TYPE SYSTEMS FOR FORMAL VERIFICATION OF CONCURRENT PROGRAMS

by

Kohei Suenaga

A Dissertation

Submitted to
the Graduate School of the University of Tokyo
in December 2007
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Akinori Yonezawa
Professor of Computer Science

ABSTRACT

Writing reliable concurrent programs is said to be more difficult than sequential programs because of the possibility of deadlocks and races. Moreover, the state-explosion problem caused by non-deterministic interleaving of threads makes it difficult to track program states.

One of the promising approaches for high software reliability is formal verification, statically verifying software before executing it. However, though static verification methods have been extensively studied for sequential programs so far, verification for concurrent programs is still immature, in that many important software features such as interrupts and dynamic creation of communication channels, which often appear in practical software, have not been investigated. Because the debugging-by-testing approach for concurrent programs is less useful than for sequential programs, static verification methods that can deal with concurrent programs with such primitives are of significant importance.

As a step to verification methods that can deal with practical programs, we propose type-based verification of two important security properties of concurrent programs: deadlock-freedom and resource usage. The presented deadlock-freedom verification is equipped with three features: (1) non-block-structured mutex primitives (2) mutable references to mutexes and (3) interrupts. Those three features, which are heavily used in real-world programs, have not been dealt with in the deadlock-freedom analyses proposed so far.

In order to design a deadlock-freedom verification for such programs, we define a concurrent calculus with those three features and a type system for the calculus. Our type system guarantees deadlock-freedom (1) by verifying that there are not circular dependencies among locking/unlocking operations and (2) by guaranteeing that an acquired lock is released exactly once. For guaranteeing the first condition, the type system assigns a natural number called *lock level* to each lock type and verifies that locks are acquired in a strict increasing order of those numbers. For the verification of the second condition in the presence of aliasing and references to a mutex, we use a kind of linear types called *capabilities/obligations*, and *ownerships*, a technique to control accesses to a reference to a mutex. We also report the result of a verification experiment of an implementation of a network protocol stack.

We next present a type-based resource usage analysis for the π-calculus. The goal of the resource usage analysis is to statically check that various resources (e.g., files, memory and sockets) are accessed according to an access protocol associated to each resource. Though resource usage analysis for sequential programs is extensively studied so far, it is not well investigated for concurrent programs, especially those involving dynamic creation/passing of communication channels and resources, which often appears in practical software.

To this end, we design a type system for a π-calculus extended with resource creation/access primitives, as an extension of the behavioral type system by Igarashi and Kobayashi. The type system guarantees the safety property that no invalid access is performed, as well as the property that necessary accesses (such as the close operation for a file) are eventually performed unless the program diverges. A sound type inference algorithm for the type system is also developed to free the programmer from the burden of writing complex type annotations. Based on the algorithm, we have implemented a prototype resource usage analyzer for the π-calculus. To the authors' knowledge, ours is the first type-based

resource usage analysis that deals with an expressive concurrent language like the π-calculus.

Contents

1 Introduction **4**

1.1 Background 4

1.2 Overview of this thesis 5

1.2.1 Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts 5

1.2.2 Resource Usage Analysis for the π-Calculus 8

1.3 Contribution 10

1.4 Outline 11

2 Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts 12

2.1 Target Language 12

2.1.1 Syntax 12

2.1.2 Operational Semantics 15

2.2 Type System 16

2.2.1 Lock Levels 16

2.2.2 Effects 16

2.2.3 Syntax of Types 18

2.2.4 Type Judgment 19

2.2.5 Type Soundness 21

2.2.6 Type Inference 24

2.3 Extension 24

2.3.1 Target Language 24

2.3.2 Type System 25

2.4 Experiment 37

2.4.1 Implementation 37

2.4.2 Deadlock-freedom verification of a device driver 38

2.4.3 Discussion 42

3 Resource Usage Analysis for the π-Calculus **46**

3.1 Processes 46

3.1.1 Syntax . . . 46
3.1.2 Operational Semantics . . . 47
3.2 Type System . . . 50
3.2.1 Types . . . 51
3.2.2 Semantics of behavioral types . . . 52
3.2.3 Typing . . . 55
3.3 Type Inference Algorithm . . . 59
3.3.1 Step 1: Extracting Constraints . . . 59
3.3.2 Step 2: Reducing Constraints . . . 62
3.3.3 Step 3: Constraint Solving . . . 62
3.3.4 Step 3-1: Construction of $N_{A,x}$ . . . 64
3.3.5 Steps 3-2 and 3-3: Construction of $N_{A,x} \,||\, M_\Phi$ and reduction of $\mathbf{traces}_x(A)$ to a reachability problem . . . 66
3.4 Extensions . . . 67
3.4.1 A Type System for the Partial Liveness Property . . . 68
3.4.2 Type Inference . . . 71
3.5 Implementation . . . 72
3.6 Discussion . . . 73

4 Related work **76**
4.1 Deadlock-freedom verification . . . 76
4.2 Calculi with interrupts . . . 77
4.3 Resource usage analysis . . . 77
4.4 Static analysis for practical software . . . 80

5 Conclusion **82**

References **84**

A Proofs of Lemma 2.2 and 2.3 **91**

B Proof of Lemma 2.4 **95**

C Properties of the Subtyping Relation **97**

D Proof of the Subject Reduction Property **106**

E Proofs of the Lemma for Theorem 3.3 **111**

F Computing a Basis of Behavioral Type **113**

Acknowledgement

I would like to express my best gratitude to Naoki Kobayashi and Akinori Yonezawa for their supporting me enormously during these five years. They have been great mentors. I have learned much about how to conduct research from them. They have also supported me mentally when I was in difficulty. The contents of this thesis have been developed through many discussions with Naoki Kobayashi.

I am also grateful to Lucian Wischik, Eijiro Sumii and Tachio Terauchi for the discussions of the technical issues of this research. The contents of Chapter 3 is a joint work with Naoki Kobayashi and Lucian Wischik.

I also thank Hiroya Matsuba for comments and advice from the viewpoint of a system software researcher, and for being a great friend since I was an undergraduate student. He let me notice the importance of dealing with interrupts in software verification.

I thank my family for their kind support since I was a child. They have always assisted me with my life choices.

This research is supported by JSPS Research Fellowships for Young Scientists.

Chapter 1

Introduction

1.1 Background

Concurrent programs, in which several threads run in an interleaving manner, are getting important as multi-processor machines and clusters get popular. Nowadays, many programs including operating systems and various servers are written as concurrent programs.

However, writing reliable concurrent programs is said to be more difficult than sequential programs because of the possibility of deadlocks and races. Moreover, the state-explosion problem caused by non-deterministic interleaving of threads makes it difficult to track program states.

In order to achieve high reliability of software, one often debugs software by testing. However, debugging by testing is insufficient for that purpose due to the possibility of missing a path that leads to an incorrect behavior. It is especially hard to test every path of concurrent programs due to non-determinism. What is worse, even if one finds an incorrect behavior of a concurrent program, one may not be able to reproduce that behavior, again due to the non-determinism, so that it may be difficult to determine the cause of the behavior.

One of the promising approaches for high software reliability is formal verification, statically verifying the software before executing it. In this approach, one provides a verifier of a certain security property (e.g., deadlock-freedom.) The verification algorithm should be mathematically proved to be sound. Before executing a program, the verifier checks the program. If the verification succeeds, then the program is guaranteed to have the security property in execution time.

Formal verification for sequential programs are well investigated so far, including type-based approaches [25, 30, 38, 45, 46, 50, 56, 57, 62, 64, 65] and abstract-interpretation-based approaches [4, 10, 42]. However, though much effort also has been paid for verification of concurrent programs, many important software features such as interrupts and runtime creation of communication channels, which often appear in real-world software, have not been well in-

vestigated. Because the debugging-by-testing approach for concurrent programs is less useful than for sequential programs, static verification methods for concurrent programs with such primitives are of significant importance.

1.2 Overview of this thesis

In this thesis, as a step to verification methods that can deal with real-world programs, we study verification of two important security properties of concurrent programs: deadlock-freedom and resource usage, and propose type-based verification methods. The presented analyses deal with features that are not dealt with in the existing verification methods. In the following, we present the overview of those two verification methods.

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

A deadlock, a state in which several threads are waiting for a lock acquired by other threads, is one of the well-known bugs that may occur in concurrent programs. Because a deadlock leads to an unintentional system halt, it is considered to be a serious problem, and much work [1, 5, 14, 15, 32, 34, 35, 54] has been done to detect deadlocks.

We present a new type-based deadlock-freedom verification. The verification method is equipped with the following three features which have not been dealt with by the existing work.

- **Feature A**: Non-block-structured mutex primitives.
- **Feature B**: Mutable references to mutexes.
- **Feature C**: Interrupts.

Those three features are heavily used in the real-world software. Non-block-structured mutex primitives, whose locking operations do not syntactically correspond to unlocking operations, are used in, for example, C programs with POSIX thread library. Mutable references to mutexes are used frequently. Interrupts, asynchronous jump to an interrupt handler, are essential in system software and appear in user applications as signal handling.

For example, consider the program in Figure 1.1. This example, based on a bug that was found in an initial version of a protocol stack implementation used in an ongoing research project on cluster computing [41], shows a typical deadlock that often occurs in a program with interrupts. 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 receives a reference to a lock which is associated with a network device, and acquires it before the function flushes the buffer. That function

```
let flush_buffer devlock =
  let data = dequeue () in
  while !data != NULL do
    (* Interrupts should be forbidden before this locking 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
  ...
```

Figure 1.1: An example of a program which may cause deadlock.

`receive_data` works as an interrupt handler and is asynchronously called when a packet arrives. Note that `receive_data` calls `flush_buffer` in order for the local buffer to be flushed as soon as the function knows there is room in the remote buffer (a similar mechanism called *congestion control* is used in TCP), so that the following control flow causes a deadlock:

$$\begin{aligned} \text{Call to } \texttt{flush\_buffer} &\rightarrow \texttt{lock(!devlock)} \\ &\rightarrow \text{an interrupt (call to } \texttt{receive\_data}) \\ &\rightarrow \text{call to } \texttt{flush\_buffer} \rightarrow \texttt{lock(!devlock)} \end{aligned}$$

To prevent the deadlock, `flush_buffer` has to disable interrupts before it acquires the device lock.

Though **Feature A**–**C** are heavily used in real-world software, the researches on deadlock-freedom verification so far [1, 5, 14, 15, 32, 34, 35, 54] have not dealt with all of those three features. Kobayashi, Saito and Sumii [32, 34, 36] studied deadlock-freedom verification for π-calculus. However, their framework lacks **Feature B** and **Feature C**. Flanagan, Abadi and Freund [1, 14, 15] and Boyapati, Lee and Rinard [5] studied type-based deadlock- and race-freedom verification for Java. However, their frameworks lack **Feature A** and **Feature C**. Permandla, Roberson and Boyapati [54] studied deadlock- and race-freedom verification for Java byte codes.

```
let f plock1 plock2 =
  unlock(!plock1); unlock(!plock2)

let main () =
  let x = newlock () in
    lock(x);
    let r1 = ref x in
    let r2 = ref x in
      f(r1, r2)
```

Figure 1.2: An example in which deadlock is caused by an aliasing.

However, their framework does not deal with **Feature C**. Importantly, **Feature C** is not dealt with by the existing researches so far.

In order to design a verification framework for programs with **Feature A**–**C**, we define a concurrent calculus which is equipped with those three features and a type system for the calculus. Essentially, our type system guarantees that there are no circular dependencies among locking/unlocking operations by assigning a natural number called *lock level* to each lock type, and by verifying that locks are acquired in a strict increasing order of the lock levels. For example, the type system rejects the program in Figure 1.1 because the lock `!devlock` may be acquired twice sequentially due to an interrupt, so that assigning a lock level to the type of `devlock` is impossible.

The actual type system is more complex than expected due to the following three difficulties.

- Possibility that an acquired lock may not be released or may be released more than once. The type system has to guarantee that `!devlock` acquired in `flush_buffer` is actually released exactly once for the program in Figure 1.1.

- Aliasing. For example, consider the program in Figure 1.2. The function `f` releases locks through the passed two references `plock1` and `plock2`. In the function `main`, two references `r1` and `r2` to one acquired lock `x` are passed to `f`. Thus, the program should be rejected because the lock `x` is released twice.

- Possibility of race conditions to a reference to a lock. For the program in Figure 1.1, suppose that there is a concurrently running thread which may overwrite the reference `devlock` while the lock is acquired. Then, the lock released by `unlock(!devlock)` may be different from one acquired by `lock(!devlock)`, so that the acquired lock may not be released.

In order to solve the first and the second difficulty, we use the idea of *obligations* and *capabilities* [32, 34, 35]. The type system assigns an *obligation* to release a lock to its type when the lock is acquired. The type system guarantees that the obligation is fulfilled by an unlocking operation exactly once. Additionally, the type system guarantees that if a lock has to be released, then only one variable bound to or one reference to the lock has the obligation to release the lock (i.e., obligations are linearly [62] manipulated.) Based on this principle, the program in Figure 1.2 is rejected because only one of `x`, `r1` or `r2` can have the obligation to release the lock, which is not possible due to the requirement on the arguments on `f` that both `plock1` and `plock2` must have an obligation.

The third difficulty is solved using the idea of *ownerships*. The type system assigns ownership information to each reference type and guarantees that (1) each thread has an appropriate ownership for read/write access and (2) if a thread has a write ownership of a reference, then no other threads access the reference. Our type system uses rational-numbered ownership information [6, 27, 61] to express those conditions concisely.

Based on the type system, we have implemented a prototype deadlock-freedom verifier for C programs. We show the result of a preliminary experiment in Section 2.4.

1.2.2 Resource Usage Analysis for the π-Calculus

Computer programs access many external resources, such as files, memory, external devices, etc. Such resources are often associated with certain access protocols; for example, an allocated memory cell should be eventually deallocated exactly once, and after the cell has been deallocated, no read/write access is allowed. Violating that protocol may cause problems such as double free, dangling pointers or memory leak. Similarly, a file should be opened, accessed several times and then should be closed. The file should not be accessed after the close operation.

A problem of statically checking that a program respects such access protocols is called *resource usage analysis* [24], and has been extensively studied. DeLine and Fähndrich [11, 12], Foster, Terauchi and Aiken [17] and Igarashi and Kobayashi [24] proposed type-based analyses. Ball and Rajamani [3] designed an analysis based on model checking technique. However, most of them focused on analysis of sequential programs, and did not treat concurrent programs, especially those involving dynamic creation/passing of communication channels and resources, which often appear in real programs.

For such problem, we propose a type-based method of resource usage analysis for *concurrent languages*. We use the π-calculus (extended with resource primitives) as a target language so that our analysis can be applied to a wide range of concurrency primitives (including those for dynamically creating and passing channels) in a uniform manner.

A main new difficulty in dealing with concurrent programs is that control structures are

more complex in concurrent programs than in sequential programs. For example, consider the following process P_1:

$$(\nu c)\,(\mathbf{read}(x).\overline{c}\langle\,\rangle \mid c(\,).\,\mathbf{close}(x))$$

In that program, after creating a communication channel c with (νc), two threads run concurrently. The first thread reads a resource x and sends a signal on channel c with $\overline{c}\langle\rangle$. The second thread waits for a signal on c with $c()$. and closes the resource x. Because of the synchronization through channel c, x is closed only after being read. To capture this kind of causal dependency between communications and resource access, we use CCS processes as extra type information (which are called behavioral types). For example, the above process is given the behavioral type $(\nu c)\,(x^R.\overline{c} \mid c.\,x^C)$, which actually captures the fact that those two threads synchronize on the channel c, so that the close access on x occurs after the read access on x.

Using the behavioral types introduced above, we can construct a type system for resource usage analysis in a manner similar to previous behavioral type systems for the π-calculus [7, 23]. The type judgment of our type system is of the form $\Gamma \triangleright P : A$, where Γ is the usual type environment and A is a behavioral type approximating the behavior of P on the free channels and resources. For example, the above process P_1 is typed $x : \mathbf{res} \triangleright P_1 : (\nu c)\,(x^R.\overline{c} \mid c.\,x^C)$. Behavioral types are also used to augment channel types. The judgment for $s(x).\,P_1$ is given by:

$$\Gamma \quad \triangleright \quad s(x).\,P_1 \quad : \quad s$$

where $\Gamma = s{:}\mathbf{chan}\langle(x{:}\mathbf{res})(\nu c)\,(x^R.\overline{c} \mid c.\,x^C)\rangle$. Here, the behavioral type of $s(x).\,P_1$ is simply a single input command s. Note that the behavior of the input continuation, the behavior of P_1, is accounted for at the behavior of a process that performs output on the channel s, not at input. The channel s has *argument type* $(x{:}\mathbf{res})(\nu c)\,(x^R.\overline{c} \mid c.\,x^C)$, which specifies that the resource sent along channel s will be read first and then closed. Using the same type environment, the output process $\overline{s}\langle r\rangle$ is typed as:

$$\Gamma, r{:}\mathbf{res} \quad \triangleright \quad \overline{s}\langle r\rangle \quad : \quad \overline{s}.\,(\nu c)\,(r^R.\overline{c} \mid c.\,r^C)$$

Here the behavioral type is an output followed by a continuation. The continuation $(\nu c)\,(r^R.\overline{c} \mid c.\,r^C)$ has been obtained by substituting r for x in the argument type of s. In this way, the types propagate information about how resources and channels passed thorough channels are accessed.

An important property of our type system is that types express abstract behavior of processes, so that certain properties of processes can be verified by verifying the corresponding properties of their types, using, for example, model checking techniques. The latter properties (of behavioral types) are more amenable to automatic verification techniques like model checking than the former ones, because the types do not have channel mobility and also because the types typically represent only the behavior of a part of the entire process.

1.3 Contribution

The main contribution of this thesis is a theoretical basis of deadlock-freedom verification and resource usage verification for concurrent programs with primitives that often appear in practical software, yet have not been dealt with by existing researches. The deadlock-freedom analysis deals with non-block-structured mutex primitives, mutable references to mutexes and interrupts. A combination of the ideas of lock levels, obligations/capabilities and ownerships enables the verification. The resource usage analysis deals with dynamic creation of channels and resources. This becomes possible by extending the behavioral type system for the π-calculus by Igarashi and Kobayashi [23] with hiding and renaming constructors, and adapting them to the problem of resource usage analysis.

Technical contribution of this thesis can be summarized as follows.

- Definition of a concurrent calculus with interrupts. Though Palsberg and Ma [52] proposed a calculus with interrupts, that calculus does not deal with concurrency. As far as we know, ours is the first formal calculus which is equipped with both concurrency and interrupts.

- An implementation of a prototype deadlock-freedom verifier based on the proposed type system. Using that verifier, we have conducted a preliminary experiment on an implementation of a protocol stack, and confirmed that a known bug is actually detected.

- Realization of fully automatic resource usage verification for concurrent programs (while making the analysis more precise than [23]). Igarashi and Kobayashi [23] gave only an abstract type system, without giving a concrete type inference algorithm. Chaki et al. [7] requires type annotations. The full automation was enabled by a combination of a number of small ideas, like inclusion of hiding and renaming as type constructors, and approximation of a CCS-like type by a Petri net (to reduce the problem of checking conformance of inferred types to resource usage specification).

- Resource usage verification of not only the usual safety property that an invalid resource access does not occur, but also an extended safety (which we call *partial liveness*) that necessary resource accesses (e.g. closing of a file) are eventually performed unless the whole process diverges. The partial liveness is not guaranteed by Chaki et al.'s type system [7]. A noteworthy point about our type system for guaranteeing the partial liveness is that it is parameterized by a mechanism that guarantees deadlock-freedom (in the sense of Kobayashi's definition [31]). So, our type system can be combined with *any* mechanism (model checking, abstract interpretation, another type system, or whatever) to verify deadlock-freedom for deadlock- or lock-freedom (e.g., Yoshida's graph type system [68]).

- Implementation of a prototype resource usage analyzer based on the proposed method. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

1.4 Outline

The structure of this thesis is as follows. In Chapter 2, we present a type-based deadlock-freedom verification for a concurrent calculus with interrupts. We first present a concurrent calculus with interrupts and *block*-structured synchronization primitives in Section 2.1.1. The type system for the calculus, presented in Section 2.2, is more simple than one we have explained in Section 1.2.1. We clarify how interrupts are dealt with using that simple setting. We then extend the framework with *non*-block-structured synchronization primitives in Section 2.3. Section 2.4 reports the result of a preliminary experiment.

In Chapter 3, we present a resource usage verification method for the π-calculus. Section 3.1 introduces an extension of the π-calculus with primitives for creating and accessing resources. Section 3.2 introduces a type system for resource usage analysis, which guarantees that well-typed processes never perform an invalid resource access. Section 3.3 gives a type inference algorithm for the type system. Section 3.4 extends the type system to guarantee that necessary resource accesses (such as closing of opened files) are eventually performed (unless the program diverges). Section 3.5 describes a prototype resource usage analyzer we have implemented based on the present work.

After discussing related work in Chapter 4, we conclude the thesis in Chapter 5.

Chapter 2

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

We present a type-based deadlock-freedom verification method for a concurrent calculus with interrupts in this chapter. Section 2.1 and 2.2 show a verification for a language with **Feature B** and **Feature C** presented in Section 1.2.1. Section 2.3 extends the framework with **Feature A**, and presents the result of an experiment. The contents of Section 2.1 and 2.2 have been presented in the 16th European Symposium on Programming [60].

2.1 Target Language

2.1.1 Syntax

The syntax of our target language is defined in Figure 2.1. 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.
- $M_1 \mid M_2$ is concurrent evaluation of M_1 and M_2. Both of M_1 and M_2 should evaluate to $()$.

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

Figure 2.1: The Syntax of Our Language.

- **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.
- $M_1 \rhd M_2$ 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, $M_1 \rhd M_2$ 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.

```
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)
```

Figure 2.2: An Encoding of the Program in Figure 1.1

- **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 2.2 shows how the example in Figure 1.1 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 2.1–2.4 below.

Example 2.1 *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.*

$$do\_something(\ldots) \triangleright interrupt\_low(\ldots) \triangleright interrupt\_high(\ldots)$$

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

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

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

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

interrupt_high can interrupt also from the initial state.

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

From the state above, interrupt_low cannot interrupt until interrupt_high(...) evaluates to a value.

Example 2.2 *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.*

$$(thread1(\ldots) \rhd handler1(\ldots)) \mid (thread2(\ldots) \rhd handler2(\ldots)) \ldots$$

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 2.3 *In the example in Figure 2.2, 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.*

$$\begin{aligned} &receive\_data(packettype, data, devlock) = \\ &\quad (\textbf{if } packettype = Room \textbf{ then } flush\_buffer(devlock) \textbf{ else } ()) \rhd \\ &\qquad\qquad\qquad\qquad receive\_data(Room, data, devlock) \end{aligned}$$

Example 2.4 *Since many real-world programs are written in C, we make design decisions of our language based on that of C. For example, function names are first-class values in our language because C allows one to use a function name as a function pointer. With this feature, we can express a runtime change of interrupt handler as follows:*

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

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.

2.1.2 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** [1].

Figure 2.3 shows the operational semantics of our language. We explain several important rules.

[1] 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 Chapter 2.2 can be used with only small changes.

- 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 2.2 leads to a deadlocked state. We write L_u for $\{\mathit{devlock}' \mapsto \mathbf{released}\}$, L_l for $\{\mathit{devlock}' \mapsto \mathbf{acquired}\}$, and M' for $\mathit{flush\_buffer\_iter}(\mathit{devlock}', \mathit{dequeue}())$. We omit $\widetilde{D}, H$ and I components of configurations.

$$
\begin{aligned}
& (L_u, \mathit{flush\_buffer}(\mathit{devlock}') \rhd \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock}')) \\
\to^* \ & (L_u, \mathbf{sync}\ \mathit{devlock}'\ \mathbf{in}\ (); M' \rhd \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock})) \\
\to \ & (L_l, \mathbf{in\_sync}\ \mathit{devlock}'\ \mathbf{in}\ (); M' \rhd \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock})) \\
\to \ & (L_l, \mathbf{in\_sync}\ \mathit{devlock}'\ \mathbf{in}\ (); M' \blacktriangleleft_{\mathit{receive\_data}(\ldots)} \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock})) \\
\to^* \ & (L_l, \mathbf{in\_sync}\ \mathit{devlock}'\ \mathbf{in}\ (); M' \blacktriangleleft_{\mathit{receive\_data}(\ldots)} \mathit{flush\_buffer}(\mathit{devlock}')) \\
\to^* \ & (L_l, \mathbf{in\_sync}\ \mathit{devlock}'\ \mathbf{in}\ (); M' \blacktriangleleft_{\mathit{receive\_data}(\ldots)} \mathbf{sync}\ \mathit{devlock}'\ \mathbf{in}\ ())
\end{aligned}
$$

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

2.2 Type System

2.2.1 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 $\forall \mathit{lev} \in \{-\infty, \infty\} \cup \mathbb{N}. -\infty \leq \mathit{lev} \leq \infty$. We write $\mathit{lev}_1 < \mathit{lev}_2$ for $\mathit{lev}_1 \leq \mathit{lev}_2 \wedge \mathit{lev}_1 \neq \mathit{lev}_2$.

2.2.2 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.3: The Operational Semantics of Our Language.

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

Figure 2.4: Syntax of types.

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.

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 2.1 (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 2.2 (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 $\sqcup$.

2.2.3 Syntax of Types

Figure 2.4 shows the syntax of types and effects. A type, represented by a meta-variable 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.

2.2.4 Type Judgment

The type judgment of our type system is $\Gamma \triangleright M : \tau \ \& \ \varphi$ 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,

$$x : \mathbf{lock}(0), y : \mathbf{lock}(1) \triangleright \mathbf{sync}\ x\ \mathbf{in}\ \mathbf{sync}\ y\ \mathbf{in}\ () : \mathbf{unit}\ \&\ (0, 1)$$

and

$$x : \mathbf{lock}(0), y : \mathbf{lock}(1) \triangleright \mathbf{sync}\ x\ \mathbf{in}\ (\mathbf{disable\_int}\ \mathbf{sync}\ y\ \mathbf{in}\ ()) : \mathbf{unit}\ \&\ (0, 0)$$

hold.

Definition 2.3 *The relation $\Gamma \triangleright M : \tau \ \& \ \varphi$ is the smallest relation closed under the rules in Figures 2.5–2.7. The predicate noIntermediate(M) in Figure 2.7 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 2.2 is rejected in our type system. From the derivation tree in Figure 2.8, *flush_buffer_iter* has a type $(\mathbf{lock}(1), \tau_d\ \mathbf{ref}) \stackrel{(1,1)}{\rightarrow} \mathbf{unit}$, where τ_d is the type of the contents of the reference *data*. Thus, *flush_buffer* has a type $\mathbf{lock}(1) \stackrel{(1,1)}{\rightarrow} \mathbf{unit}$ and *receive_data* has a type $(\tau_p, \tau_d\ \mathbf{ref}, \mathbf{lock}(1)) \stackrel{(1,1)}{\rightarrow} \mathbf{unit}$, where τ_p is the type of *packettype*.

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

$$\Gamma \triangleright () : \mathbf{unit} \ \& \ \emptyset \qquad \text{(T-Unit)}$$

$$\Gamma \triangleright n : \mathbf{int} \ \& \ \emptyset \qquad \text{(T-Int)}$$

$$\Gamma \triangleright \mathbf{true} : \mathbf{bool} \ \& \ \emptyset \qquad \text{(T-True)}$$

$$\Gamma \triangleright \mathbf{false} : \mathbf{bool} \ \& \ \emptyset \qquad \text{(T-False)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \triangleright x : \tau \ \& \ \emptyset} \qquad \text{(T-Var)}$$

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

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

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

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

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

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

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

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

Figure 2.5: Typing rules

- $\Gamma \triangleright \mathit{flush\_buffer}(\mathit{devlock}) : \mathbf{unit} \ \& \ (1, 1)$ and
- $\Gamma \triangleright \mathit{receive\_data}(\mathit{Room}, \mathit{data}, \mathit{devlock}) : \mathbf{unit} \ \& \ (1, 1)$.

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

$$\mathbf{sync} \ \mathit{devlock} \ \mathbf{in} \ ()$$

in the body of *flush_buffer_iter* is replaced by

$$\mathbf{disable\_int} \ \mathbf{sync} \ \mathit{devlock} \ \mathbf{in} \ ().$$

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

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

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

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

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

$$\frac{\Gamma \triangleright M_1 : \tau \ \&\ (lev_1, lev_2) \quad \Gamma \triangleright M_2 : \mathbf{unit} \ \&\ (lev_1', lev_2') \quad \Gamma \triangleright M : \mathbf{unit} \ \&\ (lev_1'', lev_2'') \quad lev_2 < lev_1' \quad lev_2 < lev_1'' \quad \varphi' = \varphi \sqcup (lev_1, lev_2) \sqcup (lev_1', lev_2')}{\Gamma \triangleright M_1 \blacktriangleleft_M M_2 : \tau \ \&\ \varphi'} \text{(T-InInterrupt)}$$

Figure 2.6: Typing rules

Then, *flush_buffer_iter* has a type $(\mathbf{lock}(1), \tau_d\ \mathbf{ref}) \overset{(1,-\infty)}{\to} \mathbf{unit}$ Thus, because

$$\Gamma \triangleright \mathit{flush\_buffer}(\mathit{devlock}) : \mathbf{unit} \ \&\ (1, -\infty)$$

and $-\infty < 1$ hold, the program is well-typed.

2.2.5 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 $\mathit{deadlocked}(L, M)$ defined below means that M is in a deadlocked state under L.

Definition 2.4 (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 $i = \mathbf{sync}\ x\ \mathbf{in}\ M' \wedge L(x) = \mathbf{acquired}$.*

$$\frac{\Gamma \supseteq f_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \xrightarrow{\varphi_1} \tau_1, \ldots, f_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \xrightarrow{\varphi_n} \tau_n \quad \Gamma, x_{i,1} : \tau_{i,1}, \ldots, x_{i,m_i} : \tau_{i,m_i} \triangleright M_i : \tau_i \ \&\ \varphi' \quad \varphi' \leq \varphi_i \quad \mathit{noIntermediate}(M_i)}{\Gamma \vdash_D f_i(x_{i,1}, \ldots, x_{i,m_i}) = M_i : (\tau_{i,1}, \ldots, \tau_{i,m_i}) \xrightarrow{\varphi_i} \tau_i} \quad \text{(T-FUNDEF)}$$

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

$$\frac{\begin{array}{c}\widetilde{D} = \{f_1(x_{1,1}, \ldots, x_{1,m_1}) = M_1, \ldots, f_l(x_{l,1}, \ldots, x_{l,m_l}) = M_l\} \\ H = \{y_1 \mapsto v_1, \ldots, y_k \mapsto v_k\} \\ L = \{z_1 \mapsto lck_1, \ldots, z_n \mapsto lck_n\} \\ \Gamma \vdash_D (f_i(x_{i,1}, \ldots, x_{i,m_i}) = M_i) : (\tau_{i,1}, \ldots, \tau_{i,m_i}) \xrightarrow{\varphi_i} \tau_i \quad (1 \leq i \leq l) \\ \Gamma \triangleright v_i : \tau_i' \ \&\ \emptyset \quad (1 \leq i \leq k) \\ \begin{array}{rcl}\Gamma & = & f_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \xrightarrow{\varphi_1} \tau_1, \ldots, f_l : (\tau_{l,1}, \ldots, \tau_{l,m_l}) \xrightarrow{\varphi_l} \tau_l, \\ & & y_1 : \tau_1' \ \mathbf{ref}, \ldots, y_k : \tau_k' \ \mathbf{ref}, \\ & & z_1 : \mathbf{lock}(lev_1), \ldots, z_n : \mathbf{lock}(lev_n)\end{array}\end{array}}{\vdash_{Env} (\widetilde{D}, H, L) : \Gamma} \quad \text{(T-ENV)}$$

$$\frac{\vdash_{Env} (\widetilde{D}, H, L) : \Gamma \qquad \Gamma \triangleright M : \tau \ \&\ (lev_1, lev_2)}{\vdash_C (\widetilde{D}, H, L, I, M) : \tau} \quad \text{(T-CONFIG)}$$

Figure 2.7: Typing Rules for Program and Configuration.

Here, i is defined by the following syntax.

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

In the definition above, i is a term that can be reduced by the rules in Figure 2.3. Thus, $\mathit{deadlocked}(L, M)$ means that every reducible subterm in M is a blocked lock-acquiring instruction. For example,

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

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

where

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

$$\mathcal{T}_2 = \dfrac{\Gamma \rhd \mathit{flush\_buffer\_iter} : (\textbf{lock}(1), \tau_d) \overset{(1,1)}{\rightarrow} \textbf{unit } \& \ \emptyset \quad \vdots}{\Gamma \rhd \mathit{flush\_buffer\_iter}(\mathit{devlock}, \mathit{dequeue}()) : \textbf{unit } \& \ (1,1)}$$

Figure 2.8: Derivation Tree of the body of *flush_buffer_iter*. $\Gamma = \mathit{flush\_buffer\_iter} : (\textbf{lock}(1), \tau_d \textbf{ ref}) \overset{(1,1)}{\rightarrow} \textbf{unit}, \mathit{flush\_buffer} : \textbf{lock}(1) \overset{(1,1)}{\rightarrow} \textbf{unit}, \mathit{receive\_data} : (\tau_p, \tau_d \textbf{ ref}, \textbf{lock}(1)) \overset{(1,1)}{\rightarrow} \textbf{unit}, \mathit{devlock} : \textbf{lock}(1), \mathit{data} : \tau_d$.

