\\ (\mu,\xi,(S\cdot(\lambda xM[]),V)) & \xrightarrow{()} & (\mu,\xi,(S,\{x\mapsto V\}M))\end{array}$$

During the evaluation of the thread, we record the pc as the reading level which will effect the result of current running thread. We note that this level can only increase by reading from a location. The security level of the location will be added to the reading level.

$$\begin{array}{l}(\mu,\xi,(\mathsf{pc},\mathsf{cur},S\cdot(![]),u_{l,\theta}))\\ \quad\xrightarrow{()}(\mu,\xi,(\mathsf{pc}\curlyvee l,\mathsf{cur},S,\mu(u_{l,\theta})))\end{array}$$

However, we should be careful that this level will not accumulate levels from unnecessary read operations that will not influence the current expression. For example in the evaluation of application $(MN)$ or sequence $(M;N)$, certainly the reading happened in $M$ will not influence the evaluation of $N$. So what we have to do is to "forget" about the reading level concerning computing $M$. The strategy we use here is that when we start to evaluate $(MN)$ we push frame $([]N)$ together with current pc, when we pop the frame, we should recover the reading level recorded in frame to pc. For example,

$$\begin{array}{rcl}(\mu,\xi,(\mathsf{pc},\mathsf{cur},S,(MN))) & \xrightarrow{()} & (\mu,\xi,(\mathsf{pc},\mathsf{cur},S\cdot([]\langle N\rangle^{\mathsf{pc}}),M))\\ (\mu,\xi,(l_c,l_s,S\cdot([]\langle N\rangle^{\mathsf{pc}}),V)) & \xrightarrow{()} & (\mu,\xi,(\mathsf{pc},l_s,S\cdot(\langle V\rangle^{l_c}[]),N))\\ (\mu,\xi,(l'_c,l'_sS\cdot(\langle\lambda xM\rangle^{l_c}[]),V)) & \xrightarrow{()} & (\mu,\xi,(l_c\curlyvee l'_c,l'_s,S,\{x\mapsto V\}M))\end{array}$$

We use similar strategy for evaluating sub-expression in sequence, reference creating, assignment, emit signal, suspension construct and preemption construct.

When we have evaluated the guard of the conditional construct, we keep the current reading level, to prevent implicit flow.

$$(\mu,\xi,(\mathsf{pc},\mathsf{cur},S\cdot(\mathsf{if}\ []\ N_0N_1),\mathit{tt}))\xrightarrow{()}(\mu,\xi,(\mathsf{pc},\mathsf{cur},S,N_0))$$

When we create a reference, or do an assignment, or emit a signal, we compare the level of target location to join of current reading level and suspension level, to prevent one from storing in low level location high level information:

$$(\mathsf{pc}\curlyvee\mathsf{cur})\preceq l$$

especially, in assignment we also take the level $l_0$ when evaluating the location into consideration.

When we test some signal by the when construct, we increase the current suspension level with the join of pc, cur and the level of tested signal $l$, because that the pc record the reading level that might influence the evaluation of the tested signal.

$$\begin{aligned}&(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{when}\ [\,]\ \mathsf{do}\ \langle N\rangle^{l}), s_{l'}))\\ &\quad \xrightarrow{()} (\mu, \xi, (l, \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l', S \cdot (\mathsf{when}\ s_{l'}\ \mathsf{do}\ [\,]), N))\end{aligned}$$

For watch construct it is a slight different, due to the fact that the testing of the signal only occurs at the end of each instant, then we could postpone to increase the suspension level at instant transition. Therefore we have to annotate in the stack frame the reading level used for evaluating the tested signal:

$$(\mathsf{watch}\ \langle s_{l'}\rangle^{\mathsf{pc}}\ \mathsf{do}\ [\,])$$

and the increasing of the level happens at the instant transition by $\kappa$ transformation.

$$\kappa^{\xi}(\mathsf{pc}, \mathsf{cur}, (\mathsf{watch}\ \langle s_l\rangle^{l'}\ \mathsf{do}\ [\,]) \cdot S, M) \quad = \quad \begin{cases} (\mathsf{pc}, \mathsf{cur} \curlyvee l \curlyvee l', \varepsilon, ()) & \text{if } s_l \in \xi \\ (\mathsf{pc}', \mathsf{cur}' \curlyvee l \curlyvee l', (\mathsf{watch}\ s_l\ \mathsf{do}\ [\,]) \cdot S', M') & \text{otherwise} \end{cases}$$

Finally, let us comment on the propagation of suspension level. Unlike the reading level. The suspension level gained from testing a signal will not only influence the body of reactive construct, but also the upcoming computation. Thus the suspension level is always transmitted to next step, to prevent implicit flow originated from testing a high level signal.

