A Hybrid Type System for Lock-Freedom of Mobile Processes[1]

Naoki Kobayashi
Tohoku University
and
Davide Sangiorgi
Università di Bologna

We propose a type system for lock-freedom in the π-calculus, which guarantees that certain communications will eventually succeed. Distinguishing features of our type system are: it can verify lock-freedom of concurrent programs that have sophisticated recursive communication structures; it can be fully automated; it is hybrid, in that it combines a type system for lock-freedom with local reasoning about deadlock-freedom, termination, and confluence analyses. Moreover, the type system is parameterized by deadlock-freedom/termination/confluence analyses, so that any methods (e.g. type systems and model checking) can be used for those analyses. A lock-freedom analysis tool has been implemented based on the proposed type system, and tested for non-trivial programs.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: Type systems, concurrency, mobile processes

1. INTRODUCTION

Verification of concurrent programs is very important. Concurrency is common in recent distributed environments or multi-processor machines, yet writing and debugging concurrent programs is hard because of non-determinism, deadlock, livelock, etc. Many methods have been proposed recently for verification of concurrent programs, including model checking, type systems, and separation logic. Although there are some promising reports such as verification of termination of several thousands lines of multi-threaded code [Cook et al. 2007], verification techniques for concurrent programs are still premature, compared with those for sequential programs, for which certain properties of millions of lines of code can be verified.

In this paper, we attack the problem of verifying concurrent programs that create threads and communication channels dynamically. More specifically, we choose the π-calculus [Sangiorgi and Walker 2001] as the target language, and consider the problem of verifying the lock-freedom property, which intuitively means that certain communications (or synchronizations) will eventually succeed (possibly under some fairness assumption). Lock-freedom is important for communication-centric computation models like the π-calculus; indeed, in the pure π-calculus, most liveness properties can be turned into the lock-freedom property. For example, the

[1]A preliminary summary of this paper has appeared in Proceedings of CAV 2008, Springer LNCS 5123, pp.80-93, 2008.

following properties can be reduced to instances of lock-freedom: Will the request of accessing a resource be eventually granted? In a client-server system, will a client request be eventually received from the server? And if so, will the server eventually send back an answer to the client? In multi-threaded programs, can a thread eventually acquire a lock? And if so, will the thread eventually release the lock? The lock-freedom property has also applications to other verification problems and program transformation, such as information flow analysis and program slicing (dependency analysis in general) [Honda et al. 2000; Honda and Yoshida 2007; Kobayashi 2005a]. Verification of liveness properties such as lock-freedom is notoriously hard in concurrency. In formalisms for mobile processes, such as the π-calculus, it is even harder, because of dynamic creation of threads and first-class channels. In these formalisms, *type systems* have emerged as a powerful means for disciplining and controlling the behaviors of the processes.

Type systems for lock-freedom include [Kobayashi 2002; 2005a; Acciai and Boreale 2008; Yoshida 2002; Yoshida et al. 2004]. An automatic verification tool, TyPiCal [Kobayashi 2005b], has been implemented based on Kobayashi's system [Kobayashi 2005a]. The expressive power of such type systems is, however, very limited. This shows up clearly, for instance, in the treatment of recursion. For example, even primitive recursive functions cannot be expressed in Kobayashi's lock-free type system, since it ignores value-dependent behaviors completely.

Related to lock-freedom is deadlock-freedom. In a system of threads, deadlock freedom is the property that the system can reduce further, if at least one thread is not terminated. A more refined form of deadlock can be given by focusing on certain special actions (prefixes, in the π-calculus): here deadlock-freedom says that the system can always reduce further if there is a thread with one special action ready for execution. The latter form of deadlock has been extensively studied by Kobayashi (see e.g., [Kobayashi 2006]); the resulting system has been implemented as a part of TyPiCal. Note that any process is deadlock-free if it is run with a divergent process. Unlike lock-freedom, deadlock-freedom is insufficient for applications to information flow analysis or program slicing.

In this paper, we tackle lock-freedom by pursuing a different route. We overcome limitations of previous type systems by combining the lock-freedom analysis with two other analyses: *deadlock-freedom* and *termination*. The result, therefore, is not a "pure" type system, but one that is *parametric* in the techniques employed to ensure deadlock-freedom and termination. Such techniques may themselves be based on type systems (and indeed in the paper we indicate such type systems, or develop them when needed), but could also use other methods (model checking, theorem provers, etc.). The parameterization allows us to go beyond certain limits of type systems, by appealing to other methods. For instance, a type system, as a form of static analysis, may have difficulties in handling value-dependent behaviors (even very simple ones), which are more easily dealt with by other methods such as model checking (see Section 7.3 for such an example).

Roughly, we use the deadlock-freedom analysis to ensure that a system can reduce if some of its expected communications have not yet occurred. We then apply a termination analysis to discharge the possibility of divergence and guarantee lock-freedom (i.e., the expected communication will indeed occur). The reasons

for appealing to deadlock-freedom are that powerful type-based analyzers exist (notably Kobayashi's systems [Kobayashi 2006]), and that deadlock-freedom is a safety property, which is easier than liveness to verify in other verification methods such as model checking.

A major challenge was to be able to apply the deadlock and termination analysis *locally*, to subsystems of larger systems. The local reasoning is particularly important for termination. A result forcing a global termination analysis would not be very useful in practice: first, valid concurrent programs may not terminate (e.g., operating systems); second, even if a program is terminating, it can be extremely hard to verify it if the program is large, particularly in languages for mobile processes such as the π-calculus that subsume higher-order formalisms such as the λ-calculus.

Very approximately, our hybrid rule for local reasoning looks as follows:

$$\frac{\models_{\mathtt{DF}} P \qquad \models_{\mathtt{Ter}} P}{\Delta \vdash_{\mathtt{LT}} P} \tag{$*$}$$

where $\models_{\mathtt{DF}} P$ and $\models_{\mathtt{Ter}} P$ indicate, respectively, that P is deadlock-free and terminating, and $\Delta \vdash_{\mathtt{LT}} P$ is a typing judgment for lock-freedom. The type environment Δ captures conditions, or "contracts", on the way P interacts with its environment, of the kind "P will eventually send a message on a" and "if P receives a message on a, then P is lock-free afterwards". Such contracts are necessary for the compositionality of the type system for lock-freedom (i.e., local reasoning on lock-freedom). We use Kobayashi's lock freedom types [Kobayashi 2005a], which refine those of the simply-typed π-calculus with *channel usages*, to express the contracts. Therefore we add rule $(*)$, as an "axiom", to the rules of Kobayashi's lock freedom type system [Kobayashi 2005a].

The contracts in Δ, however, are completely ignored—and are not guaranteed—in the premises of rule $(*)$. As a consequence, the resulting type system is unsound. In other words, knowing that P is deadlock-free and terminating is not sufficient to guarantee compositionality and local reasoning. As an example of missing information, P being terminating ensures that P itself has no infinite reductions; but it says nothing on the behaviour of P after it receives a message from other components in the system. (Indeed rule $(*)$ is only sound if applied globally, to the whole system.)

The first refinement we make for the soundness of rule $(*)$ is to replace deadlock-freedom and termination with more robust notions, which we call, respectively, *robust deadlock-freedom under* Δ, written $\Delta \models_{\mathtt{RD}} P$, and *robust termination*, written $\models_{\mathtt{RTer}} P$. These stronger notions approximately mean that P is deadlock-free or terminating after any substitution (P may be open, and therefore contain free variables), and any interaction with its environment; $\Delta \models_{\mathtt{RD}} P$ further ensures that P fulfills certain obligations in Δ. The problems of verifying robust deadlock-freedom and robust termination are more challenging than the ordinary ones, due to the additional requirements (e.g., quantifications over substitutions and transition sequences). Existing type systems for deadlock-freedom, notably [Kobayashi 2006],

do meet however the extra conditions for robust deadlock-freedom. We also show how to tune type systems for ordinary termination in a generic manner so to guarantee the stronger property of robust termination. Specifically, we isolate some conditions on a type system for termination that allow us to turn it into one for robust termination. We should stress nevertheless that $\Delta \models_{\mathtt{RD}} P$ and $\models_{\mathtt{RTer}} P$ are semantic requirements: our type system is parametric on the verification methods that guarantee them—one need not employ type systems.

Even with the above refinement of the deadlock-freedom and termination conditions, the hybrid rule ($*$) remains unsound. The reason is, roughly, the same as why assume-guarantee reasoning in concurrency often fails in the presence of circularity. In fact, the judgment $\Delta \vdash_{\mathtt{LT}} P$ can be considered a kind of assume-guarantee reasoning, where Δ expresses both assumptions on the environment and guarantees about P's behavior. To prevent circular reasoning, we add a condition $\mathit{nocap}(\Delta)$ that intuitively ensures us that P is independent of its environment, in the sense that P will fulfill its obligations (to perform certain input/output actions) without relying on its environment's behavior. (For example, suppose that there is an obligation to send a message on channel a. The process $\overline{a}[1]$, which sends 1 on a, is fine, since it fulfills the obligation with no assumption. On the other hand, the process $b(x).\,\overline{a}[x]$, which waits to receive a value on b before sending x on a, is not allowed since it fulfills the obligation only *on the assumption* that the environment will send a message on b.) This leads to the following hybrid rule:

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad \models_{\mathtt{RTer}} P \qquad \mathit{nocap}(\Delta)}{\Delta \vdash_{\mathtt{LT}} P} \qquad \text{(LT-HYB)}$$

The resulting type system guarantees that any well-typed process P is *weakly lock-free*, in the sense that if an input/output action is declared in P as an action that should succeed, and if $P \longrightarrow^* Q$, then the action has already succeeded in $P \longrightarrow^* Q$ or there is a further reduction sequence from Q in which the action will succeed. This is similar to the way in which success of passing a test is defined in fair should/must testing [Brinksma et al. 1995; Natarajan and Cleaveland 1995; Boreale et al. 1999], (and also in accordance with other definitions of similar forms of liveness for π-calculus such as [Yoshida 2002]).

For example, consider the process $\mathit{Server} \mid \mathit{Client}$, where:

$$\begin{array}{l}\mathit{Client} \stackrel{\text{def}}{=} (\nu r_1)\,(\overline{\mathit{fact}^{\circ}}[3, r_1] \mid r_1^{\circ}(x).\,P_1)\\ \mathit{Server} \stackrel{\text{def}}{=} (\nu \mathit{fact\_it})\,(*\mathit{fact}(n, r).\,\overline{\mathit{fact\_it}}[n, 1, r]\\ \qquad\qquad \mid *\mathit{fact\_it}(n, x, r).\,\mathbf{if}\ n = 0\ \mathbf{then}\ \overline{r}[x]\ \mathbf{else}\ \overline{\mathit{fact\_it}}[n-1, x \times n, r])\end{array}$$

The process *Server* creates an internal communication channel *fact_it* (used for computing factorial numbers in a tail-recursive manner), and waits on *fact* for a request $[n, r]$ on computing the factorial of n. Upon receiving a request, it returns the result on r. The process *Client* creates a fresh channel r_1 for receiving a reply, sends a request $[3, r_1]$ and then waits for the result on r_1. The client expects that the request will be eventually accepted (i.e., the output on *fact* should eventually succeed), and that the result will be eventually received (i.e., the output at *fact* and the input at r_1 should eventually succeed). To indicate these expectations, the two actions from the client are marked (symbol $\circ$). These properties cannot be verified

by Kobayashi's type system for lock-freedom [Kobayashi 2005a]. We can derive, however, $\Delta \models_{\mathtt{RD}} \mathit{Server}$ for a type environment Δ, which says that, upon receiving a request, *Server* either eventually sends a result or diverges. We can also verify that *Server* is terminating by using existing type systems for termination, such as [Deng and Sangiorgi 2006]. Thus, by using LT-Hyb above, we infer $\Delta \vdash_{\mathtt{LT}} \mathit{Server}$. Finally, with the typing rules for lock-freedom, we derive $\emptyset \vdash_{\mathtt{LT}} (\nu \mathit{fact})\,(\mathit{Server} \mid \mathit{Client})$, which says that the client's request will be eventually accepted and the result will also be eventually received. Note that, as termination and deadlock-freedom are applied locally, the above reasoning is valid even if the client is not terminating (e.g., P_1 is divergent).

We have also considered a stronger form of lock-freedom, guaranteeing that the marked actions will eventually succeed on the assumption that the scheduler is strongly fair (in the sense that if an action is enabled infinitely often, then the action will indeed succeed). We show that our type system can be strengthened to guarantee the strong lock-freedom by adding a condition of partial confluence to rule LT-Hyb above. Again, the partial confluence is only required locally; the whole program need not be confluent.

The verification framework outlined above for lock-freedom (including an automated robust termination analysis) has been implemented as an extension of TyPiCal program analysis tool (except the extension to strong lock-freedom; adding this on top of the present implementation would be tedious but not difficult). We have succeeded in automatically verifying several non-trivial programs, such as symbol tables and binary tree search. These examples are non-trivial because lists and trees are implemented as networks of processes connected by channels, and they grow dynamically (both channels and processes are dynamically created and linked). Recursive structures of the kind illustrated in these examples are common in programming languages for mobile processes (the examples in fact, were taken or inspired from Pict programs [Pierce and Turner 2000]).

The contributions of this paper are summarized as follows.

—The new type system for lock-freedom mentioned above, with a proof of its soundness. The system is hybrid (combining analyses for lock-freedom, deadlock-freedom, and termination), parameterized by any robust deadlock-freedom/termination analyzers, and allows local reasoning about termination and deadlock-freedom. The proof of the soundness of the type system is non-trivial because of the hybrid nature of the type system.

—A further extension of the type system for strong lock-freedom, by a combination with a form of confluence analysis. Again, the type system is parameterized by any analyzer for partial confluence, and enables local reasoning about confluence.

—A method for extending type systems for termination to guarantee robust termination.

—An implementation of an automated (weak) lock-freedom verifier based on the proposed method. It has been successfully tested on non-trivial examples.

The rest of this article is structured as follows. Section 2 introduces the target language of our type system, and gives formal definitions of deadlock-freedom, lock-freedom, and robust termination. Section 3 introduces the new type system,

obtained by combining Kobayashi's previous type system for lock-freedom with the hybrid rules mentioned above. Section 4 proves the soundness of the type system. Section 5 discusses how to extend type systems for termination to deal with the robust termination property. Section 6 briefly reports implementation and experiments. Section 7 discusses extensions of our type system. Section 8 discusses related work and Section 9 concludes.

2. TARGET LANGUAGE

This section introduces the target language of our work: a polyadic π-calculus [Milner 1993] with conditionals.

2.1 Syntax

We write $\mathcal{L}$ for the set of *links* (also called *channels*), and $\mathcal{V}$ for the (disjoint) set of *variables*. We use meta-variables $a, b, c, \ldots$ and $x, y, z, \ldots$ for links and variables, respectively. We write $\mathcal{N}$ for the set $\mathcal{L} \cup \mathcal{V} \cup \{\mathtt{true}, \mathtt{false}\}$ of *names* (sometimes called *values*), where `true` and `false` are the usual boolean values. We use meta-variables u, v, w for names. The grammar is the following:

Definition 2.1 *(Processes).* The set of processes, ranged over by P, is defined by:

$$P ::= \mathbf{0} \mid \overline{v}^{\chi}[\widetilde{w}].\, P \mid v^{\chi}(\widetilde{y}).\, P \mid (P \,|\, Q) \mid *P \mid (\nu a)\, P \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$$

Here, χ is either $\circ$ or $\bullet$, and $\widetilde{w}$ abbreviates a possibly empty sequence $w_1, \ldots, w_n$.

The process $\mathbf{0}$ does nothing. The process $\overline{v}^{\chi}[\widetilde{w}].\, P$ sends a tuple consisting of values $\widetilde{w}$ on v, and then (after the tuple has been received by some process) behaves like P. The process $v^{\chi}(\widetilde{y}).\, P$ waits for a tuple of values on v, binds $\widetilde{y}$ to them, and then behaves like P. The annotation χ in prefixes indicates whether the action is expected to succeed (symbol $\circ$) or not (symbol $\bullet$). (In the type inference of TyPiCal these annotations are actually inferred, in the sense that if the analysis succeeds then a set of prefixes that will eventually succeed is marked, see Section 6.) We call a prefix *marked* if its annotation is $\circ$. We usually omit the $\bullet$ annotation, thus for example $a(x).P$ stands for $a^{\bullet}(x).\, P$. Process $P \,|\, Q$ executes P and Q in parallel, and $*P$ behaves like infinitely many copies of P running in parallel; $(\nu a)\, P$ creates a fresh communication channel a, and then behaves like P. The process $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ behaves like P if v is `true` and Q if v is `false`.