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

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 environment 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 2.1 (Type Soundness) *If* $\vdash_P \widetilde{D}M$ *and* $(\widetilde{D}, \emptyset, \emptyset, \textbf{enabled}, M) \rightarrow^* (\widetilde{D}', H', L', I', M')$, *then* $\neg \mathit{deadlocked}(L', M')$.

The theorem above follows from Lemmas 2.1–2.4 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 2.5 $\mathit{wellformed}(L, I, M)$ *holds if and only if*

- $L(x) = \textbf{released}$ *or* $x \notin \textbf{Dom}(L)$ *implies that* M *does not contain* **in_sync** x,
- $L(x) = \textbf{acquired}$ *implies* $\mathit{AckIn}(x, M)$,
- $I = \textbf{enabled}$ *implies that* M *does not contain* **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* **in_disable_int**.

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

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

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

Lemma 2.1 *If* $\vdash_P \widetilde{D}M$*, then wellformed*$(\emptyset, \mathbf{enabled}, M)$ *and* $\vdash_C (D, \emptyset, \emptyset, \mathbf{enabled}, M)$.

Lemma 2.2 *If wellformed*(L, I, M) *and* $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$*, then wellformed*$(L', I', M')$.

Lemma 2.3 (Preservation) *If* $\vdash_C (\widetilde{D}, H, L, I, M) : \tau$ *and* $(\widetilde{D}, H, L, I, M) \to (\widetilde{D}', H', L', I', M')$*, then* $\vdash_C (\widetilde{D}', H', L', I', M') : \tau$.

Lemma 2.4 (Deadlock-Freedom) *If* $\vdash_C (\widetilde{D}, H, L, I, M) : \tau$ *and wellformed*(L, I, M)*, then* $\neg$*deadlocked*(L, M).

Proofs of those lemmas are in Appendix A and B.

2.2.6 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

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

Note that $lev < lev_1$ in (T-Sync) can be replaced by $lev + 1 \leq lev_1$. The constraints above can be solved as in Kobayashi's type-based deadlock analysis for the π-calculus [32].

2.3 Extension

We extend our framework with non-block-structured mutex primitives in this section. We first show the extension to the syntax, and then show the extended the type system.

2.3.1 Target Language

Figure 2.9 shows the syntax of the new target language. We do not give a detailed explanation here because the intuitive meaning of most constructs can be guessed from the semantics of the original language. The main difference of the new and the old language is as follows.

- Instead of *expressions* M, we use *statements*, which are ranged over by a meta-variable s, as a body of a program.
- A block-structured mutex primitive **sync** x **in** M is replaced by non-block-structured ones **lock** x and **unlock** x. **disable_int** M is also replaced by **disable_int** and **enable_int** .

$$
\begin{array}{rcl}
x, y, z, f \ldots & \in & \mathit{Var} \\
\mathit{lck} & ::= & \mathbf{acquired} \mid \mathbf{released} \\
P & ::= & \widetilde{D}s \\
D & ::= & x(\widetilde{y}) = s \\
v & ::= & () \mid \mathbf{true} \mid \mathbf{false} \mid n \mid x \\
s & ::= & \mathbf{0} \mid x(\widetilde{v}) \mid (\mathbf{if}\ v\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) \\
 & \mid & \mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in}\ s \mid \mathbf{let}\ x = !y\ \mathbf{in}\ s \mid x := y \\
 & \mid & (s_1 \mid s_2) \mid \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s \\
 & \mid & \mathbf{lock}\ x \mid \mathbf{unlock}\ x \\
 & \mid & s_1 \rhd s_2 \mid s_1 \blacktriangleleft_s s_2 \\
 & \mid & \mathbf{disable\_int} \mid \mathbf{enable\_int} \\
 & \mid & s_1; s_2 \\
E & ::= & [\,] \mid E; s \mid (E \mid s) \mid (s \mid E) \\
 & \mid & E \rhd s \mid s_1 \blacktriangleleft_s E \\
I & ::= & \mathbf{enabled} \mid \mathbf{disabled}
\end{array}
$$

Figure 2.9: Syntax

- For a technical reason in designing the type system, we introduce a statement $\mathbf{let}\ x = !y\ \mathbf{in}\ s$ as a primitive in order to name the result of a dereference. The type system forces programs not to access the reference y inside s. We write $[!y/x]s$, which is a statement obtained by replacing every free occurrence of x in s with $!y$, for $\mathbf{let}\ x = !y\ \mathbf{in}\ s$ when the distinction between those two statements do not matter.

The semantics of the new language is shown in Figure 2.10.

2.3.2 Type System

Overview

We show the main idea of the extended type system in this section. Essentially, the extended type system guarantees that locks are acquired in an strict increasing order of their levels, as in the original type system. However, there are three difficulties in the extended language.

- The possibility that a lock acquired in a function may not be released within the function. For example, consider the two programs in Figure 2.11. The *main* function in the left-hand side program acquires x after executing the function f which acquires and releases x. The *main* function in the right-hand side program also acquires x after calling f. However, f only acquires x and does not release x, which leads to a deadlock. Thus, we

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

$$(\widetilde{D}, H, L, I, \mathbf{0}; s) \to (\widetilde{D}, H, L, I, s) \quad \text{(E-Seq)}$$

$$(\widetilde{D}, H, L, I, (\mathbf{if\ true\ then}\ s_1\ \mathbf{else}\ s_2)) \to (\widetilde{D}, H, L, I, s_1) \quad \text{(E-IfTrue)}$$

$$(\widetilde{D}, H, L, I, (\mathbf{if\ false\ then}\ s_1\ \mathbf{else}\ s_2)) \to (\widetilde{D}, H, L, I, s_2) \quad \text{(E-IfFalse)}$$

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

$$(\widetilde{D}, H[x \mapsto v'], L, I, x := v) \to (\widetilde{D}, H[x \mapsto v], L, I, \mathbf{0}) \quad \text{(E-Assign)}$$

$$(\widetilde{D}, H[y \mapsto v], L, I, E[\mathbf{let}\ x = !y\ \mathbf{in}\ s]) \to (\widetilde{D}, H[y \mapsto v], L, I, E[\![v/x]s]) \quad \text{(E-LetDeref)}$$

$$\frac{x' \text{ is fresh}}{(\widetilde{D}, H, L, I, \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s) \to (\widetilde{D}, H, L[x' \mapsto \mathbf{released}], I, [x'/x]s)} \quad \text{(E-LetNewlock)}$$

$$(\widetilde{D}, H, L, I, \mathbf{0} \mid \mathbf{0}) \to (\widetilde{D}, H, L, I, \mathbf{0}) \quad \text{(E-ParEnd)}$$

$$(\widetilde{D}, H, L[x \mapsto \mathbf{released}], I, \mathbf{lock}\ x) \to (\widetilde{D}, H, L[x \mapsto \mathbf{acquired}], I, \mathbf{0}) \quad \text{(E-Lock)}$$

$$(\widetilde{D}, H, L[x \mapsto \mathbf{acquired}], I, \mathbf{unlock}\ x) \to (\widetilde{D}, H, L[x \mapsto \mathbf{released}], I, \mathbf{0}) \quad \text{(E-Unlock)}$$

$$(\widetilde{D}, H, L, \mathbf{enabled}, s_1 \rhd s_2) \to (\widetilde{D}, H, L, \mathbf{enabled}, s_1 \blacktriangleleft_{s_2} s_2) \quad \text{(E-Interrupt)}$$

$$(\widetilde{D}, H, L, I, s_1 \blacktriangleleft_{s_2} \mathbf{0}) \to (\widetilde{D}, H, L, I, s_1 \rhd s_2) \quad \text{(E-ExitInterrupt)}$$

$$(\widetilde{D}, H, L, I, \mathbf{0} \rhd s) \to (\widetilde{D}, H, L, I, \mathbf{0}) \quad \text{(E-NoInterruptHalt)}$$

$$(\widetilde{D}, H, L, I, \mathbf{disable\_int}\ ) \to (\widetilde{D}, H, L, \mathbf{disabled}, s) \quad \text{(E-DisableInterrupt)}$$

$$(\widetilde{D}, H, L, I, \mathbf{enable\_int}\ ) \to (\widetilde{D}, H, L, \mathbf{enabled}, s) \quad \text{(E-EnableInterrupt)}$$

$$\frac{(\widetilde{D}, H, L, I, s) \to (\widetilde{D}', H', L', I', s')}{(\widetilde{D}, H, L, I, E[s]) \to (\widetilde{D}', H', L', I', E[s'])} \quad \text{(E-Context)}$$

Figure 2.10: Operational Semantics

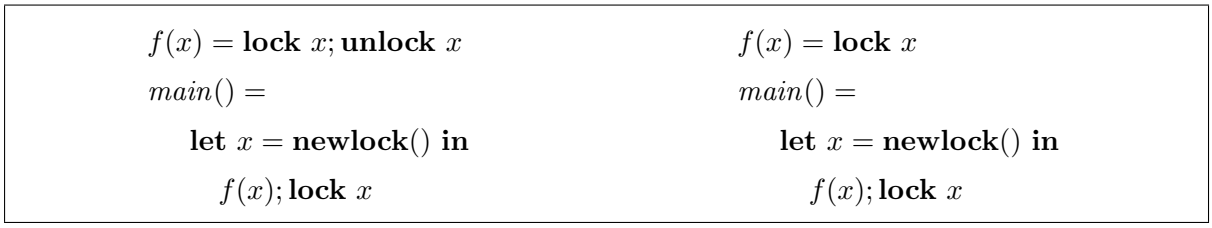


Figure 2.11: Examples that show a difficulty caused by non-block-structured mutex primitives.

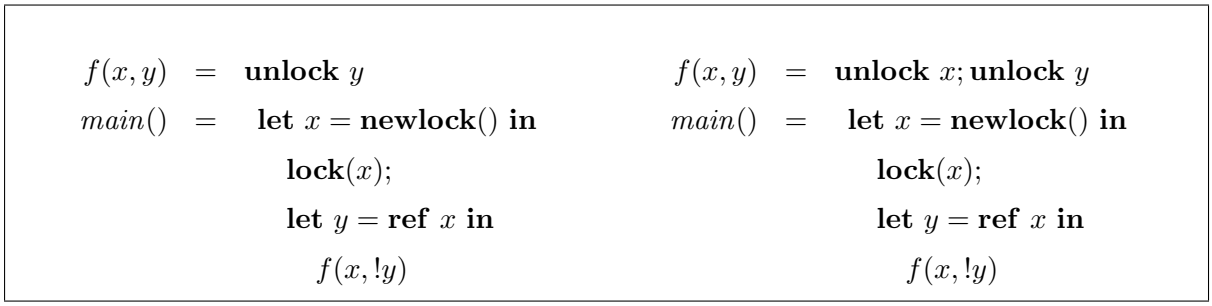


Figure 2.12: Programs that contain aliasing to a lock occurs.

should reject the latter program. In order to reject the right-hand side program, the type system has to track whether the lock x has to be released after a call to a function, which is not necessary in the original language.

- Aliasing. For example, consider the programs in Figure 2.12. The function f takes two locks x and y as arguments and releases both in the right-hand side program, while f releases only x in the left-hand side program. In the function *main*, both programs generate a fresh lock x and acquire the lock. After that, a pointer y to the lock x is generated, so that $!y$ is an alias of x. Then, the function *main* passes x and $!y$ to f. We should reject the right-hand side program because the program releases the lock generated in *main* twice. Note that such problem does not occur in the original language because it is syntactically guaranteed that an acquired lock is released exactly once in that language.

- Race conditions to a reference to a lock. Consider the program in Figure 2.13. The program generates two references r_1 and r_2, where r_1 is a reference to a lock x and r_2 is a reference to y. Then, two threads are spawned. The first thread acquires mutexes twice through r_1 and r_2, while the second thread assigns y to r_1. Due to that race to r_1, a deadlock occurs if $r_1 := y$ is executed before the first thread ends.

Instead of the effect-based approach of the original type system, the extended type system solves the first two difficulties using the idea of *capabilities* and *obligations* [32, 34, 35]. In the extended type system, each lock type has an *obligation* to release the lock, a *capability* to acquire the lock, or neither. The type system deals with the obligations and the capabilities based on the following principles.

```
let x = newlock() in
let y = newlock() in
let r1 = ref x in
let r2 = ref y in
  (lock(!r1); lock(!r2); unlock(!r2); unlock(!r1)) |
  (r1 := y)
```

Figure 2.13: A program that contains race condition to a reference to a mutex.

```
flush_buffer_iter(devlock, data) =
  if !data = Null then () else
    (lock devlock; unlock devlock); 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 statement *)
let devlock = newlock() in
let data = ref Null in
  flush_buffer(devlock) ▷ receive_data(Room, data, devlock)
```

Figure 2.14: An Encoding of the Program in Figure 1.1

1. **lock** x can be executed if and only if (1) x has a *capability* and (2) the level of every lock with an *obligation* is less than the level of x.

2. **unlock** x can be performed if and only if x has an obligation.

3. An obligation is treated linearly, that is, if an alias to a lock with an *obligation* is generated, then exactly one of the lock or the alias inherits the obligation.

Based on the principles above, the extended type system rejects the program in the right-hand side in Figure 2.11, because, due to the restriction (2) in the first principle, the level of x has to be less than itself for the program to be accepted, which is unsatisfiable. The program in Figure 2.12 is also rejected because only one of x or y has an obligation after the reference y is generated due to the third principle, while f requires both arguments to have an obligation.

In order to deal with interrupts, a capability and an obligation are accompanied by a tag that indicates whether the lock may be held while interrupts are enabled. Based on those tags, the type system infers the conditions that correspond to $lev_2 < lev_1'$ in the rule (T-InstHandler) in Figure 2.6. For example, consider the example in Figure 2.14, which encodes the program in Figure 1.1 with the extended language. For that program, the type system infers (1) that

$$
\begin{array}{rcl}
lev & \in & \{0, 1, \ldots\} \cup \{-\infty, \infty\} \\
\iota, \theta, \xi & ::= & \mathbf{DI} \mid \mathbf{EI} \\
U & ::= & \mathbf{0} \mid ob^{\iota} \mid cap^{\iota} \\
r & \in & [0, 1] \\
\tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{bool} \mid ((\tau_1, \ldots, \tau_n) \mid \iota) \xrightarrow{(\theta,\xi)}_{lev} ((\tau'_1, \ldots, \tau'_n) \mid \iota') \mid \tau\ \mathbf{ref}^r \\
 & \mid & \mathbf{lock}(lev, U)
\end{array}
$$

Figure 2.15: Syntax of types.

devlock in *flush_buffer*(*devlock*) has a capability with a tag that indicates the capability may be used while interrupts are enabled and (2) that *devlock* in *receive_data*(*Room*, *data*, *devlock*) has a capability, so that *devlock* may be acquired. Thus, the level of *devlock* must be less than itself for the program to be accepted, which is not satisfiable.

The third difficulty is solved using *ownerships*, the right of a thread to access a reference. As Boyland [6], Terauchi [61] and Kikuchi and Kobayashi [27] do, we use rational-numbered ownerships. A well-typed program obeys the following rules on ownerships in manipulating references.

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

Based on those rules, the program in Figure 2.13 is rejected because the total ownership required on the reference r_1 exceeds 1: the first thread requires an ownership more than 0 while the second thread requires 1.

Syntax

Figure 2.15 shows the syntax of types. In order to track the state of interrupts, we use *interrupt flags* ι which range over $\{\mathbf{EI}, \mathbf{DI}\}$. **EI** means that the interrupts are enabled and **DI** means disabled.

Usage, ranged over by a meta-variable U, represents a capability or an obligation assigned to a lock type. A usage **0** indicates that there is no obligation nor no capability on a lock type.

```
f(y) = disable_int ; lock y; enable_int
g() = enable_int
(* Main statement *)
let x = newlock() in
  f(x) | g()
```

Figure 2.16: A concurrent program in which a thread enables interrupts.

A usage ob^ι represents an obligation to release a lock, while a usage cap^ι represents a capability to acquire a lock.

Usages ob^ι and cap^ι are accompanied by an interrupt flag ι. The interrupt flag represents whether the lock can be held while interrupts are enabled. If ι is **DI**, then the lock can be held only while interrupts are disabled. If ι is **EI**, the lock can be held while interrupts are enabled. For example, if a lock has type $\mathbf{lock}(lev, cap^{\mathbf{DI}})$, then (1) it cannot be acquired while interrupts are enabled and (2) the program cannot enable interrupts while the lock is held. We introduce a partial order $\mathbf{DI} \leq \mathbf{EI}$, which means that the type system may conservatively assume that a lock which is held only while interrupts are disabled as one held while interrupts are enabled.

τ is a meta-variable that represents types. A type of references $\tau\ \mathbf{ref}^r$ is accompanied by an ownership r of the reference, which is a rational number in the set $[0, 1]$. A lock type $\mathbf{lock}(lev, U)$ has a usage U as well as a lock level lev.

A function type $(\widetilde{\tau} \mid \iota) \xrightarrow{(\theta,\xi)}_{lev} (\widetilde{\tau}' \mid \iota')$ consists of the following seven components.

- $\widetilde{\tau}$ and $\widetilde{\tau}'$: the types of arguments before and after execution of the functions.
- ι and ι': the interrupt flag before and after execution of the functions.
- *Context lock level lev*: the maximum level of locks which may have obligations in calling the function.
- *Self enable flag* ξ: an interrupt flag that indicates whether the function may enable interrupts during execution.
- *Context enable flag* θ: an interrupt flag that indicates whether the function can be concurrently executed with a thread that may enable interrupts.

We need θ and ξ because the interrupt state is shared among threads. For example, consider the program in Figure 2.16. Though the function f acquires the passed lock y after disabling interrupts, the type system has to assume that the capability of y may be used while interrupts are enabled (i.e., the type of y has to be $\mathbf{lock}(lev, cap^{\mathbf{EI}})$ for some lev) because f is concurrently executed with the function g which enables interrupts during execution. In that program, the

$\boxed{U_1 \otimes U_2}$

$$
\begin{aligned}
ob^{\iota} \otimes ob^{\iota'} &= \text{undefined} \\
ob^{\iota} \otimes cap^{\iota'} &= ob^{\iota \sqcup \iota'} \\
cap^{\iota} \otimes ob^{\iota'} &= ob^{\iota \sqcup \iota'} \\
cap^{\iota} \otimes cap^{\iota'} &= cap^{\iota \sqcup \iota'} \\
U \otimes \mathbf{0} = \mathbf{0} \otimes U &= U
\end{aligned}
$$

$\boxed{\omega U}$

$$
\begin{aligned}
\omega\, ob^{\iota} &= \text{undefined} \\
\omega\, cap^{\iota} &= cap^{\iota}
\end{aligned}
$$

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

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

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

$\tau \otimes \tau = \tau$
(where τ is $\mathbf{unit}, \mathbf{int}, \mathbf{bool}$, or a function type.)

$\boxed{\omega\tau}$

$\omega(\tau\ \mathbf{ref}^{r}) = (\omega\tau)\ \mathbf{ref}^{r}$

$\omega(\mathbf{lock}(lev, U)) = \mathbf{lock}(lev, \omega U)$

$\omega\tau = \tau$
(where τ is $\mathbf{unit}, \mathbf{int}, \mathbf{bool}$, or a function type.)

$\boxed{\Gamma_1 \otimes \Gamma_2, \omega\Gamma}$

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

$$
(\omega\Gamma)(x) = \omega(\Gamma(x))
$$

Figure 2.17: Definition of operators $\otimes$ and ω.

self enable flag of g and the context enable flag of f are inferred to be **EI**, from which the requirement on the type of y above is derived.

Type Judgment

The type judgment for statements in the extended type system is $(\Gamma \mid \iota) \triangleright^{(\theta,\xi)}_{lev} s \Rightarrow (\Gamma' \mid \iota')$. Type environments Γ and Γ' describe the types of free variables in s before and after execution of s. The interrupts flags ι and ι' represents the state of interrupts before and after execution of s. A context enable flag θ indicates whether there may exist other thread that may enable interrupts. A self enable flag ξ indicates whether s may enable interrupts. A lock level lev is the maximum level of locks that may be held by the context to which s belongs.

The type judgment intuitively means that locks are acquired in an strict increasing order of their levels and an acquired lock is released exactly once if the statement is executed (1) under an environment described by Γ, (2) under an interrupt state described by ι, (3) with threads whose enable/disable behavior on the interrupt state are described by θ and (4) with a

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

$$noob(\mathbf{0}) \quad \text{(NoOb-Zero)} \qquad noob(cap\iota) \quad \text{(NoOb-Unlocked)}$$

$$\frac{noob(U)}{noob(\mathbf{lock}(lev, U))} \quad \text{(NoOb-Lock)} \qquad \frac{noob(\tau)}{noob(\tau\ \mathbf{ref}^r)} \quad \text{(NoOb-Ref)}$$

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

Figure 2.18: Definition of *noob*.

$$\frac{noob(\Gamma)}{\Gamma \triangleright () : \mathbf{unit}} \quad \text{(T-Unit)} \qquad \frac{noob(\Gamma)}{\Gamma \triangleright n : \mathbf{int}} \quad \text{(T-Int)}$$

$$\frac{noob(\Gamma)}{\Gamma \triangleright \mathbf{true} : \mathbf{bool}} \quad \text{(T-True)} \qquad \frac{noob(\Gamma)}{\Gamma \triangleright \mathbf{false} : \mathbf{bool}} \quad \text{(T-False)}$$

$$\frac{noob(\Gamma)}{x : \tau, \Gamma \triangleright x : \tau} \quad \text{(T-Var)}$$

Figure 2.19: Typing rules for values.

continuation that respects types in Γ'.

The type judgment is defined as the least relation that satisfies the typing rules in Figure 2.19–2.21. In those rules, we use the following definitions.

Definition 2.6 ($\otimes$ and ω) *Operators $\otimes$ and ω are defined as in Figure 2.17.*

Definition 2.7 (Stripping) *An operator $\tau^\flat$ is defined as follows.*

$$\begin{aligned}
(\tau\ \mathbf{ref}^r)^\flat &= \tau^\flat\ \mathbf{ref}^0 \\
(\mathbf{lock}(lev, U))^\flat &= \mathbf{lock}(lev, \mathbf{0}) \\
\tau^\flat &= \tau \qquad \textit{(where } \tau \textit{ is } \mathbf{unit}, \mathbf{int}, \mathbf{bool}, \textit{ or a function type.)}
\end{aligned}$$

Definition 2.8 (No obligation) *$noob(\tau)$ is defined as the least predicate that satisfies the rules in Figure 2.18.*

Definition 2.9 *$level_U$, a function that takes a type and returns a set of lock levels, is defined as follows.*

$$\begin{aligned}
level_U(\tau\ \mathbf{ref}^r) &= level_U(\tau) \\
level_U(\mathbf{lock}(lev, U')) &= \{lev\} \qquad \textit{(where } U = U'\textit{)} \\
level_U(\tau) &= \emptyset \qquad (\tau = \mathbf{unit}, \mathbf{int}, \mathbf{bool},\ \textit{or}\ (\widetilde{\tau} \mid \iota) \xrightarrow{(\theta,\xi)}_{lev} (\widetilde{\tau}' \mid \iota'))
\end{aligned}$$

$$(\Gamma \mid \iota) \rhd_{lev}^{(\theta,\mathbf{DI})} \mathbf{0} \Rightarrow (\Gamma \mid \iota) \qquad \text{(T-Nop)}$$

$$\frac{(\Gamma_1 \mid \iota_1) \rhd_{lev}^{(\theta,\xi_1)} s_1 \Rightarrow (\Gamma_2 \mid \iota_2) \qquad (\Gamma_2 \mid \iota_2) \rhd_{lev}^{(\theta,\xi_2)} s_2 \Rightarrow (\Gamma_3 \mid \iota_3)}{(\Gamma_1 \mid \iota_1) \rhd_{lev}^{(\theta,\xi_1 \sqcup \xi_2)} s_1; s_2 \Rightarrow (\Gamma_3 \mid \iota_3)} \qquad \text{(T-Seq)}$$

$$\frac{\begin{array}{c}(x : \mathbf{lock}(lev, U_1), \Gamma \mid \iota) \rhd_{lev}^{(\theta,\xi)} s \Rightarrow (x : \mathbf{lock}(lev, U_2), \Gamma' \mid \iota') \\ noob(U_1) \qquad noob(U_2)\end{array}}{(\Gamma \mid \iota) \rhd_{lev}^{(\theta,\xi)} \mathbf{let}\ x = \mathbf{newlock}\ ()\ \mathbf{in}\ s \Rightarrow (\Gamma' \mid \iota')} \qquad \text{(T-Newlock)}$$

$$\frac{lev' \sqcup \mathbf{max}(level_{ob}(\Gamma)) < lev \qquad \iota \sqcup \theta \leq \iota'}{(\Gamma, x : \mathbf{lock}(lev, cap^{\iota'}) \mid \iota) \rhd_{lev'}^{(\theta,\mathbf{DI})} \mathbf{lock}\ x \Rightarrow (\Gamma, x : \mathbf{lock}(lev, ob^{\iota'}) \mid \iota)} \qquad \text{(T-Lock)}$$

$$\frac{\iota \sqcup \theta \leq \iota'}{(x : \mathbf{lock}(lev, ob^{\iota'}) \mid \iota) \rhd_{lev}^{(\theta,\mathbf{DI})} \mathbf{unlock}\ x \Rightarrow (x : \mathbf{lock}(lev, cap^{\iota'}) \mid \iota)} \qquad \text{(T-Unlock)}$$

$$\frac{\begin{array}{c}\Gamma_i \rhd v_i : \tau_i\ (i = 1, \ldots, n) \qquad \Gamma'_i \rhd v_i : \tau'_i\ (i = 1, \ldots, n) \\ lev' \leq lev \\ \Gamma_{pre} = (\Gamma_1 \otimes \cdots \otimes \Gamma_n) \otimes (x : ((\tau_1, \ldots, \tau_n) \mid \iota) \overset{(\theta,\xi)}{\rightarrow}_{lev} ((\tau'_1, \ldots, \tau'_n) \mid \iota')) \\ \Gamma_{post} = (\Gamma'_1 \otimes \cdots \otimes \Gamma'_n) \otimes (x : ((\tau_1, \ldots, \tau_n) \mid \iota) \overset{(\theta,\xi)}{\rightarrow}_{lev} ((\tau'_1, \ldots, \tau'_n) \mid \iota')) \\ \theta_1 \leq \theta \qquad x : \mathbf{lock}(lev'', ob^{\iota''}) \in \Gamma_{pre} \Longrightarrow \xi \leq \iota''\end{array}}{(\Gamma_{pre} \mid \iota) \rhd_{lev'}^{(\theta_1,\xi)} x(v_1, \ldots, v_n) \Rightarrow (\Gamma_{post} \mid \iota')} \qquad \text{(T-App)}$$

$$\frac{\Gamma_1 \rhd v : \mathbf{bool} \qquad (\Gamma_2 \mid \iota) \rhd_{lev}^{(\theta,\xi_1)} s_1 \Rightarrow (\Gamma \mid \iota') \qquad (\Gamma_2 \mid \iota) \rhd_{lev}^{(\theta,\xi_2)} s_2 \Rightarrow (\Gamma \mid \iota')}{(\Gamma_1 \otimes \Gamma_2 \mid \iota) \rhd_{lev}^{(\theta,\xi_1 \sqcup \xi_2)} \mathbf{if}\ v\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \Rightarrow (\Gamma \mid \iota')} \qquad \text{(T-If)}$$

Figure 2.20: Typing rules for statements (1/2).

We write $level_{ob}(\tau)$ *for* $level_{ob^{\mathbf{EI}}}(\tau) \cup level_{ob^{\mathbf{DI}}}(\tau)$, *and* $level_{cap}(\tau)$ *for* $level_{cap^{\mathbf{EI}}}(\tau) \cup level_{cap^{\mathbf{DI}}}(\tau)$.
$level_U(\Gamma)$ *is defined as* $\{lev | x : \tau \in \Gamma \wedge lev \in level_U(\tau)\}$.

$U_1 \otimes U_2$ gives the usage that means *both* obligations in U_1 and U_2 have to be fulfilled. $ob^{\iota} \otimes ob^{\iota'}$ is undefined because releasing an acquired lock twice is prohibited. ωU intuitively means $U \otimes U \otimes \cdots \otimes U$. Operators $\otimes$ and ω on usages are naturally extended to types. $\tau^{\flat}$ is an operation that strips all the capability, obligation and ownerships from τ. $noob(\tau)$ is a predicate that asserts τ does not have any obligation to fulfil. $level_U$ is a function that collects levels of locks whose usages are equal to U.

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

$$\frac{\Gamma_1 \triangleright v : \tau \qquad (x : \tau\ \mathbf{ref}^1, \Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta,\xi)} s \Rightarrow (x : \tau'\ \mathbf{ref}^1, \Gamma_2' \mid \iota') \qquad noob(\tau')}{(\Gamma_1 \otimes \Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta,\xi)} \mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in}\ s \Rightarrow (\Gamma_2' \mid \iota')} \text{ (T-Ref)}$$

$$\frac{(x : \tau, y : \tau^\flat\ \mathbf{ref}^0, \Gamma \mid \iota) \triangleright_{lev}^{(\theta,\xi)} s \Rightarrow (x : \tau_1, y : \tau_2\ \mathbf{ref}^0, \Gamma' \mid \iota') \qquad r > 0}{(y : \tau\ \mathbf{ref}^r, \Gamma \mid \iota) \triangleright_{lev}^{(\theta,\xi)} \mathbf{let}\ x = !y\ \mathbf{in}\ s \Rightarrow (y : \tau_1 \otimes \tau_2\ \mathbf{ref}^r, \Gamma' \mid \iota')} \text{ (T-Deref)}$$

$$\frac{noob(\tau_3)}{(x : \tau_3\ \mathbf{ref}^1, y : \tau_1 \otimes \tau_2 \mid \iota) \triangleright_{lev}^{(\theta,\mathbf{DI})} x := y \Rightarrow (x : \tau_1\ \mathbf{ref}^1, y : \tau_2 \mid \iota)} \text{ (T-Assign)}$$

$$(\Gamma \mid \iota) \triangleright_{lev}^{(\theta,\mathbf{DI})} \mathbf{disable\_int}\ \Rightarrow (\Gamma \mid \mathbf{DI}) \quad \text{(T-DisableInterrupt)}$$

$$\frac{x : \mathbf{lock}(lev, ob^\iota) \in \Gamma \Longrightarrow \iota = \mathbf{EI}}{(\Gamma \mid \iota) \triangleright_{lev}^{(\theta,\mathbf{EI})} \mathbf{enable\_int}\ \Rightarrow (\Gamma \mid \mathbf{EI})} \text{ (T-EnableInterrupt)}$$

$$\frac{\begin{array}{c}(\Gamma_1 \mid \iota) \triangleright_{lev}^{(\theta_1,\xi_1)} s_1 \Rightarrow (\Gamma_1' \mid \iota_1) \qquad (\Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta_2,\xi_2)} s_2 \Rightarrow (\Gamma_2' \mid \iota_2) \\ \xi_2 \sqcup \theta \leq \theta_1 \qquad \xi_1 \sqcup \theta \leq \theta_2 \\ noob(\Gamma_1') \qquad noob(\Gamma_2')\end{array}}{(\Gamma_1 \otimes \Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta,\xi_1 \sqcup \xi_2)} s_1 \mid s_2 \Rightarrow (\Gamma_1' \otimes \Gamma_2' \mid \iota_1 \sqcup \iota_2)} \text{ (T-Par)}$$

$$\frac{\begin{array}{c}(\Gamma_1 \mid \iota) \triangleright_{lev}^{(\theta,\xi_1)} s_1 \Rightarrow (\Gamma_1' \mid \iota') \\ (\Gamma_2 \mid \mathbf{EI}) \triangleright_{lev}^{(\theta,\xi_2)} s_2 \Rightarrow (\Gamma_2' \mid \iota'') \\ noob(\Gamma_2') \\ max(level_{ob}(\Gamma_1) \cup level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma_2))\end{array}}{(\Gamma_1 \otimes \omega\Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta,\xi_1 \sqcup \xi_2)} s_1 \rhd s_2 \Rightarrow (\Gamma_1' \mid \iota' \sqcup \iota'')} \text{ (T-InstHandler)}$$

$$\frac{\begin{array}{c}(\Gamma_1 \mid \iota) \triangleright_{lev}^{(\theta,\xi_1)} s_1 \Rightarrow (\Gamma_1' \mid \iota_1) \qquad (\Gamma_2 \mid \iota) \triangleright_{lev}^{(\theta,\xi_2)} s_2 \Rightarrow (\Gamma_2' \mid \iota_2) \\ (\Gamma \mid \mathbf{EI}) \triangleright_{lev}^{(\theta,\xi)} s \Rightarrow (\Gamma' \mid \iota'') \\ noob(\Gamma_2') \qquad noob(\Gamma') \\ max(level_{ob}(\Gamma_1) \cup level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma_2)) \\ max(level_{ob}(\Gamma_1) \cup level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma))\end{array}}{(\Gamma_2 \otimes \Gamma_1 \otimes \omega\Gamma \mid \iota) \triangleright_{lev}^{(\theta,\xi_1 \sqcup \xi_2 \sqcup \xi)} s_1 \blacktriangleleft_s s_2 \Rightarrow (\Gamma_1' \mid \iota')} \text{ (T-InInterrupt)}$$

$$\frac{\begin{array}{c}(\Gamma \mid \iota) \triangleright_{lev}^{(\theta,\xi)} s \Rightarrow (\Gamma' \mid \iota') \\ max(level_{ob}(\Gamma'')) < min(level_{cap}(\Gamma))\end{array}}{(\Gamma, \Gamma'' \mid \iota) \triangleright_{lev}^{(\theta,\xi)} s \Rightarrow (\Gamma', \Gamma'' \mid \iota')} \text{ (T-Weak)}$$

Figure 2.21: Typing rules for statements (2/2).