## 2.5 Secure programs

Then the definition of a secure program is straightforward: it should not run into security error, that is, violating the condition appear in the box defined in the semantics of abstract machine. One could easily see that if the condition is violated, then the computing will get stuck, with no further possible transition.

**Definition 2.1** *We denote a machine transition from one configuration $C$ to another configuration $C'$ as $C \rightarrowtail C'$ if one of following occurs*

- $C \rightarrow C'$ *(thread transition)*
- $C \longmapsto C'$ *(scheduling transition)*
- $C \hookrightarrow C'$ *(instant transition)*

To define the notion of the secure program, we introduce an unmonitored variant of the operational semantics on machine configurations,denoted by $\Rightarrow$, which is exactly defined as $\rightarrowtail$ except that we ignore all the conditions when we do "something observable" (those conditions appears in the box when we define the abstract machine). Obviously the unmonitored semantics is more flexible than the monitored one. Then then definition of a secure program is straightforward, that is all the security check during the semantics succeed. In another word, those checks are useless.

**Definition 2.2 (Secure Programs)** *A program $M$ is secure (from the confidentiality point of view) if and only if for a class of initial memory $\mu$ and signal environment $\xi$,*

$$[\mu, \xi, (\bot, \bot, M), \emptyset] \Rightarrow^{*} [\mu', \xi', t, T] \quad \textit{implies} \quad [\mu, \xi, (\bot, \bot, M), \emptyset] \rightarrowtail^{*} [\mu', \xi', t, T]$$

*such that $t$ is neither terminated (a value) nor suspended, but can not go on for further step.*

# 3 Type and effect system

In this section we show a standard type system which entails the safety property. The design of the type system follow the line of "state-oriented" approach [3, 1, 5]. Furthermore, we take the current reading level and suspension level into typing judgment, in order to type running thread. The types are defined by the following grammar:

$$\tau, \sigma, \theta \ldots \ ::= \ t \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{tsig}_l \mid \theta\mathsf{ref}_l \mid (\tau \xrightarrow{e} \sigma)$$

where $t$ is any type variables, $\mathsf{tsig}_l$ is the type of signals whose confidentiality level are $l$, and $e$ is any "security effect". The judgements of the type and effect system have the form

$$\mathsf{pc}; \mathsf{cur}; \Gamma \vdash M : e, \tau$$

where $\Gamma$ is a typing context, $e$ is a security effect, which is a triple $(e.\mathsf{c}, e.\mathsf{w}, e.\mathsf{s})$, and $\tau$ is a type. They can be explained as follows:

- $\mathsf{pc}$ is the initial confidentiality level.
- $\mathsf{cur}$ is the initial suspension level.
- $\Gamma$ maps each variable to a type.
- $e.\mathsf{c}$ is the *confidentiality* level of $M$, indicating the upper bound of the confidentiality levels of reading that may influence the final evaluation result of expression $(\mathsf{pc}, \mathsf{cur}, M)$;
- $e.\mathsf{w}$ is the *writing* level of $M$, that is the lower bound of levels of references that $(\mathsf{pc}, \mathsf{cur}, M)$ might update;
- $e.\mathsf{s}$ denotes the *suspension* level of $M$, which is the upper bound of levels of reading references or testing signals that may influence suspensions during the evaluation of $(\mathsf{pc}, \mathsf{cur}, M)$.
- $\tau$ is the type of $M$.

We denote $e.\mathsf{c} \curlyvee e.\mathsf{s}$ by $e.\mathsf{r}$.

## 3.1 Typing programs

The typing rule is given in Figure 3.

## 3.2 Typing running configurations

We also have to give types to a running configuration, in order to obtain the subject reduction property. Therefore, we extend the typing judgement to type the running thread, which have the following form

$$\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (S, M) : e, \tau$$

The rules are given in Figure 4.

To establish the safety results, we have to show how to type the configuration. For doing this we first define how to the memory. The typing judgement follows the form $\Gamma \vdash \mu$, which can be inferred by following rules:

$$\frac{}{\Gamma \vdash \emptyset} \qquad \frac{\Gamma \vdash \emptyset \quad \Gamma(u_{l,\theta}) = \theta \qquad \bot, \bot, \Gamma \vdash V : (\bot, \top\bot), \theta}{\Gamma \vdash \mu \cup \{u_{l,\theta} \mapsto V\}}$$