The prefix (νa) is a binder for link a, and the input prefix $v^{\chi}(\widetilde{y}).\, P$ is a binder for variables $\widetilde{y}$. As usual, we identify processes up to renamings of bound names/variables, and implicitly apply α-conversion. We write $\mathbf{FN}(P)$ for the set of free names (i.e., free links and variables) in P. A process P is *closed* if the set of free variables in P is empty. We often omit trailing $\mathbf{0}$, and write $\overline{v}^{\chi}[\widetilde{w}]$ for $\overline{v}^{\chi}[\widetilde{w}].\, \mathbf{0}$. We also write $\overline{v}^{\chi}.P$ and $v^{\chi}.P$ for $\overline{v}^{\chi}[\,].\, P$ and $v^{\chi}(\,).\, P$ respectively. In examples, we use an extension of the above language with natural numbers, list, etc. as they are straightforward to accommodate.

Remark 2.2. The choice operator is omitted for the sake of simplicity. We believe that the overall ideas of the hybrid type system are applicable to other variants of the π-calculus.

$$\frac{}{\Gamma \vdash_{\mathtt{ST}} \mathbf{0}} \qquad \frac{\Gamma \vdash_{\mathtt{ST}} P \qquad \Gamma \vdash_{\mathtt{ST}} Q}{\Gamma \vdash_{\mathtt{ST}} P \,|\, Q} \qquad \frac{\Gamma \vdash_{\mathtt{ST}} P}{\Gamma \vdash_{\mathtt{ST}} *P} \qquad \frac{\Gamma(v) = \mathtt{Bool} \qquad \Gamma \vdash_{\mathtt{ST}} P \qquad \Gamma \vdash_{\mathtt{ST}} Q}{\Gamma \vdash_{\mathtt{ST}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

$$\frac{\Gamma \vdash_{\mathtt{ST}} P \qquad \Gamma(v) = \sharp[\widetilde{\mathtt{S}}] \qquad \Gamma(\widetilde{w}) = \widetilde{\mathtt{S}}}{\Gamma \vdash_{\mathtt{ST}} \overline{v}^{\chi}[\widetilde{w}].\, P} \qquad \frac{\Gamma, \widetilde{y} : \widetilde{\mathtt{S}} \vdash_{\mathtt{ST}} P \qquad \Gamma(v) = \sharp[\widetilde{\mathtt{S}}]}{\Gamma \vdash_{\mathtt{ST}} v^{\chi}(\widetilde{y}).\, P} \qquad \frac{\Gamma, a : \sharp[\widetilde{\mathtt{S}}] \vdash_{\mathtt{ST}} P}{\Gamma \vdash_{\mathtt{ST}} (\nu a)\, P}$$

Fig. 1. Simple Type System

2.2 Typing

The type systems that we will propose are defined on top of the simply-typed π-calculus (ST), that we take as the basis for our work. We believe that languages of more advanced type systems could be used as basis; we preferred ST because simple and natural. The set of *simple types* is given by:

$$\mathtt{S} ::= \mathtt{Bool} \mid \sharp[\mathtt{S}_1, \ldots, \mathtt{S}_n]$$

$\sharp[\mathtt{S}_1, \ldots, \mathtt{S}_n]$ is the type of channels that are used for transmitting tuples consisting of values of types $\mathtt{S}_1, \ldots, \mathtt{S}_n$. A type judgment is of the form $\Gamma \vdash_{\mathtt{ST}} P$. A type environment Γ is a mapping from names to simple types, with the constraint that `true` and `false` are mapped to `Bool`, and that the links are mapped to channel types. $\Gamma, \widetilde{v} : \widetilde{\mathtt{S}}$ indicates the type environment obtained by extending Γ with the type assignments $\widetilde{v} : \widetilde{\mathtt{S}}$, with the understanding that for all v_i already defined in Γ it should be $\Gamma(v_i) = \mathtt{S}_i$. The typing rules are given in Figure 1.

2.3 Operational Semantics

We introduce the standard (early) labeled transition relation $P \xrightarrow{\eta} Q$ for the π-calculus. Here, η, called a transition label, is either a silent action τ (which represents an internal communication), an output action $(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$, or an input action $a[\widetilde{b}]$.

Definition 2.3 *(Transition labels).* The set of *transition labels*, ranged over by η, is given by:

$$\eta ::= \tau \mid (\nu\widetilde{c})\,\overline{a}[\widetilde{b}] \mid a[\widetilde{b}]$$

Here, $(\nu\widetilde{c})$ represents a (possibly empty) sequence $(\nu c_1) \cdots (\nu c_n)$.

$\mathbf{SN}(\eta)$, $\mathbf{FN}(\eta)$ and $\mathbf{BN}(\eta)$ are defined by:

$$\begin{array}{ll}
\mathbf{SN}(\tau) = \emptyset & \mathbf{SN}((\nu\widetilde{c})\,\overline{a}[\widetilde{b}]) = \mathbf{SN}(a[\widetilde{b}]) = \{a\} \\
\mathbf{FN}(\tau) = \emptyset & \mathbf{BN}(\tau) = \emptyset \\
\mathbf{FN}((\nu\widetilde{c})\,\overline{a}[\widetilde{b}]) = \{a, \widetilde{b}\} \setminus \{\widetilde{c}\} & \mathbf{BN}((\nu\widetilde{c})\,\overline{a}[b]) = \{\widetilde{c}\} \\
\mathbf{FN}(a[\widetilde{b}]) = \{a, \widetilde{b}\} & \mathbf{BN}(a[\widetilde{b}]) = \emptyset
\end{array}$$

We consider only transition labels η that satisfy $\mathbf{SN}(\eta) \subseteq \mathbf{FN}(\eta)$.

Definition 2.4. The labeled transition relation $\xrightarrow{\eta}$ is the least relation closed under the rules in Figure 2, plus the symmetric of the two rules for parallel composition.

A difference from the standard transition semantics is in the treatment of replication. We distinguish between replicated input processes and unrestricted replications, and ensure that a replicated input can be copied only lazily (notice the

$$\frac{}{\overline{a}^{\chi}[\widetilde{b}].\, P \xrightarrow{\overline{a}[\widetilde{b}]} P}\ (\textsc{Tr-Out}) \qquad \frac{}{a^{\chi}(\widetilde{y}).\, P \xrightarrow{a[\widetilde{b}]} [\widetilde{y} \mapsto \widetilde{b}]P}\ (\textsc{Tr-In}) \qquad \frac{}{*a^{\chi}(\widetilde{y}).\, P \xrightarrow{a[\widetilde{b}]} *a^{\chi}(\widetilde{y}).\, P \mid [\widetilde{y} \mapsto \widetilde{b}]P}\ (\textsc{Tr-RIn})$$

$$\frac{}{\textbf{if } \texttt{true} \textbf{ then } P \textbf{ else } Q \xrightarrow{\tau} P}\ (\textsc{Tr-IfT}) \qquad \frac{}{\textbf{if } \texttt{false} \textbf{ then } P \textbf{ else } Q \xrightarrow{\tau} Q}\ (\textsc{Tr-IfF})$$

$$\frac{P \xrightarrow{\eta} Q \qquad \mathbf{BN}(\eta) \cap \mathbf{FN}(R) = \emptyset}{P \mid R \xrightarrow{\eta} Q \mid R}\ (\textsc{Tr-ParL}) \qquad \frac{P_1 \xrightarrow{(\nu \widetilde{c})\, \overline{a}[\widetilde{b}]} Q_1 \qquad P_2 \xrightarrow{a[\widetilde{b}]} Q_2 \qquad \{\widetilde{c}\} \cap \mathbf{FN}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} (\nu \widetilde{c})\, (Q_1 \mid Q_2)}\ (\textsc{Tr-ComL})$$

$$\frac{P \xrightarrow{(\nu \widetilde{c})\, \overline{a}[\widetilde{b}]} Q \qquad a \in \{\widetilde{b}\} \setminus \{d, \widetilde{c}\}}{(\nu a)\, P \xrightarrow{(\nu a, \widetilde{c})\, \overline{a}[\widetilde{b}]} Q}\ (\textsc{Tr-Open}) \qquad \frac{P \xrightarrow{\eta} Q \qquad a \notin \mathbf{FN}(\eta) \cup \mathbf{BN}(\eta)}{(\nu a)\, P \xrightarrow{\eta} (\nu a)\, Q}\ (\textsc{Tr-New})$$

$$\frac{*P \mid P \xrightarrow{\eta} Q \qquad P \text{ is not an input process}}{*P \xrightarrow{\eta} Q}\ (\textsc{Tr-Rep})$$

Fig. 2. Rules of the operational semantics

difference between Tr-RIn and Tr-Rep). This distinction is required to make the robust confluence condition defined in Section 3 not too restrictive. We write $\xrightarrow{\tau}{}^{*}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$; we write $P \xrightarrow{\tau}$ and $P \xrightarrow{\tau}{}^{*}$ if there exists a process P' s.t. $P \xrightarrow{\tau} P'$ and $P \xrightarrow{\tau}{}^{*} P'$, respectively.

We extend the above transition relation to a *typed* transition relation, to show how a type environment evolves when a process performs a transition. We write $\Gamma \vdash_{\mathtt{ST}} P \xrightarrow{\eta} \Gamma' \vdash_{\mathtt{ST}} P'$ to indicate how the type environment Γ for P evolves under the transitions of P. Further, we only consider transitions well-typed under Γ; this means that, in an input, the values supplied to P should agree with the types declared in Γ. Precisely, $\Gamma \vdash_{\mathtt{ST}} P \xrightarrow{\eta} \Gamma' \vdash_{\mathtt{ST}} P'$ holds if:

(1) $\Gamma \vdash_{\mathtt{ST}} P$;

(2) $P \xrightarrow{\eta} P'$;

(3) if $\eta = \tau$ then $\Gamma = \Gamma'$; otherwise if η is an output $(\nu \widetilde{c})\, \overline{a}[\widetilde{b}]$ or an input $a[\widetilde{b}]$ and $\Gamma(a) = \sharp[\widetilde{\mathtt{S}}]$, then $\Gamma, \widetilde{b} : \widetilde{\mathtt{S}}$ is well-defined and $\Gamma' = \Gamma, \widetilde{b} : \widetilde{\mathtt{S}}$.

Note that $\Gamma \vdash_{\mathtt{ST}} P \xrightarrow{\eta} \Gamma' \vdash_{\mathtt{ST}} P'$ implies $\Gamma' \vdash_{\mathtt{ST}} P'$. We write $\Gamma_0 \vdash_{\mathtt{ST}} P_0 \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} P_k$ to mean that $\Gamma_0 \vdash_{\mathtt{ST}} P_0$, and there are $\Gamma_1, ..., \Gamma_k$ s.t. for all $i < k$ it holds that $\Gamma_i \vdash_{\mathtt{ST}} P_i \xrightarrow{\eta_{i+1}} \Gamma_{i+1} \vdash_{\mathtt{ST}} P_{i+1}$.

Remark 2.5. The reason why we use the transition semantics instead of a reduction semantics is that we need to talk about interactions of a process with the environment, for defining and reasoning about the robust termination and deadlock-freedom (the relations $\models_{\mathtt{RD}}$ and $\models_{\mathtt{RTer}}$ mentioned in Section 1).

2.4 Deadlock-Freedom and Lock-Freedom

We now define deadlock-freedom, lock-freedom, and strong lock-freedom. A prefix is *at top level* if it is not underneath another input/output prefix or underneath a replication.

Definition 2.6 *(Deadlock-freedom).* P is *deadlock-free* if, whenever $P \xrightarrow{\tau}{}^{*} Q$ and Q has at least one marked prefix at top level, then $Q \xrightarrow{\tau}$.

The above definition of deadlock-freedom is essentially the same as the one in [Kobayashi 2006]. It says that if a marked input/output is at top level, the whole process can be reduced further.

We define lock-freedom by tagging the prefix, and the transitions originating from it. Deadlock-freedom indicates only the possibility for the system to evolve further; on the other hand, lock-freedom indicates the eventual success of marked actions at top-level. In the definition of lock-freedom, we track the success of a specific action (as several marked actions may simultaneously appear at top-level) by tagging it. We then demand success for all possible taggings. We call *tagged* a process in which exactly one unguarded and unreplicated prefix—the prefix that we wish to track—has the special annotation $\Box$ (instead of $\circ$ as in the marked prefixes). Transitions of tagged processes are defined as for the untagged ones, except that the labels of transitions emanating from the tagged prefix are also tagged. For instance, we have:

$$\frac{}{a^{\Box}(\widetilde{y}).\, P \xrightarrow{a^{\Box}[\widetilde{b}]} [\widetilde{y} \mapsto \widetilde{b}]P} \qquad \frac{P_1 \xrightarrow{(\nu \widetilde{c})\, \overline{a}^{\Box}[\widetilde{b}]} Q_1 \qquad P_2 \xrightarrow{a[\widetilde{b}]} Q_2 \qquad \{\widetilde{c}\} \cap \mathbf{FN}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau^{\Box}} (\nu \widetilde{c})\, (Q_1 \mid Q_2)}$$

We call a tagged τ-transition, written $P \xrightarrow{\tau^{\Box}} P'$, a *success*.

Definition 2.7 *(Weak lock-freedom).* A tagged process P is *successful* if whenever $P \xrightarrow{\tau}{}^{*} Q$ then $Q \xrightarrow{\tau}{}^{*} \xrightarrow{\tau^{\Box}}$. (That is, no matter how P evolves, the success transition can always be taken) Given an untagged process P, the *tagging of* P is the set of tagged processes obtained from P by replacing the annotation of a marked prefix at top level with $\Box$. Process P is *(weakly) lock-free* if whenever $P \xrightarrow{\tau}{}^{*} Q$ then all processes in the tagging of Q are successful.

The above notion of lock-freedom is similar to Yoshida's linear liveness [Yoshida 2002]: The property that P *eventually answers* at x [Yoshida 2002] can be expressed as the lock-freedom of $P \mid x^{\circ}(y)$. In the definitions of deadlock and lock freedom above, the tracked prefixes are at top level. The case in which one wants to track also guarded prefixes (for instance, in lock-freedom, ensuring that any marked prefix that is not underneath a replication will eventually be consumed) can be recovered by marking also the preceding prefixes (those that are above). The resulting lock-freedom property roughly corresponds to Acciai and Boreale's notion of responsiveness [Acciai and Boreale 2008].

A sequence of transitions $\xrightarrow{\tau}$ or $\xrightarrow{\tau^{\Box}}$ is *full* if it is finite and ends with an irreducible process, or if it is infinite. A sequence of transitions is *strongly fair* if, intuitively, any τ-action that is enabled infinitely often will eventually succeed (see Kobayashi [2002] and Bidinger and Compagnoni [2009] for a formal definition of strong fairness in the π-calculus).

Definition 2.8 *(Strong lock-freedom).* P is *strongly lock-free* if whenever $P \xrightarrow{\tau}{}^{*} Q$ then every full and strongly fair transition sequence of each process in the tagging of Q contains the success transition $\xrightarrow{\tau^{\Box}}$.

We give some examples to clarify the difference between deadlock-freedom, lock-freedom, and strong lock-freedom.

EXAMPLE 2.1. Consider the following process.

$$b^\circ(\,) \mid \overline{a}[b] \mid *a(y).\,\overline{a}[y]$$

The process is deadlock-free, since a reduction on a is always enabled. It is however not lock-free, as the input on b never succeeds. □

Experts in concurrency will easily recognize the difference between weak lock-freedom and strong lock-freedom: Weak lock-freedom combines safety and liveness guarantees, by requiring that a system never reaches a state where a marked action is at top-level, but there is no sequence of τ-actions in which the marked action is consumed. On the other hand, strong lock-freedom is a purely liveness property that says that if a marked action is at top-level, the action will eventually be consumed. The example below shows the difference between weak lock-freedom and strong lock-freedom.