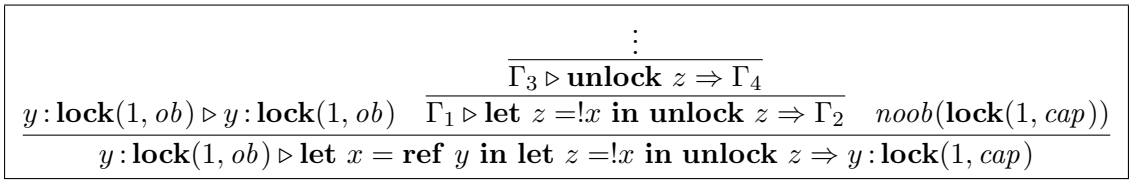


Figure 2.22: A derivation tree of **let** $x =$ **ref** y **in unlock** x under the assumption $y : \mathbf{lock}(0, ob)$, where

(T-Lock) assigns an obligation to the type of x. The condition $lev' \sqcup \mathbf{max}(level_{ob}(\Gamma)) < lev$ guarantees that the level of lock x is greater than levels of already acquired locks. Note that the condition $lev < lev_1$ in the rule (T-Sync) in Figure 2.6 corresponds to $lev' \sqcup \mathbf{max}(level_{ob}(\Gamma)) < lev$ in (T-Lock).

The condition $\iota \sqcup \theta \leq \iota'$ in (T-Lock) means that if $\iota' = \mathbf{DI}$ (i.e., x should not be acquired while interrupts are enabled), then the interrupt flag ι should be **DI** and there should not be other threads which may enable interrupts. For example, $(x : \mathbf{lock}(0, cap^{\mathbf{EI}}) \mid \mathbf{EI}) \triangleright_{lev}^{(\mathbf{DI},\mathbf{DI})}$ **lock** $x \Rightarrow (x : \mathbf{lock}(0, cap^{\mathbf{EI}}) \mid \mathbf{EI})$ holds, while $(x : \mathbf{lock}(0, cap^{\mathbf{DI}}) \mid \mathbf{EI}) \triangleright_{lev}^{(\mathbf{DI},\mathbf{DI})}$ **lock** $x \Rightarrow (x : \mathbf{lock}(0, cap^{\mathbf{DI}}) \mid \mathbf{EI})$ nor $(x : \mathbf{lock}(0, cap^{\mathbf{DI}}) \mid \mathbf{DI}) \triangleright_{lev}^{(\mathbf{EI},\mathbf{DI})}$ **lock** $x \Rightarrow (x : \mathbf{lock}(0, cap^{\mathbf{DI}}) \mid \mathbf{EI})$ does not hold. The pre-interrupt flag of the second judgment **EI** means that interrupts may be enabled when **lock** x is executed. However, the type of x is $\mathbf{lock}(0, cap^{\mathbf{DI}})$ which means that the lock may not be held while interrupts are enabled, so that the judgment does not hold. The third judgment does not hold either because its context enable flag is **EI**, so that there may be another thread which may enable interrupts.

(T-Unlock): The usage of the type of x is changed from $ob^{\iota'}$ to $cap^{\iota'}$. We need the condition $\iota \sqcup \theta \leq \iota'$ for the same reason as in the case (T-Lock).

In the rule (T-Ref), an ownership 1 is assigned to the new reference x in typing s. At the end of s, x should not have any obligation because x cannot be accessed after execution of s.

In the rule (T-Deref), all the obligations, capabilities and ownerships in the type of y are stripped from y and delegated to x in typing s. Thus, y cannot be accessed inside s. At the end of s, obligations, capabilities and ownerships of x are returned to y.

A derivation of a statement **let** $x =$ **ref** y **in let** $z =!x$ **in unlock** z under the type environment $y : \mathbf{lock}(0, ob^{\mathbf{EI}})$ in Figure 2.22 shows how (T-Ref) and (T-Deref) work. For the simplicity of the presentation, we have erased the interrupt-related contents such as **DI** and **EI** from the derivation tree. The type environments $\Gamma_1, \ldots, \Gamma_4$ in that figure is defined as follows.

$$
\begin{aligned}
\Gamma_1 &= x : \mathbf{lock}(1, ob)\ \mathbf{ref}^1, y : \mathbf{lock}(1, cap) \\
\Gamma_2 &= x : \mathbf{lock}(1, cap)\ \mathbf{ref}^1, y : \mathbf{lock}(1, cap) \\
\Gamma_3 &= x : \mathbf{lock}(1, \mathbf{0})\ \mathbf{ref}^0, y : \mathbf{lock}(1, cap), z : \mathbf{lock}(1, ob) \\
\Gamma_4 &= x : \mathbf{lock}(1, \mathbf{0})\ \mathbf{ref}^0, y : \mathbf{lock}(1, cap), z : \mathbf{lock}(1, cap).
\end{aligned}
$$

Note that the obligation of y is passed to the newly generated reference x, delegated to z and fulfilled through z.

The rule (T-Assign) guarantees that there is no obligation that must be fulfilled through the reference x because x is being overwritten. The obligation or the capability of y is distributed to x and y by the assignment. For example, both of

$$x : \mathbf{lock}(0, cap)\ \mathbf{ref}^1, y : \mathbf{lock}(0, ob) \triangleright x := y \Rightarrow x : \mathbf{lock}(0, cap)\ \mathbf{ref}^1, y : \mathbf{lock}(0, ob)$$

and

$$x : \mathbf{lock}(0, cap)\ \mathbf{ref}^1, y : \mathbf{lock}(0, ob) \triangleright x := y \Rightarrow x : \mathbf{lock}(0, ob)\ \mathbf{ref}^1, y : \mathbf{lock}(0, cap)$$

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

$$x : \mathbf{lock}(0, cap)\ \mathbf{ref}^1, y : \mathbf{lock}(0, ob) \triangleright x := y \Rightarrow x : \mathbf{lock}(0, ob)\ \mathbf{ref}^1, y : \mathbf{lock}(0, ob)$$

does not hold.

In the rule (T-Par), obligations, capabilities and ownerships in the type environments of s_1 and s_2 are added using $\otimes$ operator in the conclusion part. The condition $\xi_2 \sqcup \theta \leq \theta_1$ guarantees that, if we have $\theta_1 = \mathbf{DI}$, which means s_1 should not be executed with threads that may enable interrupts, then neither s_2 nor threads other than s_1 and s_2 should not enable interrupts (indicated by $\xi_2 = \mathbf{DI}$ and $\theta = \mathbf{DI}$ respectively.) The intuitive meaning of $\xi_1 \sqcup \theta \leq \theta_2$ is similar.

In the rules (T-DisableInterrupt) and (T-EnableInterrupt), the interrupt flags are changed to **DI** and **EI** respectively. The condition $x : \mathbf{lock}(lev, ob^\iota) \in \Gamma \Longrightarrow \iota = \mathbf{EI}$ in the rule (T-EnableInterrupt) guarantees that every lock type with an obligation in Γ is accompanied by an interrupt flag **EI**. This is necessary to guarantee the invariant that a lock with a usage $ob^{\mathbf{DI}}$ and $cap^{\mathbf{DI}}$ can be held only while interrupts are disabled.

The condition $max(level_{ob}(\Gamma_1) \cup level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma_2))$ in the rule (T-InstHandler) guarantees that the minimum level of locks that may be acquired in s_2 (i.e., $min(level_{cap}(\Gamma_2))$) should be greater than (1) the maximum level of the already acquired locks (i.e., $level_{ob}(\Gamma_1)$) and (2) the maximum level of locks that may be acquired during interrupts are enabled (i.e., $level_{cap^{\mathbf{EI}}}(\Gamma_1)$.) This condition corresponds to the condition $lev_2 < lev_1'$ of the rule (T-InstHandler) in Figure 2.6.

The rule (T-Weak) is for adding redundant variables to type environments. In that rule, the condition $max(level_{ob}(\Gamma'')) < min(level_{cap}(\Gamma))$ guarantees that if newly added lock types have obligations, then the levels of those lock types ($level_{ob}(\Gamma'')$) should be less than the level of locks that may be acquired in s ($level_{cap}(\Gamma)$.)

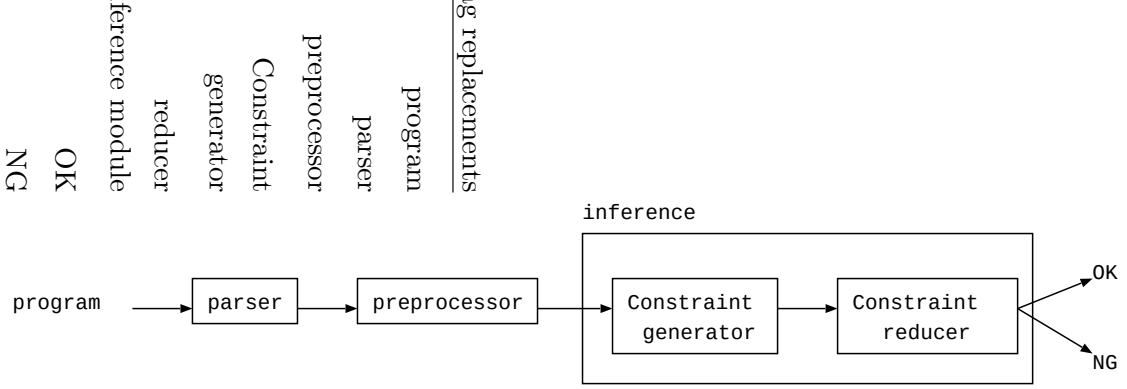


Figure 2.23: Structure of the verifier.

Type Inference

We can design a standard constraint-based type inference algorithm for our type system. We informally explain the behavior of such algorithm.

The type inference algorithm consists of two subalgorithms: constraint generation and constraint reduction. The constraint generation algorithm receives a program in which every bound variable is annotated with its simple type, and returns a set of constraints that should be satisfied for the program to be well-typed by constructing a typing derivation tree based on the rules in Figure 2.20 and 2.21. A constraint set consists of (1) inequalities between interrupt flags ($\iota_1 \leq \iota_2$) (2) inequalities between lock levels ($lev_1 \leq lev_2$) (3) equalities between usages ($\gamma = \gamma_1 \otimes \cdots \otimes \gamma_n$ where γ is a usage variable or a usage defined in Figure 2.15) (4) $noob(\gamma)$ and (5) linear inequalities among ownerships.

The generated constraints are passed to the constraint reduction algorithm. The algorithm solves constraints of the form (1)–(4) above in a standard manner. The constraints of the form (5) are solved using a linear inequality solver.

2.4 Experiment

2.4.1 Implementation

Based on the type system introduced in the previous section, we have implemented a deadlock-freedom verifier. Figure 2.23 shows the structure of the verifier. The verifier is implemented in Objective Caml [38]. The verifier first constructs the abstract syntax tree of an input program using CIL [49]. Then, the verifier applies several pre-processing, such as conversion of an assignment to a variable into one through a pointer, to the abstract syntax tree. After that, the verifier passes the tree to the type inference algorithm, which was described in Section 2.3.2. If the type inference succeeds, the verifier reports that the program is deadlock-free. Otherwise, the verifier reports the error.

The verifier does not require any type annotation. Types are automatically inferred by the type inference algorithms. However, when one passes a source file which does not contain `main` function, the user has to provide `main` which describes how the functions in the passed file are

```
/* A function that may be called while interrupts are enabled. */
int f(...) {
  ...
}

/* A function that is used as an interrupt handler. */
int handler(...) {
  ...
}
```

Figure 2.24: A source file `example.c`, which does not contain `main` function.

```
int main(int argc, char **argv) {
  request_irq(handler);
  enable_irq();
  f(...);
  return 0;
}
```

Figure 2.25: An example of a `main` function that describes how the functions in Figure 2.24 are used.

used. For example, consider the file `example.c` in Figure 2.24. Suppose that the programmer intends to call `f` while interrupts are enabled and to use `handler` as an interrupt handler. Then, one has to provide `main` function in Figure 2.25 in verifying `example.c`. Inside the `main` function, the programmer's intention that `handler` is used as an interrupt handler is expressed by a call to `request_irq` with an argument `handler`. `request_irq` is a function that installs its argument as an interrupt handler. Another intention that `f` is called while interrupts are enabled is also expressed by a call to `f` just after `enable_irq`, which enables interrupts. By inspecting the `main` function, the verifier generates constraints that correspond to those intention.

The current implementation cannot deal with a program in which obligations/capabilities arise/disappear inside structures. In order to verify such program, one has to rewrite the program that does not manipulate a lock inside structures.

2.4.2 Deadlock-freedom verification of a device driver

Using the current implementation of the verifier, we have conducted a verification experiment on a part of the source code of a protocol stack [41] presented in Section 1.2.1. The primary aim

```
int main(int argc, char **argv)
{
  struct csocknet_connection con;
  spinlock_t *devlock;

  /* csocknet_recv is used as an interrupt handler. */
  request_irq(0, csocknet_recv, 0, (int*)0, 0);

  /* csocknet_sendq_flush may be called while interrupts are enabled. */
  enable_irq();
  csocknet_sendq_flush(&con, devlock);
  return 0;
}
```

Figure 2.26: `main` function provided for the verification.

of this experiment is to clarify weaknesses of the current type system and the implementation. We do not intend to insist the current implementation is applicable to the real-world programs.

We have manually extracted the function definitions which are involved in the deadlock bug mentioned in Section 1.2.1 from the source codes in the implementation of [41], have made corrections to get the extracted source code to meet the restrictions on locks in structures explained in the previous section, and have provided a `main` function in Figure 2.26, which describes how the functions in the extracted code are intended to be used. After that work, the input program consists of 869 lines of code.

Figure 2.27 shows a part of the definition of a function `csocknet_send_to_device`, which is contained in the extracted source code. The function `csocknet_send_to_device` corresponds to `flush_buffer` presented in Figure 1.1. Due to the restriction on structures explained in the previous section, lock acquiring/releasing operations, which were originally operations on `&dev->lock`, are replaced with operations on `devlock`, a lock passed to `csocknet_send_to_device`.

If the program is compiled with a macro constant `BUGGY` defined, the function `csocknet_send_to_device` uses `spin_lock` and `spin_unlock` in acquiring/releasing `devlock` instead of `spin_lock_irq`, which disables interrupts before acquiring a lock, and `spin_unlock_irq`, which enables interrupts after releasing a lock, so that deadlock may occur due to the reason mentioned in Section 1.2.1.

Figure 2.28 shows a part of the verification result of the input program in which the macro constant `BUGGY` is defined. This output shows how the verifier reports verification failure. The

```
static int csocknet_send_to_device(struct sk_buff *skb, spinlock_t *devlock)
{
  struct net_device *dev = skb->dev;

  if (dev->type == ARPHRD_MYRI){
    int rc;
    unsigned long flags;
#ifdef BUGGY
    spin_lock(devlock);
#else
    /* Acquires a lock after disabling interrupts. */
    spin_lock_irq(devlock);
#endif
    /* Originally netif_queue_stopped(dev) */
    if (netif_queue_stopped_dis(dev)) {
#ifdef BUGGY
      spin_unlock(devlock);
#else
      spin_unlock_irq(devlock);
#endif
      return 1;
    }
    rc = dev->hard_start_xmit(skb, dev);
#ifdef BUGGY
    spin_unlock(devlock);
#else
    spin_unlock_irq(devlock);
#endif
    return rc;
  } else {
    return dev_queue_xmit(skb);
  }
}
```

Figure 2.27: A part of the input used in the experiment.

```
LESSTHAN constraints among lock levels cannot be solved.
There is a circle from ''LEVVAR 39482.
Globals:
...
csocknet_send_to_device:
FUN(prearg : ref('STRUCT sk_buff(6711), ''OVAR 6140 );
             ref(lock(''LEVVAR 39482 , lockstate('CAP, 'ENABLED)),
                 ''OVAR 39400 )
    postarg : ref('STRUCT sk_buff(34234), ''OVAR 39178 );
              ref(lock(''LEVVAR 39482 , lockstate('CAP, 'ENABLED)),
                  ''OVAR 39400 )...)
csocknet_recv:::
FUN(prearg : ref('STRUCT sk_buff(26562), ''OVAR 26733 );
             ref('STRUCT net_device(30924), ''OVAR 39382 );
             ref('STRUCT packet_type(29560), ''OVAR 39385 );
             ref('STRUCT net_device(30003), ''OVAR 39388 );
             ref(lock(''LEVVAR 39396 , lockstate('CAP, ''IVAR 28231 )),
                 ''OVAR 39393 );
             ref(lock(''LEVVAR 39482 , lockstate('CAP, 'ENABLED)),
                 ''OVAR 39400 )
    postarg : ref('STRUCT sk_buff(24703), ''OVAR 39379 );
              ref('STRUCT net_device(30924), ''OVAR 39382 );
              ref('STRUCT packet_type(29560), ''OVAR 39385 );
              ref('STRUCT net_device(30003), ''OVAR 39388 );
              ref(lock(''LEVVAR 39396 , lockstate('CAP, ''IVAR 28231 )),
                  ''OVAR 39393 );
              ref(lock(''LEVVAR 39482 , lockstate('CAP, 'ENABLED)),
                  ''OVAR 39400 )...)
...
```

Figure 2.28: A part of the output.

verifier reports that an inequality of lock levels cannot be solved, and that a lock level variable `''LEVVAR 39482` is involved in the error. The verifier also presents the inferred types of the defined functions. From that type information, we have the levels of the second parameter of `csocknet_send_to_device` (i.e., `devlock` in Figure 2.27) and the sixth parameter of an interrupt handler `csocknet_recv` are `''LEVVAR 39482`. We also have that the interrupt flag added to the usage of the second parameter of `csocknet_send_to_device` is **enabled**, so that the condition $max(level_{ob}(\Gamma_1) \cup level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma_2))$ was not satisfiable. The verifier reports no error when we do not define `BUGGY`.

2.4.3 Discussion

We have discovered the following two weaknesses of the current type system through the false-alarms reported by the verifier during the experiment.

Lack of polymorphism on interrupt flags

Consider the program in Figure 2.29. Though that program does not deadlock, the current type system rejects that program as follows.

1. The function `f` does nothing and returns, so that its inferred type is $(\mathbf{unit} \mid \iota) \to (\mathbf{unit} \mid \iota)$ for some interrupt flag ι. (We have omitted the self enable flag, the context enable flag and the context lock level from the type of `f`.) Note that the pre- and the post-interrupt flags of the type of `f` are the same.

2. Because the function `g` calls `f` while interrupts are enabled, the interrupt flag ι above has to be **EI** from the typing rules (T-EnableInterrupt), (T-Seq) and (T-App), so that the type of `f` is $(\mathbf{unit} \mid \mathbf{EI}) \to (\mathbf{unit} \mid \mathbf{EI})$, which implies interrupts may be enabled at the end of a call to `f`.

3. The function `h` disables interrupts, calls `f`, and then acquires a lock `l`. From the type of `f`, which says interrupts may be enabled after the call to `f`, the usage of the lock `l` before the lock acquiring operation should be, from (T-App), (T-Seq) and (T-Lock), $cap^{\mathbf{EI}}$, which means `l` may be acquired while interrupts are enabled, though `l` is acquired only while interrupts are disabled. (Note that `h` disables interrupts at the beginning and `f` does not change the interrupt state.)

4. The function `handler` acquires the lock `l`. Thus, the usage of `l` should be $cap^{\mathbf{EI}}$ before a call to `handler`.

5. In `main`, the function `handler` is installed as an interrupt handler and then `g` is called, which is encoded as `h`()$\triangleright$`handler`() in our calculus. Thus, the condition $max(level_{ob}(\Gamma_1) \cup$

$level_{cap^{\mathbf{EI}}}(\Gamma_1)) < min(level_{cap}(\Gamma_2))$ in the typing rule (T-InstHandler) is not satisfiable because $level_{cap^{\mathbf{EI}}}(\Gamma_1) = \{lev\}$ and $level_{cap}(\Gamma_2) = \{lev\}$.

In the experiment, we avoided false-alarms of this kind by manually duplicating the implementation of `f` as `f_dis`, and called `f` while interrupts are enabled (i.e., in `g`) and `f_dis` while disabled (i.e., in `h`.) For example, a call to `netif_queue_stopped_dis` in Figure 2.27 is a result of such duplication. We need to extend our type system with polymorphism on interrupt flags in order to avoid such manual duplication.

Lack of path-sensitivity

Consider the following definition of a function `f`.

```
void f(...) {
  int sk;

  if (sk) {
    ...
  } else {
    spin_lock(conlock);
  }
  ...
  /* The function does not change the value of sk in this part. */
  ...
  if (sk) {
    ...
  } else {
    spin_unlock(conlock);
  }
}
```

The function `f` first acquires a lock `conlock` if a local variable `sk` is false. After doing some work which does not change the value of `sk`, the function again checks the value of `sk` and unlocks `conlock` if `sk` is false. Because `sk` is a local variable, the value of `sk` is not changed by other threads or interrupt handlers, so that `conlock` is released exactly once after it is acquired. However, because our types are not path-sensitive, the verifier cannot determine the usage of `conlock` after the first branch, so that reports a type error. We need to rewrite the program as follows.

```
void f(...) {
  int sk;

  if (sk) {
    ...
    /* Code that was between the first and the second branch
       in the original source is inserted here. */
    ...
  } else {
    spin_lock(conlock);
    /* Code that was between the first and the second branch
       in the original source is inserted here. */
    spin_unlock(conlock);
  }
}
```

The code above does not require path-sensitivity on the type of `conlock`, so that the verification succeeds.

```
/* A function that does not change the interrupt state. */
int f() {
  return 0;
}

int g() {
  enable_irq();
  /* f is called while interrupts are enabled. */
  f();
  return 0;
}

spinlock_t *l;

int h() {
  disable_irq();
  f();
  spin_lock(l);
  spin_unlock(l);
}

/* An interrupt handler. */
void handler() {
  spin_lock(l);
  spin_unlock(l);
}

int main(int argc, char **argv) {
  request_irq(handler);
  h();
}
```

Figure 2.29: An example that describes a problem caused by lack of polymorphism on interrupt flags.

Chapter 3

Resource Usage Analysis for the π-Calculus

We present a type-based resource usage analysis for the π-calculus extended with resource creation/access primitives in this chapter. Section 3.1 shows the syntax and the semantics of our target language. Section 3.2 presents a type system for the verification of resource safety property and states the soundness of the type system. Section 3.3 presents a type inference algorithm for the type system. Section 3.4 shows an extension of the type system for verification of partial liveness property. Section 3.5 reports a prototype of resource usage analyzer for the presented calculus.

The contents of this chapter is presented in a journal Logical Methods in Computer Science [37]. The formalization through Section 3.1–3.4 is a joint work with Naoki Kobayashi and Lucian Wischik. The implementation presented in 3.5 is due to the author.

3.1 Processes

This chapter introduces the syntax and the operational semantics of our target language.

3.1.1 Syntax

Definition 3.1 (processes) *The set of processes is defined by the following syntax.*

$$\begin{array}{lll} P \text{ (processes)} & ::= & \mathbf{0} \mid \overline{x}\langle v_1, \ldots, v_n\rangle.\, P \mid x(y_1, \ldots, y_n).\, P \\ & & \mid (P \mid Q) \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ & & \mid (\nu x)\, P \mid *P \mid \boldsymbol{acc}_{\xi}(x).P \mid (\mathfrak{N}^{\Phi} x)P \\ v \text{ (values)} & ::= & x \mid \boldsymbol{true} \mid \boldsymbol{false} \end{array}$$

Here, x, y, and z range over a countably infinite set **Var** *of variables. ξ ranges over a set of labels called* access labels*. Φ, called a* trace set*, denotes a set of sequences of access labels that is prefix-closed. The prefixes (like (νx) and $(\mathfrak{N}^{\Phi} x)$) bind tighter than the parallel composition $\mid$.*

An access label specifies the kind of an access operation. Typical access labels that we are going to use in this paper are: I for initialization, R for read, W for write, and C for close.

Process $\mathbf{acc}_\xi(x).P$ accesses the resource x, and then behaves like P. We will often write $\mathbf{init}(x).P$, $\mathbf{read}(x).P$, $\mathbf{write}(x).P$, and $\mathbf{close}(x).P$ for $\mathbf{acc}_I(x).P$, $\mathbf{acc}_R(x).P$, $\mathbf{acc}_W(x).P$, $\mathbf{acc}_C(x).P$. Process $(\mathfrak{N}^\Phi x)P$ creates a new resource with the bound name x that should be accessed according to Φ, and then behaves like P. Φ specifies a set of acceptable sequences of operations that are allowed for the new resource x. For example, $(\mathfrak{N}^{(I(R+W)^*C)^\#} x)P$ creates a resource that should be first initialized, read or written an arbitrary number of times, and then closed. Here, $(S)^\#$ is the prefix closure of S, i.e., $\{s \mid ss' \in S\}$. We write ϵ for the empty sequence.

We often abbreviate a sequence $v_1, \ldots, v_n$ to $\widetilde{v}$, and write $\overline{x}\langle\widetilde{v}\rangle.\, P$ and $x(\widetilde{y}).\, P$ for $\overline{x}\langle v_1, \ldots, v_n\rangle.\, P$ and $x(y_1, \ldots, y_n).\, P$. We often omit trailing $\mathbf{0}$ and write $\overline{x}\langle\widetilde{v}\rangle$ and $\mathbf{acc}_\xi(x)$ for $\overline{x}\langle\widetilde{v}\rangle.\, \mathbf{0}$ and $\mathbf{acc}_\xi(x).\mathbf{0}$ respectively.

The bound and free variables of P are defined in a customary manner; also $(\mathfrak{N}^\Phi x)P$ binds x. We identify processes up to α-conversion, and assume that α-conversion is implicity applied so that bound variables are always different from each other and from free variables.

3.1.2 Operational Semantics

We now formally define the operational semantics of our process calculus The operational semantics is almost the same as the standard reduction semantics for the π-calculus, except that trace sets Φ (which represent how resources should be accessed in future) may change during reduction.

Definition 3.2 *The* structural preorder $\preceq$ *is the least reflexive and transitive relation closed under the rules in Figure 3.1 ($P \equiv Q$ stands for $(P \preceq Q) \wedge (Q \preceq P)$).*

Reminder 3.1 *As in our previous behavioural type systems for the π-calculus [23, 31, 33], the structural relation is asymmetric. If the standard, symmetric structural relation were used, the type preservation property would not hold: $\Gamma \triangleright *P \mid P : A$ does not necessarily imply $\Gamma \triangleright *P : A$) for the type system introduced in the next chapter.*

Definition 3.3 *The set of* reduction labels*, ranged over by L, is $\{x^\xi \mid x \in \mathbf{Var}\} \cup \{\tau\}$. We define target($L$) by:*

$$target(x^\xi) = \{x\} \qquad target(\tau) = \emptyset$$

Definition 3.4 *Let Φ be a set of sequences of access labels. $\Phi^{-\xi}$ is defined by: $\Phi^{-\xi} = \{s \mid \xi s \in \Phi\}$.*

$$P \mid \mathbf{0} \equiv P \qquad \text{(SP-Zero)}$$

$$P \mid Q \equiv Q \mid P \qquad \text{(SP-Commut)}$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \text{(SP-Assoc)}$$

$$*P \preceq *P \mid P \qquad \text{(SP-Rep)}$$

$$(\nu x)\, P \mid Q \preceq (\nu x)\, (P \mid Q) (\text{if } x \text{ not free in } Q) \qquad \text{(SP-New)}$$

$$(\mathfrak{N}^{\Phi} x) P \mid Q \preceq (\mathfrak{N}^{\Phi} x)(P \mid Q) (\text{if } x \text{ not free in } Q) \qquad \text{(SP-NewR)}$$

$$\frac{P \preceq P' \qquad Q \preceq Q'}{P \mid Q \preceq P' \mid Q'} \qquad \text{(SP-Par)}$$

$$\frac{P \preceq Q}{(\nu x)\, P \preceq (\nu x)\, Q} \qquad \text{(SP-CNew)}$$

$$\frac{P \preceq Q}{(\mathfrak{N}^{\Phi} x) P \preceq (\mathfrak{N}^{\Phi} x) Q} \qquad \text{(SP-CNewR)}$$

Figure 3.1: Structural Preorder

Definition 3.5 *The reduction relation $\xrightarrow{L}$ is the least relation closed under the rules in Figure 3.2.*

We write $P \longrightarrow Q$ when $P \xrightarrow{L} Q$ for some L. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

Notice that when an invalid access to a resource occurs (i.e. when the program accesses ξ but the specification Φ has no ξ-prefixes), then resource specification Φ is set to $\emptyset$ by (R-NewR1). On the other hand $\Phi \supseteq \{\epsilon\}$ indicates a resource that has been correctly used so far, and $\Phi = \{\epsilon\}$ indicates one that has been correctly and completely used.

Definition 3.6 *A process P is* resource-safe *if it does not contain a sub-expression of the form $(\mathfrak{N}^{\emptyset} x)Q$.*

We give a type system guaranteeing that any resource-safe, well-typed process cannot be reduced to a non-safe process (in other words, any resource-safe, well-typed process never performs an invalid access) in Section 3.2.