Finally, the typing rule for configuration is

$$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (S, M) : e, \tau}{\Gamma \vdash (\mathsf{pc}, \mathsf{cur}, S, M)} \qquad \frac{\Gamma \vdash \mu \quad \Gamma \vdash t \qquad \forall t_i \in T, \Gamma \vdash t_i}{\Gamma \vdash [\mu, \xi, t, T]}$$

$$\textsc{Loc}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash u_{l,\theta} : (\mathsf{pc},\top,\mathsf{cur}),\theta\mathsf{ref}_l} \qquad \textsc{Var}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma, x:\tau \vdash x : (\mathsf{pc},\top,\mathsf{cur}),\tau}$$

$$\textsc{BoolT}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash tt : (\mathsf{pc},\top,\mathsf{cur}),\mathsf{bool}} \qquad \textsc{BoolF}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash \mathit{ff} : (\mathsf{pc},\top,\mathsf{cur}),\mathsf{bool}}$$

$$\textsc{CreaSig}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash \mathsf{sig}_l : (\mathsf{pc},\top,\mathsf{cur}),\mathsf{tsig}_l} \qquad \textsc{Sig}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash s_l : (\mathsf{pc},\top,\mathsf{cur}),\mathsf{tsig}_l}$$

$$\textsc{Nil}\ \frac{}{\mathsf{pc};\mathsf{cur};\Gamma \vdash () : (\mathsf{pc},\top,\mathsf{cur}),\mathsf{unit}} \qquad \textsc{Abs}\ \frac{\bot;\bot;\Gamma, x:\tau \vdash M : e,\sigma}{\mathsf{pc};\mathsf{cur};\Gamma \vdash \lambda x M : (\mathsf{pc},\top,\mathsf{cur}),(\tau \xrightarrow{e} \sigma)}$$

$$\textsc{Cond}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\mathsf{bool} \quad e.\mathsf{c};e.\mathsf{s};\Gamma \vdash N_i : e_i,\tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : e \curlyvee e_0 \curlyvee e_1,\tau}$$

$$\textsc{App}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,(\tau \xrightarrow{e'} \sigma) \quad \mathsf{pc};e.\mathsf{s};\Gamma \vdash N : e'',\tau \quad (e.\mathsf{r} \curlyvee e''.\mathsf{r}) \preceq e'.\mathsf{w}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash MN : e \curlyvee e' \curlyvee e'',\sigma} \qquad \textsc{Seq}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\tau \quad \mathsf{pc};e.\mathsf{s};\Gamma \vdash N : e',\sigma}{\mathsf{pc};\mathsf{cur};\Gamma \vdash M;N : (\bot,e.\mathsf{w},e.\mathsf{s}) \curlyvee e',\sigma}$$

$$\textsc{Ref}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\theta \quad e.\mathsf{r} \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{ref}_{l,\theta} M) : (\mathsf{pc},e.\mathsf{w} \curlyvee l,e.\mathsf{s}),\theta\mathsf{ref}_l} \qquad \textsc{DeRef}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\theta\mathsf{ref}_l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (!M) : e \curlyvee (l,\top,\bot)}$$

$$\textsc{Assign}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\theta\mathsf{ref}_l \quad \mathsf{pc};e.\mathsf{s};\Gamma \vdash N : e',\theta \quad (e.\mathsf{r} \curlyvee e'.\mathsf{r}) \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (M := N) : (\mathsf{pc},e.\mathsf{w} \curlyvee e'.\mathsf{w} \curlyvee l,e.\mathsf{s} \curlyvee e'.\mathsf{s}),\mathsf{unit}}$$

$$\textsc{Thread}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\sigma}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{thread}\ M) : e,\mathsf{unit}} \qquad \textsc{Emit}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\mathsf{tsig}_l \quad e.\mathsf{r} \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{emit}\ M) : (\mathsf{pc},e.\mathsf{w} \curlyvee l,e.\mathsf{s}),\mathsf{unit}}$$

$$\textsc{When}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash, M : e,\mathsf{tsig}_l \quad \mathsf{pc};e.\mathsf{s} \curlyvee e.\mathsf{c} \curlyvee l;\Gamma \vdash N : e',\tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{when}\ M\ \mathsf{do}\ N) : e \curlyvee e',\tau}$$

$$\textsc{Watch}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e,\mathsf{tsig}_l \quad \mathsf{pc};e.\mathsf{s};\Gamma \vdash N : e',\tau \quad (e.\mathsf{c} \curlyvee l) \preceq e'.\mathsf{w}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{watch}\ M\ \mathsf{do}\ N) : e \curlyvee e',\tau}$$