EXAMPLE 2.2. Consider the following process P:

$$b^\circ(\,) \mid \overline{a}[b] \mid *a(y).\,(\nu c)\,(\overline{c}[y] \mid c(y).\,\overline{y}[\,] \mid c(y).\,\overline{a}[y])$$

The rightmost subprocess $(*a(y).\,\cdots)$ receives b on a and either sends a message on b or forwards b to itself non-deterministically. Since c is freshly created everytime b is received from a, the strong fairness does not guarantee that a message is eventually sent on b, and P is therefore *not* strongly lock-free. On the other hand, however, after any number of forwardings, there is a chance for a message to be sent on b; hence, P is weakly lock-free. □

The example below was inspired by Cook et al. [2007].

EXAMPLE 2.3. Consider the following process P:

$$\begin{array}{l}\overline{s}[10]\\ \mid *f(r).\,s(x).\,(\textbf{if } x=0 \textbf{ then } \overline{r} \mid \overline{s}[0] \textbf{ else } \overline{s}[x-1] \mid \overline{f}[r])\\ \mid *g.s(x).\,\overline{s}[10]\\ \mid *(\nu a)\,(\overline{f}[a] \mid a^\circ)\\ \mid *\overline{g}\end{array}$$

There are two servers, which are listening on f and g respectively. The server listening on f makes recursive calls while decrementing the value of s, until the value of s reaches 0. When the value reaches 0, it sends a reply on r. On the other hand, the server on g simply resets the value of s to 10. The process $(\nu a)\,(\overline{f}[a] \mid a^\circ)$ is a client for the server.

The process is *weakly* lock-free, since after any number of τ-transitions, the server on f can return a message on a if it is solely scheduled. The process is, however, not *strongly* lock-free, because if requests on f and g are processed in an interleaving manner (note that it is a strongly fair scheduling), then the value of s may never reach 0. □

Example 3.11 in Section 3.4 gives another example that shows the difference between weak lock-freedom and strong lock-freedom.

3. TYPE SYSTEM FOR LOCK-FREEDOM

We introduce the type systems for weak/strong lock-freedom. They are obtained by augmenting Kobayashi's type system [Kobayashi 2005a] with hybrid rules appealing to deadlock/termination/confluence analyses. We first review Kobayashi's previous type system for (strong) lock-freedom [Kobayashi 2005a] (with some simplification) in Section 3.1. We then define robust deadlock-freedom, robust termination, and robust confluence, and introduce the hybrid rules for combining deadlock-freedom analysis, termination analysis, and confluence analysis to strengthen the lock-freedom analysis.

3.1 Review of Previous Type System for Lock-Freedom

As mentioned in Section 1, to enable local reasoning about lock-freedom in terms of deadlock and termination analyses, we need to express some contracts between a process and its environment. We reuse the type judgments of Kobayashi's lock-freedom type system [Kobayashi 2005a] to express the contracts. A type judgment is of the form $\Delta \vdash_{\mathtt{LT}} P$, where Δ is a type environment, which expresses both requirements on the behavior of P, and assumptions on its environment. Ordinary channel types are extended with *usages*, which express how each communication channel is used. For example, $\sharp_{?.!}[\mathtt{Bool}]$ describes a channel that should be first used for receiving a boolean once, and then for sending a boolean once. A channel of type $\sharp_{?}[\sharp_{!}[\mathtt{Bool}]]$ should be first used for receiving a channel once, and then the received channel should be used once for sending a boolean. (! and ? express an output and an input respectively, and "." denotes the sequential composition; the whole syntax of usages is given later.)

In order to express both assumptions on the environment (like, "a process can eventually receive a message from its environment") and guarantees by the process (like, "a process will certainly send a message"), each action (! or ?) in a usage is further annotated with *capability levels* and *obligation levels*, which range over the set of natural numbers extended with ∞. If a capability level of an action is finite, then that action can be assumed to succeed (in other words, it is assumed that its co-action will be provided by the environment) whenever it becomes ready for execution (i.e., it is at top-level). If the level is infinite (∞), then no such assumption can be made. If an obligation level of an action is finite, then that action must become ready for execution, only by relying on capabilities of *smaller* levels (thus, levels are used for expressing dependencies between capabilities and obligations). If the level is infinite (∞), then there is no such obligation. For example, the type judgment $a : \sharp^{\infty}_{?_0}[\mathtt{Bool}], b : \sharp^{1}_{!_\infty}[\mathtt{Bool}] \vdash_{\mathtt{LT}} P$ means that P has a capability of level 0 to receive a boolean on channel a (but not an obligation to receive it), and P has an obligation of level 1 to send a boolean on b. (Here, the superscript of ! or ? is the obligation level, and the subscript is the capability level.) Thus, P can be $\overline{b}[\mathtt{true}]$ or $a(x).\overline{b}[x]$, but not $a(x).\mathbf{0}$. Thanks to the abstraction of process behavior by usages, the problem of checking lock-freedom of a process is reduced to that of checking whether the usage of each channel is consistent in the sense that, for each capability of level t, there is a corresponding obligation of level less than or equal to t.

In the terminology of assume-guarantee reasoning, a capability on an action may

be understood as an *assumption* that the environment will (or, has an obligation to) do its co-action, and an obligation on an action as a *guarantee* for the environment. A lower capability level expresses a stronger assumption on the environment, while a lower obligation level expresses a stronger guarantee for the environment. To avoid a circular assume-guarantee reasoning, the condition is imposed that an obligation (or, a guarantee) can depend only on capabilities (or assumptions) of smaller levels.

To understand how usages, capabilities, and obligations can be used for compositional reasoning about lock-freedom, consider the (deadlocked) process $a^\circ(x).\,\overline{b}[x] \mid b^\circ(x).\,\overline{a}[x]$. We have the following judgment for subprocesses:

$$a : \sharp^{0}_{?^{0}_{0}}[\mathtt{Bool}], b : \sharp^{1}_{!^{1}_{\infty}}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x]$$
$$a : \sharp^{1}_{!^{1}_{\infty}}[\mathtt{Bool}], b : \sharp^{0}_{?^{0}_{0}}[\mathtt{Bool}] \vdash_{\mathtt{LT}} b^\circ(x).\,\overline{a}[x]$$

The first judgment means that the process will provide an input on a (because the obligation level of the usage of a is 0), and that the process will also provide an output on b (because the obligation level of the output on b is 1), but that the output on b being provided may depend on the assumption that the input on a will succeed (because the capability level of the input on a is smaller than the obligation level of the output on b). For the entire process, we can simply combine both type environments by combining usages pointwise:

$$a : \sharp^{0}_{?^{0}_{0} \mid !^{1}_{\infty}}[\mathtt{Bool}], b : \sharp^{1}_{!^{1}_{\infty} \mid ?^{0}_{0}}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x] \mid b^\circ(x).\,\overline{a}[x]$$

Now, the capability level of the input on a (which is 0) is smaller than the obligation level of the corresponding output on a (which is 1); this indicates a failure of assume-guarantee reasoning (the assumption made by the left subprocess is not met by the guarantee by the right subprocess). Thus, we know the process may not be lock-free. On the other hand, if we replace the subprocess in the righthand side with $\overline{a}[\mathtt{true}].\,b(x)$, then we get:

$$a : \sharp^{0}_{?^{0}_{0} \mid !^{0}_{0}}[\mathtt{Bool}], b : \sharp^{1}_{!^{1}_{1} \mid ?^{1}_{1}}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x] \mid \overline{a}[\mathtt{true}].\,b^\circ(x)$$

The capability of each action is matched by the obligation of its co-action, which implies that the process is lock-free. This is similar to the standard assume-guarantee reasoning; the employment of such reasoning in the type system (to enable fully automated, compositional reasoning), together with the mobility of the π-calculus, however, inevitably make some technical details complex.

Figure 3 summarizes (a slightly simplified version of) Kobayashi's type system for lock-freedom.

The usage $\mathbf{0}$ describes channels that cannot be used at all. The usage $?^{t_1}_{t_2}.U$ describes channels that can be first used for input, and then used according to U. Similarly, the usage $!^{t_1}_{t_2}.U$ describes channels that can be first used for output, and then used according to U. The usage $U_1 \mid U_2$ describes channels that can be used according to U_1 and U_2, possibly in parallel. The usage $*U$ describes channels that can be used according to U infinitely often. We omit choice and recursive usages [Kobayashi 2005a; 2006] for the sake of simplicity.

Type $\mathtt{Bool}$ is the type of booleans. The type $\sharp_U[\widetilde{\mathrm{L}}]$ describes channels that should be used according to U for transmitting a tuple of values of types $\widetilde{\mathrm{L}}$.

In the definition of type environments, we impose the constraint that the names

Syntax:

$$\begin{array}{rcl} U \text{ (usages)} & ::= & \mathbf{0} \mid \alpha^{t_1}_{t_2}.U \mid (U_1 \mid U_2) \mid *U \\ \alpha \text{ (actions)} & ::= & ? \mid ! \\ t \text{ (levels)} & \in & \mathbf{Nat} \cup \{\infty\} \\ \mathtt{L} \text{ (usage types)} & ::= & \mathtt{Bool} \mid \sharp_U[\widetilde{\mathtt{L}}] \\ \Delta \text{ (type environments)} & ::= & v_1 : \mathtt{L}_1, \ldots, v_n : \mathtt{L}_n \end{array}$$

Operations:

$$\begin{array}{ll} \uparrow^t \mathbf{0} = \mathbf{0} & \uparrow^t \alpha^{t_1}_{t_2}.U = \alpha^{\mathbf{max}(t,t_1)}_{t_2}.U \\ \uparrow^t (U_1 \mid U_2) = \uparrow^t U_1 \mid \uparrow^t U_2 & \uparrow^t (*U) = *\uparrow^t U \end{array}$$

$$\begin{array}{ll} \uparrow \mathbf{0} = \mathbf{0} & \uparrow \alpha^{t_1}_{t_2}.U = \alpha^{t_1+1}_{t_2}.U \\ \uparrow (U_1 \mid U_2) = \uparrow U_1 \mid \uparrow U_2 & \uparrow (*U) = *\uparrow U \end{array}$$

$$\begin{array}{lll} \uparrow \mathtt{Bool} = \mathtt{Bool} & \uparrow^t \mathtt{Bool} = \mathtt{Bool} & *\mathtt{Bool} = \mathtt{Bool} \\ \uparrow(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{\uparrow U}[\widetilde{\mathtt{L}}] & \uparrow^t(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{\uparrow^t U}[\widetilde{\mathtt{L}}] & *(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{*U}[\widetilde{\mathtt{L}}] \end{array}$$

$$\mathtt{Bool} \mid \mathtt{Bool} = \mathtt{Bool} \qquad \sharp_{U_1}[\widetilde{\mathtt{L}}] \mid \sharp_{U_2}[\widetilde{\mathtt{L}}] = \sharp_{U_1 \mid U_2}[\widetilde{\mathtt{L}}]$$

$$(*\Delta)(v) = *(\Delta(v))$$

$$(\Delta_1 \mid \Delta_2)(v) = \begin{cases} \Delta_1(v) \mid \Delta_2(v) & \text{if } v \in dom(\Delta_1) \cap dom(\Delta_2) \\ \Delta_1(v) & \text{if } v \in dom(\Delta_1) \backslash dom(\Delta_2) \\ \Delta_2(v) & \text{if } v \in dom(\Delta_2) \backslash dom(\Delta_1) \end{cases}$$

$$(v : \sharp_{\alpha^{t_o}_{t_c}}[\widetilde{\mathtt{L}}]; \Delta)(w) = \begin{cases} \sharp_{\alpha^{t_o}_{t_c}.U}[\widetilde{\mathtt{L}}] & \text{if } w = v \wedge \Delta(v) = \sharp_U[\widetilde{\mathtt{L}}] \\ \sharp_{\alpha^{t_o}_{t_c}}[\widetilde{\mathtt{L}}] & \text{if } w = v \wedge v \notin dom(\Delta) \\ \uparrow^{t_c+1} \Delta(w) & \text{if } w \in dom(\Delta) \setminus \{v\} \end{cases}$$

Subtyping:

$$\frac{}{\mathbf{Top}(\mathtt{Bool})} \qquad \frac{U \leq \mathbf{0}}{\mathbf{Top}(\sharp_U[\widetilde{\mathtt{L}}])} \qquad \frac{}{\mathtt{Bool} \leq \mathtt{Bool}} \qquad \frac{U \leq U'}{\sharp_U[\widetilde{\mathtt{L}}] \leq \sharp_{U'}[\widetilde{\mathtt{L}}]}$$

$$\frac{\mathtt{L}_i \leq \mathtt{L}'_i \text{ (for } i = 1, \ldots, m) \qquad \mathbf{Top}(\mathtt{L}_k) \text{ (for } k = m+1, \ldots, n)}{v_1 : \mathtt{L}_1, \ldots, v_m : \mathtt{L}_m, v_{m+1} : \mathtt{L}_{m+1}, \ldots, v_n : \mathtt{L}_n \leq v_1 : \mathtt{L}'_1, \ldots, v_m : \mathtt{L}'_m}$$

Typing:

$$\frac{\Delta_1 \vdash_{\mathrm{LT}} P \qquad t_c = \infty \Rightarrow \chi = \bullet}{v : \sharp_{!^0_{t_c}}[\widetilde{\mathtt{L}}]; (\Delta_1 \mid \widetilde{w} : \uparrow\widetilde{\mathtt{L}}) \vdash_{\mathrm{LT}} \overline{v}^{\chi}[\widetilde{w}].\, P} \text{ (LT-Out)}$$

$$\frac{\Delta, \widetilde{y} : \widetilde{\mathtt{L}} \vdash_{\mathrm{LT}} P \qquad t_c = \infty \Rightarrow \chi = \bullet}{v : \sharp_{?^0_{t_c}}[\widetilde{\mathtt{L}}]; \Delta \vdash_{\mathrm{LT}} v^{\chi}(\widetilde{y}).\, P} \text{ (LT-In)}$$

$$\frac{}{\emptyset \vdash_{\mathrm{LT}} \mathbf{0}} \text{ (LT-Zero)} \qquad \frac{\Delta_1 \vdash_{\mathrm{LT}} P_1 \qquad \Delta_2 \vdash_{\mathrm{LT}} P_2}{\Delta_1 \mid \Delta_2 \vdash_{\mathrm{LT}} P_1 \mid P_2} \text{ (LT-Par)}$$

$$\frac{\Delta' \vdash_{\mathrm{LT}} P \qquad \Delta \leq \Delta'}{\Delta \vdash_{\mathrm{LT}} P} \text{ (LT-Weak)} \qquad \frac{\Delta \vdash_{\mathrm{LT}} P}{*\Delta \vdash_{\mathrm{LT}} *P} \text{ (LT-Rep)}$$

$$\frac{\Delta, a : \sharp_U[\widetilde{\mathtt{L}}] \vdash_{\mathrm{LT}} P \qquad rel(U)}{\Delta \vdash_{\mathrm{LT}} (\nu a)\, P} \text{ (LT-New)}$$

$$\frac{\Delta \vdash_{\mathrm{LT}} P \qquad \Delta \vdash_{\mathrm{LT}} Q}{\Delta \mid (v : \mathtt{Bool}) \vdash_{\mathrm{LT}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \text{ (LT-If)}$$

Fig. 3. Kobayashi's type system for lock-freedom [Kobayashi 2005a]

`true` and `false` are always mapped to `Bool`, and that any links are mapped to channel types. We often omit the bindings `true`:`Bool` and `false`:`Bool`, and write $\emptyset$ for the type environment `true`:`Bool`, `false`:`Bool`.

We explain some key typing rules below. In the rule LT-In, the type environment $v:\sharp^{?0}_{\tilde{t}}[\widetilde{\mathrm{L}}];\Delta$ captures the condition that v is first used for input, and then v and other channels are used according to Δ. The obligation level of the input action on v is 0, since the input is immediately performed, without relying on any capabilities. For example, if $a:\sharp^{!1}_{\infty}[\mathtt{Bool}], b:\sharp^{!0}_{\infty}[\mathtt{Bool}], x:\mathtt{Bool} \vdash_{\mathtt{LT}} P$, then we can obtain $a:\sharp^{?0\,.\,!1}_{?2\,.\,\infty}[\mathtt{Bool}], b:\sharp^{!3}_{\infty}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^{\circ}(x).\widetilde{P}$ by using LT-In. Note that the obligation level of the output action on b has been raised to 3, since $a^{\circ}(x).P$ tries to exercise the capability of level 2 to receive a value from a, before fulfilling the obligation on b.