Example 3.1 *The following process first creates a resource x that should be first initialized, read an arbitrary number of times, and then closed. It then spawns four processes; they synchronize*

$$\overline{x}\langle\widetilde{z}\rangle.\,P \mid x(\widetilde{y}).\,Q \xrightarrow{\tau} P \mid [\widetilde{z}/\widetilde{y}]Q \qquad \text{(R-Com)}$$

$$\mathbf{acc}_{\xi}(x).P \xrightarrow{x^{\xi}} P \qquad \text{(R-Acc)}$$

$$\frac{P \xrightarrow{L} Q}{P \mid R \xrightarrow{L} Q \mid R} \qquad \text{(R-Par)}$$

$$\mathbf{if\ true\ then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} P \qquad \text{(R-IfT)}$$

$$\mathbf{if\ false\ then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} Q \qquad \text{(R-IfF)}$$

$$\frac{P \xrightarrow{L} Q \qquad x \notin target(L)}{(\nu x)\,P \xrightarrow{L} (\nu x)\,Q} \qquad \text{(R-New)}$$

$$\frac{P \xrightarrow{x^{\xi}} Q}{(\mathfrak{N}^{\Phi}x)P \xrightarrow{\tau} (\mathfrak{N}^{\Phi^{-\xi}}x)Q} \qquad \text{(R-NewR1)}$$

$$\frac{P \xrightarrow{L} Q \qquad x \notin target(L)}{(\mathfrak{N}^{\Phi}x)P \xrightarrow{L} (\mathfrak{N}^{\Phi}x)Q} \qquad \text{(R-NewR2)}$$

$$\frac{P \preceq P' \qquad P' \xrightarrow{L} Q' \qquad Q' \preceq Q}{P \xrightarrow{L} Q} \qquad \text{(R-SP)}$$

Figure 3.2: Reduction Relation

through channels c_1 and c_2, so that x is accessed in a valid order.

$$\begin{aligned}
&(\mathfrak{N}^{(IR^*C)^{\#}}x)(\nu c_1)\,(\nu c_2)\,\big(\\
&\quad \boldsymbol{init}(x).(\overline{c_1}\langle\,\rangle \mid \overline{c_1}\langle\,\rangle) && \text{/* initialize x, and send signals */}\\
&\quad \mid c_1(\,).\,\boldsymbol{read}(x).\overline{c_2}\langle\,\rangle && \text{/* wait for a signal on } c_1,\\
&\qquad\qquad && \text{\quad then read } x, \text{ and signal on } c_2\text{*/}\\
&\quad \mid c_1(\,).\,\boldsymbol{read}(x).\overline{c_2}\langle\,\rangle && \text{/* wait for a signal on } c_1,\\
&\qquad\qquad && \text{\quad then read } x, \text{ and signal on } c_2\text{*/}\\
&\quad \mid c_2(\,).\,c_2(\,).\,\boldsymbol{close}(x)\,\big) && \text{/* wait on } c_2, \text{ then close } x\ \text{*/}
\end{aligned}$$

□

Example 3.2 *The following program is prototypical of recursive functions. There is a replicated service which listens on channel s; it either terminates the recursion by sending a message back on the reply channel r, or it recursively invokes a sub-instance of itself which will reply on a private channel r'. In this example each recursive step does a $\boldsymbol{read}(x)$. The following program use an integer to decide whether or not to recurse. Though our language does not have integers and operations on them as primitives, it is trivial to extend our language and type system with those primitives.*

$$\begin{aligned}
&(\nu s)\,\big(\ *(s(n,x,r).\,\mathbf{if}\ n=0\ \mathbf{then}\ \overline{r}\langle\rangle\\
&\qquad\qquad \mathbf{else}\ (\nu r')\,(\overline{s}\langle n-1,x,r'\rangle \mid r'().\,\boldsymbol{read}(x).\overline{r}\langle\rangle)\\
&\qquad \mid\ (\mathfrak{N}^{(IR^*C)^{\#}}x)(\nu r)\,(\boldsymbol{init}(x).\overline{s}\langle 100,x,r\rangle \mid r(\,).\,\boldsymbol{close}(x))\,\big)
\end{aligned}$$

The above program corresponds to the following higher-level program:

```
init(x); parbegin read(x); read(x) parend; close(x)
```

Example 3.3 *Consider the following producer/consumer program:*[1]

$$
\begin{array}{l}
(\nu producer)\,(\nu consumer) \\
*(producer(b,p,c).\, p().\, \boldsymbol{acc}_{\mathbf{P}}(b).(\overline{c}\langle\rangle \mid \overline{producer}\langle b,p,c\rangle)) \mid \\
*(consumer(b,p,c).\, c().\, \boldsymbol{acc}_{\mathbf{G}}(b).(\overline{p}\langle\rangle \mid \overline{producer}\langle b,p,c\rangle)) \mid \\
(\mathfrak{N}^{((\mathbf{P}\ \mathbf{G})^*)^{\#}}\, buf)(\nu x)\,(\nu y) \\
\quad *(\overline{producer}\langle buf, x, y\rangle) \mid *(\overline{consumer}\langle buf, x, y\rangle) \mid \overline{x}\langle\rangle
\end{array}
$$

The first two processes $*(producer(b,p,c).\,\cdots)$ *and* $*(consumer(b,p,c).\,\cdots)$ *define the behavior of producers and consumers. A producer repeatedly waits to receive a signal on* p*, performs a put on the buffer* b *(by* $\boldsymbol{acc}_{\mathbf{P}}(b)$*), and then sends a signal on* c*. A consumer repeatedly waits to receive a signal on* c*, performs a get on the buffer* b *(by* $\boldsymbol{acc}_{\mathbf{P}}(b)$*), and then sends a signal on* p*. The third process creates a new buffer on which put and get should be applied only alternately, creates two channels* x *and* y *used for synchronization, and runs infinitely many producers and consumers.*

Reminder 3.2 *We treat resources as primitives in this paper, but we could alternatively express a resource as a tuple of channels, each of which corresponds to each access operation. For example, the resource in Example 3.1 can be expressed as a tuple consisting of three channels* ***init***, ***read***, *and* ***close****. If we did so, we could directly reuse the previous type systems [7, 23] to infer some of the properties discussed in this paper (with different precision). Treating resources as primitives, however, simplifies the type systems introduced in later chapters and clarifies the essence: if we expressed a resource as a tuple of channels, we would need primitives for simultaneous creation of multiple channels as in [23], and need to care about whether communications on the resource access channels succeed or not. On the other hand, our resource access primitives are non-blocking, which simplifies in particular the extended type system discussed in Section 3.4.*

3.2 Type System

This section introduces a type system that prevents invalid access to resources. The type system in this section does not guarantee a liveness property that all the necessary accesses are eventually made; extensions to guarantee that property are discussed in Section 3.4.

[1] This is an example taken from an ealier version of [51] and modified.

3.2.1 Types

We first introduce the syntax of types. We use two categories of types: value types and behavioral types. The latter describes how a process accesses resources and communicates through channels. As mentioned in Section 1.2.2, we use CCS processes for behavioral types.

Definition 3.7 (types) *The sets of* value types σ *and* behavioral types A *are defined by:*

$$\begin{array}{rcl} \sigma & ::= & \boldsymbol{bool} \mid \boldsymbol{res} \mid \boldsymbol{chan}\langle(x_1:\sigma_1,\ldots,x_n:\sigma_n)A\rangle \\ A & ::= & \boldsymbol{0} \mid \alpha \mid a.A \mid x^{\xi}.A \mid \tau.A \mid (A_1 \mid A_2) \mid A_1 \oplus A_2 \mid *A \\ & & \mid \langle y_1/x_1,\ldots,y_n/x_n\rangle A \mid (\nu x)\,A \mid \mu\alpha.A \mid A\uparrow_S \mid A\downarrow_S \\ a \text{ (communication labels)} & ::= & x \mid \overline{x} \end{array}$$

A behavioral type A, which is a CCS process, describes what kind of communication and resource access a process may perform. $\mathbf{0}$ describes a process that performs no communication or resource access. The types $x.\,A$, $\overline{x}.\,A$, $x^{\xi}.A$ and $\tau.A$ describe processes that first perform an action and then behave according to A; the actions are, respectively, an input on x, an output on x, an access operation ξ on x, and the invisible action. $A_1 \mid A_2$ describes a process that performs communications and resource access according to A_1 and A_2, possibly in parallel. $A_1 \oplus A_2$ describes a process that behaves according to either A_1 or A_2. $*A$ describes a process that behaves like A an arbitrary number of times, possibly in parallel. $\langle y_1/x_1,\ldots,y_n/x_n\rangle A$, abbreviated to $\langle\widetilde{y}/\widetilde{x}\rangle A$, denotes simultaneous renaming of $\widetilde{x}$ with $\widetilde{y}$ in A. $(\nu x)\,A$ describes a process that behaves like A for some hidden channel x. For example, $(\nu x)\,(x.\,\overline{y} \mid \overline{x})$ describes a process that performs an output on y after the invisible action on x. The type $\mu\alpha.A$ describes a process that behaves like a recursive process defined by $\alpha \stackrel{\triangle}{=} A$.[2] The type $A\uparrow_S$ describes a process that behaves like A, except that actions whose targets are in S are replaced by the invisible action τ, while $A\downarrow_S$ describes a process that behaves like A, except that actions whose targets are not in S are replaced by τ. The formal semantics of behavioral types is defined later using labeled transition semantics.

As for value types, **bool** is the type of booleans. **res** is the type of resources. The type $\mathbf{chan}\langle(x_1:\sigma_1,\ldots,x_n:\sigma_n)A\rangle$, abbreviated to $\mathbf{chan}\langle(\widetilde{x}:\widetilde{\sigma})A\rangle$, describes channels carrying tuples consisting of values of types $\sigma_1,\ldots,\sigma_n$. Here the type A approximates how a receiver on the channel may use the elements $x_1,\ldots,x_n$ of each tuple for communications and resource access. For example, $\mathbf{chan}\langle(x:\mathbf{res},y:\mathbf{res})x^R.y^C\rangle$ describes channels carrying a pair of resources, where a party who receives the actual pair (x',y') will first read x' and then close y'. We sometimes omit $\widetilde{\sigma}$ and write $\mathbf{chan}\langle(\widetilde{x})A\rangle$ for $\mathbf{chan}\langle(\widetilde{x}:\widetilde{\sigma})A\rangle$. When $\widetilde{x}$ is empty, we also write $\mathbf{chan}\langle\rangle$.

[2]The replication $*A$ and $\mu\alpha.(A \mid \alpha)$ have the same semantics in this section, but they are differentiated in Section 3.4 by the predicate *disabled*.

Note that $\langle \widetilde{y}/\widetilde{x} \rangle$ is treated as a constructor rather than an operator for performing the actual substitution. We write $[\widetilde{y}/\widetilde{x}]$ for the latter throughout this paper. $\langle \widetilde{y}/\widetilde{x} \rangle A$ is slightly different from the *relabeling* of the standard CCS [44]: $\langle y/x \rangle (x \,|\, \overline{y})$ allows the communication on y, but the relabeling of CCS does not. This difference calls for the introduction of a special transition label $\{x, \overline{y}\}$ in Section 3.2.2.

Definition 3.8 *The set of* free variables *of A, written $\mathbf{FV}(A)$, is defined by:*

$$\begin{array}{rcl}
\mathbf{FV}(\boldsymbol{0}) & = & \emptyset \\
\mathbf{FV}(\alpha) & = & \emptyset \\
\mathbf{FV}(x.\, A) & = & \{x\} \cup \mathbf{FV}(A) \\
\mathbf{FV}(\overline{x}.\, A) & = & \{x\} \cup \mathbf{FV}(A) \\
\mathbf{FV}(x^{\xi}.A) & = & \{x\} \cup \mathbf{FV}(A) \\
\mathbf{FV}(\tau.A) & = & \mathbf{FV}(A) \\
\mathbf{FV}(A_1 \,|\, A_2) & = & \mathbf{FV}(A_1) \cup \mathbf{FV}(A_2) \\
\mathbf{FV}(A_1 \oplus A_2) & = & \mathbf{FV}(A_1) \cup \mathbf{FV}(A_2) \\
\mathbf{FV}(*A) & = & \mathbf{FV}(A) \\
\mathbf{FV}(\langle \widetilde{y}/\widetilde{x} \rangle A) & = & (\mathbf{FV}(A) \backslash \{\widetilde{x}\}) \cup \{\widetilde{y}\} \\
\mathbf{FV}((\nu x)\, A) & = & \mathbf{FV}(A) \backslash \{x\} \\
\mathbf{FV}(\mu\alpha.A) & = & \mathbf{FV}(A) \\
\mathbf{FV}(A\!\uparrow_S) & = & \mathbf{FV}(A) \backslash S \\
\mathbf{FV}(A\!\downarrow_S) & = & \mathbf{FV}(A) \cap S
\end{array}$$

As defined above, $(\nu x)\, A$, $\langle \widetilde{y}/\widetilde{x} \rangle A$, and $A\!\uparrow_S$ bind x, $\widetilde{x}$, and the variables in S respectively. We identify behavioral types up to renaming of bound variables. In the rest of this paper, we require that every channel type $\mathbf{chan}\langle (x_1 : \sigma_1, \ldots, x_n : \sigma_n) A \rangle$ must satisfy $\mathbf{FV}(A) \subseteq \{x_1, \ldots, x_n\}$. For example, $\mathbf{chan}\langle (x{:}\mathbf{res}) x^R \rangle$ is a valid type but $\mathbf{chan}\langle (x{:}\mathbf{res}) y^R \rangle$ is not.[3]

3.2.2 Semantics of behavioral types

We give a labeled transition relation $\xrightarrow{l}$ for behavioral types. The transition labels l (distinct from the reduction labels L of Definition 3.3) are

$$l \;::=\; x \mid \overline{x} \mid x^{\xi} \mid \tau \mid \{x, \overline{y}\}$$

The label $\{x, \overline{y}\}$ indicates the potential to react in the presence of a substitution that identifies x and y. We also extend *target* to the function on transition labels by:

$$\mathit{target}(x) = \mathit{target}(\overline{x}) = \{x\} \qquad \mathit{target}(\{x, \overline{y}\}) = \{x, y\}$$

[3] This constraint can be removed if we assume that the free variables in $\mathit{codom}(\Gamma)$ never clash with the bound variables of P in the judgment form $\Gamma \triangleright P : A$ given later. In particular, we need an implicit assumption $\{\widetilde{y}\} \cap \mathbf{FV}(\Gamma) = \emptyset$ in Figure 3.4, (T-In).

$$a.A \xrightarrow{a} A \qquad x^{\xi}.A \xrightarrow{x^{\xi}} A \qquad \tau.A \xrightarrow{\tau} A \tag{Tr-Act}$$

$$\frac{A_1 \xrightarrow{l} A_1'}{A_1|A_2 \xrightarrow{l} A_1'|A_2} \qquad \frac{A_2 \xrightarrow{l} A_2'}{A_1|A_2 \xrightarrow{l} A_1|A_2'} \tag{Tr-Par1}$$

$$\frac{A_1 \xrightarrow{x} A_1' \quad A_2 \xrightarrow{\overline{y}} A_2'}{A_1|A_2 \xrightarrow{\{x,\overline{y}\}} A_1'|A_2'} \qquad \frac{A_1 \xrightarrow{\overline{y}} A_1' \quad A_2 \xrightarrow{x} A_2'}{A_1|A_2 \xrightarrow{\{x,\overline{y}\}} A_1'|A_2'} \tag{Tr-Par2}$$

$$\frac{A \xrightarrow{\{x,\overline{x}\}} A'}{A \xrightarrow{\tau} A'} \tag{Tr-Com}$$

$$\frac{A_1 \xrightarrow{l} A_1'}{A_1 \oplus A_2 \xrightarrow{l} A_1'} \qquad \frac{A_2 \xrightarrow{l} A_2'}{A_1 \oplus A_2 \xrightarrow{l} A_2'} \tag{Tr-Or}$$

$$\frac{A \,|\, *A \xrightarrow{l} A'}{*A \xrightarrow{l} A'} \tag{Tr-Rep}$$

$$\frac{[\mu\alpha.A/\alpha]A \xrightarrow{l} A'}{\mu\alpha.A \xrightarrow{l} A'} \tag{Tr-Rec}$$

$$\frac{A \xrightarrow{l} A'}{\langle \widetilde{y}/\widetilde{x}\rangle A \xrightarrow{[\widetilde{y}/\widetilde{x}]l} \langle \widetilde{y}/\widetilde{x}\rangle A'} \tag{Tr-Rename}$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap \{x\} = \emptyset}{(\nu x)\, A \xrightarrow{l} (\nu x)\, A'} \tag{Tr-Hiding}$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A\!\uparrow_S \xrightarrow{\tau} A'\!\uparrow_S} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A\!\uparrow_S \xrightarrow{l} A'\!\uparrow_S} \tag{Tr-Exclude}$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A\!\downarrow_S \xrightarrow{l} A'\!\downarrow_S} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A\!\downarrow_S \xrightarrow{\tau} A'\!\downarrow_S} \tag{Tr-Project}$$

Figure 3.3: Transition semantics of behavioral types

The transition relation $\xrightarrow{l}$ on behavioral types is the least relation closed under the rules in Figure 3.3. We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$. We also write $\overset{l}{\Longrightarrow}$ for $\Longrightarrow \xrightarrow{l} \Longrightarrow$.

Reminder 3.3 *$(\nu x)\, A$ should not be confused with $A\!\uparrow_{\{x\}}$. $(\nu x)\, A$ is the hiding operator of CCS, while $A\!\uparrow_{\{x\}}$ just replaces any actions on x with τ [23]. For example, $(\nu x)\,(x.\, y^{\xi})$ cannot make any transition, but $(x.\, y^{\xi})\!\uparrow_{\{x\}} \xrightarrow{\tau} \xrightarrow{y^{\xi}} \boldsymbol{0}\!\uparrow_{\{x\}}$.*

The set $\mathbf{traces}_x(A)$ defined below is the set of possible access sequences on x described by A.

Definition 3.9 (traces)

$$\boldsymbol{traces}_x(A) = \{\xi_1 \ldots \xi_n \mid A{\downarrow_{\{x\}}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} A'\}$$

Note that $\mathbf{traces}_x(A)$ is prefix-closed (hence a trace set) by definition.

We define the subtyping relation $A_1 \leq A_2$ below. Intuitively, $A_1 \leq A_2$ means that a process behaving according to A_1 can also be viewed as a process behaving according to A_2. To put in another way, $A_1 \leq A_2$ means that A_2 simulates A_1.We define $\leq$ for only *closed* types, i.e., those not containing free type variables.

Definition 3.10 (subtyping) *The subtyping relation $\leq$ on closed behavioral types is the largest relation such that $A_1 \leq A_2$ and $A_1 \stackrel{l}{\longrightarrow} A_1'$ implies $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_1' \leq A_2'$ for some A_2'.*

We often write $A_1 \geq A_2$ for $A_2 \leq A_1$, and write $A_1 \approx A_2$ for $A_1 \leq A_2 \wedge A_2 \leq A_1$.

Reminder 3.4 *Note that the subtyping relation defined here is the converse of the one used in Igarashi and Kobayashi's generic type system [23]. This is due to two different, dual views on behavioral types. Here, we think of behavioral types as describing the behavior of processes. On the other hand, Igarashi and Kobayashi [23] think of behavioral types as describing the assumption on the environment about what kind of process is accepted by the environment. Because of this difference, they write behavioral types on the lefthand side of $\triangleright$, and write $A_1 \& A_2$ for non-deterministic choice instead of $A_1 \oplus A_2$.*

Reminder 3.5 *Depending on what property the type system should guarantee, a finer subtyping relation may need to be chosen. For example, the above definition allows $(x^W.\boldsymbol{0}) \mid (x^W.\boldsymbol{0}) \leq x^W.x^W.\boldsymbol{0}$. We may want to disallow this relation if we want to infer a property like "no simultaneous writes on x can occur."*

The following properties are satisfied by $\leq$. For proofs, see Appendix C.

Lemma 3.1

1. *$\leq$ is a precongruence, i.e., $\leq$ is closed under any behavioral type constructor.*
2. *If $A_1 \leq A_2$, then $\boldsymbol{traces}_x(A_1) \subseteq \boldsymbol{traces}_x(A_2)$ for any x.*
3. *$B_1 \oplus B_2 \leq A$ if and only if $B_1 \leq A$ and $B_2 \leq A$.*
4. *If $[B/\alpha]A \leq B$, then $\mu\alpha.A \leq B$.*

3.2.3 Typing

We consider two kinds of judgments, $\Gamma \triangleright v : \sigma$ for values, and $\Gamma \triangleright P : A$ for processes. Γ is a mapping from a finite set of variables to value types. In $\Gamma \triangleright P : A$, the type environment Γ describes the types of the variables, and A describes the possible behaviors of P. For example, $x : \mathbf{chan}\langle(b : \mathbf{bool})\mathbf{0}\rangle \triangleright P : \overline{x} \,|\, \overline{x}$ implies that P may send booleans along the channel x twice. The judgment $y : \mathbf{chan}\langle(x : \mathbf{chan}\langle(b : \mathbf{bool})\mathbf{0}\rangle)\overline{x}\rangle \triangleright Q : y$ means that Q may perform an input on y once, and then it may send a boolean on the received value. Note that in the judgment $\Gamma \triangleright P : A$, the type A is an approximation of the behavior of P on free channels. P may do less than what is specified by A, but must not do more; for example, $x : \mathbf{chan}\langle(\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle : \overline{x} \,|\, \overline{x}$ holds but $x : \mathbf{chan}\langle(\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle.\,\overline{x}\langle\,\rangle : \overline{x}$ does not. Because of this invariant, if A does not perform any invalid access, neither does P.

We write $dom(\Gamma)$ for the domain of Γ. We write $\emptyset$ for the empty type environment, and write $x_1 : \tau_1, \ldots, x_n : \tau_n$ (where $x_1, \ldots, x_n$ are distinct from each other) for the type environment Γ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, and $\Delta(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. We define the *value judgment* relation $\Gamma \triangleright v{:}\sigma$ to be the least relation closed under

$$\Gamma,\, x{:}\sigma \triangleright x{:}\sigma \qquad \Gamma \triangleright \mathbf{true}{:}\mathbf{bool} \qquad \Gamma \triangleright \mathbf{false}{:}\mathbf{bool}.$$

We write $\Gamma \triangleright \widetilde{v}{:}\widetilde{\sigma}$ as an abbreviation for $(\Gamma \triangleright v_1{:}\sigma_1) \wedge \cdots \wedge (\Gamma \triangleright v_n{:}\sigma_n)$.

Definition 3.11 *The type judgment relation $\Gamma \triangleright P : A$ is the least relation closed under the rules given in Figure 3.4.*

We explain key rules below.

In rule (T-Out), the first premise $\Gamma \triangleright P : A_2$ implies that the continuation of the output process behaves like A_2, and the second premise $\Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$ implies that the tuple of values $\widetilde{v}$ being sent may be used by an input process according to $\langle\widetilde{v}/\widetilde{y}\rangle A_1$. Therefore, the whole behavior of the output process is described by $\overline{x}.\,(\langle\widetilde{v}/\widetilde{y}\rangle A_1 \,|\, A_2)$. Here, $\langle v_1/x_1, \ldots, v_n/x_n\rangle A$ stands for $\langle v_{i_1}/x_{i_1}, \ldots, v_{i_k}/x_{i_k}\rangle A$ where $\{v_{i_1}, \ldots, v_{i_k}\} = \{v_1, \ldots, v_n\} \backslash \{\mathbf{true}, \mathbf{false}\}$. For example,$\langle\mathbf{true}/x, y/z\rangle A$ stands for $\langle y/z\rangle A$. Note that, as in previous behavioral type systems [7, 23], the resource access and communications made on $\widetilde{v}$ by the receiver of $\widetilde{v}$ are counted as the behavior of the output process (see Remark 3.7).

In rule (T-In), the first premise implies that the continuation of the input process behaves like A_2. Following previous behavioral type systems [7, 23], we split A_2 into two parts: $A_2{\downarrow_{\{\widetilde{y}\}}}$ and $A_2{\uparrow_{\{\widetilde{y}\}}}$. The first part describes the behavior on the received values $\widetilde{y}$ and is taken into account in the channel type. The second part describes the resource access and communications

performed on other values, and is taken into account in the behavioral type of the input process. The condition $A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1$ requires that the access and communication behavior on $\widetilde{y}$ conforms to A_1, the channel arguments' behavior.

In (T-NEW), the premise implies that P behaves like A, so that $(\nu x)\, P$ behaves like $(\nu x)\, A$. Here, we only require that x is a channel, unlike in the previous behavioral type systems for the π-calculus [23, 33]. That is because we are only interested in the resource access behavior; the communication behavior is used only for accurately inferring the resource access behavior.

In (T-NEWR), we check that the process's behavior A conforms to the resource usage specification Φ.

Rule (T-SUB) allows the type A' of a process to be replaced by its approximation A.

We remark that weakening of Γ can be derived (Appendix D, Lemma D.1) and so is not needed as a rule.

The following example shows how information about the usage of resources by an input process is propagated to an output process.

Example 3.4 *Let us consider* $(\mathfrak{N}^{\Phi} x)P$*, where*

$$\Phi = (R^* C)^{\#}$$
$$P = (\nu y)\,(\overline{y}\langle x, x\rangle \mid y(z_1, z_2).\, \mathbf{read}(z_1).\mathbf{close}(z_2)).$$

Let $\Gamma = y : \mathbf{chan}\langle (z_1, z_2) z_1^R.z_2^C \rangle, x : \mathbf{res}$*. Then, the following judgment holds for the output and input processes.*

$$\Gamma \triangleright \overline{y}\langle x, x\rangle : \overline{y}.\, x^R.x^C$$
$$\Gamma \triangleright y(z_1, z_2).\, \mathbf{read}(z_1).\mathbf{close}(z_2) : y.\, \mathbf{0}$$

Here, we have used subtyping relations $\langle x/z_1, x/z_2\rangle z_1^R.z_2^C \approx x^R.x^C$ *and* $z_1^R.z_2^C{\uparrow_{\{z_1,z_2\}}} \approx \mathbf{0}$*. By using (*T-PAR*) and (*T-NEW*), we obtain*

$$x : \mathbf{res} \triangleright P : (\nu y)\,(\overline{y}.\, x^R.x^C \mid y)$$

*Using (*T-SUB*) with* $(\nu y)\,(\overline{y}.\, x^R.x^C \mid y) \approx x^R.x^C$ *we get*

$$x : \mathbf{res} \triangleright P : x^R.x^C$$

Since $\mathbf{traces}_x(x^R.x^C)) \subseteq (R^* C)^{\#}$*, we obtain* $\emptyset \triangleright (\mathfrak{N}^{\Phi} x)P : \mathbf{0}$ *by using (*T-NEWR*) and (*T-SUB*).* □

$$\Gamma \triangleright \mathbf{0} : \mathbf{0} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad \Gamma \triangleright \widetilde{v} : \widetilde{\sigma}}{\Gamma \triangleright \overline{x}\langle\widetilde{v}\rangle.\, P : \overline{x}.\, (\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)} \qquad \text{(T-Out)}$$

$$\frac{\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1}{\Gamma \triangleright x(\widetilde{y}).\, P : x.\, (A_2{\uparrow_{\{\widetilde{y}\}}})} \qquad \text{(T-In)}$$

$$\frac{\Gamma \triangleright P_1 : A_1 \qquad \Gamma \triangleright P_2 : A_2}{\Gamma \triangleright P_1 \mid P_2 : A_1 \mid A_2} \qquad \text{(T-Par)}$$

$$\frac{\Gamma \triangleright P : A}{\Gamma \triangleright *P : *A} \qquad \text{(T-Rep)}$$

$$\frac{\Gamma \triangleright v : \mathbf{bool} \qquad \Gamma \triangleright P : A \qquad \Gamma \triangleright Q : A}{\Gamma \triangleright \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : A} \qquad \text{(T-If)}$$

$$\frac{\Gamma, x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \triangleright P : A_2}{\Gamma \triangleright (\nu x)\, P : (\nu x)\, A_2} \qquad \text{(T-New)}$$

$$\frac{\Gamma \triangleright P : A \qquad \Gamma \triangleright x : \mathbf{res}}{\Gamma \triangleright \mathbf{acc}_\xi(x).P : x^\xi.A} \qquad \text{(T-Acc)}$$

$$\frac{\Gamma, x : \mathbf{res} \triangleright P : A \qquad \mathbf{traces}_x(A) \subseteq \Phi}{\Gamma \triangleright (\mathfrak{N}^\Phi x)P : A{\uparrow_{\{x\}}}} \qquad \text{(T-NewR)}$$

$$\frac{\Gamma \triangleright P : A' \qquad A' \leq A}{\Gamma \triangleright P : A} \qquad \text{(T-Sub)}$$

Figure 3.4: Typing Rules

Example 3.5 *Recall Example 3.2:*

$$\begin{aligned}
P &= (\nu s)\,(*s(n, x, r).\, P_1 \mid (\mathfrak{N}^\Phi x)P_2) \\
P_1 &= \mathbf{if}\ n = 0\ \mathbf{then}\ \overline{r}\langle\rangle \\
&\quad\ \mathbf{else}\ (\nu r')\,(\overline{s}\langle n - 1, x, r'\rangle \mid r'().\, \textbf{\textit{read}}(x).\overline{r}\langle\rangle) \\
P_2 &= (\nu r)\,(\textbf{\textit{init}}(x).\overline{s}\langle 100, x, r\rangle \mid r().\, \textbf{\textit{close}}(x)) \\
\Phi &= (IR^*C)^\#
\end{aligned}$$

Let $A_1 = \mu\alpha.(\overline{r} \oplus (\nu r')\,(\langle r'/r\rangle\alpha | r'.\, x^R.\overline{r})$ *and let* $\Gamma = s{:}\boldsymbol{chan}\langle(n{:}\boldsymbol{int},\, x{:}\boldsymbol{res},\, r{:}\boldsymbol{chan}\langle\rangle)\, A_1\rangle$. *Then*

$$\Gamma, n{:}\boldsymbol{int}, x{:}\boldsymbol{res}, r{:}\boldsymbol{chan}\langle\rangle \triangleright P_1 : A_1$$
$$\Gamma \triangleright *s(n,x,r).\, P_1 : *s.\, (A_1{\uparrow_{\{n,x,r\}}}) \approx *s$$
$$\Gamma \triangleright P_2 : (\nu r)\,(x^I.A_1 | r.\, x^C)$$

So long as $\boldsymbol{traces}_x((\nu r)\,(x^I.A_1|r.\, x^C)) \subseteq \Phi$, *we obtain* $\emptyset \triangleright P : \boldsymbol{0}$. *See Section 3.3.3 for the algorithm that establishes* $\boldsymbol{traces}_x(\cdot) \subseteq \Phi$. □

Reminder 3.6 *The type* A_1 *in the example above demonstrates how recursion, hiding, and renaming are used together. In general, in order to type a recursive process of the form* $*s(x).\,(\nu y)\,(\cdots \overline{s}\langle y\rangle \cdots)$, *we need to find a type that satisfies* $(\nu y)\,(\cdots \langle y/x\rangle A \cdots) \leq A$. *Moreover, for the type inference (in Section 3.3), we must find the* least *such* A. *Thanks to the type constructors for recursion, hiding, and renaming, we can always do that:* A *can be expressed by* $\mu\alpha.(\nu y)\,(\cdots \langle y/x\rangle\alpha \cdots)$ *(recall Lemma 3.1.4).*

Reminder 3.7 *A reader may wonder why the rules (*T-Out*) and (*T-In*) are asymmetric, in the sense that information about the continuation of a receiver process is transferred to a sender process but not vice versa. That design choice comes from the observation that a channel or resource exchanged between a sender and a receiver are, in general, statically known only to the sender, so that we have to put information about the behavior on the channel or resource into the type of the sender. For example, consider the process* $((\nu y)\,(\overline{x}\langle y\rangle \mid \cdots) \mid x(z).\,\overline{z}\langle\,\rangle$. *Since the receiver* $x(z).\,\overline{z}\langle\,\rangle$ *is not in the scope of* y, *we have to put the information that* y *will be used for output into the type of the sender* $\overline{x}\langle y\rangle$ *(as* $\overline{x}.\,\overline{y}$*). It is still useful and possible to recover the symmetry in the treatment of senders and receivers to some extent: see Section 8 of our previous paper [23].*

The following theorem states that no well-typed process performs an invalid access to a resource.

Theorem 3.1 (type soundness (safety)) *Suppose that* P *is safe. If* $\Gamma \triangleright P : A$ *and* $P \longrightarrow^* Q$, *then* Q *is safe.*

Proof 3.1 *We make use of the following lemma:*

- ***Subject-reduction.*** *If* $P \xrightarrow{L} P'$ *and* $\Gamma \triangleright P : A$ *then* $A \xRightarrow{L} A'$ *and* $\Gamma \triangleright P' : A'$.
  Proof: see Appendix D.

For the proof of the theorem, we focus on just a single reduction step. By the Lemma we know that judgements are preserved by reduction; we must show that safety is also preserved, by

*induction on the derivation of reduction. The only interesting case is (*R-NewR1*),* $(\mathfrak{N}^{\Phi}x)P \xrightarrow{\tau} (\mathfrak{N}^{\Phi^{-\xi}}x)P'$*, since the other rules do not alter trace-sets* Φ*. In this case, we are given* $\Gamma \triangleright P : A$*,* $\boldsymbol{traces}_x(A) \subseteq \Phi$*, and* $P \xrightarrow{x^{\xi}} P'$*. By the Lemma,* $A \xRightarrow{x^{\xi}} A'$ *for some* $\Gamma \triangleright P' : A'$*. Assume* $(\mathfrak{N}^{\Phi}x)P$ *is safe; hence so is* P*; by the induction hypothesis so is* P'*. From the conditions* $\boldsymbol{traces}_x(A) \subseteq \Phi$ *and* $A \xRightarrow{x^{\xi}} A'$*, we get* $\xi \in \boldsymbol{traces}_x(A) \subseteq \Phi$*, so that* $\epsilon \in \Phi^{-\xi} \neq \emptyset$*. So,* $(\mathfrak{N}^{\Phi^{-\xi}}x)P'$ *is safe.*

3.3 Type Inference Algorithm

This chapter discusses an algorithm which takes a closed process P as an input and checks whether $\emptyset \triangleright P : \mathbf{0}$ holds. As in similar type systems [24, 33], the algorithm consists of the following steps.

1. Extract constraints on type variables based on the (syntax-directed version of) typing rules.
2. Reduce constraints to trace inclusion constraints of the form $\{\mathbf{traces}_{x_1}(A_1) \subseteq \Phi_1, \ldots, \mathbf{traces}_{x_n}(A_n) \subseteq \Phi_n\}$
3. Decide whether the constraints are satisfied.

The algorithm for Step 3 is sound but not complete.

We give an overview of each step below. The first two steps are almost the same as those in the previous work.

3.3.1 Step 1: Extracting Constraints

The typing rules presented in Section 3.2 can be transformed to the syntax-directed typing rules shown in Figure 3.5. In the figure, $\Gamma_1 \cup \Gamma_2$ is the type environment obtained by merging both bindings, and defined only if $\Gamma_1(x) = \Gamma_2(x)$ for every $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$. Type equality here is syntactic equality up to α-renaming. And $wd(\Gamma_1 \cup \Gamma_2)$ means that $\Gamma_1 \cup \Gamma_2$ is well-defined. The two sets of typing rules are equivalent in the following sense: If $\Gamma \triangleright P : A$ is derivable, then there exists A' such that $A' \leq A$ holds and $\Gamma \triangleright_{sd} P : A'$ is derivable. Conversely, if $\Gamma \triangleright_{sd} P : A$ is derivable, so is $\Gamma \triangleright P : A$.