Figure 3: The type and effect system

$$\textsc{Active}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e, \theta}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\varepsilon, M) : e, \theta}$$

$$\textsc{Cond-E}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \mathsf{bool} \quad e.\mathsf{c};e.\mathsf{s};\Gamma \vdash N_i : e_i, \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{if}\ []\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1)\cdot S, M) : e \curlyvee e_0 \curlyvee e_1, \tau}$$

$$\textsc{App-E1}\ \frac{\begin{array}{c}\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, (\tau \xrightarrow{e'} \sigma) \\ l;e.\mathsf{s};\Gamma \vdash N : e'', \tau \quad (e.\mathsf{r} \curlyvee e''.\mathsf{r}) \preceq e'.\mathsf{w}\end{array}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (([]\langle N\rangle^{l})\cdot S, M) : e \curlyvee e' \curlyvee e'', \sigma}$$

$$\textsc{App-E2}\ \frac{\begin{array}{c}\bot;\bot;\Gamma \vdash V : \bot, (\tau \xrightarrow{e'} \sigma) \\ \mathsf{pc};\mathsf{cur};\Gamma \vdash (S, N) : e'', \tau \quad (l \curlyvee e''.\mathsf{r}) \preceq e'.\mathsf{w}\end{array}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\langle V\rangle^{l}[])\cdot S, N) : (l, \top, \bot) \curlyvee e' \curlyvee e'', \sigma}$$

$$\textsc{Seq-E}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \tau \quad l;e.\mathsf{s};\Gamma \vdash N : e', \sigma}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (([];\langle N\rangle^{l})\cdot S, M) : (\bot, e.\mathsf{w}, e.\mathsf{s}) \curlyvee e', \sigma}$$

$$\textsc{Ref-E}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \theta \quad e.\mathsf{r} \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\langle \mathsf{ref}_{l,\theta}\rangle^{l'}[])\cdot S, M) : (l', l \curlywedge e.\mathsf{w}, e.\mathsf{s}), \theta\mathsf{ref}_{l}}$$

$$\textsc{DeRef-E}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (, S, M) : e, \theta\mathsf{ref}_{l}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (([!])\cdot S, M) : e \curlyvee (l, \top, \bot)}$$

$$\textsc{Assign-E1}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \theta\mathsf{ref}_{l} \quad l', e.\mathsf{s};\Gamma \vdash N : e', \theta \quad (e.\mathsf{r} \curlyvee e'.\mathsf{r}) \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (([] := \langle N\rangle^{l'})\cdot S, M) : (l', e.\mathsf{w} \curlywedge e'.\mathsf{w} \curlywedge l, e.\mathsf{s} \curlyvee e'.\mathsf{s}), \mathsf{unit}}$$

$$\textsc{Assign-E2}\ \frac{\bot;\bot;\Gamma \vdash u_{l,\theta} : \bot, \theta\mathsf{ref}_{l} \quad \mathsf{pc};\mathsf{cur};\Gamma \vdash N : e', \theta \quad (l_0 \curlyvee e'.\mathsf{r}) \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\langle u_{l,\theta}\rangle^{l_0} :=^{l_1} [])\cdot S, N) : (l', e'.\mathsf{w} \curlywedge l, e'.\mathsf{s}), \mathsf{unit}}$$

$$\textsc{Emit-E}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \mathsf{tsig}_{l} \quad e.\mathsf{r} \preceq l}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\langle \mathsf{emit}\rangle^{l'}[])\cdot S, M) : (l', e.\mathsf{w} \curlywedge l, e.\mathsf{s}), \mathsf{unit}}$$

$$\textsc{When-E1}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \mathsf{tsig}_{l} \quad l';e.\mathsf{s} \curlyvee e.\mathsf{c} \curlyvee l;\Gamma \vdash N : e', \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{when}\ []\ \mathsf{do}\ \langle N\rangle^{l'})\cdot S, M) : (e'.\mathsf{c}, e.\mathsf{w} \curlywedge e'.\mathsf{w}, e.\mathsf{s} \curlyvee e'.\mathsf{s}), \tau}$$

$$\textsc{When-E2}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{when}\ s_{l}\ \mathsf{do}\ [])\cdot S, M) : e, \tau}$$

$$\textsc{Watch-E1}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \mathsf{tsig}_{l} \quad l';e.\mathsf{s} \curlyvee e.\mathsf{c} \curlyvee l;\Gamma \vdash N : e', \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{watch}\ []\ \mathsf{do}\ \langle N\rangle^{l'})\cdot S, M) : (e'.\mathsf{c}, e.\mathsf{w} \curlywedge e'.\mathsf{w}, e.\mathsf{s} \curlyvee e'.\mathsf{s}), \tau}$$

$$\textsc{Watch-E2}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S_0, M) : e, \tau \quad l \curlyvee l' \preceq e.\mathsf{w}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{watch}\ \langle s_{l}\rangle^{l'}\ \mathsf{do}\ [])\cdot S_0, M) : e \curlyvee (\bot, \top, l \curlyvee l'), \tau}$$

$$\textsc{Watch-E3}\ \frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{watch}\ s_{l}\ \mathsf{do}\ [])\cdot S, M) : e, \tau}$$

Figure 4: Typing running thread

### 3.3 Type safety

To obtain the type safety results, we follow the stand approach: to prove a "subjection property" property, and further show that insecure program are not typable.