The rule LT-Out for output is similar: $v:\sharp^{!0}_{!\tilde{t}}[\widetilde{\mathrm{L}}];(\Delta_1 \mid \widetilde{w}:\uparrow\widetilde{\mathrm{L}})$ captures the condition that v is first used for output. The part $\widetilde{w}:\uparrow\widetilde{\mathrm{L}}$ expresses the usage of $\widetilde{w}$ by the process that receives $\widetilde{w}$. The operation $\uparrow$ ensures that the obligation level of actions on channels $\widetilde{w}$ is decreased by one when $\widetilde{w}$ is passed on v. For example, we can derive $a:\sharp^{?0}_{0}[\sharp^{!2}_{\infty}[\,]], b:\sharp^{!1}_{0}[\sharp^{!1}_{\infty}[\,]] \vdash_{\mathtt{LT}} a(x).\overline{b}[x]$, but not $a:\sharp^{?0\,.\,!0}_{?0\,.\,!0}[\sharp^{!2}_{\infty}[\,]] \vdash_{\mathtt{LT}} a(x).\overline{a}[x]$. Although x is received as a channel of type $\sharp^{!2}_{\infty}[\,]$, it has to be sent as a channel of type $\sharp^{!1}_{\infty}[\,]$, with the obligation level being decremented. This condition prevents a process from infinitely delegating obligations. While this is sufficient for ensuring (strong) lock-freedom, it is too restrictive; for example, in a recursive process $*a(n,x).(\cdots\overline{a}[n-1,x]\cdots)$, the obligation level of x must be ∞. Attempts of overcoming this limitation have led us to the hybrid type system in this paper.

In the rule LT-New, the condition $rel(U)$, which is defined below (in Definition 3.4), checks that each capability of an action is matched by an obligation of its co-action. This serves as a "sanity check" for assume-guarantee reasoning. For example, we can derive

$$b:\sharp^{!1}_{1}{}_{\mid}{}^{?1}_{1}[\mathtt{Bool}] \vdash_{\mathtt{LT}} (\nu a)\,(a^{\circ}(x).\overline{b}[x] \mid \overline{a}[\mathtt{true}].b^{\circ}(x)),$$

from

$$a:\sharp^{?0}_{0}{}_{\mid}{}^{!0}_{0}[\mathtt{Bool}], b:\sharp^{!1}_{1}{}_{\mid}{}^{?1}_{1}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^{\circ}(x).\overline{b}[x] \mid \overline{a}[\mathtt{true}].b^{\circ}(x),$$

but we cannot derive

$$b:\sharp^{!1}_{\infty}{}_{\mid}{}^{?0}_{0}[\mathtt{Bool}] \vdash_{\mathtt{LT}} (\nu a)\,(a^{\circ}(x).\overline{b}[x] \mid b^{\circ}(x).\overline{a}[x])$$

from

$$a:\sharp^{?0}_{0}{}_{\mid}{}^{!1}_{\infty}[\mathtt{Bool}], b:\sharp^{!1}_{\infty}{}_{\mid}{}^{?0}_{0}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^{\circ}(x).\overline{b}[x] \mid b^{\circ}(x).\overline{a}[x]$$

because the input obligation on a is not matched by the output obligation on a.

The rule LT-Weak allows us to replace a type environment Δ with Δ' if Δ' represents a more liberal usage of channels. For example, from $a:\sharp^{!0}_{\infty}[\mathtt{Bool}] \vdash_{\mathtt{LT}} P$, we can derive $a:\sharp^{!1}_{0}[\mathtt{Bool}] \vdash_{\mathtt{LT}} P$. The subusage $U \le U'$ used in the definition of $\Delta \le \Delta'$ means that U represents a more liberal usage of channels than U'. The definition of the subusage relation $U \le U'$ is rather complex, hence omitted; see Kobayashi [2005a]. We list here some derived rules for $U \le U'$ ($U \sim U'$ means

$U \leq U'$ and $U' \leq U$):

$$U \sim U \,|\, \mathbf{0} \qquad U_1 \,|\, U_2 \sim U_2 \,|\, U_1 \qquad U_1 \,|\, (U_2 \,|\, U_3) \sim (U_1 \,|\, U_2) \,|\, U_3$$

$$\alpha^{\infty}_{t_c}.U \leq \mathbf{0} \qquad \frac{t'_o \leq t_o \qquad t_c \leq t'_c \qquad U \leq U'}{\alpha^{t_o}_{t_c}.U \leq \alpha^{t'_o}_{t'_c}.U'} \qquad \frac{U_1 \leq U'_1 \qquad U_2 \leq U'_2}{U_1 \,|\, U_2 \leq U'_1 \,|\, U'_2}$$

The rules on the first line say that the usages form the commutative monoid with the operation $|$ and the unit element $\mathbf{0}$. The leftmost rule on the second line says that if the obligation level is infinite, then there is no obligation, so that a channel of that usage need not be used at all. The rule in the middle of the second line allows us to replace an obligation with a stronger one (i.e. an obligation of a smaller level), and a capability (or an assumption on the environment) with a weaker one.

We now define the relation $rel(U)$ by using auxiliary operations and relations. In the definitions below, $\overline{\alpha}$ denotes the dual action of α, i.e., $\overline{\alpha} =!$ if $\alpha =?$, and $\overline{\alpha} =?$ if $\alpha =!$.

Definition 3.1. The transition relation $U \xrightarrow{l_u} U'$ (where $l_u \in \{!, ?, \tau\}$) is the least relation closed under the following rules:

$$\frac{}{\alpha^{t_1}_{t_2}.U \xrightarrow{\alpha} U} \qquad \frac{*U \,|\, U \xrightarrow{l_u} U'}{*U \xrightarrow{l_u} U'} \qquad \frac{U_1 \xrightarrow{l_u} U'_1}{U_1 \,|\, U_2 \xrightarrow{l_u} U'_1 \,|\, U_2}$$

$$\frac{U_2 \xrightarrow{l_u} U'_2}{U_1 \,|\, U_2 \xrightarrow{l_u} U_1 \,|\, U'_2} \qquad \frac{U_1 \xrightarrow{!} U'_1 \qquad U_2 \xrightarrow{?} U'_2}{U_1 \,|\, U_2 \xrightarrow{\tau} U'_1 \,|\, U'_2} \qquad \frac{U_1 \xrightarrow{?} U'_1 \qquad U_2 \xrightarrow{!} U'_2}{U_1 \,|\, U_2 \xrightarrow{\tau} U'_1 \,|\, U'_2}$$

Definition 3.2 *(Capabilities).* The *input and output capability levels* of usage U, written $cap_?(U)$ and $cap_!(U)$, are defined by:

$$cap_\alpha(\mathbf{0}) = cap_\alpha(\overline{\alpha}^{t_o}_{t_c}.U) = \infty \qquad cap_\alpha(\alpha^{t_o}_{t_c}.U) = t_c$$
$$cap_\alpha(*U) = cap_\alpha(U) \qquad cap_\alpha(U_1 \,|\, U_2) = \mathbf{min}(cap_\alpha(U_1), cap_\alpha(U_2))$$

Definition 3.3 *(Obligations).* The *input and output obligation levels* of a usage U, written $ob_?(U)$ and $ob_!(U)$, are defined by:

$$ob_\alpha(\mathbf{0}) = ob_\alpha(\overline{\alpha}^{t_o}_{t_c}.U) = \infty \qquad ob_\alpha(\alpha^{t_o}_{t_c}.U) = t_o$$
$$ob_\alpha(*U) = ob_\alpha(U) \qquad ob_\alpha(U_1 \,|\, U_2) = \mathbf{min}(ob_\alpha(U_1), ob_\alpha(U_2))$$

Definition 3.4 *(Reliability).* We write $con_\alpha(U)$ when $ob_{\overline{\alpha}}(U) \leq cap_\alpha(U)$. U is consistent, written $con(U)$, if both $con_?(U)$ and $con_!(U)$ hold. A usage U is *reliable*, written $rel(U)$, if $con(U')$ holds for any U' such that $U \xrightarrow{\tau}{}^{*} U'$.

Intuitively, $cap_\alpha(U)$ represents the level of the strongest assumption (thus, the lowest capability level) made about whether a co-action of α is provided by some process, and $ob_\alpha(U)$ represents the level of the strongest obligation to do the action α. The predicate $rel(U)$ means that all the assumptions made in U are met by the corresponding obligations (or, guarantees).

EXAMPLE 3.1. Let U be $?^1_2.?^0_2 \mid !^2_1 \mid !^2_2$. Then, the set $\{U' \mid U \stackrel{\tau}{\longrightarrow}^* U'\}$ is:

$$\{U, ?^0_2 \mid \mathbf{0} \mid !^2_2,\ ?^0_2 \mid !^2_1 \mid \mathbf{0},\ \mathbf{0} \mid \mathbf{0} \mid \mathbf{0}\}.$$

$rel(U)$ holds since $con(U')$ holds for each element U' of the set above. For example, for U, $cap_\alpha(U)$ and $ob_\alpha(U)$ are calculated as follows.

$$\begin{array}{l} cap_?(U) = \mathbf{min}(cap_?(?^1_2.?^0_2), cap_?(!^2_1), cap_?(!^2_2)) = \mathbf{min}(2, \infty, \infty) = 2 \\ cap_!(U) = \mathbf{min}(cap_!(?^1_2.?^0_2), cap_!(!^2_1), cap_!(!^2_2)) = \mathbf{min}(\infty, 1, 2) = 1 \\ ob_?(U) = \mathbf{min}(ob_?(?^1_2.?^0_2), ob_?(!^2_1), ob_?(!^2_2)) = \mathbf{min}(1, \infty, \infty) = 1 \\ ob_!(U) = \mathbf{min}(ob_?(?^1_2.?^0_2), ob_?(!^2_1), ob_?(!^2_2)) = \mathbf{min}(\infty, 2, 2) = 2 \end{array}$$

$con(U)$ holds because $ob_!(U) \leq cap_?(U)$ and $ob_?(U) \leq cap_!(U)$ hold. □

EXAMPLE 3.2. Consider the following process P:

$$a(x).\,\overline{x}[\,] \mid (\nu r)\,(\overline{a}[r] \mid r^\circ(\,)).$$

It is typed as follows.

$$\dfrac{\dfrac{\dfrac{}{x : \sharp_{!^0_\infty}[\,] \vdash \overline{x}[\,]}}{a : \sharp_{?^0_\infty}[\sharp_{!^0_\infty}[\,]] \vdash a(x).\,\overline{x}[\,]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \mathbf{0}}}{a : \sharp_{!^0_0}[\sharp_{!^0_\infty}[\,]], r : \sharp_{!^1_\infty}[\,] \vdash \overline{a}[r]}}{a : \sharp_{!^\infty_0}[\sharp_{!^0_\infty}[\,]], r : \sharp_{!^1_\infty}[\,] \vdash \overline{a}[r]} \qquad \dfrac{\dfrac{}{\emptyset \vdash \mathbf{0}}}{r : \sharp_{?^0_1}[\,] \vdash r^\circ(\,)}}{r : \sharp_{?^\infty_1}[\,] \vdash r^\circ(\,)}}{a : \sharp_{!^\infty_0}[\sharp_{!^0_\infty}[\,]], r : \sharp_{!^1_\infty \mid ?^\infty_1}[\,] \vdash \overline{a}[r] \mid r^\circ(\,)}}{a : \sharp_{!^\infty_0}[\sharp_{!^0_\infty}[\,]] \vdash (\nu r)\,(\overline{a}[r] \mid r^\circ(\,))}}{a : \sharp_{?^0_\infty \mid !^\infty_0}[\sharp_{!^0_\infty}[\,]] \vdash a(x).\,\overline{x}[\,] \mid (\nu r)\,(\overline{a}[r] \mid r^\circ(\,))}$$

□

Remark 3.5. The main omission from the original type system for lock-freedom [Kobayashi 2005a] is recursion and choice on usages. The omission of those features are just for the sake of simplicity, and the new type system is sound in the presence of them. Recursion and choice on usages are necessary for automatic type inference.

3.2 Robust Deadlock-Freedom/Termination/Confluence

To enable local reasoning in the new type system for lock-freedom that we will present, we introduce a strengthening of the notions of deadlock-freedom, termination, and confluence.

3.2.1 *Robust Termination.* We first define robust termination. For the sake of simplicity, we define robust termination using simple type environments, rather than lock-freedom type environments. A substitution $\sigma = [\widetilde{w}/\widetilde{x}]$ *respects* $\Gamma = \widetilde{v} : \widetilde{\mathtt{S}}$ if $\sigma\Gamma = \widetilde{\sigma v} : \widetilde{\mathtt{S}}$ is well-defined. A substitution σ is *closing for* Γ if σ respects Γ and $\sigma\Gamma$ has no variables. A process is robustly terminating if it cannot diverge, after any sequence of transitions that conform to the base type system ST. The reason why, in the definition of robust termination, we consider only transitions that are well-typed under the ST system (as opposed, for instance, to the arbitrary untyped transitions of the operational semantics of processes) is the following. We wish to apply the analysis of robust termination only locally, to subcomponents of larger

systems. These subcomponents are typed with termination types, but they interact with the rest of the system whose components only respect the ST types.

Definition 3.6 *(Robust termination).* A process P is *terminating* if there is no infinite internal transition sequence $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \cdots$. A closed process P is *robustly terminating under* Γ if $\Gamma \vdash_{\mathtt{ST}} P$ and, for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\Gamma \vdash_{\mathtt{ST}} P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is terminating. An (open) process P is *robustly terminating under* Γ, written $\Gamma \models_{\mathtt{RTer}} P$, if σP is robustly terminating under $\sigma\Gamma$ for every closing substitution σ for Γ.

Example 3.3. Let P be $\overline{a}[c] \mid a(x).(*b.\overline{x} \mid \overline{b})$. P is terminating, as the only τ-transition sequence is

$$P \xrightarrow{\tau} \mathbf{0} \mid (*b.\overline{c} \mid \overline{b}) \xrightarrow{\tau} \mathbf{0} \mid ((*b.\overline{c} \mid \overline{c}) \mid \mathbf{0}).$$

P is however *not* robustly terminating under $a : \sharp[\sharp[\,]], b : \sharp[\,], c : \sharp[\,]$, since P has a transition:

$$P \xrightarrow{a[b]} \overline{a}[c] \mid (*b.\overline{b} \mid \overline{b})$$

and the reduct is not terminating. □

3.2.2 *Robust Deadlock-Freedom.* We say that Δ is closed if $dom(\Delta) \cap \mathcal{V} = \emptyset$. We write $rel(\mathtt{L})$ if either $\mathtt{L} = \mathtt{Bool}$, or $\mathtt{L}$ is a channel type $\sharp_U[\widetilde{\mathtt{L}}]$ and $rel(U)$. We write $rel(\Delta)$ if $rel(\Delta(v))$ for every $v \in dom(\Delta)$.

In the definition of robust deadlock-freedom below, the first condition say that P is deadlock-free when it is executed by itself, and that P either fulfills its obligations or reduces further. The other conditions say that the robust deadlock-freedom is preserved by substitutions and transitions. The relation $\Delta \xrightarrow{\eta} \Delta'$ used in the definition expresses the increase/decrease of capabilities/obligations in Δ by the transition η. For example, $a : \sharp_{?_\infty^0}[\sharp_{!_\infty^1}[\mathtt{Bool}]] \xrightarrow{a[b]} a : \sharp_{\mathbf{0}}[\sharp_{!_\infty^1}[\mathtt{Bool}]], b : \sharp_{!_\infty^1}[\mathtt{Bool}]$ holds (where the usage $\mathbf{0}$ indicates that the channel cannot be used at all).

Definition 3.7 *(Robust deadlock-freedom).* The relation $\Delta \models_{\mathtt{RD}} P$ is the largest relation such that $\Delta \models_{\mathtt{RD}} P$ implies all of the following conditions.

(1) If Δ is closed and $rel(\Delta)$, then:
   —P is deadlock-free
   —If $ob_!(\Delta(a)) \neq \infty$, then either $P \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}$ or $P \xrightarrow{\tau}$.
   —If $ob_?(\Delta(a)) \neq \infty$, then either $P \xrightarrow{a[\widetilde{b}]}$ or $P \xrightarrow{\tau}$.
(2) If $[v \mapsto a]\Delta$ is well-defined, then $[v \mapsto a]\Delta \models_{\mathtt{RD}} [v \mapsto a]P$.
(3) If $P \xrightarrow{\eta} P'$ and, furthermore, when η is an input, all names received are fresh, then $\Delta \xrightarrow{\eta} \Delta'$ and $\Delta' \models_{\mathtt{RD}} P'$ for some Δ'.