Based on the syntax-directed rules, we obtain the algorithm in Figure 3.6, which takes a process P and outputs a triple consisting of a type environment Γ, a behavioral type A, and a set C of constraints. In Figure 3.6, $\Gamma_1 \otimes \cdots \otimes \Gamma_n$ is defined to be (Γ, C) where Γ and C are given by:

$$dom(\Gamma) = dom(\Gamma_1) \cup \cdots \cup dom(\Gamma_n)$$
$$\Gamma(x) = \Gamma_i(x) \text{ where } x \in dom(\Gamma_i) \backslash (dom(\Gamma_1) \cup \cdots \cup dom(\Gamma_{i-1}))$$
$$C = \{\Gamma_i(x) = \Gamma_j(x) \mid x \in dom(\Gamma_i) \cap dom(\Gamma_j)\}$$

$$\emptyset \triangleright_{sd} \mathbf{0} : \mathbf{0} \qquad \text{(T-SD-Zero)}$$

$$\frac{\Gamma_0 \triangleright_{sd} P : A_2 \qquad \Gamma_i \triangleright v_i : \sigma_i \text{ (for each } i \in \{1, \ldots, n\})}{\Gamma_0 \cup \widetilde{\Gamma} \cup (x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle) \triangleright_{sd} \overline{x}\langle\widetilde{v}\rangle.\, P : \overline{x}.\, (\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)} \qquad \text{(T-SD-Out)}$$

$$\frac{\Gamma \triangleright_{sd} P : A_2 \qquad A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1 \qquad wd(\Gamma \cup \widetilde{y} : \widetilde{\sigma})}{(\Gamma \backslash \{\widetilde{y}\}) \cup x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \triangleright_{sd} x(\widetilde{y}).\, P : x.\, A_2{\uparrow_{\{\widetilde{y}\}}}} \qquad \text{(T-SD-In)}$$

$$\frac{\Gamma_1 \triangleright_{sd} P_1 : A_1 \qquad \Gamma_2 \triangleright_{sd} P_2 : A_2}{\Gamma_1 \cup \Gamma_2 \triangleright_{sd} P_1 \mid P_2 : A_1 \mid A_2} \qquad \text{(T-SD-Par)}$$

$$\frac{\Gamma \triangleright_{sd} P : A}{\Gamma \triangleright_{sd} *P : *A} \qquad \text{(T-SD-Rep)}$$

$$\frac{\Gamma_0 \triangleright v : \mathbf{bool} \qquad \Gamma_1 \triangleright_{sd} P : A_1 \qquad \Gamma_2 \triangleright_{sd} Q : A_2 \qquad A_1 \leq A \qquad A_2 \leq A}{\Gamma_0 \cup \Gamma_1 \cup \Gamma_2 \triangleright_{sd} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : A} \qquad \text{(T-SD-If)}$$

$$\frac{\Gamma \triangleright_{sd} P : A_2 \qquad wd(\Gamma \cup (x : \mathbf{chan}\langle(\widetilde{x} : \widetilde{\tau})A_1\rangle))}{\Gamma \backslash \{x\} \triangleright_{sd} (\nu x)\, P : (\nu x)\, A_2} \qquad \text{(T-SD-New)}$$

$$\frac{\Gamma \triangleright_{sd} P : A}{\Gamma \cup (x : \mathbf{res}) \triangleright_{sd} \mathbf{acc}_{\xi}(x).P : x^{\xi}.A} \qquad \text{(T-SD-Acc)}$$

$$\frac{\Gamma \triangleright_{sd} P : A \qquad \mathbf{traces}_x(A) \subseteq \Phi \qquad wd(\Gamma \cup (x : \mathbf{res}))}{\Gamma \backslash \{x\} \triangleright_{sd} (\mathfrak{N}^{\Phi} x)P : A'{\uparrow_{\{x\}}}} \qquad \text{(T-SD-NewR)}$$

Figure 3.5: Syntax Directed Typing Rules

The triple (Γ, A, C) output by PT satisfies the following properties:

- $\theta\Gamma \triangleright P : \theta A$ holds for any substitution θ such that $\models \theta C$.
- If $\Gamma' \triangleright P : A'$, then there exists a substitution θ such that $\theta\Gamma \subseteq \Gamma'$ and $\theta A \leq A'$.

Here, Γ and A may contain variables representing unknown behavioral types and value types. C is a set of constraints on them, and the substitution θ above replaces them with closed behavioral types and value types. Intuitively, the triple (Γ, A, C) expresses a set of type judgments for P. The first property above says that the triple contains only valid judgments, while the second property says that every valid judgment is subsumed by the triple.

We do not give a formal proof of the above properties; As usual, they can be proved by induction on the structure of P.

$$
\begin{array}{l}
PTv(x) = (x : \rho, \rho) \quad (\text{where } \rho \text{ fresh}) \\
PTv(b) = (\emptyset, \mathbf{bool}) \quad \text{if } b \in \{\mathbf{true}, \mathbf{false}\}
\end{array}
$$

$$
\begin{array}{lll}
PT(\mathbf{0}) = (\emptyset, \mathbf{0}, \emptyset) & & \\
PT(\overline{x}\langle\widetilde{v}\rangle.\, P_0) = & \mathbf{let} & (\Gamma_i, \sigma_i) = PTv(v_i) \\
 & & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma, C) = \Gamma_0 \otimes (x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})\alpha\rangle) \otimes \Gamma_1 \otimes \cdots \otimes \Gamma_n \\
 & \mathbf{in} & (\Gamma, \overline{x}.\, ([\widetilde{v}/\widetilde{y}]\alpha \mid A_0), C) \quad (\text{where } \alpha \text{ fresh}) \\
PT(x(\widetilde{y}).\, P_0) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma_1, C_1) = \Gamma_0 \otimes (x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\rho})\alpha\rangle) \otimes (\widetilde{y} : \widetilde{\rho}) \\
 & \mathbf{in} & (\Gamma \backslash \widetilde{y}, x.\, A_0 \uparrow_{\{\widetilde{y}\}}, C_0 \cup C_1 \cup \{\alpha \geq A_0 \downarrow_{\{\widetilde{y}\}}\}) \quad (\text{where } \alpha,\ \widetilde{\rho} \text{ fresh}) \\
PT(P_0 \mid P_1) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma_1, A_1, C_1) = PT(P_1) \\
 & & (\Gamma_2, C_2) = \Gamma_0 \otimes \Gamma_1 \\
 & \mathbf{in} & (\Gamma_2, A_0 \mid A_1, C_0 \cup C_1 \cup C_2) \\
PT(\mathbf{if}\ v\ \mathbf{then}\ P_0\ \mathbf{else}\ P_1\ ) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma_1, A_1, C_1) = PT(P_1) \\
 & & (\Gamma_2, \sigma) = PTv(v) \\
 & & (\Gamma, C_2) = \Gamma_0 \otimes \Gamma_1 \otimes \Gamma_2 \\
 & & \mathbf{in}\ (\Gamma, A_0 \oplus A_1, C_0 \cup C_1 \cup C_2 \cup \{\sigma = \mathbf{bool}\}) \\
PT((\nu x)\, P_0) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & C_1 = \mathbf{if}\ x \in dom(\Gamma_0) \mathbf{then}\ \{\mathbf{isChan}(\Gamma_0(x))\} \mathbf{else}\ \emptyset \\
 & \mathbf{in} & (\Gamma_0 \backslash \{x\}, (\nu x)\, A_0, C_0 \cup C_1) \\
PT(*P_0) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & \mathbf{in} & (\Gamma_0, *A_0, C_0) \\
PT(\mathbf{acc}_\xi(x).P_0) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma_1, C_1) = \Gamma_0 \otimes (x : \mathbf{res}) \\
 & \mathbf{in} & (\Gamma_1, x^\xi.A_0, C_0 \cup C_1) \\
PT((\mathfrak{N}^\Phi x)P_0) = & \mathbf{let} & (\Gamma_0, A_0, C_0) = PT(P_0) \\
 & & (\Gamma_1, C_1) = \Gamma_0 \otimes (x : \mathbf{res}) \\
 & \mathbf{in} & (\Gamma_1 \backslash \{x\}, A_0 \uparrow_{\{x\}}, C_0 \cup C_1 \cup \{\mathbf{traces}_x(A_0) \subseteq \Phi\})
\end{array}
$$

Figure 3.6: An algorithm for constraint extraction.

3.3.2 Step 2: Reducing Constraints

Given a closed process P, $PT(P)$ produces a triple $(\emptyset, A, C)$. The set C of constraints consists of unification constraints on value types (where all the behavioral types occurring in them are variables), constraints of the form $\mathbf{isChan}(\sigma)$ (which means that σ is a channel type), subtype constraints on behavioral types of the form $\alpha \geq A$, and constraints of the form $\mathbf{traces}_x(A) \subseteq \Phi$. We can remove the first two kinds of constraints (unification constraints on value types and $\mathbf{isChan}(\sigma)$) by applying the standard unification algorithm. Thus, we obtain the following constraints:

$$\{\alpha_1 \geq A_1, \ldots, \alpha_n \geq A_n,$$
$$\mathbf{traces}_{x_1}(B_1) \subseteq \Phi_1, \ldots, \mathbf{traces}_{x_m}(B_m) \subseteq \Phi_m\}$$

Here, we can assume that $\alpha_1, \ldots, \alpha_n$ are different from each other, since $\alpha \geq A_1$ and $\alpha \geq A_2$ can be replaced with $\alpha \geq A_1 \oplus A_2$ by Lemma 3.1. We can also assume that $\{\alpha_1, \ldots, \alpha_n\}$ contains all the type variables in the constraint, since otherwise we can always add the tautology $\alpha \geq \alpha$. Each subtype constraint $\alpha \geq A$ can also be replaced by $\alpha \geq \mu\alpha.A$, by using Lemma 3.1. Therefore, the above constraints can be further reduced, by Lemma 3.1, to:

$$\{\mathbf{traces}_{x_1}([\widetilde{A}'/\widetilde{\alpha}]B_1) \subseteq \Phi_1, \ldots, \mathbf{traces}_{x_m}([\widetilde{A}'/\widetilde{\alpha}]B_m) \subseteq \Phi_m\}$$

Here, $A'_1, \ldots, A'_n$ are the least solutions for the subtype constraints.

Thus, we have reduced type checking to the validity of trace inclusion constraints of the form $\mathbf{traces}_x(A) \subseteq \Phi$.

Example 3.6 *Recall Example 3.2. By applying the algorithm PT and the first part of Step 2, we obtain the following constraints:*

$$\begin{aligned}
&\boldsymbol{traces}_x((\nu r)\,(x^I.\overline{s}.\,\alpha_1 \,|\, r.\,x^C)) \subseteq (IR^*C)^\# \\
&\alpha_1 \geq \overline{r}.\,\alpha_2 \oplus (\nu r')\,(\overline{s}.\,\langle r'/r\rangle\alpha_1 \,|\, r'.\,x^R.\overline{r}.\,\alpha_2)\downarrow_{\{n,x,r\}} \\
&\alpha_2 \geq \alpha_2
\end{aligned}$$

By applying the second part of Step 2, we obtain $\boldsymbol{traces}_x(A_1) \subseteq (IR^*C)^\#$ *where*

$$\begin{aligned}
A_1 &= (\nu r)\,(x^I.\overline{s}.\,A_2 \,|\, r.\,x^C) \\
A_2 &= \mu\alpha_1.\overline{r}.\,A_3 \oplus (\nu r')\,(\overline{s}.\,\langle r'/r\rangle\alpha_1 \,|\, r'.\,x^R.\overline{r}.\,A_3)\downarrow_{\{n,x,r\}} \\
A_3 &= \mu\alpha_2.\alpha_2.
\end{aligned}$$

3.3.3 Step 3: Constraint Solving

We present an approximate algorithm for checking a trace inclusion constraint $\mathbf{traces}_x(A) \subseteq \Phi$ when the trace set Φ is a regular language. (Actually, we can extend the algorithm to deal with the case where Φ is a deterministic Petri net language: see Remark 3.8.)

We first describe the algorithm with an example. In Example 3.6 above, we have reduced the typability of the process to the equivalent constraint $\mathbf{traces}_x(A_1) \subseteq \Phi$ where $\Phi = (IR^*C)^\#$ and

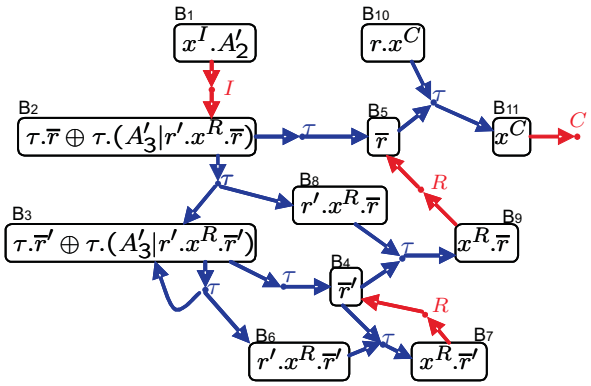
$$\begin{aligned} A_1{\downarrow_{\{x\}}} &\approx (\nu r)\,(x^I.A_2'' \mid r.\,x^C) \\ A_2'' &= \overline{r} \oplus (\nu r')\,(\langle r'/r\rangle A_2'' \mid r'.\,x^R.\overline{r}) \end{aligned}$$

Here, we have removed $A_3 = \mu\alpha.\alpha$ since $A_3 \approx \mathbf{0}$.

Step 3-1. Approximate the behavior of $A_1{\downarrow_{\{x\}}}$ by a Petri net [55] $N_{A_1,x}$. This part is similar to the translation of usage expressions into Petri nets in Kobayashi's previous work [28, 33, 35]. Since the behavioral types are more expressive (having recursion, hiding, and renaming), however, we need to approximate the behavior of a behavioral type unlike in the previous work. In this case $A_1{\downarrow_{\{x\}}}$ is infinite. To make it tractable we make a sound approximation A_1' by pushing (ν) to top level, and we eliminate $\langle r'/r\rangle$:

$$\begin{aligned} A_1' &= (\nu r, r')\,(x^I.A_2' \mid r.\,x^C) \\ A_2' &= \overline{r} \oplus (A_3' \mid r'.\,x^R.\overline{r}) \\ A_3' &= \overline{r'} \oplus (A_3' \mid r'.\,x^R.\overline{r'}) \end{aligned}$$

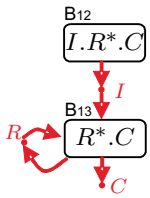
Then $N_{A_1',x}$ is as pictured. (Here we treat $A_1 \oplus A_2$ as $\tau.A_1 \oplus \tau.A_2$ for clarity. We also use a version of Petri nets with labeled transitions.)



The rectangles are the places of the net, and the dots labeled by τ, x^R, etc. are the transitions of the net. Write i_x for the number of tokens at node B_x. The behavior A_1' corresponds to the initial marking $\{i_1{=}1,\ i_{10}{=}1\}$. We say that the nodes $\widetilde{B}$ together with the restricted names (r, r') constitute a *basis* for A_1'. Note here that $\mathbf{traces}_x(A_1) \subseteq \mathbf{traces}_x(A_1') = \mathbf{ptraces}(N_{A_1',x})$ where $\mathbf{ptraces}(N_{A_1',x})$ is the set of traces of the Petri net. Thus, $\mathbf{ptraces}(N_{A_1',x}) \subseteq \Phi$ is a sufficient condition for $\mathbf{traces}_x(A_1) \subseteq \Phi$. The key point here is that A_1' still has infinite states, but all its reachable states can be expressed in the form $(\nu r, r')\,(i_1 B_1 \mid \cdots \mid i_{11} B_{11})$ (where $i_k B_k$

is the parallel composition of i_k copies of B_k), a linear combination of finitely many processes $\widetilde{B}$. That is why we could express A'_1 by the Petri net as above.

Step 3-2. Construct a deterministic, minimized automaton M_Φ that accepts the language Φ. Here the initial marking is $\{i_{12}{=}1\}$.



Step 3-3. Construct another Petri net $N_{A'_1,x} \parallel M_\Phi$ from $N_{A'_1,x}$ and M_Φ, which simulates the behavior of P_A and M_Φ simultaneously, so that the problem of $\mathbf{traces}_x(A'_1)(= \mathbf{ptraces}(N_{A'_1,x})) \subseteq \Phi$ is equivalent to a reachability problem of $N_{A'_1,x} \parallel M_\Phi$. In the example, $N_{A'_1,x} \parallel M_\Phi$ has the initial marking $\{i_1{=}1,\ i_{10}{=}1,\ i_{12}{=}1\}$ and transitions such as $B_1|B_{12} \xrightarrow{I} B_2|B_{13}$. $\mathbf{ptraces}(N_{A'_1,x}) \subseteq \Phi$ if and only if the following unsafe state is unreachable.

$$(i_1{>}0 \wedge i_{12}{=}0) \vee (i_7{>}0 \wedge i_{13}{=}0) \vee (i_9{>}0 \wedge i_{13}{=}0) \vee (i_{11}{>}0 \wedge i_{13}{=}0)$$

To explain, if $i_1 > 0 \wedge i_{12}{=}0$ then the behavior is able to make an R transition but the specification automaton M_Φ is not able.

Step 3-4. Use an approximate algorithm to decide the reachability problem of $N_{A'_1,x} \parallel M_\Phi$, in a manner similar to Kobayashi's type-based analyzer `TyPiCal` [28] for the π-calculus.

The above steps 3-1, 3-2, and 3-3 are described in more detail below. See Section 3.5 for Step 3-4.

3.3.4 Step 3-1: Construction of $N_{A,x}$

We first introduce the notion of a *basis*. The basis is analogous to that of a vector space; Each state is expressed as a linear combination of elements of the basis.

Definition 3.12 *A pair* $(\{y_1, \ldots, y_m\}, \{B_1, \ldots, B_n\})$ *is a* basis *of* A *if all of the following conditions are satisfied:*

- $A \approx (\nu y_1) \cdots (\nu y_m)\,(i_1 B_1 \mid \cdots \mid i_n B_n)$ *for some* $i_1, \ldots, i_n \in \mathbf{Nat}$.
- *If* $B_j \xrightarrow{l} C$, *then there exist* $i_1, \ldots, i_n \in \mathbf{Nat}$ *such that* $C \approx i_1 B_1 \mid \cdots \mid i_n B_n$.
- *For each* B_j, *there are only finitely many* C *(up to* $\approx$*) such that* $B_j \xrightarrow{l} C$.

Note that if $(\{\widetilde{y}\}, \{B_1, \ldots, B_n\})$ is a basis of A, then whenever $A \Longrightarrow A'$, there exist $i_1, \ldots, i_n$ such that $A' \approx (\nu \widetilde{y})\,(i_1 B_1 \mid \cdots \mid i_n B_n)$. Let us write $\mathit{Index}(C)$ for $(i_1, \ldots, i_n)$ such that $C \approx i_1 B_1 \mid \cdots i_n B_n$. (If there are more than one such tuple, $\mathit{Index}(C)$ picks one among them.)

Therefore, if $A{\downarrow_{\{x\}}}$ has a basis, the behavior of $A{\downarrow_{\{x\}}}$ is simulated by the (labeled) Petri net $N_{A,x,(\{\widetilde{y}\},\{\widetilde{B}\})}$ given below. Here, we use a process-like syntax to represent the elements of a Petri net rather than the standard tuple notation (P, T, F, W, M_0). A marking state m which has i_k tokens for each place p_k ($k \in \{1, \ldots, n\}$) is written $i_1 p_1 \mid \cdots \mid i_n p_n$. A transition that consumes a marking m_1 and produces m_2 is expressed by $m_1 \xrightarrow{\gamma} m_2$, where γ is the label of the transition.

- The set P of places is $\{p_{B_1}, \ldots, p_{B_n}\}$.
- The initial marking m_I is $i_1 p_{B_1} \mid \cdots \mid i_n p_{B_n}$
  where $A{\downarrow_{\{x\}}} \approx (\nu\widetilde{y})\,(i_1 B_1 \mid \cdots \mid i_n B_n)$.
- The set of transitions consists of:
  - $p_{B_j} \xrightarrow{\tau} i_1 p_{B_1} \mid \cdots \mid i_n p_{B_n}$
    where $\mathit{Index}(C) = (i_1, \ldots, i_n)$, for each $B_j \xrightarrow{\tau} C$.
  - $p_{B_j} \xrightarrow{\xi} i_1 p_{B_1} \mid \cdots \mid i_n p_{B_n}$
    where $\mathit{Index}(C) = (i_1, \ldots, i_n)$, for each $B_j \xrightarrow{x^\xi} C$.
  - $p_{B_j} \mid p_{B_{j'}} \xrightarrow{\tau} (i_1 + i'_1) p_{B_1} \mid \cdots \mid (i_n + i'_n) p_{B_n}$ where $\mathit{Index}(C) = (i_1, \ldots, i_n)$ and $\mathit{Index}(C') = (i'_1, \ldots, i'_n)$, for each pair of transitions $B_j \xrightarrow{\overline{z}} C$ and $B_{j'} \xrightarrow{z} C'$ such that $z \in \{\widetilde{y}\}$.

Below, we omit the basis and just write $N_{A,x}$ for $N_{A,x,(\{\widetilde{y}\},\{\widetilde{B}\})}$. Let us write $\mathbf{ptraces}(N_{A,x})$ for the set:

$$\{\xi_1 \cdots \xi_k \mid m_I \stackrel{\xi_1}{\Longrightarrow} \cdots \stackrel{\xi_k}{\Longrightarrow} m'\}$$

where $\stackrel{\xi}{\Longrightarrow}$ means $\xrightarrow{\tau}{}^* \xrightarrow{\xi} \xrightarrow{\tau}{}^*$. By the construction of $N_{A,x}$, $\mathbf{ptraces}(N_{A,x}) = \mathbf{traces}_x(A)$.

The construction of $N_{A,x}$ outlined above can be applied only when a basis of $A{\downarrow_x}$ can be found (by some heuristic algorithm). If $A{\downarrow_x}$ has no basis or cannot be found, we approximate $A{\downarrow_x}$ by moving all the ν-prefixes to the top-level; for example, $y.\,(\nu x)\,A$, $*(\nu x)\,A$ and $\mu\alpha.(\nu x)\,A$ are replaced by $(\nu x)\,(y.\,A)$, $(\nu x)\,{*}A$, and $(\nu x)\,\mu\alpha.A$ respectively. Let A' be the approximation of $A{\downarrow_{\{x\}}}$. It is easy to prove that A' is a sound approximation of $A{\downarrow_{\{x\}}}$, in the sense that $\mathbf{traces}_x(A) \subseteq \mathbf{traces}_x(A')$.

We can compute a basis of A' as follows (see Appendix F for more details). Since ν-prefixes do not appear inside recursion, we can first eliminate the constructors $\cdot{\uparrow_S}$, $\cdot{\downarrow_S}$, and $\langle\widetilde{y}/\widetilde{x}\rangle$. Let $(\nu\widetilde{y})\,A''$ be the resulting expression, where A'' does not contain $\cdot{\uparrow_S}$, $\langle\widetilde{y}/\widetilde{x}\rangle$, or $(\nu x)\,$. Let $\mathbf{B}$ be the set of behavioral types that are subexpressions of the behavioral types obtained from A'' by expanding recursive types and do not contain "unnecessary" unfolding $[\mu\alpha.A/\alpha]A$. Then, $\mathbf{B}$ is a finite set, and $(\{\widetilde{y}\}, \mathbf{B})$ is a basis of A'. We can therefore construct a Petri net $N_{A',x}$. By

the construction, $\mathbf{ptraces}(N_{A',x}) = \mathbf{traces}_x(A') \supseteq \mathbf{traces}_x(A)$, so that $\mathbf{ptraces}(N_{A',x}) \subseteq \Phi$ is a sufficient condition for $\mathbf{traces}_x(A) \subseteq \Phi$.

3.3.5 Steps 3-2 and 3-3: Construction of $N_{A,x} \,\|\, M_\Phi$ and reduction of $\mathbf{traces}_x(A)$ to a reachability problem

Let $P_{N_{A,x}}$ and $T_{N_{A,x}}$ be the sets of places and transitions of $N_{A,x}$ respectively. Let M_Φ be a minimized deterministic automaton[4] that accepts Φ, and let Q_Φ be its set of states and δ_Φ be its transition function.

Definition 3.13 *The* composition *of $N_{A,x}$ and M_Φ, written $N_{A,x} \,\|\, M_\Phi$, is defined as follows:*

- *The set of places is $P_{N_{A,x}} \cup Q_\Phi$*
- *The set of transitions is:*

$$\{(m|q) \xrightarrow{\xi} (m'|q') \mid (m \xrightarrow{\xi} m') \in T_{N_{A,x}} \wedge \delta_\Phi(q,\xi) = q'\}$$
$$\cup \{m \xrightarrow{\tau} m' \mid (m \xrightarrow{\tau} m') \in T_{N_{A,x}}\}$$

- *Initial state is $m_I \,|\, q_I$ where m_I is the initial state of $N_{A,x}$ and q_I is the initial state of M_Φ.*

Now, $\mathbf{ptraces}(N_{A,x}) \subseteq \Phi$ can be reduced to the reachability problems of $N_{A,x} \,\|\, M_\Phi$.

Theorem 3.2 ***ptraces***$(N_{A,x}) \subseteq \Phi$ *if and only if no marking $m \,|\, q$ that satisfies the following conditions is reachable:*

- *$m \xrightarrow{\xi} m'$ for some m' and ξ in $N_{A,x}$.*
- *$\delta_\Phi(q,\xi)$ is undefined.*

Thus, we can reduce $\mathbf{ptraces}(N_{A,x}) \subseteq \Phi$ to a finite set of reachability problems of $N_{A,x} \,\|\, M_\Phi$. Hence $\mathbf{ptraces}(N_{A,x}) \subseteq \Phi$ is decidable [43].

Corollary 3.1 ***ptraces***$(N_{A,x}) \subseteq \Phi$ *if and only if for every transition rule of the form $m_1 \xrightarrow{\xi} m_2$ of $N_{A,x}$ and q such that $\delta_\Phi(q,\xi)$ is undefined, no marking m such that $m \geq m_1 \,|\, q$ is reachable by $N_{A,x} \,\|\, M_\Phi$.*

Reminder 3.8 *We can actually extend the above algorithm for checking* ***traces***$_x(A) \subseteq \Phi$ *to deal with the case where Φ belongs to the class of deterministic Petri net languages (more precisely, the class of P-type languages of λ-free, deterministic Petri nets [53, 55]). If Φ is the P-type language of a λ-free, deterministic Petri net, then its complement $\overline{\Phi}$ is a Petri net*

[4]Note that since Φ is prefix-closed, all the states of the minimized automaton are accepting states.

language [53]. Therefore, we can construct a Petri net that accepts the intersection of the language of $N_{A,x}$ and $\overline{\Phi}$ [55]), so that $\boldsymbol{ptraces}(N_{A,x}) \subseteq \Phi$ can be reduced to the emptiness problem of the Petri net, which is decidable due to the decidability of the reachability problem.

Some of the useful resource usage specifications are not regular languages but are deterministic Petri net language. For example, consider a stack-like resource on which, at any point of program execution, the number of times the operation pop has been performed is less than the number of times push has been performed. Such specification is expressible as a deterministic Petri net language.

3.4 Extensions

The type system given so far guarantees that no invalid resource access is performed, but not that any necessary access is performed eventually; for example, the type system does *not* guarantee that a file is eventually closed. We discuss extensions of the type system to guarantee such properties.

We are interested in type systems that satisfy either *partial liveness*[5] or the stronger *liveness* property:

- partial liveness: If $P \longrightarrow^* Q$ and $Q \not\longrightarrow$, then Q does not contain any resource to which some access *must* be performed.
- liveness: In any fair reduction sequence $P \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$, P eventually performs all the necessary resource access. (Here, a reduction sequence is fair if an input or output action that is infinitely enabled will eventually succeed. Without the fairness assumption, no process can satisfy the liveness property in the presence of a divergent process $(\nu x)\,(\overline{x}\langle\,\rangle \mid *x(\,).\,\overline{x}\langle\,\rangle$, which is too restrictive.)

Our idea is to take the resource type system from the previous sections, and combine it with some existing system that annotates those communications that eventually succeed. Specifically, this existing system might be (1) deadlock-freedom [33, 35], which guarantees that the annotated communications eventually succeed unless the process diverges; the combination would then guarantee partial liveness. Or the existing system could be (2) lock-freedom [31, 33], which guarantees that the annotated communications eventually succeed even in the presence of divergence (assuming a strongly fair scheduler); the combination would then guarantee full liveness.

To formally state which resource access *must* be performed, we extend the trace sets.

[5]This is not a standard term; actually, the partial liveness here can be viewed as the safety property that no 'bad' state is reachable such that the necessary accesses have not yet been performed but the system cannot make any move.

Definition 3.14 *An* extended trace set *is a set of sequences of access labels, possibly ending with a special label* $\downarrow$*, that is closed under the prefix operation.*

Intuitively, the special label $\downarrow$ means that no further resource access need to be performed. For example, the trace set $(\{C\downarrow, RC\downarrow\})^{\#}$ means that the close operation needs to be performed, while $(\{\downarrow, R\downarrow, C\downarrow, RC\downarrow\})^{\#}$ means that the close operation need not be performed.

Now we can state the partial liveness property more formally. We write $(\widetilde{\nu}\widetilde{\mathfrak{N}})$ for a (possibly empty) sequence of ν- and $\mathfrak{N}$-binders.

Definition 3.15 *A process* P *is* partially live *if* $\downarrow \in \Phi$ *whenever* $P \longrightarrow^* \preceq (\widetilde{\nu}\widetilde{\mathfrak{N}})(\mathfrak{N}^{\Phi}x)Q \not\longrightarrow$.

3.4.1 A Type System for the Partial Liveness Property

We extend the syntax of processes to allow each input and output prefix to be annotated with information about whether the communication is guaranteed to succeed.

Definition 3.16 ((extended) processes) *The set of (extended) processes is given by:*

$$\begin{array}{lll} t \text{ (attributes)} & ::= & \mathbf{c} \mid \emptyset \\ P & ::= & \overline{x}_t\langle y_1, \ldots, y_n\rangle.\, P \mid x_t(y_1, \ldots, y_n).\, P \mid \cdots \end{array}$$

The attribute $\mathbf{c}$ indicates that when the annotated input or output operation appears at the top-level, the operation will succeed unless the whole process diverges, while $\emptyset$ does not give such a guarantee. We often omit tag $\emptyset$.

We assume that there exists a type system guaranteeing that any well-typed process is *well-annotated* in the sense of Definition 3.17 below. There are indeed such type systems [29, 33, 35]. Moreover, the static analysis tool `TyPiCal` [28] can automatically infer the annotations.

Definition 3.17 P *is* active*, written* $active(P)$*, if* $P \preceq (\widetilde{\nu}\widetilde{\mathfrak{N}})(\overline{x}_{\mathbf{c}}\langle\widetilde{v}\rangle.\, Q \mid R)$ *or* $P \preceq (\widetilde{\nu}\widetilde{\mathfrak{N}})(x_{\mathbf{c}}(\widetilde{y}).\, Q \mid R)$*. Additionally,* P *is* well-annotated*, written* $well\_annotated(P)$*, if for any* P' *such that* $P \longrightarrow^* P'$ *and* $active(P')$*, there exists* P'' *such that* $P' \longrightarrow P''$*.*

For example, $\overline{x}_{\mathbf{c}}\langle\,\rangle.\, \mathbf{0} \mid x_{\mathbf{c}}(\,).\, \overline{y}_{\emptyset}\langle\,\rangle.\, \mathbf{0}$ is well-annotated, but $\overline{x}_{\mathbf{c}}\langle\,\rangle.\, \mathbf{0} \mid x_{\mathbf{c}}(\,).\, \overline{y}_{\mathbf{c}}\langle\,\rangle.\, \mathbf{0}$ is not. Note that $x_{\emptyset}(\,).\, \overline{x}_{\mathbf{c}}\langle\,\rangle.\, \mathbf{0}$ *is* well-annotated since, although the output never succeeds, it does not appear at the top-level.