**Lemma 3.1 (Typing Values)** *For all $V \in \mathcal{V}$, if $\mathsf{pc};\mathsf{cur};\Gamma \vdash V : e, \tau$, then $e = (\mathsf{pc}, \top, \mathsf{cur})$.*

**Proof** Straightforward by typing rules Loc, Var, BoolT, BoolF, Sig, Nil, and Abs. ∎

**Lemma 3.2 (Strengthening Initial Effect)** *If $\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e, \tau$, and $l_c, l_s \preceq e.\mathsf{w}$, then $(\mathsf{pc} \curlyvee l_c);(\mathsf{cur} \curlyvee l_s);\Gamma \vdash M : e_0, \tau$, such that $e_0 = e \curlyvee (l_c, \top, l_s)$.*

**Proof** By induction on the height of inference tree. We examine the case of rule Cond here.

$$\frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash M : e, \mathsf{bool} \quad e.\mathsf{c};e.\mathsf{s};\Gamma \vdash N_i : e_i, \tau}{\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : e \curlyvee e_0 \curlyvee e_1, \tau}$$

By induction hypothesis, we have

$$\mathsf{pc} \curlyvee l_c;\mathsf{cur} \curlyvee l_s;\Gamma \vdash M : e \curlyvee (l_c, \top, l_s), \mathsf{bool}$$

and

$$e.\mathsf{c} \curlyvee l_c;e.\mathsf{s} \curlyvee l_s;\Gamma \vdash N_i : e_i \curlyvee (l_c, \top, l_s), \tau$$

We can conclude that

$$\frac{\begin{array}{c}\mathsf{pc} \curlyvee l_c;\mathsf{cur} \curlyvee l_s;\Gamma \vdash M : e \curlyvee (l_c, \top, l_s), \mathsf{bool} \\ e.\mathsf{c} \curlyvee l_c;e.\mathsf{s} \curlyvee l_s;\Gamma \vdash N_i : e_i \curlyvee (l_c, \top, l_s), \tau\end{array}}{\begin{array}{c}\mathsf{pc};\mathsf{cur};\Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : \\ e \curlyvee e_0 \curlyvee e_1 \curlyvee (l_c, \top, l_s), \tau\end{array}}$$

Other cases are similar. ∎

**Lemma 3.3 (Replacement Lemma)** *If $\mathsf{pc};\mathsf{cur};\Gamma \vdash (S \cdot S_0, M) : e, \tau$, then there exists $e_0$ and $\sigma$, such that $\mathsf{pc};\mathsf{cur};\Gamma \vdash (S_0, M) : e_0, \sigma$, and if $\mathsf{pc}';\mathsf{cur}';\Gamma \vdash (S_0', M') : e_1, \sigma$ with $e_1 \preceq e_0$, then $\mathsf{pc}';\mathsf{cur}';\Gamma \vdash (S \cdot S_0', M') : e', \tau$ for some $e'$ such that $e' \preceq e$.*

**Proof** By induction on the typing inference. ∎

**Lemma 3.4 (Substitution Lemma)** *If $\bot;\bot;\Gamma, x : \tau \vdash M : e, \sigma$, and $\bot;\bot;\Gamma \vdash V : \bot, \tau$, then $\bot;\bot;\Gamma \vdash \{x \mapsto V\}M : e, \tau$.*

**Proof** The lemma is quite standard, proof by induction on the structure of $M$ and the typing inference. ∎

**Proposition 3.5 (Subject Reduction)** *If $\Gamma \vdash [\mu, \xi, t, T]$, and $[\mu, \xi, t, T] \rightarrow [\mu', \xi', t', T']$, then there exist $\Gamma'$, such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash [\mu', \xi', t', T']$.*

**Proof** We need only exam the cases when the transition comes from a thread transition or a instant transition. For scheduling transition, it is trivial from the typing judgement of configuration.

Assuming that $t = (\mathsf{pc}, \mathsf{cur}, S, M)$, then by typing rules of configuration, we have that

$$\mathsf{pc};\mathsf{cur};\Gamma \vdash (S, M) : e, \theta$$

For the thread transitions, we proceed case by case on the thread transitions.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1))) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{if}\ []\ N_0 N_1), M))}$

  By Lemma 3.3, we have

$$\dfrac{\dfrac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash M : e, \mathsf{bool} \quad e.\mathsf{c}; e.\mathsf{s}; \Gamma \vdash N_i : e_i, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : e_2, \sigma}}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1)) : e_2, \sigma}$$

  and then we have

$$\dfrac{\dfrac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash M : e, \mathsf{bool}}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, M) : e, \mathsf{bool}} \quad e.\mathsf{c}; e.\mathsf{s}; \Gamma \vdash N_i : e_i, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\mathsf{if}\ []\ N_0 N_1), M) : e_2, \sigma}$$