We say that P is *robustly deadlock-free* under Δ if $\Delta \models_{\mathtt{RD}} P$ holds.

The relation $\Delta \xrightarrow{\eta} \Delta'$ discussed above is defined by:

$$\frac{}{\Delta \xrightarrow{\tau} \Delta}$$

$$\frac{U \xrightarrow{\tau} U'}{\Delta, a:\sharp_U[\widetilde{\mathtt{L}}] \xrightarrow{\tau} \Delta, a:\sharp_{U'}[\widetilde{\mathtt{L}}]}$$

$$\frac{U \xrightarrow{?} U'}{\Delta, a:\sharp_U[\widetilde{\mathtt{L}}] \xrightarrow{a[\widetilde{b}]} \Delta \mid \widetilde{b}:\widetilde{\mathtt{L}}, a:\sharp_{U'}[\widetilde{\mathtt{L}}]}$$

$$\frac{U \xrightarrow{!} U' \qquad \Delta, \widetilde{c}:\widetilde{\mathtt{L}_c} \leq \Delta' \mid \widetilde{b}:\widetilde{\mathtt{L}} \qquad rel(\widetilde{\mathtt{L}_c})}{\Delta, a:\sharp_U[\widetilde{\mathtt{L}}] \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]} \Delta', a:\sharp_{U'}[\widetilde{\mathtt{L}}]}$$

EXAMPLE 3.4. Let Δ be $a:\sharp_{?^0_\infty}[\sharp_{!^1_\infty}[\mathtt{Bool}]]$. In order for $\Delta \models_{\mathtt{RD}} P$ to hold, either $P \xrightarrow{a[b]} P'$ or $P \xrightarrow{\tau} P'$ must hold for some P' and b by the 3rd clause of condition (1) of Definition 3.7. In the former case, $\Delta' \models_{\mathtt{RD}} P'$ must hold for $\Delta' = a:\sharp_{\mathbf{0}}[\sharp_{!^1_\infty}[\mathtt{Bool}]], b:\sharp_{!^1_\infty}[\mathtt{Bool}]$ by condition (3), which implies that P' must eventually send a boolean on b unless it diverges. As a whole, $\Delta \models_{\mathtt{RD}} P$ means that P will eventually perform an input on a, and then send a boolean on the received channel, unless P at some point diverges.

Thus, all of the following three processes are robustly deadlock-free under Δ (where Ω is a divergent process $(\nu c)\,(\overline{c} \mid *c.\overline{c})$):

$$a(x).\,\overline{x}[\mathtt{true}] \qquad a(x).\,\Omega \qquad \Omega$$

The following process is however not robustly deadlock-free:

$$a(x).\,(\nu c)\,(\overline{c} \mid c.\mathbf{0} \mid c.\overline{x}[\mathtt{true}]),$$

because after receiving a channel x on a, the process may be blocked without sending a boolean on x. □

3.2.3 *Robust Confluence.* We introduce the notion of *partial confluence*, which means that any τ-transition commutes with any other transitions. To formally state the partial confluence, we assume that each prefix is uniquely labeled as in Bidinger and Compagnoni [2009], and extend the transition relation to $\xrightarrow{\eta,S}$ where S is the set of the labels of the prefixes involved in the transition. For example, the rules for input and communication become:

$$a^{\chi,\xi}(\widetilde{y}).\,P \xrightarrow{a[\widetilde{b}],\{\xi\}} [\widetilde{y} \mapsto \widetilde{b}]P$$

$$\frac{P' \text{ is a relabeling of } P}{*a^{\chi,\xi}(\widetilde{y}).\,P \xrightarrow{a[\widetilde{b}],\{\xi\}} *a^{\chi,\xi}(\widetilde{y}).\,P \mid [\widetilde{y} \mapsto \widetilde{b}]P'}$$

$$\frac{P_1 \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}],S_1} Q_1 \qquad P_2 \xrightarrow{a[\widetilde{b}],S_2} Q_2 \qquad \{\widetilde{c}\} \cap \mathbf{FN}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau,S_1 \cup S_2} (\nu\widetilde{c})\,(Q_1 \mid Q_2)}$$

Robust confluence indicates partial confluence after any sequence of transitions that conform to the base type system ST.

Definition 3.8 *(Robust confluence).* A process P is *partially confluent*, if whenever $P_1 \overset{\tau,S_1}{\longleftarrow} P \overset{\eta,S_2}{\longrightarrow} P_2$, either $\eta = \tau \wedge S_1 = S_2$, or $P_1 \overset{\eta,S_2}{\longrightarrow} \equiv \overset{\tau,S_1}{\longleftarrow} P_2$. (Here, $\equiv$ is the least relation closed under the commutativity and associativity of $|$.) A process P is *robustly confluent* under Γ, written $\Gamma \models_{\mathtt{RConf}} P$, if $\Gamma \vdash_{\mathtt{ST}} P$ and for any closing substitution σ that respects Γ and for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\sigma\Gamma \vdash_{\mathtt{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is partially confluent.

Example 3.5. Let P be $\overline{a} \mid a.\overline{b}$. P is not partially confluent, as the transitions $P \xrightarrow{\tau} \mathbf{0} \mid \overline{b}$ and $P \xrightarrow{\overline{a}} \mathbf{0} \mid a.\overline{b}$ do not commute. The process $(\nu a)\, P$ is however robustly confluent under $b : \sharp[\,]$. The process *Server* in Section 1 is also robustly confluent under $\mathit{fact} : \sharp[\mathtt{Nat}, \sharp[\mathtt{Nat}]]$. □

3.2.4 *Verification Methods for Robust Deadlock-Freedom and Confluence.* While termination, deadlock-freedom, and confluence are frequently discussed in the literature, we are not aware of previous work that defines the robust counterparts above and verification methods for them.

Robust deadlock-freedom is guaranteed by Kobayashi's type system for deadlock-freedom [Kobayashi 2006]:

Theorem 3.9. *If $\Delta \vdash_{\emptyset} P$ in the type system of Kobayashi [2006][2], then $\Delta \models_{\mathtt{RD}} P$.*

The proof is similar to the type soundness proof in Kobayashi [2006], hence omitted. (A difference is that Kobayashi [2006] proves the soundness based on the reduction semantics, while we need to prove it based on the labeled transition semantics.) In applications of robust deadlock-freedom, it is often the case that the environment Δ needed is of a restricted form, so that $\Delta \models_{\mathtt{RD}} P$ then boils down to the verification of a few simple behavioral properties for which other type systems and model checkers can also be used. For example, if Δ is $a : \sharp^{\mathtt{IO}}_{!^{\infty}}[\mathtt{Bool}]$, then $\Delta \models_{\mathtt{RD}} P$ only means that P is deadlock-free and P will eventually send a boolean on a unless it diverges. Robust confluence is guaranteed, for instance, by types systems for linear channels [Kobayashi et al. 1999] and race-freedom [Terauchi and Aiken 2008]; other static analysis methods such as model checking and abstract interpretation [Feret 2005] could also be used. Verification of robust termination is discussed in Section 5.

3.3 Hybrid Typing Rules

We now introduce the new rules LT-Hyb (for weak lock-freedom), and SLT-Hyb (for strong lock-freedom).

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}} P \qquad \mathit{nocap}(\Delta)}{\Delta \vdash_{\mathtt{LT}} P} \tag{LT-Hyb}$$

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}} P \qquad \mathit{Erase}(\Delta) \models_{\mathtt{RConf}} P \qquad \mathit{nocap}(\Delta)}{\Delta \vdash_{\mathtt{SLT}} P} \tag{SLT-Hyb}$$

[2]Kobayashi's type system [Kobayashi 2006] uses pairs instead of tuples; so strictly speaking, we need to encode tuples into pairs in the judgment $\Delta \vdash_{\emptyset} P$.

Here, $\mathit{Erase}(\Delta)$ is the simple type environment obtained from Δ by removing all usage annotations. The condition $\mathit{nocap}(\Delta)$ holds if, intuitively, Δ describes a process that fulfills its obligations without relying on the environment. As mentioned in Section 1, this is used to avoid circular, unsound, assume-guarantee reasoning. The definition is subtle; for nested channel types, the nocap condition depends on whether a channel is used for input or output. For example, $\mathit{nocap}(\sharp^{?0}_{\infty}[\sharp^{!0}_{\infty}[\,]])$ holds but $\mathit{nocap}(\sharp^{!0}_{\infty}[\sharp^{!0}_{\infty}[\,]])$ does not.

Definition 3.10 *(nocap).* We write $\mathit{nocap}(U)$ when all the (syntactic occurrences of) capability levels in U are ∞, and write $\mathit{noob}(U)$ when all the (syntactic occurrences of) obligation levels in U are ∞. The relations are extended to those on types, which are inductively defined by the following rules.

$$\frac{}{\mathit{nocap}(\mathtt{Bool})} \quad \text{(NoCap-Bool)}$$

$$\frac{\mathit{nocap}(U) \qquad \mathit{mode}(U,?) \Rightarrow \mathit{nocap}(\widetilde{\mathtt{L}}) \qquad \mathit{mode}(U,!) \Rightarrow \mathit{noob}(\widetilde{\mathtt{L}})}{\mathit{nocap}(\sharp_U[\widetilde{\mathtt{L}}])} \quad \text{(NoCap-Ch)}$$

$$\frac{}{\mathit{noob}(\mathtt{Bool})} \quad \text{(NoOb-Bool)}$$

$$\frac{\mathit{noob}(U) \qquad \mathit{mode}(U,?) \Rightarrow \mathit{noob}(\widetilde{\mathtt{L}}) \qquad \mathit{mode}(U,!) \Rightarrow \mathit{nocap}(\widetilde{\mathtt{L}})}{\mathit{noob}(\sharp_U[\widetilde{\mathtt{L}}])} \quad \text{(NoOb-Ch)}$$

Here, $\mathit{mode}(U,\alpha)$ means that U contains α. We write $\mathit{nocap}(\Delta)$ when $\mathit{nocap}(\Delta(v))$ for any $v \in \mathit{dom}(\Delta)$.

Notice the interplay between *nocap* and *noob*. For example, $\mathit{noob}(\mathtt{L})$ is required for $\mathit{nocap}(\sharp^{!0}_{\infty}[\mathtt{L}])$, since $\mathtt{L}$ is the type of a channel that is *exported* to the environment. On the other hand, $\mathit{nocap}(\mathtt{L})$ is required for $\mathit{nocap}(\sharp^{?0}_{\infty}[\mathtt{L}])$ since $\mathtt{L}$ is the type of a channel that is *imported* from the environment.

Example 3.6. $\mathit{nocap}(\sharp_{\mathbf{0}}[\widetilde{\mathtt{L}}])$ and $\mathit{noob}(\sharp_{\mathbf{0}}[\widetilde{\mathtt{L}}])$ hold for any $\widetilde{\mathtt{L}}$; since neither $\mathit{mode}(\mathbf{0},?)$ nor $\mathit{mode}(\mathbf{0},!)$ holds, the second and third premises of rules NoCap-Ch and NoOb-Ch are void.

$\mathit{nocap}(\sharp^{?0}_{\infty}[\sharp^{!0}_{\infty}[\,]])$ can be derived from $\mathit{nocap}(\sharp^{!0}_{\infty}[\,])$ by using NoCap-Ch. $\mathit{nocap}(\sharp^{!0}_{\infty}[\sharp^{!\infty}_{0}[\,]])$ can be derived from $\mathit{noob}(\sharp^{!\infty}_{0}[\,])$ by using NoCap-Ch. Note that $\mathit{nocap}(\sharp^{!0}_{\infty}[\sharp^{!0}_{\infty}[\,]])$ does not hold: the rightmost premise of NoCap-Ch requires $\mathit{noob}(\sharp^{!0}_{\infty}[\,])$, but that is not the case.

□

Example 3.7. $\Delta_1 = a : \sharp^{?0}_{\infty}[\sharp^{!0}_{\infty}[\,]], b : \sharp^{?0}_{\infty}[\,]$ satisfies $\mathit{nocap}(\Delta_1)$. On the other hand, $\Delta_2 = a : \sharp^{?1}_{\infty}[\sharp^{!0}_{\infty}[\,]], b : \sharp^{?\infty}_{0}[\,]$ does not satisfy $\mathit{nocap}(\Delta_2)$. □

To see why the $\mathit{nocap}(\Delta)$ condition is necessary, consider the process $P_1 \mid P_2$, where

$$P_1 \stackrel{\text{def}}{=} *a(x).\,\overline{b}[x] \qquad P_2 \stackrel{\text{def}}{=} \overline{a}[c] \mid *b(x).\,\overline{a}[x].$$

Let us define Δ_1 and Δ_2 by:

$$\Delta_1 \stackrel{\text{def}}{=} a : \sharp^{0}_{*?\infty}[\sharp^{1}_{!1_\infty}[\,]], b : \sharp^{\infty}_{*!0}[\sharp^{1}_{!1_\infty}[\,]]$$
$$\Delta_2 \stackrel{\text{def}}{=} a : \sharp^{\infty}_{*!0}[\sharp^{1}_{!1_\infty}[\,]], b : \sharp^{0}_{*?\infty}[\sharp^{1}_{!1_\infty}[\,]], c : \sharp^{1}_{!1_\infty}[\,]$$

Then, we have $\Delta_1 \models_{\mathtt{RD}} P_1$ and $\Delta_2 \models_{\mathtt{RD}} P_2$. P_1 and P_2 are robustly terminating, i.e., $\mathit{Erase}(\Delta_1) \models_{\mathtt{RTer}} P_1$ and $\mathit{Erase}(\Delta_2) \models_{\mathtt{RTer}} P_2$. If there were no other conditions, we would obtain $\Delta_1 \vdash_{\mathtt{LT}} P_1$ and $\Delta_2 \vdash_{\mathtt{LT}} P_2$, from which the following wrong judgment would be obtained:

$$\emptyset \vdash_{\mathtt{LT}} (\nu c)\,(c^{\circ} \mid (\nu a)\,(\nu b)\,(P_1 \mid P_2)).$$

The problem with the example is that P_1 and P_2 assume each other that the other process will fulfill an obligation to execute the input on a or b, and to use the received channel for output.

Based on the observation above, we require by $\mathit{nocap}(\Delta)$ that P must not rely on the environment fulfilling any obligation.

Remark 3.11. Weakening the *nocap* condition, or finding situations in which it can be removed, appears delicate. For instance, the example of P_1 and P_2 above might suggest that *nocap* is not needed if LT-Hyb is applied only once in a typing derivation. That is, however, *unsound*. Let P be $*a(x).\,b.\overline{a}[x]$ and Δ be $b : \sharp^{\infty}_{*?0}[\,], a : \sharp^{0}_{*?\infty.!0^{\infty}}[\sharp^{1}_{!1_\infty}[\mathtt{Bool}]]$. Then we have $\Delta \models_{\mathtt{RD}} P$ and $\mathit{Erase}(\Delta) \models_{\mathtt{RTer}} P$. Without the *nocap* condition, we would get $\Delta \vdash_{\mathtt{LT}} P$, from which we would obtain a wrong conclusion:

$$\emptyset \vdash_{\mathtt{LT}} (\nu a, b)\,(P \mid *\overline{b} \mid \overline{a}[c] \mid c^{\circ}).$$

As this example suggests, if the *nocap* condition is weakened, the condition of robust termination must be strengthened to recover the type soundness. A more interesting weakening of *nocap* is mentioned in Section 9.

In the rule for strong lock-freedom, the robust confluence ensures that once a marked prefix is enabled, it cannot be disabled by any other transitions. See Example 3.11 for an non-trivial example, for which the rule LT-Hyb fails to guarantee strong lock-freedom.

We write $\Delta \vdash_{\mathtt{LT}} P$ if it is derivable by using the typing rules in Section 3.1 and LT-Hyb, and write $\Delta \vdash_{\mathtt{SLT}} P$ if it is derivable by using SLT-Hyb instead of LT-Hyb.

3.4 Examples

This section shows several examples to demonstrate our hybrid type system.