Now we introduce the type system that guarantees the partial liveness. We extend the behavioral types by extending each input, output, or τ-action with an attribute to indicate whether the action is guaranteed to succeed.

$$A \quad ::= \quad \overline{x}_t.\, A \mid x_t.\, A \mid \tau_t.A \mid \cdots$$

$$
\begin{array}{rll}
\mathit{disabled}(\mathbf{0}, S) & & \\
\mathit{disabled}(x^{\xi}.A, S) & \text{if} & \mathit{disabled}(A, S) \text{ and } x \notin S \\
\mathit{disabled}(a_{\mathbf{c}}.A, S) & \text{if} & \mathit{disabled}(A, S) \\
\mathit{disabled}(a_{\emptyset}.A, S) & & \\
\mathit{disabled}(\tau_{\mathbf{c}}.A, S) & \text{if} & \mathit{disabled}(A, S) \\
\mathit{disabled}(\tau_{\emptyset}.A, S) & & \\
\mathit{disabled}(A_1 \mid A_2, S) & \text{if} & \mathit{disabled}(A_1, S) \text{ and } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(A_1 \oplus A_2, S) & \text{if} & \mathit{disabled}(A_1, S) \text{ or } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(*A, S) & \text{if} & \mathit{disabled}(A, S) \\
\mathit{disabled}((\nu x)\, A, S) & \text{if} & \mathit{disabled}(A, S \backslash \{x\}) \\
\mathit{disabled}(A\uparrow_{S'}, S) & \text{if} & \mathit{disabled}(A, S \backslash S') \\
\mathit{disabled}(A\downarrow_{S'}, S) & \text{if} & \mathit{disabled}(A, S \cap S') \\
\mathit{disabled}(\langle \widetilde{y}/\widetilde{x} \rangle A, S) & \text{if} & \mathit{disabled}(A, \{z \mid [\widetilde{y}/\widetilde{x}]z \in S\}) \\
\mathit{disabled}(\mu\alpha.A, S) & \text{if} & \mathit{disabled}([\mu\alpha.A/\alpha]A, S)
\end{array}
$$

Figure 3.7: The definition of $\mathit{disabled}(A, S)$

For example, a process having type $\overline{x}_{\mathbf{c}}.\,\overline{x}_{\emptyset}.\,\mathbf{0}$ implies that the process may send values on x twice, and that the first send is guaranteed to succeed (i.e., the sent value will be received by some process), while there is no such guarantee for the second send.

The transition semantics of behavioral types is unchanged; The attribute t is just ignored.

We revise the definitions of the subtype relation and the traces by using the following predicate $\mathit{disabled}(A, S)$. Intuitively, this means that A describes a process that may get blocked without accessing any resources in S.

Definition 3.18 *$\mathit{disabled}(A, S)$ is the least binary relation between extended behavioral types and sets of variables closed under the rules in Figure 3.7.*

Definition 3.19 *The set $\boldsymbol{etraces}_x(A)$ of extended traces is:*

$$
\begin{array}{l}
\{\xi_1 \cdots \xi_n \downarrow \mid \exists B.A\downarrow_{\{x\}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} B \wedge \mathit{disabled}(B, \{x\})\} \\
\cup \{\xi_1 \cdots \xi_n \mid \exists B.A\downarrow_{\{x\}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} B\}
\end{array}
$$

Here, $A\downarrow_{\{x\}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} B \wedge \mathit{disabled}(B, \{x\})$ means that ξ_n may be the last access to x, so that $\downarrow$ is attached to the sequence $\xi_1 \cdots \xi_n$. By definition, $\mathbf{etraces}_x(A)$ is prefix-closed.

$$\frac{\Gamma \rhd_{pl} P : A_2 \qquad \Gamma \rhd_{pl} x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \qquad \Gamma \rhd_{pl} \widetilde{v} : \widetilde{\sigma}}{\Gamma \rhd_{pl} \overline{x}_t\langle\widetilde{v}\rangle.\, P : \overline{x}_t.\, (\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)} \quad \text{(ET-Out)}$$

$$\frac{\Gamma, \widetilde{y} : \widetilde{\sigma} \rhd_{pl} P : A_2 \qquad \Gamma \rhd x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \qquad A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1}{\Gamma \rhd_{pl} x_t(\widetilde{y}).\, P : x_t.\, (A_2{\uparrow_{\{\widetilde{y}\}}})} \quad \text{(ET-In)}$$

$$\frac{\Gamma, x : \mathbf{res} \rhd_{pl} P : A \qquad \mathbf{etraces}_x(A) \subseteq \Phi}{\Gamma \rhd_{pl} (\mathfrak{N}^{\Phi} x)P : A{\uparrow_{\{x\}}}} \quad \text{(ET-NewR)}$$

Figure 3.8: Typing Rules for Partial Liveness

Definition 3.20 *$A_1 \leq A_2$ is the largest relation on closed behavioral types that satisfies the following properties:*

- *If $A_1 \xrightarrow{l} A_1'$ then there exists A_2' such that $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_1' \leq A_2'$.*
- *disabled(A_1, S) implies disabled(A_2, S) for any set S of variables.*

Note that by the definition, $A_1 \leq A_2$ implies $\mathbf{etraces}_x(A_1) \subseteq \mathbf{etraces}_x(A_2)$.

The typing rules are the same as those in Section 3.2, except for the rules shown in Figure 3.8. The only changes are that attributes have been attached to (ET-Out) and (ET-In), and that $\mathbf{traces}_x(A{\downarrow_{\{x\}}})$ has been replaced by $\mathbf{etraces}_x(A{\downarrow_{\{x\}}})$ in (ET-NewR). An important invariant maintained by the typing rules is that the type of an input/output process is annotated with $\mathbf{c}$ only if the process itself is annotated with $\mathbf{c}$. For example, we cannot derive $x : \mathbf{chan}\langle\rangle \rhd_{pl} \overline{x}_{\emptyset}\langle\,\rangle : \overline{x}_{\mathbf{c}}$.

The following theorem states the soundness of the extended type system.

Theorem 3.3 *If well_annotated(P) and $\emptyset \rhd_{pl} P : A$, then P is partially live.*

Proof 3.2 *We make use of three lemmas. The first two show that typing and well-annotatedness are preserved by reduction. The third means that the type of a process properly captures the possibility of the process being blocked.*

- ***Subject reduction.*** *If $\Gamma \rhd_{pl} P : A$ and $P \xrightarrow{L} Q$, then there exists some B such that $\Gamma \rhd_{pl} Q : B$ and $A \stackrel{L}{\Longrightarrow} B$.*
  Proof: See Appendix D.
- ***Well-annotatedness.*** *If well_annotated(P) and $P \longrightarrow^* \preceq Q$, then well_annotated$(Q)$.*
  Proof: trivial by definition of well_annotated(P).

- ***Disabled.*** *If well_annotated(P) and $\Gamma \triangleright_{pl} P : A$ with **bool** $\notin codom(\Gamma)$, then $P \not\longrightarrow$ implies disabled(A, S) for any S.*
  Proof: See Appendix E.

Now we are ready to prove the theorem. Suppose that $P \longrightarrow^* (\widetilde{\nu}\widetilde{\mathfrak{N}})(\mathfrak{N}^{\Phi}x)Q \not\longrightarrow$ and well_annotated(P), $\emptyset \triangleright_{pl} P : A$. We have to show $\downarrow \in \Phi$. By subject-reduction *we obtain $\emptyset \triangleright_{pl} (\widetilde{\nu}\widetilde{\mathfrak{N}})(\mathfrak{N}^{\Phi}x)Q : A'$ for some A'. By the inversion of the typing rules, we get $\widetilde{y} : \widetilde{\textbf{\textit{res}}}, \widetilde{z} : \widetilde{\sigma}, x : \textbf{\textit{res}} \triangleright_{pl} Q : B$ and $\textbf{\textit{traces}}_x(B) \subseteq \Phi$ for some sequence $\widetilde{\sigma}$ of channel types. (Here, $\widetilde{y}$ and $\widetilde{z}$ are the variables bound by $\mathfrak{N}$.) By* well-annotatedness *we also have well_annotated$((\widetilde{\nu}\widetilde{\mathfrak{N}})(\mathfrak{N}^{\Phi}x)Q)$, which implies well_annotated(Q). Thus, by* Disabled*, we get disabled(B, S) for any S, which implies disabled$(B{\downarrow_{\{x\}}}, \{x\})$. So, we have $\downarrow \in \textbf{\textit{etraces}}_x(B) \subseteq \Phi$ as required.*

Example 3.7 *An annotated version of Example 3.5:*

$$\begin{aligned}
P &= (\nu s)\,(*s_{\mathbf{c}}(n,x,r).\,P_1 \mid (\mathfrak{N}^{\Phi}x)P_2) \\
P_1 &= \textbf{if } n = 0 \textbf{ then } \overline{r}_{\mathbf{c}}\langle\rangle \\
&\quad\ \textbf{else } (\nu r')\,(\overline{s}_{\mathbf{c}}\langle n-1, x, r'\rangle. \mid r'_{\mathbf{c}}().\,\textbf{\textit{read}}(x).\overline{r}_{\mathbf{c}}\langle\rangle) \\
P_2 &= (\nu r)\,(\textbf{\textit{init}}(x).\overline{s}_{\mathbf{c}}\langle 100, x, r\rangle \mid r_{\mathbf{c}}().\,\textbf{\textit{close}}(x)) \\
\Phi &= (IR^*C\downarrow)^{\#}
\end{aligned}$$

is well-annotated. Suppose

$$\begin{aligned}
A_1 &= \mu\alpha.(\overline{r}_{\mathbf{c}} \oplus (\nu r')\,(\langle r'/r\rangle\alpha | r'_{\mathbf{c}}.\,x^R.\overline{r}_{\mathbf{c}}) \\
\Gamma &= s{:}\textbf{\textit{chan}}\langle(b{:}\textbf{int},\ x{:}\textbf{\textit{res}},\ r{:}\textbf{\textit{chan}}\langle\rangle)\,A_1\rangle.
\end{aligned}$$

Then

$$\begin{aligned}
&\Gamma \triangleright P_1 : A_1 \\
&\Gamma \triangleright *s_{\mathbf{c}}(n,x,r).\,P_1 : *s_{\mathbf{c}}.\,(A_1{\uparrow_{\{n,x,r\}}}) \approx *s_{\mathbf{c}} \\
&\Gamma \triangleright P_2 : (\nu r)\,(x^I.A_1 | r_{\mathbf{c}}.\,x^C).
\end{aligned}$$

So long as $\textbf{\textit{etraces}}_x((\nu r)\,(x^I.A_1 \mid r_{\mathbf{c}}.\,x^C.)) \subseteq \Phi$, we obtain $\emptyset \triangleright P : \textbf{\textit{0}}$. □

3.4.2 Type Inference

The type inference algorithm for the extended type system is almost the same as the algorithm for the basic type system discussed in Section 3.3. The only changes are:

- In the constraint generaltion algorithm PT, attribute annotations for input and ouptut processes are propagated to types. For example, the case for output processes becomes:

$$\begin{aligned}
&PT(\overline{x}^t\langle\widetilde{v}\rangle.\,P_0) = \\
&\quad \textbf{let } \ (\Gamma_i, \sigma_i) = PTv(v_i) \\
&\qquad (\Gamma_0, A_0, C_0) = PT(P_0) \\
&\qquad (\Gamma, C) = \Gamma_0 \otimes (x : \textbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})\alpha\rangle) \otimes \Gamma_1 \otimes \cdots \otimes \Gamma_n \\
&\quad \textbf{in } (\Gamma, \overline{x}_t.\,([\widetilde{v}/\widetilde{y}]\alpha \mid A_0), C) \text{ (where } \alpha \text{ fresh)}
\end{aligned}$$

- The constraint $\mathbf{traces}_x(A) \subseteq \Phi$ is replaced by $\mathbf{etraces}_x(A) \subseteq \Phi$.

The second change forces us to adjust the reduction of the constraint to the reachability problem of Petri nets (recall step 3 of the algorithm in Section 3.3). First, we need to use $\mathbf{eptraces}(N_{A,x})$ defined below, which corresponds to $\mathbf{etraces}_x(A)$, instead of $\mathbf{ptraces}(N_{A,x})$ in the reduction.

Definition 3.21 *$\boldsymbol{eptraces}(N_{A,x})$ is the set*

$$\{\xi_1 \cdots \xi_k \mid m_I \stackrel{\xi_1}{\Longrightarrow} \cdots \stackrel{\xi_k}{\Longrightarrow} m'\} \cup \{\xi_1 \cdots \xi_k \downarrow \mid m_I \stackrel{\xi_1}{\Longrightarrow} \cdots \stackrel{\xi_k}{\Longrightarrow} m' \wedge \mathit{pdisabled}(m', \{x\})\}$$

where m_I is the initial marking of $N_{A,x}$. $\mathit{pdisabled}(m, S)$ means that $\mathit{disabled}(A, S)$ holds for the behavioral type A expressed by m.

Second, the construction of an automaton needs to be adjusted so that it accepts extended traces. For example, the automaton used in the explanation of Step 3-2 in Section 3.3 is replaced by the one that accepts $IR^*C\downarrow$.

With these changes, the validity of a constraint $\mathbf{etraces}_x(A) \subseteq \Phi$ is reduced to the reachability problem of a Petri net $N_{A,x} \parallel M_\Phi$ where composition of a Petri net $N_{A,x}$ and an automaton M_Φ is defined in the same manner as Definition 3.13.

Theorem 3.4 *$\boldsymbol{eptraces}(N_{A,x}) \subseteq \Phi$ if and only if no marking $m \,|\, q$ that satisfies the following conditions is reachable:*

- *$\mathit{pdisabled}(m, \{x\})$.*
- *$\delta_\Phi(q, \downarrow)$ is undefined.*

3.5 Implementation

We have implemented a prototype resource usage analyzer based on the extended type system described in Section 3.4. We have tested all the examples given in the present paper. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

The analyzer takes a pi-calculus program as an input, and uses `TyPiCal`[28] to annotate each input or output action with an attribute on whether the action is guaranteed to succeed automatically (recall the syntax of extended processes in Section 3.4). The annotated program is then analyzed based on the algorithm described in Section 3.3.

The followings are some design decisions we made in the current implementation. We restrict the resource usage specification (Φ) to the regular languages, although in future we may extend it based on Remark 3.8. In Step 3-1 of the algorithm for checking $\mathbf{etraces}_x(A) \subseteq \Phi$, we blindly approximate A by pushing all of its ν-prefixes to the top-level. In future we might utilize an

existing model checker to handle the case where A is already finite. In Step 3-4 for solving the reachability problems of Petri nets, we approximate the number of tokens in each place by an element of the finite set $\{0, 1, 2, \text{“3 or more”}\}$. That approximation reduces Petri nets to finite state machines, so we can use BDD to compute an approximation of the reachable states.

Figure 3.9 shows a part of a successful run of the analyzer. The first process (on the second line) of the input program runs a server, which returns a new, initialized resource. We write `!` and `?` for output and input actions. The resource access specification is here expressed by the number 1 of `newR 1, x`, which refers to the built-in specification $(I(R+W)^*C\downarrow)^{\#}$. The second process runs infinitely many client processes, each of which sends a request for a new resource, and after receiving it, reads and closes it. The third process (on the 6th line) is a tail-recursive version of the replicated service in Example 3.2. Here, a boolean is passed as the first argument of s instead of an integer, as the current system is not adapted to handle integers; it does not affect the analysis, since the system ignores the value and simply inspects both branches of the conditional. Note that the program creates infinitely many resources and has infinitely many states. The first output is the annotated version of the input program produced by `TyPiCal`, where `!!` and `??` are an output and an input with the attribute **c** (recall Section 3.4).

The remaining part shows the trace inclusion constraint and the constructed Petri net. The final line reports that the verification has succeeded, which implies that both the safety property (in Section 3.2) and the partial liveness property (in Section 3.4) are satisfied.

3.6 Discussion

We have presented a resource usage verification for the π-calculus extended with resource manipulation primitives. Though we believe the idea of extracting resource-wise behavior of a program using a type system and verifying the safety of the behavior can be applied to practical software, it is difficult to apply our current framework directly to practical software because the current calculus lacks many primitives which are in standard programming languages. In this section, we roughly discuss how the idea of our verification can be applied to primitives in more practical language. We discuss particularly how function calls, references and interrupts can be dealt with. More detailed discussion or formalization of such extensions are left as future work.

Function calls We can deal with function calls in almost the same manner as communications on channels. A function type would be of the form $(\tau_1, \ldots, \tau_n) \xrightarrow{A} \tau$. This function type is accompanied by a behavioral type A which describes how the arguments of the function are accessed after the function call. Note that A on that function type corresponds to a behavioral type attached to a channel type of the current type system, which describes how communicated

Input:

```
new create,s in
  *(create?(r).newR 1,x in acc(x,init).r!(x))
| *(new r in create!(r)
     | r?(y).new c in s!(false,y,c) | s!(false,y,c)
                    | c?().c?().acc(y,close))
| *(s?(b,x,r).if b then r!()
              else acc(x,read).s!(b,x,r))
```

Output:

```
(*** The result of lock-freedom analysis ***)
new create, s in
  *create??(r). newR 1,x in acc(x, I). r!!(x)
| *(new r in create!!(r)
     | r??(y).new c in s!!(false,y,c) | s!!(false,y,c)
                    | c??().c??().acc(y,close))
...
(*** Constraints ***)
etrace(x,acc(x, init).(c!! & acc(x, read). $16 | $16 |
         c??. c??. acc(x, close). 0)) is included in 1
...
(*** initial marking ***)
1 * 11 | 1 * 7
(*** 14 Places ***)
0: c!!. 0
...
(*** 9 Transitions ***)
(x,close): 1*12 | 1*10 -> -1*12 | 1*13 | -1*10 | 1*1
...
No error found
```

Figure 3.9: A Sample Run of the Analyzer.

values will be used after the communication.

References Dealing with mutable references is important because it is necessary in modelling a program in which processes communicate using shared memory. In order to deal with references, behavioral types should be extended to be able to express communications and resource accesses through references.

A challenge is race-condition on a reference. Due to a similar reason to the difficulty in references to mutexes described in Section 2.3.2, we need to prevent race on a reference to avoid unexpected change of a reference content by concurrently running threads. For that purpose, we may be able to use the ownership idea introduced in Section 2.3.2, or some external race-freedom analysis.

Interrupts We can deal with interrupts by extending our calculus with an interrupt primitive $P_1 \rhd P_2$ which intuitively means an execution of an interrupt handler P_2 may interrupt P_1. We also need to extend behavioral types with $A_1 \rhd A_2$. In verification of partial-liveness property, we have to use a deadlock-freedom analysis which takes interrupts into account, such as one presented in Chapter 2.

In Step 3 of constraint solving phase in Section 3.3.3, we should consider how to judge whether a trace inclusion constraint $\mathbf{traces}_x(A) \subseteq \Phi$ holds when A contains an interrupt $A_1 \rhd A_2$. A possible solution is to extend the encoding of a behavioral type with Petri-net used in Section 3.3.3 with interrupts. Such extended encoding may need to make an approximation to A.

Chapter 4

Related work

4.1 Deadlock-freedom verification

Much work on deadlock-freedom verification [5, 32, 34, 35, 54] has been done so far. As we have mentioned in Section 1.2.1, our deadlock-freedom analysis deals with primitives that have been ignored in the existing researches.

Kobayashi, Saito and Sumii [32, 34, 35] 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 expressions in order to detect circular dependency among input/output operations to channels. However, their framework does not deal with interrupts. If we ignore interrupts, our usages can be seen as a shrunk form of their usage expressions; following their encoding [32], $\mathbf{lock}(lev, ob)$ corresponds to $()/ * I^{\infty}_{lev}.O^{lev}_{\infty}$ and $\mathbf{lock}(lev, cap)$ to $()/O^{lev}_{\infty} | * I^{\infty}_{lev}.O^{lev}_{\infty}$ and $\mathbf{lock}(lev, \mathbf{0})$ to $()/\mathbf{0}$.

Boyapati, Lee and Rinard [5] proposed a type-based deadlock- and race-freedom verification of Java programs. In their framework, programmers annotate each object with a partially-ordered lock level. Then the type system guarantees that locks are acquired in a strict decreasing order. The type system also provides a special ordering mechanism for tree-structured objects in order to deal with programs that sequentially acquire locks in such objects in a parent-to-child order. The difference between our type system and theirs is that (1) their type system deals with race-freedom (2) their type system is equipped with polymorphism on lock levels (3) our type system deals with non-block-structured mutex primitives and interrupts and (4) their type system requires annotation on lock ordering, while ours not.

Permandla, Roberson and Boyapati [54] proposed a type-based deadlock- and race-freedom verification for Java virtual machine language. They also adopt lock-level-based approach. Though they deal with non-block-structured synchronization primitives, they do not deal with interrupts.

Several model-checking-based verification methods [20, 21, 63, 67] can deal with concurrency,

and some of them [20, 21] deal with non-block-structured mutex primitives. However, none of them deals with interrupts.

4.2 Calculi with interrupts

Chatterjee et al. have proposed a calculus that is equipped with interrupts [8, 52]. 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 $(lev, taglevel)$, where $taglevel$ is a map from tags to lock levels. $taglevel(t)$ 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 Chapter 2.1.

Asynchronous exceptions [19, 39] 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. [39] extended Concurrent Haskell [26] 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.

4.3 Resource usage analysis

Resource usage analysis and similar analyses have recently been studied extensively, and a variety of methods from type systems to model checking have been proposed [2, 11, 12, 17, 24, 40, 59]. However, only a few of them deal with concurrent languages. To our knowledge, none of them deal with the partial liveness property (or the total liveness property) that we discussed in Section 3.4. Nguyen and Rathke [51] propose an effect-type system for a kind of resource usage analysis for functional languages extended with threads and monitors. In their language, neither resources nor monitors can be created dynamically. On the other hand, our target

language is π-calculus, so that our type system can be applied to programs that may create infinitely many resources (due to the existence of primitives for dynamic creation of resources: recall the example in Figure 3.9), and also to programs that use a wide range of communication and synchronization primitives. Capability-based type systems can deal with concurrency to a certain degree ([11], Section 4.2), by associating each resource with a unique capability to access the resource. The type system can control the resource access order, by ensuring the uniqueness of the capability and keeping track of what access is currently allowed by each capability. In this approach, however, resource accesses are completely serialized and programmers have to care about appropriately passing capabilities between threads. Capability-based type systems [11, 12] also require rather complex type annotations. Igarashi and Kobayashi's type system for resource usage analysis for λ-calculus [24] can be extended to deal with threads, by introducing the following typing rule:

$$\frac{\Gamma_1 \triangleright M_1 : \tau_1 \qquad \Gamma_2 \triangleright M_2 : \tau_2}{\Gamma_1 \otimes \Gamma_2 \triangleright \mathit{spawn}(M_1); M_2}$$

Here, $\Gamma_1 \otimes \Gamma_2$ describes resources that are used according to Γ_1 and Γ_2 that are used in an interleaving manner. However, it is not obvious how to accurately capture information about possible synchronizations between M_1 and M_2.

Model checking technologies [3] can of course be applicable to concurrent languages, but they suffer from the state explosion problem, especially for expressive concurrent languages like π-calculus, where resources and communication channels can be dynamically created and passed around. Appropriate abstraction must be devised for effectively performing the resource usage analysis for the π-calculus with model checking. Actually, our type-based analysis can be considered a kind of abstract model checking. The behavioral types extracted by (the first two steps of) the type inference algorithm are abstract concurrent programs, each of which captures the access behavior on each resource. Then, conformance of the abstract program with respect to the resource usage specification is checked as a model checking problem. It would be interesting to study a relationship between the abstraction through our behavioral type and the abstraction techniques for concurrent programs used in the model checking community. From that perspective, an advantage of our approach is that our type, which describes a resource-wise behavior, has much smaller state space than the whole program. In particular, if infinitely many resources are dynamically created, the whole program has infinite states, but it is often the case that our behavioral types are still finite (indeed so for the example in Figure 3.9). The limitation of our current analysis is that programs can be abstracted in only one way; on the other hand, the usual abstract model checking techniques refine abstraction step by step until the verification succeeds.

Technically, closest to our type system are that of Igarashi and Kobayashi [23] and that of Chaki, Rajamani, and Rehof [7]. Those type systems are developed for checking the communication behavior of a process, but by viewing a set of channels as a resource, it is possible to use those type systems directly for the resource usage analysis. We summarize below similarities and differences between those type systems [7, 23] and the type system in the present paper.

(1) Whether types are supplied by the programmer or inferred automatically: Types are inferred automatically in Igarashi and Kobayashi's generic type [23] and the type system of the present paper, but the type of each channel must be annotated with in Chaki et al.'s type system. The annotated type contains information about how the values (channels, in particular) sent along the channel are used by senders and receivers, and that information is used to make the type checking process compositional. For the purpose of the resource usage analysis discussed here, we think that it is a burden for programmers to declare how channels are going to be used, since their primary concern is how resources are accessed, not channels. Ideal would be to allow the user to specify some types and infer the others, like in ML. For that purpose, we need to develop an algorithm to check the conformance $A \leq B$ of an inferred type A to a declared type B. That seems generally harder to decide than the trace inclusion constraint $\mathbf{traces}_x(A) \subseteq \Phi$, but we expect to be able to develop a sound algorithm by properly restricting the language of declared types.

(2) The languages used as behavioral types: All the three type systems use a fragment of CCS as the language of types to check cross-channel dependency of communications. The types in Igarashi and Kobayashi's generic type system for the π-calculus [23], however, lacks hiding, so that their type system cannot be applied to obtain precise information about resource usage. In fact, their analysis would fail even for the program in Example 3.1. Chaki et al.'s type system does use hiding, but lacks renaming as a constructor. Without the renaming constructor, the most general type does not necessarily exist, which hinders automatic type inference (recall Remark 3.6).

(3) Algorithms for checking the conformance of inferred types with respect to specifications: In Igarashi and Kobayashi's generic type system, how to check conformance of inferred types with respect to the user-supplied specifications was left open, and only suggested that it could be solved as a model checking problem. In Chaki et al.'s type system [7], the conformance is expressed as $A \models F$ (for checking the global behavior, where F is an LTL-formula) and $A \leq A'$ (for checking the conformance of declared types with respect to inferred types). In their type checker PIPER [7], those conditions are verified using SPIN, so that A is restricted to a finite-state process. Corresponding to the conformance check of the above work is the check of trace inclusion constraints $\mathbf{traces}_x(A) \subseteq \Phi$. Our algorithm based on the reduction to Petri nets works even when A has infinite states.

(4) The guaranteed properties: Both Igarashi and Kobayashi's generic type [23] and the extended type system of the present paper can guarantee a certain lock-freedom property, that necessary communications or resource accesses are eventually performed (unless the whole process diverges), while Chaki et al.'s type system and the type system in Section 3.2 of the present paper do not. The guaranteed properties depend on the choice of the language of behavioral types and the subtyping relation. In the latter type systems, the ordinary simulation relation is used, so that a process's type describes only an upper-bound of the possible behavior of the process, not a lower-bound of the behavior like a certain resource access is eventually performed. Rajamani et al. [18, 58] recently introduced a more elaborate notion of simulation relation called "stuck-free conformance." Even with the stuck-free conformance relation, however, their type system [7] still cannot guarantee the lack of deadlock-freedom of a process. On the other hand, by relying on an external analysis to check deadlock-freedom, the extension in Section 3.4 keeps the typing rules and the subtyping relation simple, while achieving the guarantee that necessary resource accesses are eventually performed unless the whole process diverges.

Kobayashi's type systems for deadlock-freedom and livelock-freedom [31, 33, 35] and its implementation [28] form the basis of the extended type systems for partial and total liveness properties discussed in Section 3.4, and are used for producing well-annotated programs. Conversely, the behavioral types introduced in this paper can be used to refine the type systems for deadlock-freedom and livelock-freedom. Yoshida and Honda have also studied type systems that can guarantee certain lock-freedom properties [22, 68, 69]. So, their type systems can also be used for checking whether programs are well-annotated in the sense of Section 3.4.

In Section 3.4, we have utilized the existing analysis for deadlock-freedom to enhance the result of the resource usage analysis. Other type systems for concurrent languages may also be useful. For example, the type system for atomicity [9] can be used to infer the atomicity of a sequence of actions in a source program. By using the atomicity information, we may be able to reduce the state space of behavioral types and check the trace inclusion relation $\mathbf{etraces}_x(A) \subseteq \Phi$ more efficiently.

4.4 Static analysis for practical software

Much effort is being paid [1, 13, 15, 16, 47, 48, 66] to statically analyze large-scale practical software to detect bugs. Those researches are related to ours because our final aim is also to provide a verification framework for practical concurrent software. Among those researches, verification based on type qualifiers by Foster et al. [16], an error detection method using SAT solvers [66] and an inconsistency inference by Dillig et al. [13] do not deal with concurrency and interrupts. Though researches on race detection by Naik et al. [47, 48] deal with concurrency, they do not

deal with interrupts. The race-freedom verification by Flanagan , Abadi and Freund [1, 15] also deals with concurrency. However, their framework lacks non-block-structured mutex primitives and interrupts. In addition, some of those researches [13, 47, 48, 66] sacrifice soundness, so that those analyses do not guarantee the security properties which each analysis checks.

Chapter 5

Conclusion

We have presented type-based verification methods for deadlock-freedom and resource usage of concurrent programs with primitives that are often used in practical programs: non-block-structured mutex primitives, mutable references and interrupts for the deadlock-freedom verification and dynamic creation of resources and channels for the resource usage verification. Though those primitives are heavily used in practical software, they have not been dealt with by existing researches.

The presented deadlock-freedom verification guarantees that there are no circular dependencies among locking/unlocking operations and that an acquired lock is released exactly once. The former property is guaranteed by assigning a lock level to each lock type. In order to guarantee the latter property in the presence of aliasing and the possibility of race on a reference to a mutex, we use the idea of obligations/capabilities and rational-numbered ownerships. We have also implemented a prototype deadlock-freedom verifier and conducted an experiment.

The type system for resource usage analysis guarantees safety and partial liveness by abstracting the behavior of processes using behavioral types. We have also developed a sound (but incomplete because of the last phase for deciding the trace inclusion relation $\mathbf{traces}_x(A) \subseteq \Phi$) algorithm for it in order to liberate programmers from the burden of writing complex type annotations. We have also implemented a prototype resource usage analyzer based on the algorithm.

The most important future work is to assess the effectiveness of our verification methods by further experiments, and to improve our framework based on the result. For the deadlock-freedom analysis, we have shown the result of an experiment in Section 2.4. The experiment has revealed several weaknesses of the current type system such as the necessity of polymorphism on interrupt flags. After making extension to deal with those problems, we plan to conduct further experiments using larger software such as device drivers and operating system kernels. For the resource usage analysis, there is still a gap between our current framework and practical software written in, for example, the C language. We plan to extend our framework to conduct

the verification of practical software.

Another future work is to develop more user-friendly verifiers. The current implementations of the presented analyses only report an error when the verification fails. Considering applying our verification to large software, the verifier needs to show information to help programmers find the cause of an error.

We also consider refining the calculus we have presented in Section 2.1 to model actual behavior of software more precisely. For example, **disable_int** of the current calculus does not block, though the interrupt disabling primitives used in the Linux kernel may block when an interrupt handler is running. We may need to add a primitive for obtaining current interrupt state to our calculus to express the behavior of `spin_lock_irqsave` and `spin_unlock_irqrestore` primitives used in the Linux kernel.

We also plan to develop a verification method for other crucial safety properties such as race-freedom and atomicity.

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] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods 2004*, volume 2999 of *Springer-Verlag*, pages 1–20, 2004.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antonie Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.

[5] 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.

[6] John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium (SAS 2003)*, pages 55–72, 2003.

[7] Sagar Chaki, Sriram Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 45–57, 2002.

[8] 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.

[9] Shaz Qadeer Cormac Flanagan. A type and effect system for atomicity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[12] Robert DeLine and Manuel Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[14] Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.

[15] 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.

[16] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.

[17] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[18] Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In *CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer-Verlag, 2004.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, June 2005.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, pages 262–274, 2003.

[21] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[22] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2002.

[23] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[24] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005. Preliminary summary appeared in Proceedings of POPL 2002.

[25] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[26] 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.

[27] Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Proceedings of the Fifth ASIAN Symposium on Programming Languages and Systems*, November 2007.

[28] Naoki Kobayashi. Typical: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.

[29] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[30] Naoki Kobayashi. Quasi-linear types. Technical Report 98-02, Department of Information Science, University of Tokyo, 1998. Available through `http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html`.

[31] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

[32] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.

[33] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[34] 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.

[35] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, August 2000.

[36] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. Technical Report TR00-01, Dept. Info. Sci., Univ. of Tokyo, January 2000. Available from `http://www.kb.cs.titech.ac.jp/~kobayasi/`. A summary has appeared in Proceedings of CONCUR 2000, Springer LNCS1877, pp.489-503, 2000.

[37] Naoki Kobayashi, Kohei Suenaga, and Lucian Wischik. Resource usage analysis for the π-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[38] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective-Caml system, release 3.10: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, France, May 2007. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[39] 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.

[40] Kim Marriott, Peter J. Stuckey, and Martin Sulzmann. Resource usage verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, volume 2895 of *Lecture Notes in Computer Science*, pages 212–229, 2003.

[41] 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.

[42] Laurent Mauborgne. ASTRÉE: Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–

392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004.

[43] Ernst W. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13(3):441–461, 1984.

[44] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[45] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[46] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[47] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, January 2007.

[48] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006.