  By Lemma 3.3 again we can conclude that

$$\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (S \cdot (\mathsf{if}\ []\ N_0 N_1), M) : e, \theta$$

  We omit all the other cases of "administrative" transitions which consist in pushing a frame into the stack.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{if}\ []\ N_0 N_1), b)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, N_i))}$

  where $b = tt$ and $i = 0$ or $b = ff$ and $i = 1$. By Lemma 3.1 and Lemma 3.3, we have

$$\dfrac{\dfrac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash b : (\mathsf{pc}, \top, \mathsf{cur}), \mathsf{bool}}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, b) : (\mathsf{pc}, \top, \mathsf{cur}), \mathsf{bool}} \quad \mathsf{pc}; \mathsf{cur}; \Gamma \vdash N_i : e_i, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\mathsf{if}\ []\ N_0 N_1), M) : e_2, \sigma}$$

  and then we have

$$\dfrac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash N_i : e_i, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, N_i) : e_i, \sigma}$$

  where $e_i \preceq e_2$, then by Lemma 3.3 we can conclude.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \lambda x M \rangle^l []), V)) \xrightarrow{()} (\mu, \xi, (\mathsf{pc} \curlyvee l, \mathsf{cur}, S, \{x \mapsto V\} M))}$

  By Lemma 3.1 and Lemma 3.3, we have

$$\dfrac{\dfrac{\bot; \bot; \Gamma, x : \tau \vdash M : e_1, \sigma}{\bot; \bot; \Gamma \vdash (\lambda x M) : \bot, (\tau \xrightarrow{e_1} \sigma)} \quad \mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, V) : (\mathsf{pc}, \top, \mathsf{cur}), \tau \quad (l \curlyvee \mathsf{pc} \curlyvee \mathsf{cur}) \preceq e_1.\mathsf{w}}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\langle \lambda x M \rangle^l []), V) : e_0, \sigma}$$

  where $e_0 = e_1 \curlyvee (l \curlyvee \mathsf{pc}, \top, \mathsf{cur})$. Then by Lemma 3.4, we have

$$\bot; \bot; \Gamma \vdash \{x \mapsto V\} M : e_1, \sigma$$

  then by Lemma 3.2, we could infer

$$\mathsf{pc} \curlyvee l; \mathsf{cur}; \Gamma \vdash \{x \mapsto V\} M : e_1', \sigma$$

  where $e_1' \preceq e_0$, then by Lemma 3.3 we can conclude.

- $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot ([]; \langle N \rangle^l), V)) \xrightarrow{()} (\mu, \xi, (l, \mathsf{cur}, S, N))$

By Lemma 3.3, we have

$$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, V) : (\mathsf{pc}, \top, \mathsf{cur}), \tau \quad l; \mathsf{cur}; \Gamma \vdash N : e_0, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (([]; \langle N \rangle^l), V) : e_0, \sigma}$$

which implies the following

$$\frac{l; \mathsf{cur}; \Gamma \vdash N : e_0, \sigma}{l; \mathsf{cur}; \Gamma \vdash (\varepsilon, N) : e_0, \sigma}$$

Then by Lemma 3.3 again we can conclude this case.

- $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{ref}_{l,\theta} \rangle^{l'} []), V)) \xrightarrow{()} (\mu \cup \{u_{l,\theta} \mapsto V\}, \xi, (l', \mathsf{cur}, S, u_{l,\theta}))$

By Lemma 3.3 and 3.1, we have

$$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, V) : (\mathsf{pc}, \top, \mathsf{cur}), \theta \quad \mathsf{pc} \curlyvee \mathsf{cur} \preceq l}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\langle \mathsf{ref}_{l,\theta} \rangle^{l'} []), V) : (l', l, \mathsf{cur}), \theta \mathsf{ref}_l}$$

Then we can show that

$$l'; \mathsf{cur}; \Gamma \vdash (\varepsilon, u_{l,\theta}) : (l', \top, \mathsf{cur}), \theta \mathsf{ref}_l$$

Then by Lemma 3.3 we can conclude. Furthermore, we have to show that $\Gamma \vdash \mu \cup \{u_{l,\theta} \mapsto V\}$. Since $u_{l,\theta}$ is fresh, it is trivial.

- $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (![]), u_{l,\theta})) \xrightarrow{()} (\mu, \xi, (\mathsf{pc} \curlyvee l, \mathsf{cur}, S, V))$

By Lemma 3.3, we have

$$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, u_{l,\theta}) : (\mathsf{pc}, \top, \mathsf{cur}), \theta \mathsf{ref}_l}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((![]) \cdot \varepsilon, u_{l,\theta}) : (\mathsf{pc}, \top, \mathsf{cur}) \curlyvee (l, \top, \bot), \theta}$$

Then we can show that

$$\mathsf{pc} \curlyvee l; \mathsf{cur}; \Gamma \vdash (\varepsilon, V) : (\mathsf{pc} \curlyvee l, \top, \mathsf{cur}), \theta$$

By Lemma 3.3 we can conclude.