EXAMPLE 3.8. Recall the process *Server* in Section 1.

$$\begin{array}{l} \mathit{Server} \stackrel{\text{def}}{=} \\ \quad (\nu \mathit{fact\_it}) \\ \quad (*\mathit{fact}(n,r).\,\overline{\mathit{fact\_it}}[n,1,r] \\ \quad\quad \mid *\mathit{fact\_it}(n,x,r). \\ \quad\quad\quad \mathbf{if}\ n=0\ \mathbf{then}\ \overline{r}[x]\ \mathbf{else}\ \overline{\mathit{fact\_it}}[n-1,x\times n,r]) \end{array}$$

Let us define *Clients* by:

$$\mathit{Clients} \stackrel{\text{def}}{=} *(\nu r_1)\,(\overline{\mathit{fact}}^{\circ}[\mathtt{rnd}(), r_1] \mid {r_1}^{\circ}(x).\,\mathbf{0})$$

Here, `rnd`() is a primitive for generating random natural numbers.

Let Δ be $\mathit{fact} : \sharp_{*?^0_\infty}[\mathtt{Nat}, \sharp_{!^0_\infty}[\mathtt{Nat}]]$. Then, we have:

$$\Delta \models_{\mathtt{RD}} \mathit{Server} \quad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}} \mathit{Server} \quad \mathit{Erase}(\Delta) \models_{\mathtt{RConf}} \mathit{Server} \quad \mathit{nocap}(\Delta)$$

Thus, by using SLT-HYB, we obtain $\Delta \vdash_{\mathtt{SLT}} \mathit{Server}$.

On the other hand, *Clients* is typed as follows:

$$\dfrac{\dfrac{\dfrac{\mathit{fact} : \sharp_{!^\infty_0}[\mathtt{Nat}, \sharp_{!^0_\infty}[\mathtt{Nat}]], r_1 : \sharp_{!^1_\infty}[\mathtt{Nat}] \vdash \overline{\mathit{fact}}^{\circ}[\mathtt{rnd}(), r_1] \quad r_1 : \sharp_{?^\infty_1}[\mathtt{Nat}] \vdash {r_1}^{\circ}(x).\,\mathbf{0}}{\mathit{fact} : \sharp_{!^\infty_0}[\mathtt{Nat}, \sharp_{!^0_\infty}[\mathtt{Nat}]], r_1 : \sharp_{!^1_\infty \mid ?^\infty_1}[\mathtt{Nat}] \vdash \overline{\mathit{fact}}^{\circ}[\mathtt{rnd}(), r_1] \mid {r_1}^{\circ}(x).\,\mathbf{0}}}{\mathit{fact} : \sharp_{!^\infty_0}[\mathtt{Nat}, \sharp_{!^0_\infty}[\mathtt{Nat}]] \vdash (\nu r_1)\,(\overline{\mathit{fact}}^{\circ}[\mathtt{rnd}(), r_1] \mid {r_1}^{\circ}(x).\,\mathbf{0})}}{\mathit{fact} : \sharp_{*!^\infty_0}[\mathtt{Nat}, \sharp_{!^0_\infty}[\mathtt{Nat}]] \vdash \mathit{Clients}}$$

From the judgments for *Server* and *Clients* above, we obtain:

$$\emptyset \vdash_{\mathtt{SLT}} (\nu \mathit{fact})\,(\mathit{Server} \mid \mathit{Clients}).$$

This means that all the clients can eventually receive replies. Note that the whole process diverges (since there are infinitely many clients), but we can derive strong lock-freedom by local reasoning based on SLT-HYB. □

EXAMPLE 3.9. Consider the following process `BSystem`.

$$\begin{array}{l} \mathtt{BServer} \stackrel{\text{def}}{=} (\nu \mathtt{bcastit})\,(*\mathtt{bcast}(z).\,\overline{\mathtt{bcastit}}[z] \\ \quad\quad \mid *\mathtt{bcastit}(z).\,\mathbf{if}\ \mathbf{null}(z)\ \mathbf{then}\ \mathbf{0} \\ \quad\quad\quad\quad \mathbf{else\ let}\ x = \mathbf{hd}(z)\ \mathbf{in}\ \ (\overline{x} \mid \overline{x} \mid \overline{\mathtt{bcastit}}[\mathbf{tl}(z)])) \\ \mathtt{BSystem} \stackrel{\text{def}}{=} (\nu \mathtt{bcast}, \mathtt{rec})\,(\mathtt{BServer} \\ \mid *\mathtt{rec}(z).\,\mathbf{if}\ \mathbf{null}(z)\ \mathbf{then}\ \mathbf{0} \\ \quad\quad\quad\quad \mathbf{else\ let}\ x = \mathbf{hd}(z)\ \mathbf{in}\ \ (x^{\circ} \mid \overline{\mathtt{rec}}[\mathbf{tl}(z)])) \\ \mid (\nu c_1, c_2, c_3)\,(\,\overline{\mathtt{rec}}^{\circ}[c_1; c_2; c_3] \mid \overline{\mathtt{bcast}}^{\circ}[c_1; c_2; c_3] \mid {c_1}^{\circ} \mid {c_2}^{\circ} \mid {c_3}^{\circ}) \end{array}$$

This example uses lists as first-order values, with the usual operations for them. The system has two servers: the server $\mathtt{bcast}(z)$, which broadcast a message twice to each channel in the list z; the server $\mathtt{rec}(z)$, which listens on all the channels in the list z. The two services are invoked with a list made of three channels c_1, c_2, c_3, on which the clients also receive. All the receive operations on c_1, c_2, c_3 are expected to succeed. The success of the receive operations relies on the correct inspection of the lists by the two recursive servers, including the correct use of each channel in the lists (for instance, lock-freedom would fail if `bcast` did not use, or used only once,

$$
\begin{array}{rcl}
G & \stackrel{\text{def}}{=} & *p(x,y,n,s).x(t,r). \\
 & & \quad \textbf{if } t = s \textbf{ then } \overline{r}[n] \,|\, \overline{p}[x,y,n,s] \\
 & & \quad \textbf{else if } y = \texttt{nil} \textbf{ then } \overline{r}[n+1] \;\; | \, \nu c(\overline{p}[c,\texttt{nil},n+1,t] \,|\, \overline{p}[x,c,n,s]) \\
 & & \quad\quad \textbf{else } \overline{y}[t,r].\,\overline{p}[x,y,n,s] \\
\texttt{ST}_0 & \stackrel{\text{def}}{=} & (\nu p)\,(G \,|\, \overline{p}[a,\texttt{nil},1,s_0]) \\
\texttt{ST}_m & \stackrel{\text{def}}{=} & ST_0 \,|\, *(\nu r_1)\,(\overline{a}^{\circ}[\texttt{rnd\_string}(), r_1] \,|\, {r_1}^{\circ}(x).\,\mathbf{0})
\end{array}
$$

Fig. 4. A symbol table

some of the channels in its list). Forwarding of a request from `bcast` to `bcastit` is necessary to get the last condition. Actually, the forwarding can be removed if $nocap(\Delta)$ is extended to $nocap_{\Lambda}(\Delta)$ as discussed in Section 4.

Let $\Delta = \texttt{bcast} : \sharp_{*?^0_\infty}[\sharp_{!^0_\infty} \,|\, !^0_\infty[\,]\ \texttt{List}]$. Then, we have:

$$\Delta \models_{\texttt{RD}} \texttt{BServer} \quad \mathit{Erase}(\Delta) \models_{\texttt{RTer}} \texttt{BServer} \quad \mathit{Erase}(\Delta) \models_{\texttt{RConf}} \texttt{BServer} \quad \mathit{nocap}(\Delta)$$

Thus, by using SLT-Hyb, we get $\Delta \vdash_{\texttt{SLT}} \texttt{BServer}$. By applying the rules for the `LT` type system to the rest of the process, we get $\emptyset \vdash_{\texttt{SLT}} \texttt{BSystem}$. □

Example 3.10. This example is from [Jones 1993]. It is about the implementation of a symbol table as a chain of cells. In Figure 4, G is a generator for cells; ST_0 is the initial state of the symbol table with only one cell; ST_m is the system consisting of the symbol table and clients of it, where `rnd_string`() is random generator of strings, used for a compact representation of a potentially infinite number of clients. The request and answer actions from the clients are marked so as to indicate that we expect them to succeed in the lock-freedom analysis.

Every cell of the chain stores a pair (n, s), where s is a string and n is a key identifying the position of the cell in the chain. A cell is equipped with two channels so as to be connected to its left and right neighbors. The first cell has a public left channel a to communicate with the environment and the last cell has a right channel `nil` to mark the end of the chain. Once received a query for string t, the table lets the request ripple down the chain until either t is found in a cell, or the end of the chain is reached, which means that t is a new string and thus a new cell is created to store t. In both cases, the key associated to t is returned as a result. There is parallelism in the system: many requests can be rippling down the chain at the same time.

Let Δ be: $a : \sharp_{*?^0_\infty}[\texttt{String}, \sharp_{!^0_\infty}[\texttt{Nat}]]$. Then, we have:

$$\Delta \models_{\texttt{RD}} \texttt{ST}_0 \qquad \mathit{Erase}(\Delta) \models_{\texttt{RTer}} \texttt{ST}_0 \qquad \mathit{Erase}(\Delta) \models_{\texttt{RConf}} \texttt{ST}_0 \qquad \mathit{nocap}(\Delta)$$

By using SLT-Hyb, we get $\Delta \vdash_{\texttt{SLT}} \texttt{ST}_0$. By using rules for `LT` type system, we obtain $\emptyset \vdash_{\texttt{SLT}} \texttt{ST}_m$. □

Example 3.11. This example shows a binary search tree data structure, offering services for inserting and searching natural numbers. Each node of the tree is implemented as a process that has: a state, given by the integer stored in the node and pointers to the left and right subtree and that contain, respectively, smaller and greater integers; channels for the insert and search operations. In Figure 5, G is a generator of new nodes, which can then grow and originate a tree, and where: i and s will be the insertion and search channels; `state` stores the state of the node.

Initially the node is a leaf. `TreeInit` is the initial tree, with an empty state and public channels `insert` and `search` to communicate with the environment. Once received a query for an integer n, the tree lets the request ripple down the nodes, following the order on the integers to find the right path, until either t is found in a node, or the end of the tree is reached, which, in the case of an insert, means that n is a new integer and the node a leaf, and thus the leaf becomes a node that stores n and two new leaves are created. As in the symbol table example, many requests can be rippling down the tree at the same time; moreover, requests can even overtake each other.

As to lock-freedom, the example is interesting for at least two reasons. (1) The tree exhibits a syntactically challenging form. The process G has a sophisticated structure of intertwined recursive inputs: the replicated input at `newtree` has outputs at `newtree` itself in its body; similarly, the replicated inputs at i and s have, in the body, outputs at sibling channels (the names for interrogations of the two following subtrees); further, also the imperative channel `state` takes place in the recursions at i and s. (2) Semantically, the tree is a dynamic structure, which can grow to finite but unbounded length, depending on the number of requests it serves. Moreover, the tree has a high parallelism involving independent threads of activities and where: the paths followed by the threads on the tree are partially overlapping; threads can proceed at different speeds (i.e., requests can overtake each other). The number of steps that the tree takes to serve a request from a client depends on the height of the tree, on the number of internal threads in the tree, and on the value of the request.

Let Δ be $\mathtt{insert}:\sharp_{*?0}^{\infty}[\mathtt{Nat},\sharp_{!0}^{\infty}[\,]],\mathtt{search}:\sharp_{*?0}^{\infty}[\mathtt{Nat},\sharp_{!0}^{\infty}[\mathtt{Bool}]]$. Then, we have:

$$\Delta \models_{\mathtt{RD}} \mathtt{TreeInit} \quad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}} \mathtt{TreeInit} \quad \mathit{nocap}(\Delta)$$

Thus, by using LT-Hyb, we obtain $\Delta \vdash_{\mathtt{LT}} \mathtt{TreeInit}$. By applying rules for `LT` to the rest of the system, we get $\Delta \vdash_{\mathtt{LT}} \mathtt{System}$.

Note that SLT-Hyb is not applicable since `TreeInit` is not robustly confluent (because, when multiple requests arrive simultaneously, there can be a race on the channel `state`). Indeed, the example is NOT strongly lock-free! A search request may never be replied if the request is overtaken by insertion requests so often that the tree grows faster than the search request goes down the tree. Thus, a stronger scheduling assumption is necessary for this implementation to work properly.

□

In all the examples, robust termination is guaranteed by the type system described in Section 5.

Example 3.12. In Example 3.11 we showed an implementation of a binary search tree that is weakly, but not strongly, lock-free. Figure 6 shows a strongly lock-free implementation. The server $\mathtt{TreeInit}'$ receives requests along channel a one by one. A request is either of the form $\mathbf{insert}(n,r)$ or $\mathbf{search}(n,r)$. Unlike the system in Example 3.11, requests cannot be overtaken, although there is still parallelism (multiple requests can go down the tree simultaneously). $\mathtt{TreeInit}'$ is robustly confluent; note that the only τ-transitions inside $\mathtt{TreeInit}'$ are on channels `leaf`, `node`, `left`, and `right`, and that the first two of them are replicated

$$
\begin{array}{rcl}
G & \stackrel{\text{def}}{=} & *\texttt{newtree}(i,s).(\nu\texttt{state})\ \ \overline{\texttt{state}}[\mathbf{leaf}] \\
& & |\, *i(n,r).\texttt{state}(x). \quad \texttt{/*** insertion ***/} \\
& & \quad \mathbf{match}\ x\ \mathbf{with} \\
& & \quad\ \ \mathbf{leaf}\ \rightarrow \\
& & \quad\ \ \texttt{/*** if t is a leaf, turns it into a node having two new leaves ***/} \\
& & \quad\ \ (\nu\texttt{left\_i},\texttt{left\_s},\texttt{right\_i},\texttt{right\_s}) \\
& & \quad\ \ \overline{\texttt{newtree}}[\texttt{left\_i},\texttt{left\_s}] \\
& & \quad\ \ |\, \overline{\texttt{newtree}}[\texttt{right\_i},\texttt{right\_s}] \quad \texttt{/*** create two leaves ***/} \\
& & \quad\ \ |\, \overline{\texttt{state}}[\mathbf{node}(n,\texttt{left\_i},\texttt{left\_s},\texttt{right\_i},\texttt{right\_s})] \\
& & \quad\ \ \texttt{/*** change to an internal node ***/} \\
& & \quad\ \ |\, \overline{r}[] \quad \texttt{/*** notify the completion of insertion ***/} \\
& & \quad \|\ \mathbf{node}(n_1,i_l,s_l,i_r,s_r)\ \rightarrow \quad \texttt{/*** if t is a node ***/} \\
& & \quad\ \ \mathbf{if}\ n = n_1\ \mathbf{then}\ \overline{r}[] \\
& & \quad\ \ \texttt{/*** if } n \texttt{ is in the node, then stop, to avoid duplicates ***/} \\
& & \quad\ \ \mathbf{else\ if}\ n < n_1\ \mathbf{then}\ \overline{i_l}[n,r] \\
& & \quad\ \ \texttt{/*** if } n < t_1 \texttt{ then insert } n \texttt{ into the left subtree ***/} \\
& & \quad\ \ \mathbf{else}\ \overline{i_r}[n,r] \\
& & \quad\ \ \texttt{/*** otherwise, insert } n \texttt{ into the right subtree ***/} \\
& & \quad\ \ |\, \overline{\texttt{state}}[x] \\
& & |\, *s(n,r).\texttt{state}(x).(\overline{\texttt{state}}[x] \quad \texttt{/*** search ***/} \\
& & \quad |\, \mathbf{match}\ x\ \mathbf{with} \\
& & \quad\ \ \mathbf{leaf}\ \rightarrow\ \overline{r}[\texttt{false}] \\
& & \quad\ \ \|\ \mathbf{node}(n_1,i_l,s_l,i_r,s_r) \rightarrow \\
& & \quad\ \ \quad \mathbf{if}\ n_1 = n\ \mathbf{then}\ \overline{r}[\texttt{true}]\ \mathbf{else\ if}\ n < n_1\ \mathbf{then}\ \overline{s_l}[n,r]\ \mathbf{else}\ \overline{s_r}[n,r]) \\
\texttt{TreeInit} & \stackrel{\text{def}}{=} & (\nu\texttt{newtree})\,(G \,|\, \overline{\texttt{newtree}}[\texttt{insert},\texttt{search}]) \\
\texttt{System} & \stackrel{\text{def}}{=} & (\nu\texttt{insert},\texttt{search}) \\
& & (\texttt{TreeInit} \,|\, *(\nu r_1)\,(\overline{\texttt{insert}}^{\circ}[\texttt{rnd}(),r_1] \,|\, {r_1}^{\circ}) \,|\, *(\nu r_2)\,(\overline{\texttt{search}}^{\circ}[\texttt{rnd}(),r_2] \,|\, {r_2}^{\circ}(x)))
\end{array}
$$

Fig. 5. A binary tree

input channels, and the others are linearized channels. Thus, we can derive

$$a : \sharp_{*?0}^{0}{}_{\infty}\,[\mathtt{L}] \vdash_{\mathtt{SLT}} \mathtt{TreeInit}'$$

where

$$\mathtt{L} \stackrel{\text{def}}{=} \langle \mathbf{insert} : [\mathtt{Nat}, \sharp_{!0}^{0}{}_{\infty}\,[\,]], \mathbf{search} : [\mathtt{Nat}, \sharp_{!0}^{0}{}_{\infty}\,[\mathtt{Nat}]] \rangle.$$

Here, L is a variant type describing requests of the form $\mathbf{insert}(n,r)$ or $\mathbf{search}(n,r)$. By using the typing rules for SLT, we can derive:

$$\emptyset \vdash_{\mathtt{SLT}} \mathtt{System}'.$$

Thus, we can verify that $\mathtt{System}'$ is strongly lock-free. □

4. TYPE SOUNDNESS

We show the soundness of the type system in this section. The following theorems are the main results of this paper.