[49] George Necula, Scott McPeak, Westley Weimer, Ben Liblit, and Matt Harren. *CIL – Infrastructure for C Program Analysis and Transformation*. California, USA. `http://manju.cs.berkeley.edu/cil/`.

[50] George C. Necula. Proof-carrying code. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[51] Nicholas Nguyen and Julian Rathke. Typed static analysis for concurrent, policy-based, resource access control. draft.

[52] 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.

[53] Elisabeth Pelz. Closure properties of deterministic petri nets. In *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 371–382. Springer-Verlag, 1987.

[54] Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java virtual machine language. In *Proceedings of*

the 2007 ACM SIGPLAN/SIGBED conference on languages compilers and tools, page 10, 2007.

[55] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[56] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, February 2002.

[57] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* MIT Press, December 2004.

[58] Sriram K. Rajamani and Jakob Rehof. Models for contract conformance. In *ISOLA2004, First International Symposium on Leveraging Applications of Formal Methods*, 2004.

[59] Christian Skalka and Scott Smith. History effects and verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128, 2004.

[60] Kohei Suenaga and Naoki Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Proceedings of 16th European Symposium on Programming (ESOP 2007)*, pages 490–504, March 2006.

[61] Tachio Terauchi. *Types for Deterministic Concurrency.* PhD thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, August 2006.

[62] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.

[63] Willem Visser, Klaus Havelund, and Seungjoon Park. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[64] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods.* North Holland, 1990.

[65] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 214–227, 1999.

[66] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3):1–43, May 2007.

[67] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 27–40, January 2001.

[68] Nobuko Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–387. Springer-Verlag, 1996.

[69] Nobuko Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. Technical Report 2002-20, MSC Technical Report, University of Leicester, April 2002.

Appendix A

Proofs of Lemma 2.2 and 2.3

Lemma A.1 (Weakening) *If* $\Gamma \vdash M : \tau \ \& \ \varphi$ *and* $\Gamma' \supseteq \Gamma$*, then* $\Gamma' \vdash M : \tau \ \& \ \varphi$*.*

Proof A.1 *By induction on the derivation of* $\Gamma \vdash M : \tau \ \& \ \varphi$*.*

Lemma A.2 *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* $\tau = \tau'$ **ref** *or* $\tau = \mathbf{lock}(lev)$*, then* v *is a variable.*

Proof A.2 *Case analysis on the rule use for deriving* $\Gamma \vdash v : \tau \ \& \ \emptyset$*.*

Lemma A.3 (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 A.3 *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 A.2,* $v_i = y$ *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 2.2. From $(\widetilde{D}, H, L, I, M) \rightarrow (\widetilde{D}', H', L', I', M')$, $M = E[M_1]$ and $M' = E[M_1']$ for some E, M_1 and M_1'. We perform case analysis on the rule used for deriving $(\widetilde{D}, H, L, I, E[M_1]) \rightarrow (\widetilde{D}', H', L', I', E[M_1'])$. We examine only important cases.

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

- (E-Unlock): $L' = L[x \mapsto \textbf{released}]$ and $M' = E[v]$. Thus, it is sufficient to show that E does not contain **in_sync** x, which immediately follows from $\mathit{wellformed}(L, I, E[M_1])$ and $L(x) = \textbf{acquired}$.

- (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 2.3. We prove the following proposition which is stronger than Lemma 2.3.

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

From $(\widetilde{D}, H, L, I, M) \rightarrow (\widetilde{D}', H', L', I', M')$, 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 $(\widetilde{D}, H, L, I, M) \rightarrow (\widetilde{D}', H', L', I', M')$. 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 A.3.

– Case (E-Interrupt): In this case,

* (1) $M_1 = M_1' \rhd M_2'$ and
* (2) $I = \mathbf{enabled}$

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) $\Gamma \vdash M_2' : \mathbf{unit} \ \& \ (lev_2, lev_2')$
* (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 A.1, 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.

Appendix B

Proof of Lemma 2.4

This chapter proves Lemma 2.4.

Lemma B.1 *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 B.2 *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 B.3 *If* $wellformed(L, I, E[M_1 \blacktriangleleft_{M'} M_2])$*, then* M_1 *does not contain* **in_disable_int**.

Proof B.1 *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* $E[M_1 \blacktriangleleft_{M'} M_2] = E''[\textbf{in\_disable\_int}\ M''']$ *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 B.4 *If* $\Gamma, x : \textbf{lock}(lev) \vdash M_1 : \tau \ \& \ (lev_1, lev_2)$ *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 $M_1 = E[\textbf{in\_sync}\ x\ \textbf{in}\ M']$ or $M_1 = E[M'_1 \blacktriangleleft_{M'} M'_2]$ for some E, M'_1, M' and M'_2. Here, E does not contain **in_disable_int** E' from Lemma B.3. 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 2.4. 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 B.1, 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 B.2, 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 B.1 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 B.1, 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 B.2, $M_2 = v$ 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 B.1, 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 $\neg deadlocked(L, M)$.

Appendix C

Properties of the Subtyping Relation

This section states and proves the properties of the subtyping relation, which are used in the proof of type soundness (Theorems 3.1 and 3.3, in particular the proofs of the lemmas in Appendix D), and in the type inference algorithm described in Section 3.3 (in particular, for transforming constraints on behavioral types).

Actually, there are two subtyping relations; the basic one in Definition 3.10 and the extended one in Definition 3.20. Since the proofs are almost the same, we state and prove the properties of the basic and extended ones simultaneously. In a few places, we have an additional condition to check for the extended case. Such places will be marked by "**Extended case only**." When we are discussing the basic case, attributes attached to actions should be ignored. We also omit them even for the extended case when they are not important.

Lemma C.1 (Simulation relation)

1. *The subtyping relation is reflexive and transitive.*
2. *(Simulation-up-to) Let $\mathcal{R}$ be a relation on behavioral types such that whenever $A_1 \mathcal{R} A_2$ then (i) $A_1 \stackrel{l}{\rightarrow} A_1'$ implies $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_1' \mathcal{R} \leq A_2'$ for some A_2' and (ii) disabled(A_1, S) implies disabled(A_2, S). Then $\mathcal{R} \subseteq \leq$. Condition (ii) is required only for the extended case.*

Proof C.1 ***Part 1*** *is trivial by the definition. To show* ***Part 2****, suppose $\mathcal{R}$ is a simulation up to. We show that $\mathcal{R}' = (\mathcal{R} \leq) \cup \mathcal{R}$ is a simulation, i.e., whenever $A_1 \mathcal{R}' A_2$, (i) $A_1 \stackrel{l}{\longrightarrow} A_1'$ implies $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_1' \mathcal{R}' A_2'$ for some A_2' and (**Extended case only**) (ii) disabled(A_1, S) implies disabled(A_2, S). Suppose $A_1 \mathcal{R}' A_2$. The case where $A_1 \mathcal{R} A_2$ is trivial by the definition of the simulation-up-to. To check the other case, suppose $A_1 \mathcal{R} A_3 \leq A_2$. To show (i), suppose also that $A_1 \stackrel{l}{\longrightarrow} A_1'$. Since $\mathcal{R}$ is a simulation up to, there exists A_3' such that $A_3 \stackrel{l}{\Longrightarrow} A_3'$ and $A_1' \mathcal{R} \leq A_3'$. By $A_3 \stackrel{l}{\Longrightarrow} A_3'$ and $A_3 \leq A_2$, we have A_2' such that $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_3' \leq A_2'$. Since $\leq$ is transitive, we have $A_1' \mathcal{R} \leq A_2'$, which implies $A_1' \mathcal{R}' A_2'$.*

Extended case only: *To show (ii), suppose disabled*(A_1, S)*. Since* $\mathcal{R}$ *is a simulation up to, we have disabled*(A_3, S)*, which implies disabled*(A_2, S)*.*

Lemma C.2 (Structural congruence)

1. $A|\mathbf{0} \approx A$
2. $A|B \approx B|A$
3. $A|(B|C) \approx (A|B)|C$
4. $A \oplus B \approx B \oplus A$
5. $A \oplus (B \oplus C) \approx (A \oplus B) \oplus C$
6. $*A \approx A|*A$
7. $(\nu x)(A|B) \approx (\nu x)A \,|\, B$ *if* $x \notin \mathbf{FV}(B)$
8. $(\nu x)(A \oplus B) \approx (\nu x)A \oplus B$ *if* $x \notin \mathbf{FV}(B)$
9. $[\mu\alpha.A/\alpha]A \approx \mu\alpha.A$

Proof C.2 *These proofs are all standard.*

We next show that $\leq$ is a precongruence. We first show it for some basic type constructors.

Lemma C.3 (Precongruence, simple cases) *If* $A \leq A'$ *then*

1. $A|B \leq A'|B'$ *if* $B \leq B'$
2. $\langle x/y\rangle A \leq \langle x/y\rangle A'$
3. $(\nu x)A \leq (\nu x)A'$
4. $A{\uparrow_S} \leq A'{\uparrow_S}$
5. $A{\downarrow_S} \leq A'{\downarrow_S}$

Proof C.3 *These follow from the fact that the following relations are all simulations-up-to.*

$$\begin{aligned}
\mathcal{R}_1 &= \{(A \,|\, B, A' \,|\, B') \mid A \leq A', B \leq B'\} \\
\mathcal{R}_2 &= \{(\langle \widetilde{y}/\widetilde{x}\rangle A, \langle \widetilde{y}/\widetilde{x}\rangle A') \mid A \leq A'\} \\
\mathcal{R}_3 &= \{((\nu x)\, A, (\nu x)\, A') \mid A \leq A'\} \\
\mathcal{R}_4 &= \{(A{\uparrow_S}, A'{\uparrow_S}) \mid A \leq A'\} \\
\mathcal{R}_5 &= \{(A{\downarrow_S}, A'{\downarrow_S}) \mid A \leq A'\}
\end{aligned}$$

We now show that $\leq$ is closed under arbitrary type constructors. $\mathbf{FTV}(B)$ below is the set of free (i.e., not bound by μ) behavioral type variables.

Lemma C.4 (Precongruence, general cases) *If $A \leq A'$ and $\mathbf{FTV}(B) \subseteq \{\alpha\}$, then $[A/\alpha]B \leq [A'/\alpha]B$.*

Proof C.4 *Let $\mathcal{R} = \{([A/\alpha]B,\, [A'/\alpha]B)\}$. We will prove (i) if $[A/\alpha]B \xrightarrow{l} B_1$ then $[A'/\alpha]B \stackrel{l}{\Longrightarrow} B'_1$ with $B_1\ \mathcal{R} \leq\ B'_1$, by induction on the derivation of $[A'/\alpha]B \xrightarrow{l} B'_1$. We will also prove (ii) disabled$([A/\alpha]B, S)$ implies disabled$([A'/\alpha]B, S)$, by induction on the structure of B in the extended case. In other words, $\mathcal{R}$ is a simulation-up-to. Hence (Lemma C.1.2) it is in $\leq$.*

We start with (i), with case analysis on the last rule used. If $B = \alpha$, then the required condition follows immediately from $A \leq A'$. So we consider the case $B \neq \alpha$ below.

1. *Case (*TR-ACT*). In this case, $B = l.B_x$, so*

$$[A/\alpha]B = l.[A/\alpha]B_x \xrightarrow{l} [A/\alpha]B_x = B_1.$$

We also have

$$[A'/\alpha]B = l.[A/\alpha]B_x \xrightarrow{l} [A'/\alpha]B_x = B'_1.$$

By construction of $\mathcal{R}$, we have $B_1\ \mathcal{R}\ B'_1\ \leq\ B'_1$ as required.

2. *Case (*TR-PAR1*). We show only the left case. $B = B_x|B_y$ and we assumed $[A/\alpha]B_x \xrightarrow{l} B_{x1}$ to make*

$$[A/\alpha]B = [A/\alpha]B_x|[A/\alpha]B_y \xrightarrow{l} B_{x1}|[A/\alpha]B_y = B_1.$$

By the induction hypothesis, $[A'/\alpha]B_x \stackrel{l}{\Longrightarrow} B'_{x1}$ with $B_{x1}\ \mathcal{R} \leq\ B'_{x1}$. (Note that α is not free in B_{x1} or B'_{x1}.) That gives

$$[A'/\alpha]B = [A'/\alpha]B_x|[A'/\alpha]B_y \stackrel{l}{\Longrightarrow} B'_{x1}|[A'/\alpha]B_y = B'_1.$$

It remains to prove $B_1\ \mathcal{R} \leq\ B'_1$. By the condition $B_{x1}\ \mathcal{R} \leq\ B'_{x1}$, there exists C such that

$$B_{x1} = [A/\alpha]C \qquad [A'/\alpha]C\ \leq\ B'_{x1}$$

So, we get:

$$\begin{aligned} B_1 &= [A/\alpha](C \,|\, B_y)\ \mathcal{R}\ [A'/\alpha](C \,|\, B_y) \\ &= [A'/\alpha]C \,|\, [A'/\alpha]B_y\ \leq\ B'_{x1} \,|\, [A'/\alpha]B_y = B'_1. \end{aligned}$$

Here, we have used Lemma C.3, Part 1.

3. *Case (*TR-PAR2*). We show only the left case. $B = B_x|B_y$ and we assumed $[A/\alpha]B_x \xrightarrow{x} B_{x1}$ and $[A/\alpha]B_y \xrightarrow{\overline{y}} B_{y1}$ to make*

$$[A/\alpha]B = [A/\alpha]B_x|[A/\alpha]B_y \xrightarrow{\{x,\overline{y}\}} B_{x1}|B_{y1} = B_1.$$

By the induction hypothesis, $[A'/\alpha]B_x \stackrel{x}{\Longrightarrow} B'_{x1}$ *and* $[A'/\alpha]B_y \stackrel{x}{\Longrightarrow} B'_{y1}$ *with* $B_{x1} \mathcal{R} \leq B'_{x1}$ *and* $B_{y1} \mathcal{R} \leq B'_{y1}$*. That gives*

$$[A'/\alpha]B = [A'/\alpha]B_x \,|\, [A'/\alpha]B_y \stackrel{\{x,\overline{y}\}}{\Longrightarrow} B'_{x1} \,|\, B'_{y1} = B'_1.$$

It remains to prove $B_1 \mathcal{R} \leq B'_1$*. From* $B_{x1} \mathcal{R} \leq B'_{x1}$ *and* $B_{y1} \mathcal{R} \leq B'_{y1}$*, there exist* C_x *and* C_y *such that*

$$\begin{array}{ll} B_{x1} = [A/\alpha]C_x & [A'/\alpha]C_x \leq B'_{x1} \\ B_{y1} = [A/\alpha]C_y & [A'/\alpha]C_y \leq B'_{y1} \end{array}$$

Hence, $B_1 = [A/\alpha](C_x \,|\, C_y) \mathcal{R} [A'/\alpha](C_x \,|\, C_y) \leq B'_1$.

4. *Cases (*Tr-Com*) and (*Tr-Or*). These cases follow immediately from the induction hypothesis.*

5. *Case (*Tr-Rep*). Then* $B = *B_x$ *and* $[A/\alpha]B = *[A/\alpha]B_x \stackrel{l}{\longrightarrow}$*.* $[A/\alpha]B \stackrel{l}{\longrightarrow} B_1$ *must have been derived from*

$$[A/\alpha](B_x \,|\, *B_x) = [A/\alpha]B_x \,|\, *[A/\alpha]B_x \stackrel{l}{\longrightarrow} B_1.$$

By the induction hypothesis, There exists B'_1 *such that* $B_1 \mathcal{R} \leq B'_1$ *and* $[A'/\alpha](B_x \,|\, *B_x) \stackrel{l}{\Longrightarrow} B'_1$*. Using (*Tr-Rep*), we get* $[A'/\alpha]B \stackrel{l}{\Longrightarrow} B'_1$ *as required.*

6. *Case (*Tr-Rec*). Then, we have* $B = \mu\beta.B_x$ *to make*

$$[A/\alpha]B = \mu\beta.[A/\alpha]B_x \stackrel{l}{\longrightarrow} B_1$$

where we assumed $[\mu\beta.[A/\alpha]B_x/\beta][A/\alpha]B_x \stackrel{l}{\longrightarrow} B_1$*. But* β *does not clash with* A *or* α *so these two substitutions swap around, giving*

$$[A/\alpha][\mu\beta.B_x/\beta]B_x \stackrel{l}{\longrightarrow} B_1.$$

By the induction hypothesis,

$$[A'/\alpha][\mu\beta.B_x/\beta]B_x \stackrel{l}{\Longrightarrow} B'_1$$

with $B_1 \mathcal{R} \leq B'_1$*. Hence*

$$[A'/\alpha]B = \mu\beta.[A'/\alpha]B_x \stackrel{l}{\Longrightarrow} B'_1$$

as required.

7. *Case (*Tr-Rename*). Then,* $B = \langle\widetilde{y}/\widetilde{x}\rangle B_x$*.* $[A/\alpha]B \stackrel{[\widetilde{y}/\widetilde{x}]l}{\longrightarrow} \langle\widetilde{y}/\widetilde{x}\rangle B_{x1} = B_1$ *must have been derived from* $[A/\alpha]B_x \stackrel{l}{\longrightarrow} B_{x1}$*. From the induction hypothesis, we get*

$$[A'/\alpha]B_x \stackrel{l}{\longrightarrow} B'_{x1} \qquad B_{x1} \mathcal{R} \leq B'_{x1}.$$

Let $B'_1 = \langle \widetilde{y}/\widetilde{x}\rangle B'_{x1}$. *It remains to prove* $B_1 \mathcal{R} \leq B'_1$. *By* $B_{x1} \mathcal{R} \leq B'_{x1}$, *there exists* C *such that*

$$B_{x1} = [A/\alpha]C \qquad [A'/\alpha]C \leq B'_{x1}.$$

So, we have:

$$\begin{aligned} B_1 &= [A/\alpha]\langle \widetilde{y}/\widetilde{x}\rangle C \mathcal{R} [A'/\alpha]\langle \widetilde{y}/\widetilde{x}\rangle C \\ &= \langle \widetilde{y}/\widetilde{x}\rangle [A'/\alpha]C \leq \langle \widetilde{y}/\widetilde{x}\rangle B'_{x1} = B'_1. \end{aligned}$$

Here, we used the fact that $\leq$ *is preserved by* $\langle \widetilde{y}/\widetilde{x}\rangle$ *(Lemma C.3, Part 2).*

8. *Cases (*Tr-Hiding*), (*Tr-Exclude*), and (*Tr-Project*): Similar to (*Tr-Rename*). We use the fact that* $\leq$ *is preserved by* ν, $\cdot\downarrow_S$, *and* $\cdot\uparrow_S$ *(Lemma C.3).*

Extended case only: *We also need to show disabled*$([A/\alpha]B, S)$ *implies disabled*$([A'/\alpha]B, S)$. *This follows by straightforward induction on the structure of* B.

Lemma C.5 (Substitution)

1. $\langle \widetilde{y}/\widetilde{x}\rangle \mathbf{0} \approx \mathbf{0}$
2. $\langle \widetilde{y}/\widetilde{x}\rangle (a.A) \approx ([\widetilde{y}/\widetilde{x}]a).\langle \widetilde{y}/\widetilde{x}\rangle A$
3. $\langle \widetilde{y}/\widetilde{x}\rangle (z^{\xi}.A) \approx ([\widetilde{y}/\widetilde{x}]z)^{\xi}.\langle \widetilde{y}/\widetilde{x}\rangle A$
4. $\langle \widetilde{y}/\widetilde{x}\rangle (A|B) \approx \langle \widetilde{y}/\widetilde{x}\rangle A \mid \langle \widetilde{y}/\widetilde{x}\rangle B$
5. $\langle \widetilde{y}/\widetilde{x}\rangle (A \oplus B) \approx \langle \widetilde{y}/\widetilde{x}\rangle A \oplus \langle \widetilde{y}/\widetilde{x}\rangle B$
6. $\langle \widetilde{y}/\widetilde{x}\rangle (*A) \approx *(\langle \widetilde{y}/\widetilde{x}\rangle A)$
7. $\langle \widetilde{y}/\widetilde{x}\rangle \langle b/a\rangle A \approx \langle [\widetilde{y}/\widetilde{x}]b/a\rangle \langle \widetilde{y}/\widetilde{x}\rangle A$ *if* $target(a) \cap \{\widetilde{x}, \widetilde{y}\} = \emptyset$
8. $\langle \widetilde{y}/\widetilde{x}\rangle (\nu z)A \approx (\nu z)(\langle \widetilde{y}/\widetilde{x}\rangle A)$ *if* $\{z\} \cap \{x, y\} = \emptyset$
9. $\langle \widetilde{y}/\widetilde{x}\rangle (A\uparrow_S) \approx (\langle \widetilde{y}/\widetilde{x}\rangle A)\uparrow_S$, *and*
   $\langle \widetilde{y}/\widetilde{x}\rangle (A\downarrow_S) \approx \langle \widetilde{y}/\widetilde{x}\rangle A\downarrow_S \approx A\downarrow_S$,
   if $S \cap \{x, y\} = \emptyset$
10. $\langle \widetilde{y}/\widetilde{x}\rangle (A\uparrow_S) \approx A\uparrow_S$, *if* $\{\widetilde{x}\} \subseteq S$

Proof C.5 *Most parts are straightforward, although* ***Part 4*** *is non-obvious in the case of labels* $\{x, \overline{y}\}$. *For Part 4, we construct a relation* $\mathcal{S} = \{(\langle \widetilde{y}/\widetilde{x}\rangle (A|B), \langle \widetilde{y}/\widetilde{x}\rangle A|\langle \widetilde{y}/\widetilde{x}\rangle B)\}$ *and prove* $\mathcal{S}$ *and* $\mathcal{S}^{-1}$ *are simulations. The interesting case is when we infer*

$$\langle \widetilde{y}/\widetilde{x}\rangle A|\langle \widetilde{y}/\widetilde{x}\rangle B \xrightarrow{\tau} \langle \widetilde{y}/\widetilde{x}\rangle A'|\langle \widetilde{y}/\widetilde{x}\rangle B'$$

from

$$A \xrightarrow{z_1} A' \qquad B \xrightarrow{\overline{z_2}} B' \qquad [\widetilde{y}/\widetilde{x}]z_1 = [\widetilde{y}/\widetilde{x}]z_2.$$

This gives

$$A|B \xrightarrow{\{z_1,\overline{z_2}\}} A'|B'.$$

Hence

$$\langle\widetilde{y}/\widetilde{x}\rangle(A|B) \xrightarrow{\{[\widetilde{y}/\widetilde{x}]z_1,[\widetilde{y}/\widetilde{x}]\overline{z_2}\}} \langle\widetilde{y}/\widetilde{x}\rangle(A'|B').$$

And hence as required

$$\langle\widetilde{y}/\widetilde{x}\rangle(A|B) \xrightarrow{\tau} \langle\widetilde{y}/\widetilde{x}\rangle(A'|B').$$

Part 9. *Here we construct* $\mathcal{S} = \{(\langle\widetilde{y}/\widetilde{x}\rangle(A\!\uparrow_S),\ (\langle\widetilde{y}/\widetilde{x}\rangle A)\!\uparrow_S)\}$ *where* S *does not clash with* $\{\widetilde{x},\widetilde{y}\}$*, and we prove that* $\mathcal{S}$ *and* $\mathcal{S}^{-1}$ *are simulations. We focus on two cases.*

1. *Suppose* $(\langle\widetilde{y}/\widetilde{x}\rangle A)\!\uparrow_S \xrightarrow{[\widetilde{y}/\widetilde{x}]l} (\langle\widetilde{y}/\widetilde{x}\rangle A')\!\uparrow_S$ *is inferred from* $A \xrightarrow{l} A'$ *and* $target([\widetilde{y}/\widetilde{x}]l)\cap S = \emptyset$*. We must infer that* $\langle\widetilde{y}/\widetilde{x}\rangle(A\!\uparrow_S) \xrightarrow{[\widetilde{y}/\widetilde{x}]l} \langle\widetilde{y}/\widetilde{x}\rangle(A'\!\uparrow_S)$*. This requires* $target(l)\cap S = \emptyset$*, which we prove as follows. It is assumed that* S *does not clash, so* $\{\widetilde{x},\widetilde{y}\}\cap S = \emptyset$*. We also have* $target([\widetilde{y}/\widetilde{x}]l)\cap S = \emptyset$*, and so* $[\widetilde{y}/\widetilde{x}](target(l))\cap S = \emptyset$*. Let* $T = target(l)$*. Suppose* $z \in T$*. Then either* $z \in \widetilde{x}$ *so* $z \notin S$*, or* $z \in \widetilde{y}$ *so* $z \notin S$*, or* $z \notin \{\widetilde{x},\widetilde{y}\}$ *so* $z \in [\widetilde{y}/\widetilde{x}]T$ *so* $z \notin S$*. In all cases* $z \notin S$*, so* $T\cap S = \emptyset$ *as required.*

2. *Suppose* $(\langle\widetilde{y}/\widetilde{x}\rangle A)\!\uparrow_S \xrightarrow{\tau} (\langle\widetilde{y}/\widetilde{x}\rangle A')\!\uparrow_S$ *is inferred from* $A \xrightarrow{l} A'$ *and* $target([\widetilde{y}/\widetilde{x}]l) \subseteq S$*. We must infer* $\langle\widetilde{y}/\widetilde{x}\rangle(A\!\uparrow_S) \xrightarrow{\tau} \langle\widetilde{y}/\widetilde{x}\rangle(A'\!\uparrow_S)$*. This requires* $target(l) \subseteq S$*, which we prove as follows. Once again let* $T = target(l)$*. We have* $\{\widetilde{x},\widetilde{y}\}\cap S = \emptyset$ *and* $[\widetilde{y}/\widetilde{x}]T \subseteq S$*. Suppose* $z \in T$*. Then* $[\widetilde{y}/\widetilde{x}]z \in [\widetilde{y}/\widetilde{x}]T$*, and* $[\widetilde{y}/\widetilde{x}]z \in S$*. Either* $z \in \widetilde{x}$ *so* $y \in S$*, which is a contradiction. Or* $z \notin \widetilde{x}$*, so* $[\widetilde{y}/\widetilde{x}]z = z \in S$*. Hence* $T \subseteq S$ *as required.*

Lemma C.6 (Exclusion and Projection)

1. $\boldsymbol{0}\!\uparrow_S \approx \boldsymbol{0}$ $\quad$ $\boldsymbol{0}\!\downarrow_S \approx \boldsymbol{0}$
2. $(a_t.A)\!\uparrow_S \approx a_t.(A\!\uparrow_S)$ $\quad$ $(a_t.A)\!\downarrow_S \approx \tau_t.(A\!\downarrow_S)$ $\quad$ *if* $target(a)\cap S=\emptyset$
3. $(a_t.A)\!\uparrow_S \approx \tau_t.A\!\uparrow_S$ $\quad$ $(a_t.A)\!\downarrow_S \approx a_t.A\!\downarrow_S$ $\quad$ *if* $target(a)\subseteq S$
4. $(z^{\xi}.A)\!\uparrow_S \approx z^{\xi}.(A\!\uparrow_S)$ $\quad$ $(z^{\xi}.A)\!\downarrow_S \approx \tau_c.A\!\downarrow_S$ $\quad$ *if* $target(z^{\xi})\cap S=\emptyset$
5. $(z^{\xi}.A)\!\uparrow_S \approx \tau_c.A\!\uparrow_S$ $\quad$ $(z^{\xi}.A)\!\downarrow_S \approx z^{\xi}.(A\!\downarrow_S)$ $\quad$ *if* $target(z^{\xi})\subseteq S$
6. $(A|B)\!\uparrow_S \approx A\!\uparrow_S \mid B\!\uparrow_S$ $\quad$ $(A|B)\!\downarrow_S \approx A\!\downarrow_S \mid B\!\downarrow_S$
7. $(A\oplus B)\!\uparrow_S \approx A\!\uparrow_S \oplus B\!\uparrow_S$ $\quad$ $(A\oplus B)\!\downarrow_S \approx A\!\downarrow_S \oplus B\!\downarrow_S$
8. $(*A)\!\uparrow_S \approx *(A\!\uparrow_S)$ $\quad$ $(*A)\!\downarrow_S \approx *(A\!\downarrow_S)$

9. $(A\uparrow_S)\uparrow_T \approx A\uparrow_{S\cup T}$ $\quad (A\downarrow_S)\downarrow_T \approx A\downarrow_{S\cap T}$

10. $A\uparrow_S \approx A$ $\quad A\downarrow_S \leq \mathbf{0}$ $\quad$ *if* $\mathbf{FV}(A)\cap S=\emptyset$

11. $A\uparrow_S \leq \mathbf{0}$ $\quad A\downarrow_S \approx A$ $\quad$ *if* $\mathbf{FV}(A)\subseteq S$

Proof C.6 *Straightforward.*

Lemma C.7 (Simulation)

1. *If* $A_1 \leq A_2$ *then* $\boldsymbol{traces}_x(A_1) \subseteq \boldsymbol{traces}_x(A_2)$ *for any* x.
2. *If* $A \xrightarrow{\{x,\overline{y}\}} A'$ *then* $A \xrightarrow{x} \xrightarrow{\overline{y}} A'$.
3. $A \leq A \oplus B$
4. $A \oplus A \leq A$
5. $A \leq A\uparrow_S \mid A\downarrow_S$
6. *If* $[B/\alpha]A \leq B$ *then* $\mu\alpha.A \leq B$
7. $B_1 \oplus B_2 \leq A$ *if and only if* $B_1 \leq A$ *and* $B_2 \leq A$

Proof C.7 *These proofs are largely standard.*

Part 1 *follows immediately from the definitions of subtyping and traces.*

Part 6. *Suppose* $[B/\alpha]A \leq B$. *Let* $\mathcal{R}$ *be*

$$\{([\mu\alpha.A/\alpha]A', [B/\alpha]A') \mid \mathbf{FTV}(A') = \{\alpha\}\}.$$

By Lemma C.3.2, It suffices to prove that $\mathcal{R}$ *is a simulation up to.*

Suppose that $[\mu\alpha.A/\alpha]A' \mathcal{R} [B/\alpha]A'$ *and* $[\mu\alpha.A/\alpha]A' \xrightarrow{l} A''$. *We show that there exists* B' *such that* $[B/\alpha]A' \stackrel{l}{\Longrightarrow} B'$ *and* $A'' \mathcal{R} \leq B'$ *by induction on the derivation of* $[\mu\alpha.A/\alpha]A' \xrightarrow{l} A''$, *with case analysis on the last rule used. We show main cases; the other cases are similar or straightforward.*

- *Case (*TR-Act*):* $[\mu\alpha.A/\alpha]A' \xrightarrow{l} A''$ *is derived from* $l.\,[\mu\alpha.A/\alpha]A_1 \xrightarrow{l} [\mu\alpha.A/\alpha]A_1$ *where* $A' = l.\,A_1$ *and* $A'' = [\mu\alpha.A/\alpha]A_1$. *Thus,* $[B/\alpha]A' = l.\,[B/\alpha]A_1 \xrightarrow{l} [B/\alpha]A_1$.
- *Case (*TR-Par1*):* $[\mu\alpha.A/\alpha]A' \xrightarrow{l} A''$ *is derived from* $[\mu\alpha.A/\alpha]A_1 \xrightarrow{l} A_1'$ *where* $A' = A_1 \mid A_2$ *and* $A'' = A_1' \mid [\mu\alpha.A/\alpha]A_2$. *By the induction hypothesis, there exists* B_1' *such that* $[B/\alpha]A_1 \stackrel{l}{\Longrightarrow} B_1'$ *and* $A_1' \mathcal{R} \leq B_1'$. *Thus, we have*

$[B/\alpha]A' \stackrel{l}{\Longrightarrow} B'_1 \,|\, [B/\alpha]A_2$. *It remains to show* $A'' = A'_1 \,|\, [\mu\alpha.A/\alpha]A_2 \;\mathcal{R}\leq\; B'_1 \,|\, [B/\alpha]A_2$. *From* $A'_1 \;\mathcal{R}\leq\; B'_1$, *we get*

$$A'_1 = [\mu\alpha.A/\alpha]C \qquad [B/\alpha]C \;\leq\; B'_1$$

for some C. *So,*

$$A'' = A'_1 \,|\, [\mu\alpha.A/\alpha]A_2 = [\mu\alpha.A/\alpha](C \,|\, A_2)$$
$$\mathcal{R}\; [B/\alpha](C \,|\, A_2) = [B/\alpha]C \,|\, [B/\alpha]A_2 \;\leq\; B'_1 \,|\, [B/\alpha]A_2$$

- *Case (*TR-PAR2*):* $[\mu\alpha.A/\alpha]A' \stackrel{\{x,\overline{y}\}}{\longrightarrow} A''$ *is derived from* $[\mu\alpha.A/\alpha]A_1 \stackrel{x}{\longrightarrow} A'_1$ *and* $[\mu\alpha.A/\alpha]A_2 \stackrel{\overline{y}}{\longrightarrow} A'_2$ *where* $A' = A_1 \,|\, A_2$ *and* $A'' = A'_1 \,|\, A'_2$. *From the induction hypothesis, there exist* B'_1 *and* B'_2 *such that* $[B/\alpha]A_1 \stackrel{x}{\Longrightarrow} B'_1$ *and* $A'_1 \;\mathcal{R}\leq\; B'_1$ *and* $[B/\alpha]A_2 \stackrel{\overline{y}}{\Longrightarrow} B'_2$ *and* $A'_2 \;\mathcal{R}\leq\; B'_2$. *Thus, we have* $[B/\alpha]A' \stackrel{\{x,\overline{y}\}}{\Longrightarrow} B'_1 \,|\, B'_2$. *From* $A'_1 \;\mathcal{R}\leq\; B'_1$ *and* $A'_2 \;\mathcal{R}\leq\; B'_2$, *we get* $A'_1 \,|\, A'_2 \;\mathcal{R}\leq\; B'_1 \,|\, B'_2$ *as required.*

- *Case (*TR-REC*):*
  - *Case* $A' = \mu\beta.A_1$*:* $[\mu\alpha.A/\alpha]A' \stackrel{l}{\longrightarrow} A''$ *is derived from*

    $$[\mu\alpha.A/\alpha][\mu\beta.A_1/\beta]A_1$$
    $$= [\mu\beta.[\mu\alpha.A/\alpha]A_1/\beta][\mu\alpha.A/\alpha]A_1 \stackrel{l}{\longrightarrow} A''.$$

    Here, we assumed without loss of generality that β *is not free in* A *and* B. *Thus, by the induction hypothesis, there exists* B' *such that*

    $$[\mu\beta.[B/\alpha]A_1/\beta][B/\alpha]A_1 = [B/\alpha][\mu\beta.A_1/\beta]A_1 \stackrel{l}{\Longrightarrow} B'$$

    and $A'' \;\mathcal{R}\leq\; B'$. *Using (*TR-REC*), we obtain* $[B/\alpha]A' = \mu\beta.[B/\alpha]A_1 \stackrel{l}{\Longrightarrow} B'$ *as required.*
  - *Case* $A' = \alpha$*:* $[\mu\alpha.A/\alpha]A'$ *is equal to* $\mu\alpha.A$. *From* $\mu\alpha.A \leq B$, *there exists* B' *such that* $B \stackrel{l}{\Longrightarrow} B'$ *and* $A'' \leq B'$ *as required.*

Extended case only: *We also need to prove that disabled*$([\mu\alpha.A/\alpha]A', S)$ *implies disabled*$([B/\alpha]A', S)$ *for any* A'. *This is proved by induction on the derivation of disabled*$([\mu\alpha.A/\alpha]A', S)$. *We show the only non-trivial case, where disabled*$([\mu\alpha.A/\alpha]A', S)$ *has been derived by using the last rule in Figure 3.7. The other cases follow immediately from the induction hypothesis.*

There are two cases to consider.