- $(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle u_{l,\theta} \rangle^{l_0} :=^{l_1} []), V)) \xrightarrow{()} (\mu[u_{l,\theta} \mapsto V], \xi, (l_1, \mathsf{cur}, S, ()))$

By Lemma 3.3, we have

$$\frac{\bot; \bot; \Gamma \vdash u_{l,\theta} : \bot, \theta \mathsf{ref}_l \quad \mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, V) : (\mathsf{pc}, \top, \mathsf{cur}), \theta \quad (l_0 \curlyvee \mathsf{pc} \curlyvee \mathsf{cur}) \preceq l}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\langle u_{l,\theta} \rangle^{l_0} :=^{l_1} []), V) : (l_1, l, \mathsf{cur}), \mathsf{unit}}$$

Then we can show that

$$l_1; \mathsf{cur}; \Gamma \vdash (\varepsilon, ()) : (l_1, \top, \mathsf{cur}), \mathsf{unit}$$

By Lemma 3.3 we can conclude, and $\Gamma \vdash \mu[u_{l,\theta} \mapsto V]$, since $V$ is well-typed.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\langle \mathsf{emit} \rangle^{l_0} []), s_l)) \xrightarrow{()} (\mu, \xi \cup \{s_l\}, (l_0, \mathsf{cur}, S, ()))}$

  By Lemma 3.3, we have

  $$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, s_l) : (\mathsf{pc}, \top, \mathsf{cur}), \mathsf{tsig}_l \quad (\mathsf{pc} \curlyvee \mathsf{cur}) \preceq l}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\langle \mathsf{emit} \rangle^{l'} []), s_l) : (l', l, \mathsf{cur}), \mathsf{unit}}$$

  Then we can show that

  $$l'; \mathsf{cur}; \Gamma \vdash (\varepsilon, ()) : (l', \top, \mathsf{cur}), \mathsf{unit}$$

  By Lemma 3.3 again we can conclude.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, (\mathsf{thread}\ M))) \xrightarrow{t} (\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S, ()))}$

  By Lemma 3.3, we can have

  $$\frac{\dfrac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash M : e_0, \sigma}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\mathsf{thread}\ M) : e_0, \mathsf{unit}}}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, \mathsf{thread}\ M) : e_0, \mathsf{unit}}$$

  Since $t = (\mathsf{pc}, \mathsf{cur}, \varepsilon, M)$, then it is typable.

- $\boxed{(\mu, \xi, (\mathsf{pc}, \mathsf{cur}, S \cdot (\mathsf{when}\ []\ \mathsf{do}\ \langle N \rangle^l), s_{l'})) \xrightarrow{()} (\mu, \xi, (l, \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l', S \cdot (\mathsf{when}\ s_{l'}\ \mathsf{do}\ []), N))}$

  By Lemma 3.3, we have

  $$\frac{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (\varepsilon, s_{l'}) : (\mathsf{pc}, \top, \mathsf{cur}), \mathsf{tsig}_{l'} \quad l; \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l'; \Gamma \vdash N : e', \tau}{\mathsf{pc}; \mathsf{cur}; \Gamma \vdash ((\mathsf{when}\ []\ \mathsf{do}\ \langle N \rangle^l), s_{l'}) : e', \tau}$$

  we can show that

  $$\frac{l; \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l'; \Gamma \vdash (\varepsilon, N) : e', \tau}{l; \mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l'; \Gamma \vdash ((\mathsf{when}\ s_{l'}\ \mathsf{do}\ []), N) : e', \tau}$$

  By Lemma 3.3 again, we can conclude.

For the scheduling, since the transition will not actually change any thread, the it will not affect the typability of the configuration.

It remains to show the case for instant transition. In this case, it is suffice to show that the transformation $\kappa^\xi$ preserves the typability of running threads under same (or weaker) security effect $e$. That is, if we have

$$\mathsf{pc}; \mathsf{cur}; \Gamma \vdash (S, M) : e, \tau$$

and

$$\kappa^\xi(\mathsf{pc}, \mathsf{cur}, S, M) = (\mathsf{pc}', \mathsf{cur}', S', M')$$

Then we show that, for some $e' \preceq e$

$$\mathsf{pc}'; \mathsf{cur}'; \Gamma \vdash (S', M') : e', \tau$$

We prove by induction on the length of the stack $S$ in $(S, M)$.