THEOREM 4.1 (WEAK LOCK-FREEDOM). *If* $\emptyset \vdash_{\mathtt{LT}} P$, *then* P *is (weakly) lock-free.*

$$
\begin{array}{rcl}
G' & \stackrel{\mathrm{def}}{=} & *\mathtt{leaf}(x).x(req). \\
& & \quad (\mathbf{match}\ req\ \mathbf{with} \\
& & \qquad \mathbf{insert}(n,r)\ \rightarrow \\
& & \qquad\quad (\nu \mathtt{left},\mathtt{right})\,(\overline{r}\,|\,\overline{\mathtt{node}}^{\circ}[n,x,\mathtt{left},\mathtt{right}]\,|\,\overline{\mathtt{leaf}}^{\circ}[\mathtt{left}]\,|\,\overline{\mathtt{leaf}}^{\circ}[\mathtt{right}]) \\
& & \qquad ||\,\mathbf{search}(n,r)\ \rightarrow\ \overline{r}[\mathtt{false}]\,|\,\overline{\mathtt{leaf}}^{\circ}[x]) \\
& & \quad |\,*\mathtt{node}(n_1,x,x_l,x_r).x(req). \\
& & \quad (\mathbf{match}\ req\ \mathbf{with} \\
& & \qquad \mathbf{insert}(n,r)\ \rightarrow \\
& & \qquad\quad \mathbf{if}\ n=n_1\ \mathbf{then}\ \overline{r}\,|\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r] \\
& & \qquad\quad \mathbf{else\ if}\ n<n_1\ \mathbf{then}\ \overline{x_l}^{\circ}[\mathbf{insert}(n,r)].\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r] \\
& & \qquad\qquad \mathbf{else}\ \overline{x_r}^{\circ}[\mathbf{insert}(n,r)].\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r] \\
& & \qquad ||\,\mathbf{search}(n,r)\ \rightarrow \\
& & \qquad\quad \mathbf{if}\ n=n_1\ \mathbf{then}\ \overline{r}[\mathtt{true}]\,|\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r] \\
& & \qquad\quad \mathbf{else\ if}\ n<n_1\ \mathbf{then}\ \overline{x_l}^{\circ}[\mathbf{search}(n,r)].\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r] \\
& & \qquad\qquad \mathbf{else}\ \overline{x_r}^{\circ}[\mathbf{search}(n,r)].\,\overline{\mathtt{node}}^{\circ}[n_1,x,x_l,x_r])
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{TreeInit}' & \stackrel{\mathrm{def}}{=} & (\nu \mathtt{leaf},\mathtt{node})\,(G'\,|\,\overline{\mathtt{leaf}}^{\circ}[a]) \\
\mathtt{System}' & \stackrel{\mathrm{def}}{=} & (\nu a)\,(\mathtt{TreeInit}' \\
& & \quad |\,*(\nu r_1)\,(\overline{a}^{\circ}[\mathbf{insert}(\mathtt{rnd}(),r_1)]\,|\,{r_1}^{\circ})\,|\,*(\nu r_2)\,(\overline{a}^{\circ}[\mathbf{search}(\mathtt{rnd}(),r_2)]\,|\,{r_2}^{\circ}(x)))
\end{array}
$$

Fig. 6. A strongly lock-free implementation of binary trees

THEOREM 4.2 (STRONG LOCK-FREEDOM). *If* $\emptyset \vdash_{\mathtt{SLT}} P$*, then* P *is strongly lock-free.*

The rest of this paper is devoted to the proofs of Theorems 4.1 and 4.2. Readers who are not interested in the proof may safely skip the rest of this section.

Basically, as in the previous type system [Kobayashi 2005a], Theorem 4.1 follows from *type preservation*, which means that typing is preserved by any transition, and *progress*, which means that if a tagged process P is well-typed, then $P \stackrel{\tau}{\longrightarrow}{}^{*} \stackrel{\tau^{\square}}{\longrightarrow}$. The existence of the hybrid rule LT-HYB, however, poses significant challenges in the proof. First, while it was enough to show type preservation by τ-transitions in the previous type systems, because of LT-HYB, we have to show that typing is preserved by *any* transitions (including output/input transitions). Second, in the type system discussed so far, typing is actually *not* preserved by transitions, so that we have to extend the type system in a non-trivial way. To see why, suppose that a judgment $\Delta \vdash_{\mathtt{LT}} P$ is derived by using LT-HYB. In order for the judgment derived by LT-HYB to be preserved by transitions, we need to require that $\Delta \models_{\mathtt{RD}} P$ and $\mathit{nocap}(\Delta)$ with $P \stackrel{\eta}{\longrightarrow} Q$ imply $\Delta' \models_{\mathtt{RD}} Q$ *and* $\mathit{nocap}(\Delta')$ for some Δ'. The latter condition $\mathit{nocap}(\Delta')$, however, does not hold in general. For example, let $P = (\nu c)\,(\overline{a}[c]\,|\,*c(\,)\,|\,\overline{c}^{\circ}[\,])$ and $\Delta = a : \sharp_{!^{\infty}_{0}}[\sharp_{!^{\infty}_{0}}[\,]]$, with $\eta = (\nu c)\,\overline{a}[c]$ and $Q = \mathbf{0}\,|\,*c(\,)\,|\,\overline{c}^{\circ}[\,]$. Then, Q is typed under $\Delta' = a : \sharp_{\mathbf{0}}[\sharp_{!^{\infty}_{0}}[\,]], c : \sharp_{*^{?0}_{\infty}\,|\,!^{\infty}_{0}}[\,]$, but $\mathit{nocap}(\Delta')$ does not hold because c's usage contains $!^{\infty}_{0}$.

To overcome the problem above, we first extend the type system in Section 4.1. We then prove *type preservation* and *progress* for the extended type system in Sections 4.2 and 4.3. Theorem 4.1 then follows as a corollary of the two properties. Theorem 4.2 is proved in Appendix C.

4.1 Extended Typing

A key observation to solve the above problem is that although the type environment Δ' of Q contains a capability, that capability is matched by Q's own obligation $?^0_\infty$, and Q does not expect any obligatory behavior from the environment; the transition $P \xrightarrow{(\nu c)\,\overline{a}[c]} Q$ has exported only a capability (to use c for output) to the environment.

Based on the observation above, we extend the type judgment with an additional parameter Λ, which expresses an assumption about what capabilities/obligations the environment owns. The resulting type judgment form is $\Delta \vdash^{\Lambda}_{\mathrm{LT}} P$. The condition $nocap(\Delta)$ in LT-Hyb is replaced by $nocap_\Lambda(\Delta)$.

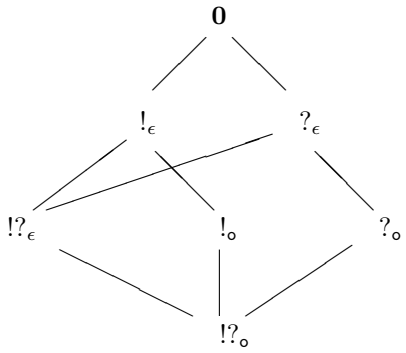
Λ is a mapping from the set $\mathcal{N}$ of names to the set of *modes*, defined by:

$$m \text{ (modes)} \ ::= \mathbf{0} \mid ?_a \mid !_a \mid !?_a$$
$$a ::= \epsilon \mid \mathsf{o}$$

Intuitively, Λ expresses how (for input or output) each channel may be used by the environment of P, and $\Delta \vdash^{\Lambda}_{\mathrm{LT}} P$ means that P is well-typed under that assumption. We write $a_1 : m_1, \ldots, a_n : m_n$ for the mapping Λ such that $\Lambda(a_i) = m_i$ and $\Lambda(b) = !?_\mathsf{o}$ for $b \notin \{a_1, \ldots, a_n\}$. We write $\bot$ for the mapping Λ such that $\Lambda(a) = !?_\mathsf{o}$ for every $a \in \mathcal{L}$. For the sake of simplicity, we assume that variables are always mapped to $!?_\mathsf{o}$.

A mode m can be considered an abstraction of usages (which are again abstractions of communication behaviors on each channel). Intuitively, $a : ?_a$ means that the environment may perform an input on a. The attribute a expresses whether the process relies on the environment performing the input. $a : ?_\epsilon$ means that the process definitely does not rely on the environment performing the input, while $a : ?_\mathsf{o}$ means that the process may rely on the environment. We often omit ϵ and just write $?$, $!$, $!?$ for $?_\epsilon$, $!_\epsilon$, $!?_\epsilon$.

We define the *submode* relation $m_1 \leq m_2$ as shown below (An upper mode is greater than a lower mode):



We extend the submode relation to that on mode environments by:

$$\Lambda_1 \leq \Lambda_2 \iff \forall a \in \mathcal{L}.\Lambda_1(a) \leq \Lambda_2(a)$$

We replace the condition $nocap(\Delta)$ with the condition $nocap_\Lambda(\Delta)$ defined below.

Definition 4.3. $nocap_m(\mathtt{L})$ is defined by:

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad \Delta \models_{\mathtt{RTer}} P \qquad nocap_{\Lambda}(\Delta)}{\Delta \vdash^{\Lambda}_{\mathrm{LT}} \langle P \rangle^{T}}\ (\text{ELT-Hyb}) \qquad \frac{\Delta' \vdash^{\Lambda'}_{\mathrm{LT}} P \qquad \Delta \le \Delta' \qquad \Lambda' \le \Lambda}{\Delta \vdash^{\Lambda}_{\mathrm{LT}} P}\ (\text{ELT-Weak})$$

$$\frac{\Delta_1 \vdash^{\perp}_{\mathrm{LT}} P \qquad t_c = \infty \Rightarrow \chi = \bullet}{v : \sharp^{!0}_{t_c}\,[\mathtt{L}]; (\Delta_1 \mid \widetilde{w} : \uparrow\widetilde{\mathtt{L}}) \vdash^{\perp}_{\mathrm{LT}} \overline{v}^{\chi}[\widetilde{w}].\, P}\ (\text{ELT-Out}) \qquad \frac{\Delta \vdash^{\perp}_{\mathrm{LT}} P}{*\Delta \vdash^{\perp}_{\mathrm{LT}} *P}\ (\text{ELT-Rep})$$

$$\frac{}{\emptyset \vdash^{\Lambda}_{\mathrm{LT}} \mathbf{0}}\ (\text{ELT-Zero}) \qquad \frac{\Delta, v : \mathtt{L} \vdash^{\perp}_{\mathrm{LT}} P \qquad t_c = \infty \Rightarrow \chi = \bullet}{v : \sharp^{?0}_{t_c}\,[\mathtt{L}]; \Delta \vdash^{\perp}_{\mathrm{LT}} v^{\chi}(y).\, P}\ (\text{ELT-In})$$

$$\frac{\Delta, a : \sharp_U[\mathtt{L}] \vdash^{\Lambda}_{\mathrm{LT}} P \qquad rel(U)}{\Delta \vdash^{\Lambda\{a \mapsto !?_{\circ}\}}_{\mathrm{LT}} (\nu a)\, P}\ (\text{ELT-New}) \qquad \frac{\Delta \vdash^{\perp}_{\mathrm{LT}} P \qquad \Delta \vdash^{\perp}_{\mathrm{LT}} Q}{\Delta \mid (v : \mathtt{Bool}) \vdash^{\perp}_{\mathrm{LT}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\text{ELT-If})$$

$$\frac{\Delta_1 \vdash^{\Lambda_1}_{\mathrm{LT}} P_1 \qquad \Delta_2 \vdash^{\Lambda_2}_{\mathrm{LT}} P_2 \qquad \Lambda_2 \le Modes(\Delta_1) \qquad \Lambda_1 \le Modes(\Delta_2)}{\Delta_1 \mid \Delta_2 \vdash^{\Lambda_1 \sqcup \Lambda_2}_{\mathrm{LT}} P_1 \mid P_2}\ (\text{ELT-Par})$$

Fig. 7. Extended Typing Rules for Lock-Freedom

$$\frac{}{nocap_m(\mathtt{Bool})}$$

$$\frac{!?_{\epsilon} \le m \vee nocap(U) \qquad (mode(U, ?) \wedge m \le !_{\epsilon}) \Rightarrow nocap(\widetilde{\mathtt{L}}) \qquad (mode(U, !) \wedge m \le ?_{\epsilon}) \Rightarrow noob(\widetilde{\mathtt{L}})}{nocap_m(\sharp_U[\widetilde{\mathtt{L}}])}$$

We write $nocap_{\Lambda}(\Delta)$ if $nocap_{\Lambda(a)}(\Delta(a))$ for each $a \in dom(\Delta)$.

For the example given in the beginning of this subsection, Q is typed as $\Delta' \vdash^{\Lambda'}_{\mathrm{DT}} Q$ where $\Lambda' = a : ?_{\epsilon}, c : !_{\epsilon}$. By the definition above, $nocap_{\Lambda'}(\Delta')$ holds.

We also extend the syntax of processes in order to make applications of LT-Hyb explicit.

$$P ::= \cdots \mid \langle P \rangle^{T}$$

The typing rules for the extended judgments are given in Figure 7. A key change from the type system in Section 3 is that the condition $nocap(\Delta)$ in LT-Hyb has been replaced by a weaker condition $nocap_{\Lambda}(\Delta)$. Note also that rule ELT-Par requires (by the conditions $\Lambda_2 \le Modes(\Delta_1)$ and $\Lambda_1 \le Modes(\Delta_2)$) that P_1 conforms to the assumption Λ_2 on the behavior of P_2's environment, and vice versa. Here, $Modes(\Delta)$, defined below, maps the type environment to the corresponding mode environment.

Definition 4.4. $Modes(U)$ is defined by:

$$\begin{array}{l} Modes(\mathbf{0}) = \mathbf{0} \\ Modes(\alpha^{t_1}_{t_2}.U) = \begin{cases} \alpha_{\circ} \sqcap Modes(U) & \text{if } t_1 \neq \infty \\ \alpha_{\epsilon} \sqcap Modes(U) & \text{if } t_1 = \infty \end{cases} \\ Modes(U_1 \mid U_2) = Modes(U_1) \sqcap Modes(U_2) \\ Modes(*U) = Modes(U) \end{array}$$

Here, $m_1 \sqcap m_2$ is the greatest lower bound of m_1 and m_2.

Modes(L) is defined by:

$$\begin{aligned} &\mathit{Modes}(\mathtt{Bool}) = \mathbf{0} \\ &\mathit{Modes}(\sharp_U[\widetilde{\mathtt{L}}]) = \mathit{Modes}(U). \end{aligned}$$

Modes(Δ) is defined by:

$$\mathit{Modes}(\Delta)(a) = \begin{cases} \mathit{Modes}(\Delta(a)) & \text{if } a \in \mathit{dom}(\Delta) \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

4.2 Type Preservation

We now show that the extended typing relation is preserved by reduction.