- *Case where* $A' = \alpha$*: Then,* $[\mu\alpha.A/\alpha]A' = \mu\alpha.A$ *and disabled*$(\mu\alpha.A, S)$ *must have been deduced from*

  disabled$([\mu\alpha.A/\alpha]A, S)$. *By the induction hypothesis, we have disabled*$([B/\alpha]A, S)$. *By the assumption* $[B/\alpha]A \leq B$, *we have disabled*(B, S) *as required (note that* $[B/\alpha]A' = B$ *in this case).*

- *Case where* $A' = \mu\beta.C$. *Let* C' *be* $[\mu\alpha.A/\alpha]C$. *Then,* $[\mu\alpha.A/\alpha]A' = \mu\beta.C'$, *and* $\mathit{disabled}(\mu\beta.C', S)$ *must have been derived from* $\mathit{disabled}([\mu\beta.C'/\beta]C', S)$. *Here, we note*

$$[\mu\beta.C'/\beta]C' = [\mu\alpha.A/\alpha][\mu\beta.C/\beta]C.$$

So, from the induction hypothesis, we get
$\mathit{disabled}([B/\alpha][\mu\beta.C/\beta]C, S)$, *i.e.,*

$$\mathit{disabled}([\mu\beta.[B/\alpha]C/\beta][B/\alpha]C, S).$$

By using the last rule of Figure 3.7, we get $\mathit{disabled}([B/\alpha]A', S)$ *as required.*

Appendix D

Proof of the Subject Reduction Property

In this section, we prove the subject reduction property used in the proofs of Theorems 3.1 and 3.3. As in Appendix C, we prove it for the basic and extended cases simultaneously.

Lemma D.1 (Weakening) *1. If $\Gamma \triangleright v : \sigma$ and $x \notin dom(\Gamma)$, then $\Gamma, x : \sigma' \triangleright v : \tau$.*

2. If $\Gamma \triangleright P : A$ and $x \notin \mathbf{FV}(P)$ and x not in $dom(\Gamma)$ or $\mathbf{FV}(A)$ then $\Gamma,\, x : \sigma \triangleright P : A$.

Proof D.1 *Part 1 is straightforward. Part 2 is proved by straightforward induction on the derivation of $\Gamma \triangleright P : A$.*

Lemma D.2 (Judgement substitution)

1. *(For values) If $\Gamma, \widetilde{x} : \widetilde{\sigma} \triangleright y : \sigma$ and $\Gamma \triangleright \widetilde{v} : \widetilde{\sigma}$ then $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]y : \sigma$.*

2. *(For processes) If $\Gamma, \widetilde{x} : \widetilde{\sigma} \triangleright P : A$ and $\Gamma \triangleright \widetilde{v} : \widetilde{\sigma}$ then $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]P : \langle \widetilde{v}/\widetilde{x} \rangle A$.*

Proof D.2 ***Part 1.*** *Either $y = x_i$ for some i, in which case $[\widetilde{v}/\widetilde{x}]y = v_i$ and $\sigma = \sigma_i$, so that the result follows from $\Gamma \triangleright \widetilde{v} : \widetilde{\sigma}$. Or $y \notin \widetilde{x}$, in which case $[\widetilde{v}/\widetilde{x}]y = y$ and $y : \sigma$ is in Γ. We remark that types σ never have free names.*

Part 2. *By induction on the derivation of $\Gamma \triangleright P : A$. Most cases follow straightforwardly on Lemma C.5. We consider four particular cases.*

1. *Case (*T-Sub*), where $\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright P : A$ is inferred from*

$$\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright P : A' \qquad A' \leq A$$

From the induction hypothesis, $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]P : \langle \widetilde{v}/\widetilde{x} \rangle A'$. By Lemma C.3.2 and assumption $A' \leq A$ we get $\langle \widetilde{v}/\widetilde{x} \rangle A' \leq \langle \widetilde{v}/\widetilde{x} \rangle A$, and hence as required $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]P : \langle \widetilde{v}/\widetilde{x} \rangle A$.

2. *Case (*T-NewR*), where $\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright (\mathfrak{N}^{\Phi} z)P : A\!\uparrow_{\{z\}}$ is inferred from*

$$\Gamma, \widetilde{x} : \widetilde{\tau}, z : \boldsymbol{res} \triangleright P : A \qquad \boldsymbol{traces}_z(A) \subseteq \Phi$$

Assume by alpha-renaming that z does not clash with $\widetilde{x}$ or $\widetilde{v}$. From Lemma C.5.9 we get $A{\downarrow_{\{z\}}} \approx (\langle\widetilde{v}/\widetilde{x}\rangle A){\downarrow_{\{z\}}}$, giving $\boldsymbol{traces}_z(A) = \boldsymbol{traces}_z(\langle\widetilde{v}/\widetilde{x}\rangle A)$ and hence $\boldsymbol{traces}_z(\langle\widetilde{v}/\widetilde{x}\rangle A) \subseteq \Phi$. From $\Gamma \triangleright \widetilde{v} : \widetilde{\tau}$ and Lemma D.1, we get $\Gamma, z : \boldsymbol{res} \triangleright \widetilde{v} : \widetilde{\tau}$. So, by the induction hypothesis, $\Gamma, z : \boldsymbol{res} \triangleright [\widetilde{v}/\widetilde{x}]P : \langle\widetilde{v}/\widetilde{x}\rangle A$. These two together give

$$\Gamma \triangleright (\mathfrak{N}^{\Phi} z)[\widetilde{v}/\widetilde{x}]P : (\langle\widetilde{v}/\widetilde{x}\rangle A){\uparrow_{\{z\}}}.$$

For the process $(\mathfrak{N}^{\Phi} z)[\widetilde{v}/\widetilde{x}]P$, we can push the substitution out by definition of the substitution operator and because $z \notin \{\widetilde{x}, \widetilde{v}\}$. For the behavior $(\langle\widetilde{v}/\widetilde{x}\rangle A){\uparrow_{\{z\}}}$ we use Lemma C.5.9 to push it out. Hence as required,

$$\Gamma \triangleright [\widetilde{v}/\widetilde{x}](\mathfrak{N}^{\Phi} z)P : \langle\widetilde{v}/\widetilde{x}\rangle(A{\uparrow_{\{z\}}}).$$

Extended case only: *Just replace* ***traces*** *with* ***etraces*** *in the above reasoning.*

3. *Case (*T-Out*), where $\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright \overline{z}\langle w\rangle.\, P : \overline{z}.\, (\langle\widetilde{w}/\widetilde{y}\rangle A_1 | A_2)$ is inferred from*

$$\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright P : A_2 \qquad \Gamma, \widetilde{x} : \widetilde{\tau} \triangleright \widetilde{w} : \widetilde{\sigma}$$
$$\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright z : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$$

Part 1 implies $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]\widetilde{w} : \widetilde{\sigma}$ and $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]z : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$. From the induction hypothesis, we get $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]P : \langle\widetilde{v}/\widetilde{x}\rangle A_2$. These three give

$$\Gamma \triangleright \overline{[\widetilde{v}/\widetilde{x}]z}\langle[\widetilde{v}/\widetilde{x}]\widetilde{w}\rangle.\, [\widetilde{v}/\widetilde{x}]P : \overline{[\widetilde{v}/\widetilde{x}]z}.\, (\langle[\widetilde{v}/\widetilde{x}]\widetilde{w}/\widetilde{y}\rangle A_1 | \langle\widetilde{v}/\widetilde{x}\rangle A_2)$$

For the process we push the substitution out by definition of the substitution operator. For the behavior we push it out using several parts of Lemma C.5.

4. *Case (*T-In*), where $\Gamma, \widetilde{x} : \widetilde{\tau} \triangleright z(\widetilde{y}).\, P : z.\, (A_2{\uparrow_{\{\widetilde{y}\}}})$ is inferred from*

$$\Gamma, \widetilde{y} : \widetilde{\sigma}, \widetilde{x} : \widetilde{\tau} \triangleright P : A_2 \qquad \Gamma, \widetilde{x} : \widetilde{\tau} \triangleright z : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$$
$$A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1$$

We use three deductions. First from Part 1 we get $\Gamma \triangleright [\widetilde{v}/\widetilde{x}]z : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$. Second, from assumption $A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1$ and Lemma C.3.2 we get $\langle\widetilde{v}/\widetilde{x}\rangle(A_2{\downarrow_{\{\widetilde{y}\}}}) \leq \langle\widetilde{v}/\widetilde{x}\rangle A_1$. The substitution on the right disappears because $\mathbf{FV}(A_1) \subseteq \{\widetilde{y}\}$ and we can assume by alpha-renaming that $\widetilde{y}$ does not clash with $\{\widetilde{x}, \widetilde{v}\}$. The substitution on the left can be pushed inside by Lemma C.5.9. These together give $(\langle\widetilde{v}/\widetilde{x}\rangle A_2){\downarrow_{\{\widetilde{y}\}}} \leq A_1$. And third, from the induction hypothesis we get $\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright [\widetilde{v}/\widetilde{x}]P : \langle\widetilde{v}/\widetilde{x}\rangle A_2$. These three give

$$\Gamma \triangleright [\widetilde{v}/\widetilde{x}]z(\widetilde{y}).\, [\widetilde{v}/\widetilde{x}]P : [\widetilde{v}/\widetilde{x}]z.\, ((\langle\widetilde{v}/\widetilde{x}\rangle A_2){\uparrow_{\{\widetilde{y}\}}})$$

As in the previous case we push the substitution out in the process and the behavior to get, as required,

$$\Gamma \triangleright [\widetilde{v}/\widetilde{x}](z(\widetilde{y}).\, P) : \langle\widetilde{v}/\widetilde{x}\rangle(z.\, (A_2{\uparrow_{\{\widetilde{y}\}}})).$$

Lemma D.3 (Subject-reduction)

1. *If* $\Gamma \triangleright P : A$ *and* $P \preceq Q$ *then* $\Gamma \triangleright Q : A$.
2. *(Subject-reduction) If* $P \xrightarrow{L} P'$ *and* $\Gamma \triangleright P : A$ *then* $A \stackrel{L}{\Longrightarrow} A'$ *and* $\Gamma \triangleright P' : A'$ *for some* A'.

Proof D.3 ***Part 1.*** *By induction on the derivation of* $P \preceq Q$. *Most cases use Lemma C.2. The case for* $(\nu x)\, P|Q \preceq (\nu x)\,(P|Q)$ *uses Lemma D.1. The only interesting case is that for* $(\mathfrak{N}^{\Phi} x)P|Q \preceq (\mathfrak{N}^{\Phi} x)(P|Q)$ *with* $x \notin \mathbf{FV}(Q)$. *The judgement* $\Gamma \triangleright (\mathfrak{N}^{\Phi} x)P|Q : A$ *must have been inferred from*

$$\Gamma, x : \textbf{\textit{res}} \triangleright P : A_3 \qquad \textbf{\textit{traces}}_x(A_3) \subseteq \Phi \qquad A_3{\uparrow_{\{x\}}} \leq A_1$$
$$\Gamma \triangleright Q : A_2 \qquad A_1|A_2 \leq A$$

From these and Lemma D.1, we infer

$$\Gamma, x : \textbf{\textit{res}} \triangleright P|Q : A_3|A_2.$$

By alpha-renaming assume $x \notin \mathbf{FV}(A_2)$. *By Lemmas C.6.6 and C.6.11 we get* $(A_3|A_2){\downarrow_{\{x\}}} \approx A_3{\downarrow_{\{x\}}}|A_2{\downarrow_{\{x\}}} \leq A_3{\downarrow_{\{x\}}}$, *and then by Lemma C.7.1 we get* $\textbf{\textit{traces}}_x(A_3|A_2) \subseteq \textbf{\textit{traces}}_x(A_3)$, *and so* $\textbf{\textit{traces}}_x(A_3|A_2) \subseteq \Phi$. *This gives*

$$\Gamma \triangleright (\mathfrak{N}^{\Phi} x)(P|Q) : (A_3|A_2){\uparrow_{\{x\}}}.$$

Finally $(A_3|A_2){\uparrow_{\{x\}}} \leq A_3{\uparrow_{\{x\}}}|A_2 \leq A_1 \,|\, A_2 \leq A$. *This gives as required*

$$\Gamma \triangleright (\mathfrak{N}^{\Phi} x)(P|Q) : A.$$

Extended case only: *Just replace* ***traces*** *with* ***etraces*** *in the above reasoning.*

Part 2. *By induction on the derivation of* $P \xrightarrow{L} P'$. *We show main cases. The other cases are straightforward.*

- *Case (*R-Com*): We are given*

$$\Gamma \triangleright \overline{x}\langle \widetilde{v} \rangle.\, P_1 \,|\, x(\widetilde{y}).\, P_2 : A.$$

This must have been deduced from

$$\Gamma \triangleright \overline{x}\langle \widetilde{v} \rangle.\, P_1 : A_1$$
$$\Gamma \triangleright x(\widetilde{y}).\, P_2 : A_2$$
$$A_1|A_2 \leq A. \tag{D.1}$$

$\Gamma \triangleright \overline{x}\langle\widetilde{v}\rangle.\, P_1 : A_1$ *and* $\Gamma \triangleright x(\widetilde{y}).\, P_2 : A_2$ *must have been deduced from*

$$\Gamma \triangleright P_1 : A_3 \tag{D.2}$$

$$\Gamma \triangleright x : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_4\rangle \tag{D.3}$$

$$\Gamma \triangleright v_i : \sigma_i \tag{D.4}$$

$$\overline{x}.\,(\langle\widetilde{v}/\widetilde{y}\rangle A_4 \,|\, A_3) \;\leq\; A_1 \tag{D.5}$$

and

$$\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright P_2 : A_5 \tag{D.6}$$

$$\Gamma \triangleright x : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_4\rangle$$

$$A_5{\downarrow_{\{\widetilde{y}\}}} \;\leq\; A_4 \tag{D.7}$$

$$x.\,(A_5{\uparrow_{\{\widetilde{y}\}}}) \;\leq\; A_2 \tag{D.8}$$

respectively. We must show $A \Rightarrow A'$ *and* $\Gamma \triangleright P_1|[\widetilde{v}/\widetilde{y}]P_2 : A'$ *for some* A'*. We pick some* A' *such that* $A \Rightarrow A'$ *and* $A' \geq \langle\widetilde{v}/\widetilde{y}\rangle A_4|A_3|A_5{\uparrow_{\{\widetilde{y}\}}}$*. The existence of such* A' *is guaranteed by* $A \geq \overline{x}.\,(\langle\widetilde{v}/\widetilde{y}\rangle A_4|A_3)|x.\,(A_5{\uparrow_{\{\widetilde{y}\}}}) \longrightarrow \langle\widetilde{v}/\widetilde{y}\rangle A_4|A_3|A_5{\uparrow_{\{\widetilde{y}\}}}$*, which follows from (B.1) and (B.5) and (B.8), and the definition of the subtyping relation (Definition 3.10). It remains to prove* $\Gamma \triangleright P_1|[\widetilde{v}/\widetilde{y}]P_2 : A'$*. We start with the judgment (B.6),*

$$\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright P_2 : A_5.$$

By Lemma D.2.2,

$$\Gamma \triangleright [\widetilde{v}/\widetilde{y}]P_2 : \langle\widetilde{v}/\widetilde{y}\rangle A_5.$$

Hence

$$\Gamma \triangleright P_1|[\widetilde{v}/\widetilde{y}]P_2 : A_3|\langle\widetilde{v}/\widetilde{y}\rangle A_5.$$

Therefore, the required result $\Gamma \triangleright P_1 \,|\, [\widetilde{v}/\widetilde{y}]P_2 : A'$ *follows by (*T-Sub*), if we show* $A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle A_5 \leq A'$*. It follows by:*

$$\begin{array}{rcll}
A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle A_5 & \leq & A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle(A_5{\downarrow_{\{\widetilde{y}\}}} \,|\, A_5{\uparrow_{\{\widetilde{y}\}}}) & \textit{(Lemma C.7.5)} \\
& \leq & A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle(A_5{\downarrow_{\{\widetilde{y}\}}}) \,|\, \langle\widetilde{v}/\widetilde{y}\rangle(A_5{\uparrow_{\{\widetilde{y}\}}}) & \textit{(Lemma C.5.4)} \\
& \leq & A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle(A_5{\downarrow_{\{\widetilde{y}\}}}) \,|\, A_5{\uparrow_{\{\widetilde{y}\}}} & \textit{(Lemma C.5.10)} \\
& \leq & A_3 \,|\, \langle\widetilde{v}/\widetilde{y}\rangle A_4|A_5{\uparrow_{\{\widetilde{y}\}}} & \textit{(assumption B.7 above)} \\
& \leq & A' & \textit{(the definition of } A'\textit{).}
\end{array}$$

- *Case (*R-Acc*): We are given* $\Gamma \triangleright \boldsymbol{acc}_{\xi}(x).P_1 : A$*. This must have been derived from*
  - $\Gamma \triangleright P_1 : A_1$
  - $\Gamma \triangleright x : \boldsymbol{res}$

– $x^{\xi}.A_1 \leq A$.

We have to show that

– $\Gamma \triangleright P_1 : A'$

– $A \stackrel{x^{\xi}}{\Longrightarrow} A'$.

Let A' be a behavioral type that satisfies $A \stackrel{x^{\xi}}{\Longrightarrow} A'$ and $A' \geq A_1$. Such A' is guaranteed to exist by $A \geq x^{\xi}.A_1 \stackrel{x^{\xi}}{\longrightarrow} A_1$. Then, $\Gamma \triangleright P_1 : A'$ follows from $\Gamma \triangleright P_1 : A_1$ and $A' \geq A_1$.

- *Case (*R-NewR1*): We are given $\Gamma \triangleright (\mathfrak{N}^{\Phi} x)P_1 : A$ This must have been derived from*

  – $\Gamma, x : \boldsymbol{res} \triangleright P_1 : A_1$

  – $\boldsymbol{traces}_x(A_1) \subseteq \Phi$

  – $A \geq A_1 \uparrow_{\{x\}}$.

  We have to show that there exists A' such that

  – $\Gamma \triangleright (\mathfrak{N}^{\Phi^{-\xi}} x)P_1' : A'$

  – $A \Longrightarrow A'$

  where $P_1 \stackrel{x^{\xi}}{\longrightarrow} P_1'$.

  *By the induction hypothesis, there exists A_1' that satisfies $\Gamma, x : \boldsymbol{res} \triangleright P_1' : A_1'$ and $A_1 \stackrel{x^{\xi}}{\Longrightarrow} A_1'$. Using (*Tr-Project*), we get $A_1 \downarrow_{\{x\}} \stackrel{x^{\xi}}{\Longrightarrow} A_1' \downarrow_{\{x\}}$. So, from the definition of **traces** and $\boldsymbol{traces}_x(A_1) \subseteq \Phi$, we get $\boldsymbol{traces}_x(A_1') \subseteq \Phi^{-\xi}$. By using (*T-NewR*), we get $\Gamma \triangleright (\mathfrak{N}^{\Phi^{-\xi}} x)P_1' : A_1' \uparrow_{\{x\}}$.*

  *It remains to show there exists A' such that $A_1' \uparrow_{\{x\}} \leq A'$ and $A \Longrightarrow A'$. That follows from $A \geq A_1 \uparrow_{\{x\}} \Longrightarrow A_1' \uparrow_{\{x\}}$. Here, the latter relation follows from $A_1 \stackrel{x^{\xi}}{\Longrightarrow} A_1'$ and rule (*Tr-Exclude*).*

  ***Extended case only:** Just replace **traces** with **etraces** in the above reasoning.*

- *Case (*R-SP*): This follows immediately from Part 1 and the induction hypothesis.*

Appendix E

Proofs of the Lemma for Theorem 3.3

This chapter gives a proof of the lemma "**Disabled**" used in the proof of Theorem 3.3.

Lemma E.1 (Disabled) *If well_annotated*(P) *and* $\Gamma \triangleright_{pl} P : A$ *with* $\boldsymbol{bool} \notin codom(\Gamma)$*, then* $P \not\longrightarrow$ *implies disabled*(A, S) *for any* S*.*

Proof E.1 *We first note that well_annotated*(P) *and* $P \not\longrightarrow$ *imply* $\neg active(P)$ *by the definition of well_annotated*(P)*. So, it is sufficient to show (i)*$\Gamma \triangleright_{pl} P : A$*, (ii)*$P \not\longrightarrow$*, (iii)*$\neg active(P)$*, and (iv)* $\boldsymbol{bool} \notin codom(\Gamma)$ *imply disabled*(A, S) *for any* S*. We prove this by induction on the derivation of* $\Gamma \triangleright_{pl} P : A$*, with case analysis on the last rule.*

- *Case (*T-Zero*): In this case,* $A = \boldsymbol{0}$*, so we have disabled*(A, S) *for any* S*.*
- *Case (*T-Out*): In this case,* $P = \overline{x}_t\langle\widetilde{v}\rangle.\, P_1$ *and* $A = \overline{x}_t.\, (\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)$*. Since* $\neg active(P)$*,* $t = \emptyset$*. So, we have disabled*(A, S) *for any* S*.*
- *Case (*T-In*): In this case,* $P = x_t(\widetilde{y}).\, P_1$ *and* $A = x_t.\, (A_2{\uparrow_{\{\widetilde{y}\}}})$*. Since* $\neg active(P)$*,* $t = \emptyset$*. So, we have disabled*(A, S) *for any* S*.*
- *Case (*T-Par*): In this case,* $P = P_1 \mid P_2$ *and* $A = A_1 \mid A_2$ *with* $\Gamma \triangleright_{pl} P_1 : A_1$ *and* $\Gamma \triangleright_{pl} P_2 : A_2$*. Note that* $P \not\longrightarrow$ *implies* $P_1 \not\longrightarrow$ *and* $P_2 \not\longrightarrow$*.* $\neg active(P)$ *implies* $\neg active(P_1)$ *and* $\neg active(P_2)$*. So, by the induction hypothesis, we get disabled*(A_1, S) *and disabled*(A_2, S) *for any* S*, which implies disabled*(A, S)*.*
- *Case (*T-Rep*): In this case,* $P = *P_1$ *and* $A = *A_1$*, with* $\Gamma \triangleright_{pl} P_1 : A_1$*.* $\neg active(P)$ *and* $P \not\longrightarrow$ *imply* $\neg active(P_1)$ *and* $P_1 \not\longrightarrow$*. So, by the induction hypothesis, we get disabled*(A_1, S) *for any* S*, which also implies disabled*(A, S) *as required.*
- *Case (*T-If*): This case cannot happen; by the condition (iv),* P *must be of the form* **if** ***true*** **then** P_1 **else** P_2 *or* **if** ***false*** **then** P_1 **else** P_2 *, which contradicts with* $P \not\longrightarrow$*.*

- *Case (*T-New*): In this case,* $P = (\nu x)\, P_1$, $A = (\nu x)\, A_2$, *and* $\Gamma, x : \boldsymbol{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \triangleright P_1 : A_2$. $\neg active(P)$ *and* $P \not\longrightarrow$ *imply* $\neg active(P_1)$ *and* $P_1 \not\longrightarrow$. *So, by the induction hypothesis, we get disabled*(A_2, S) *for any* S. *By the definition of disabled*$(\cdot, S)$, *we get disabled*(A, S).

- *Case (*T-Acc*): This case cannot happen, since* P *must be of the form* $\boldsymbol{acc}_\xi(x).P_1$, *which contradicts with* $P \not\longrightarrow$.

- *Case (*T-NewR*): Similar to the case for (*T-New*).*

- *Case (*T-Sub*):* $\Gamma \triangleright_{pl} P : A$ *must be derived from* $\Gamma \triangleright_{pl} P : A'$ *for some* $A' \leq A$. *By the induction hypothesis, for any* S, *we get disabled*(A', S). *By the condition* $A' \leq A$, *we have disabled*(A, S) *for any* S.

Appendix F

Computing a Basis of Behavioral Type

This section is an appendix for Section 3.3.4. Let A be a behavioral type of the form $(\nu\widetilde{y})\,B$, where B does not contain any ν-prefix. Such A can be obtained by pushing all the ν-prefixes out to the top-level, as described in Section 3.3.4. We show how to compute a basis of A below.

The constructor $\cdot\uparrow_S$ can be eliminated by running the algorithm $\mathit{ElimUp}^{\emptyset,\emptyset}(B,\emptyset)$ below.

$$
\begin{aligned}
&\mathit{ElimUp}^{F,D}(\mathbf{0},S) = \mathbf{0}\\
&\mathit{ElimUp}^{F,D}(\alpha,S) =\\
&\quad\begin{cases} A & \text{if } F(\alpha,S) = A\\ \mu\beta.\mathit{ElimUp}^{F\{(\alpha,S)\mapsto\beta\},D}(D(\alpha),S) & \text{if } (\alpha,S)\notin \mathit{dom}(F)\end{cases}\\
&\mathit{ElimUp}^{F,D}(l.A,S) = (l\backslash S).\mathit{ElimUp}^{F,D}(A,S)\\
&\mathit{ElimUp}^{F,D}(A_1 \mid A_2,S) = \mathit{ElimUp}^{F,D}(A_1,S) \mid \mathit{ElimUp}^{F,D}(A_2,S)\\
&\mathit{ElimUp}^{F,D}(A_1 \oplus A_2,S) =\\
&\quad \mathit{ElimUp}^{F,D}(A_1,S) \oplus \mathit{ElimUp}^{F,D}(A_2,S)\\
&\mathit{ElimUp}^{F,D}(*A,S) = *\mathit{ElimUp}^{F,D}(A,S)\\
&\mathit{ElimUp}^{F,D}(\langle\widetilde{y}/\widetilde{x}\rangle A,S) = \mathit{ElimUp}^{F,D}(A,\{z \mid [\widetilde{y}/\widetilde{x}]z \in S\})\\
&\mathit{ElimUp}^{F,D}(\mu\alpha.A,S) = \mu\alpha.\mathit{ElimUp}^{F\{(\alpha,S)\mapsto\alpha\},D\{\alpha\mapsto A\}}(A,S)\\
&\mathit{ElimUp}^{F,D}(A\uparrow_{S_1},S) = \mathit{ElimUp}^{F,D}(A,S\cup S_1)\\
&\mathit{ElimUp}^{F,D}(A\downarrow_{S_1},S) = \mathit{ElimUp}^{F,D}(A,S)\downarrow_{S_1}
\end{aligned}
$$

Here, $l\backslash S$ is τ if $\mathit{target}(l)\subseteq S$ and l otherwise. D keeps recursive definitions and F is a cache for avoiding repeated computation. If A does not contain ν-prefixes, $\mathit{ElimUp}^{\emptyset,\emptyset}(B,\emptyset)$ always terminates since S can range over a finite set (which is the powerset of $\mathbf{FV}(B)$). The constructor $\cdot\downarrow_S$ can be removed in the same manner.

We can further eliminate the renaming constructor $\langle \widetilde{y}/\widetilde{x} \rangle$ by using the following algorithm.

$$
\begin{array}{l}
\mathit{ElimRen}^{F,D}(\mathbf{0}, \theta) = \mathbf{0} \\
\mathit{ElimRen}^{F,D}(\alpha, \theta) = \\
\quad \begin{cases} A & \text{if } F(\alpha, \theta) = A \\ \mu\beta.\mathit{ElimRen}^{F\{(\alpha,\theta)\mapsto\beta\},D}(D(\alpha), \theta) & \text{if } (\alpha, \theta) \notin \mathit{dom}(F) \end{cases} \\
\mathit{ElimRen}^{F,D}(l.A, \theta) = \theta l.\mathit{ElimRen}^{F,D}(A, \theta) \\
\mathit{ElimRen}^{F,D}(A_1 \mid A_2, \theta) = \mathit{ElimRen}^{F,D}(A_1, \theta) \mid \mathit{ElimRen}^{F,D}(A_2, \theta) \\
\mathit{ElimRen}^{F,D}(A_1 \oplus A_2, \theta) = \\
\quad \mathit{ElimRen}^{F,D}(A_1, \theta) \oplus \mathit{ElimRen}^{F,D}(A_2, \theta) \\
\mathit{ElimRen}^{F,D}(*A, \theta) = *\mathit{ElimRen}^{F,D}(A, \theta) \\
\mathit{ElimRen}^{F,D}(\langle \widetilde{y}/\widetilde{x} \rangle A, \theta) = \mathit{ElimRen}^{F,D}(A, \theta \circ [\widetilde{y}/\widetilde{x}]) \\
\mathit{ElimRen}^{F,D}(\mu\alpha.A, \theta) = \mu\alpha.\mathit{ElimRen}^{F\{(\alpha,\theta)\mapsto\alpha\},D\{\alpha\mapsto A\}}(A, \theta)
\end{array}
$$

By applying the above algorithms to $A = (\nu\widetilde{y})\, B$, we obtain an equivalent type $A' = (\nu\widetilde{y})\, B'$, where B' does not contain any ν-prefixes, $\cdot\downarrow_S$, $\cdot\uparrow_S$, or $\langle \widetilde{y}/\widetilde{x} \rangle$. So, only elements of $\mathbf{Atoms}(B')$ defined below (modulo folding/unfolding of recursive types) can appear in transitions of B. So, $(\{\widetilde{y}\}, \mathbf{Atoms}(B'))$ forms a basis of A.

Definition F.1 *Let A be a behavioral type that does not contain any ν-prefix, $\cdot\downarrow_S$, $\cdot\uparrow_S$, or $\langle \widetilde{y}/\widetilde{x} \rangle$. The set of* atoms $\mathbf{Atoms}(A)$ *is the least set that satisfies the following conditions.*

$$
\begin{array}{l}
\mathbf{Atoms}(l.A) \supseteq \{l.A\} \cup \mathbf{Atoms}(A) \\
\mathbf{Atoms}(A_1 \mid A_2) \supseteq \mathbf{Atoms}(A_1) \cup \mathbf{Atoms}(A_2) \\
\mathbf{Atoms}(A_1 \oplus A_2) \supseteq \{A_1 \oplus A_2\} \cup \mathbf{Atoms}(A_1) \cup \mathbf{Atoms}(A_2) \\
\mathbf{Atoms}(*A) \supseteq \{*A\} \cup \mathbf{Atoms}(A) \\
\mathbf{Atoms}(\mu\alpha.A) \supseteq \{\mu\alpha.A\} \cup \mathbf{Atoms}([\mu\alpha.A/\alpha]A)
\end{array}
$$