- $S = \varepsilon$. It is trivial since $\kappa^\xi(\mathsf{pc}, \mathsf{cur}, \varepsilon, M) = (\mathsf{pc}, \mathsf{cur}, \varepsilon, M)$.
- $S = (\mathsf{when}\ s_l\ \mathsf{do}\ []) \cdot S_0$. If $s_l \notin \xi$, then it is trivial similar to above case. If $s_l \in \xi$. By induction hypothesis and typing rule WHEN-E2, we can conclude.

- $S = (\mathsf{watch}\ \langle s_l\rangle^{l'}\ \mathsf{do}\ [])\cdot S_0, M)$ we have

$$\frac{\mathsf{pc};\mathsf{cur};\Gamma \vdash (S_0, M) : e, \tau \quad l \curlyvee l' \preceq e.\mathsf{w}}{\mathsf{pc};\mathsf{cur};\Gamma \vdash ((\mathsf{watch}\ \langle s_l\rangle^{l'}\ \mathsf{do}\ [])\cdot S_0, M) : e \curlyvee (\bot, \top, l \curlyvee l'), \tau}$$

If $s_l \in \xi$, then we have

$$\mathsf{pc};\mathsf{cur} \curlyvee l \curlyvee l'\Gamma \vdash (\varepsilon, ()) : (\mathsf{pc}, \top, \mathsf{cur} \curlyvee l \curlyvee l'), \mathsf{unit}$$

where $(\mathsf{pc}, \top, \mathsf{cur} \curlyvee l \curlyvee l') \preceq e \curlyvee (\bot, \top, l \curlyvee l')$.
If $s_l \notin \xi$, then by induction hypothesis, we have

$$\mathsf{pc}';\mathsf{cur}';\Gamma \vdash (S_0', M') : e', \tau$$

where $e' \preceq e$, which implies $l \curlyvee l' \preceq e'.\mathsf{w}$. By Lemma 3.2, we have

$$\frac{\mathsf{pc}';\mathsf{cur}' \curlyvee l \curlyvee l';\Gamma \vdash (S_0', M') : e' \curlyvee (\bot, \top, l \curlyvee l'), \tau}{\mathsf{pc}';\mathsf{cur}' \curlyvee l \curlyvee l';\Gamma \vdash ((\mathsf{watch}\ s_l\ \mathsf{do}\ [])\cdot S_0', M') : e' \curlyvee (\bot, \top, l \curlyvee l'), \tau}$$

- Other cases are simple. ∎

**Lemma 3.6 (insecure program)** *The following insecure programs (threads) are not typable*

- $(\mathsf{pc}, \mathsf{cur}, S\cdot(\langle \mathsf{ref}_{l,\theta}\rangle^{l'}), V)$, *where* $(\mathsf{pc} \curlyvee \mathsf{cur}) \not\preceq l$
- $(\mathsf{pc}, \mathsf{cur}, S\cdot(\langle u_{l,\theta}\rangle^{l_0} :=^{l_1} []), V)$, *where* $(\mathsf{pc} \curlyvee \mathsf{cur} \curlyvee l_0) \not\preceq l$
- $(\mathsf{pc}, \mathsf{cur}, S\cdot(\langle \mathsf{emit}\rangle^{l_0}), s_l)$, *where* $(\mathsf{pc} \curlyvee \mathsf{cur}) \not\preceq l$

**Proof** Straightforward by type rules Ref-E, Assign-E2, and Emit-E. ∎

Finally, we could state our safety result.

**Theorem 3.7 (Type Safety)** *If $M$ is typable in typing context $\Gamma$ for initial effect $(\bot, \bot)$, that is, $\bot, \bot, \Gamma \vdash M : e, \tau$ for some $e$ and $\tau$. Then $M$ is secure with the class of typable memory $\{\mu | \tau \vdash \mu\}$.*

**Proof** Direct consequence of Proposition 3.5 and Lemma 3.6. ∎

# 4 Conclusion

In this paper we investigate the information flow problem in a reactive programming setting, which augments the multi-threaded functional language with reactive constructs to achieve cooperative synchronization between threads. We point out new covert channel in this programming paradigm. Furthermore, we define the security property as a safety property by means of dynamically checking information flow during evaluation. Finally, we show a type system to ensure that typable programs do not need these dynamical checks during the computing.

# References


[1] G. Boudol and M. Kolundzija. Access Control and Declassification. In *Computer Network Security*, volume 1 of *CCIS*, pages 85–98. Springer-Verlag, 2007.

[2] Gérard Boudol. ULM: A core programming model for global computing: (extended abstract). In *ESOP*, pages 234–248, 2004.

[3] Gérard Boudol. On typing information flow. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2005.

[4] Gérard Boudol. Secure information flow as a safety property. submitted, 2008.

[5] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *CSFW*, pages 226–240. IEEE Computer Society, 2005.