A type environment and a mode environment may be changed by the transition. For instance, for the example given at the beginning of the previous subsection, P's type environment and mode environment are $\Delta = a : \sharp_{!^{0}_{\infty}}[\sharp_{!^{\infty}_{0}}[\,]]$ and $\Lambda = a : ?_\epsilon$, while those of Q are $\Delta' = a : \sharp_{\mathbf{0}}[\sharp_{!^{\infty}_{0}}[\,]], c : \sharp_{*?^{0}_{\infty}\,|\,!^{\infty}_{0}}[\,]$ and $\Lambda' = a : ?_\epsilon, c : !_\epsilon$. Similarly, suppose that $a : \sharp_{?^{\infty}_{0}}[\sharp_{!^{1}_{\infty}}[\,]] \vdash^{\Lambda}_{\mathtt{LT}} P$ and $P \xrightarrow{a[b]} Q$. Since P imports the capability and obligation on b by consuming the input capability on P, the type environment of Q is $a : \sharp_{\mathbf{0}}[\sharp_{!^{1}_{\infty}}[\,]], b : \sharp_{!^{1}_{\infty}}[\,]$. Such changes of type environments and mode environments are captured by the relation $\Delta \xrightarrow{\eta} \Delta'$ defined in Section 3.2 and the relation $\Lambda \xrightarrow{\eta} \Lambda'$ defined below. We write $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$ for $\Delta \xrightarrow{\eta} \Delta'$ and $\Lambda \xrightarrow{\eta} \Lambda'$.

$$\Lambda \xrightarrow{\tau} \Lambda$$

$$\Lambda \xrightarrow{a[\widetilde{b}]} \Lambda$$

$$\Lambda \xrightarrow{(\nu \widetilde{c})\,\overline{a}[\widetilde{b}]} \Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b} : \widetilde{\mathtt{L}})$$

Here, $\Lambda_1 \sqcap \Lambda_2$ is the greatest lower bound of Λ_1 and Λ_2 (with respect to the submode relation).

The predicate $\mathit{enabled}(\Lambda, \Delta, \eta)$ defined below means that the transition η is enabled under the type environment Δ and the mode environment Λ. Note that, for example, the action $\overline{a}[b]$ is not possible if $\Lambda(a) = \mathbf{0}$, because the environment cannot perform an input action on a. That is expressed by the condition $\Lambda(a) \leq ?_\epsilon$ in the third rule below.

Definition 4.5. The predicate $\mathit{enabled}(\Lambda, \Delta, \eta)$ is defined by:

$$\frac{}{\mathit{enabled}(\Lambda, \Delta, \tau)}$$

$$\frac{\Delta(a) = \sharp_U[\widetilde{\mathtt{L}}] \qquad \Delta \,|\, \widetilde{b} : \widetilde{\mathtt{L}} \text{ well-defined} \qquad \Lambda \leq \mathit{Modes}(a : \sharp_{!_\epsilon}[\widetilde{\mathtt{L}}], \widetilde{b} : \widetilde{\mathtt{L}})}{\mathit{enabled}(\Lambda, \Delta, a[\widetilde{b}])}$$

$$\frac{\Lambda(a) \leq ?_\epsilon \qquad \Delta(a) = \sharp_U[\widetilde{\mathtt{L}}]}{\mathit{enabled}(\Lambda, \Delta, (\nu \widetilde{c})\,\overline{a}[\widetilde{b}])}$$

Now we state the main lemma.

LEMMA 4.6 TYPE PRESERVATION. *If* $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$, *enabled*$(\Lambda, \Delta, \eta)$, *and* $P \xrightarrow{\eta} Q$, *then there exists* Δ' *and* Λ' *such that* $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} Q$ *and* $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$.

PROOF. See Appendix A. □

4.3 Progress and Lock-Freedom

The progress property is stated as follows.

LEMMA 4.7 PROGRESS. *Let* P *be a tagged process. If* $\emptyset \vdash^{\Lambda}_{\mathtt{LT}} P$, *then* $P \xrightarrow{\tau}{}^{*} \xrightarrow{\tau^{\square}}$.

PROOF. See Appendix B □

We can now prove the lock-freedom theorem (Theorem 4.1).

PROOF THEOREM 4.1. Suppose that $\emptyset \vdash_{\mathtt{LT}} P$ and $P \xrightarrow{\tau}{}^{*} Q$. We need to show that any process Q' in the tagging of Q is successful. By Lemma 4.6, we have $\emptyset \vdash^{\perp}_{\mathtt{LT}} Q'$. (Note that replacement of $\circ$ with $\square$ does not affect the typability.) Suppose $Q' \xrightarrow{\tau}{}^{*} R$. Then, by using Lemma 4.6 again, we get $\emptyset \vdash^{\perp}_{\mathtt{LT}} R$. Since R must be tagged (note that only $\xrightarrow{\tau}$ cannot discharge $\square$), by using Lemma 4.7, we get $R \xrightarrow{\tau}{}^{*} \xrightarrow{\tau^{\square}}$. Thus, Q' is successful. □

See Appendix C for the proof of Theorem 4.2.

5. TYPES FOR ROBUST TERMINATION

In this section, we discuss type systems for guaranteeing robust termination. *Termination* of a term means that all its reduction sequences are of finite length. *Robust termination* (Definition 3.6) guarantees that termination is maintained when the process interacts with its environment. Termination is strictly weaker than robust termination. Consider for instance the term

$$P \stackrel{\text{def}}{=} \overline{c}[b] \mid c(x).(\overline{x} \mid *a.\overline{x}) \tag{1}$$

The process P has one reduction only, and therefore it is terminating. It is indeed typable in the simplest of the type systems in [Deng and Sangiorgi 2006]. However, P is not robustly terminating. It can interact with other processes via the input at c and, in doing so, it may receive a resulting in the non-terminating derivative

$$\overline{c}[b] \mid \overline{a} \mid *a.\overline{a}$$

It is precisely because of input prefixes, as shown in this example, that processes typable in [Deng and Sangiorgi 2006] (as well as other type systems for termination) may not be robustly terminating.

The objective here is to guarantee robust termination by re-using existing type systems for termination. Precisely, we wish to add some extra conditions to the type systems for termination capable of ensuring the stronger property of robust termination. For the sake of simplicity, we impose a restriction that replication can be applied only to input prefixes (so that a process like $*\overline{a}$ is forbidden). This restriction does not affect the expressiveness of the calculus and is indeed very

common in languages based on the π-calculus; dealing with arbitrary replications would complicate substantially the termination type systems.

We explain the idea of the extra conditions on a very simple type system for termination, namely the first of the type systems in [Deng and Sangiorgi 2006], which we recall (and revise) in the next subsection.

5.1 The type systems in [Deng and Sangiorgi 2006], revisited

We recall the type systems in [Deng and Sangiorgi 2006], as we appeal to them for the termination analysis of most of the examples in this paper. In [Deng and Sangiorgi 2006] these type systems are expressed *à la Church*—each name is assigned a type a priori—and exploit this in making use of some special functions that scan the whole syntax of a process looking for certain typed patterns of occurrences of names. We revise the systems, using an approach *à la Curry* and avoiding these complex functions.

There are four type systems in [Deng and Sangiorgi 2006], plus combinations of some of them. We discuss the first system, which is the simplest, and the fourth, as it does not fit the condition for robust termination in Lemma 5.3; we only hint at the others.

The first system, `Lev`, is obtained by making a mild modification to the types and typing rules of the simply typed π-calculus: a *level* information, which is a natural number, is added to each channel type. The levels are used to define a weight for each process. We call *active* an output that is not underneath a replication. The weight of a process is the multiset consisting of the levels of all active outputs. The type system guarantees that the weight strictly decreases under reduction (with respect to the standard multiset ordering), by imposing the constraint that, in a replicated input, the level of the input name should be strictly greater than that of any name that is used in output in the body of the replication (and that is not under some inner replications). For instance, a typing of the process P in (1) would assign b a level that is the same as that of x but smaller than that of a; the level of c could be anything, as the input at this channel is not replicated (there could also be several outputs at x underneath the replication at a; if there were an output at c, however, then the level of c should be smaller than that of a). The grammar of the types of `Lev` is:

$$V \ ::= \ \mathtt{Bool} \ \big| \ \sharp^n[\widetilde{V}] \qquad n \in \mathtt{Nat}$$

A judgment in `Lev` takes the form $\Theta \vdash^n_{\mathtt{Lev}} P$. We write $\Theta \vdash_{\mathtt{Lev}} P$ if $\Theta \vdash^n_{\mathtt{Lev}} P$ holds for some level n. Intuitively, n in a judgment $\Theta \vdash^n_{\mathtt{Lev}} P$ represents the level of the innermost replication enclosing P, so that the level of every active output must be smaller than n. The typing rules are similar to those of the simply-typed π-calculus, except for the following rules for output, input, and replicated input.

$$\frac{\Theta(p) = \sharp^{n_2}[\widetilde{V}] \quad \Theta \vdash \widetilde{v} : \widetilde{V} \qquad \Theta \vdash^{n_1}_{\mathtt{Lev}} P \qquad n_2 < n_1}{\Theta \vdash^{n_1}_{\mathtt{Lev}} \overline{p}[\widetilde{v}].P} \qquad \text{(Lev-Out)}$$

$$\frac{\Theta(p) = \sharp^{n_2}[\widetilde{V}] \qquad \Theta, \widetilde{x} : \widetilde{V} \vdash^{n_1}_{\mathtt{Lev}} P}{\Theta \vdash^{n_1}_{\mathtt{Lev}} p(\widetilde{x}).P} \qquad \text{(Lev-In)}$$

$$\frac{\Theta(p) = \sharp^{n_2}[\widetilde{V}] \qquad \Theta, \widetilde{x} : \widetilde{V} \vdash^{n_2}_{\mathtt{Lev}} P}{\Theta \vdash^{n_1}_{\mathtt{Lev}} *p(\widetilde{x}).P} \qquad \text{(Lev-RIn)}$$

The rule Lev-Out ensures that the levels of all active outputs are smaller than n. Note the difference between Lev-In and Lev-RIn; the level n_1 of the judgment does not change in Lev-In, while in Lev-RIn, the level of the judgment changes from n_1 to n_2 (as the innermost replication enclosing P is $*p(\widetilde{x})$, which has level n_2).

Example 5.1. Recall the process P given at the beginning of this section:

$$\overline{c}[b] \mid c(x).(\overline{x} \mid *a.\overline{x}).$$

It is typed as follows.

$$\dfrac{\Theta \vdash^1_{\mathtt{Lev}} \overline{c}[b] \qquad \dfrac{\dfrac{\Theta, x : \sharp^0[\,] \vdash^1_{\mathtt{Lev}} \overline{x} \quad \dfrac{\Theta, x : \sharp^0[\,] \vdash^1_{\mathtt{Lev}} \overline{x}}{\Theta, x : \sharp^0[\,] \vdash^1_{\mathtt{Lev}} *a.\overline{x}}}{\Theta, x : \sharp^0[\,] \vdash^1_{\mathtt{Lev}} \overline{x} \mid *a.\overline{x}}}{\Theta \vdash^1_{\mathtt{Lev}} c(x).(\overline{x} \mid *a.\overline{x})}}{\Theta \vdash^1_{\mathtt{Lev}} P}$$

Here, $\Theta = a : \sharp^1[\,], c : \sharp^0[\sharp^0[\,]]$. □

The main limitation of Lev is that, in certain cases, an input $*p(\widetilde{x}).P$ cannot have outputs at p, or at names with the same type as p, in the body P. Because of this limitation, for instance, Example 3.10 cannot be typed (note that p is used for output in the body of the replicated input $*p(x, y, n, s).$). The other type systems of [Deng and Sangiorgi 2006] allow more freedom by using more sophisticated types and weight measure, and exploiting techniques from term-rewriting based on lexicographical and multiset ordering.

In particular, the fourth type system, PO, introduces a notion of partial order on channels. Roughly, the partial order makes it possible to type patterns $*q(\widetilde{y}).(\cdots \overline{p}[\widetilde{v}] \cdots)$, where the output at p is not under inner replications, in which the level of p is equal to that of q (hence the pattern is not typable, for instance, in the system Lev), but p is smaller than q in the partial order.[3] This pattern appears in Example 3.11 of the binary tree (in the insert, the replicated input at i followed by the outputs at i_l and i_r towards the children nodes; and similarly in the search). Thus, PO judgments are of the form $\Theta; \mathcal{R} \vdash^n_{\mathtt{PO}} P$ where n is a level information and $\mathcal{R}$ a partial order on the names in Θ. The type of a channel may be decorated with a partial order, which expresses partial order requirements on the tuples of values exchanged along that channel; for instance the requirement that the second component should always be smaller than the third, or smaller than a certain channel.[4]

[3] We are simplifying the explanation; for instance, the input of q need not be the first input of the replication.

[4] The latter possibility, reminiscent of dependent types, was not actually present in [Deng and Sangiorgi 2006], but represents a straightforward extension, at least if names with dependent types cannot be communicated; this possibility is needed in the typing of binary tree example.

5.2 Conditions for robust termination

As an example, we first illustrate the conditions for robust termination on the system Lev of the previous section.

Given a type environment Θ, we write $\mathtt{CTypes}(\Theta)$ for the set of channel types used in Θ. That is, for each channel type assignment $v : T$ in Θ we place T and all channel type subcomponents of T in $\mathtt{CTypes}(\Theta)$. For instance, if T is $\sharp^{n_1}[\sharp^{n_2}[\mathtt{Bool}], \mathtt{Bool}]$ then T and $\sharp^{n_2}[\mathtt{Bool}]$ should be in $\mathtt{CTypes}(\Theta)$.

Let *Erase* be the function that strips off the level information from the Lev types and returns simple types. The condition that we add for the robust termination of a process P under Γ (where Γ is an ST type environment) is the following: there is Θ s.t. $\Theta \vdash_{\mathtt{Lev}} P$, $\mathit{Erase}(\Theta) = \Gamma$, and *Erase* is injective on all types used in Θ (that is, $\mathtt{CTypes}(\Theta)$). Injectivity is maintained under the (Γ-typed) transitions of P because:

—Lev has the subject reduction property, therefore any τ-derivative of P remains typed in Θ;

—an input or output derivative of P is typed under a type environment that extends Θ with types that already appear in Θ (for instance, in case of the input of a fresh name at c, the type for the fresh name is extracted from the type of c in Θ).

The robust termination for P under Γ immediately follows from the termination properties of Lev and the above invariance under transitions, which guarantees typability in Lev after any sequence of ST-typed transitions.

In the process P of (1), which is not robustly terminating, the above conditions fail because any Lev typing for P must have assignments $c : \sharp^{\ell}[\sharp^{m}[\,]], a : \sharp^{n}[\,]$ for levels ℓ, m, and n with $n > m$; *Erase* is not injective on $\mathtt{CTypes}(\Theta)$, for it returns the same simple types on $\sharp^{m}[\,]$ and $\sharp^{n}[\,]$.

Generalizing the above reasoning, we define some abstract conditions with which a type system for termination also guarantees robust termination (Lemma 5.3); we then discuss refinements of the conditions (Section 5.3).

We denote by Ter a generic type system for termination, and with $\Theta \vdash_{\mathtt{Ter}} P$ a judgment in Ter, ignoring possible additional information in the judgment (such as the levels of Lev), for this information is not relevant in the results below. We assume that the judgment is closed under renaming, i.e., if $\Theta, p : T \vdash_{\mathtt{Ter}} P$ and q is fresh (i.e., it does not appear in Θ or P), then $\Theta, q : T \vdash_{\mathtt{Ter}} [p \mapsto q]P$.

Definition 5.1. Let f be a function from the types of Ter to those of ST. We say that $\Theta \vdash_{\mathtt{Ter}} P$ is *f-admissible* if both $\Theta \vdash_{\mathtt{Ter}} P$ and $f(\Theta) \vdash_{\mathtt{ST}} P$ hold and, for all substitutions σ that are closing for $f(\Theta)$, whenever $\sigma f(\Theta) \vdash_{\mathtt{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} P'$, there is a closed Θ' s.t. $\Theta' \vdash_{\mathtt{Ter}} P'$. (Where $f(\Theta)$ is the ST type environment obtained by replacing each type assignment $v : T$ in Θ with $v : f(T)$.)

f-admissibility ensures us that f can be used to turn a typing $\Theta \vdash_{\mathtt{Ter}} P$ into a valid ST typing and, furthermore, typing in Ter is preserved under (ST-typed) transitions, hence we have:

THEOREM 5.2. *Suppose* Ter *is a type system that guarantee termination (i.e., whenever* $\Delta \vdash_{\mathtt{Ter}} Q$*, for* Δ *closed, then* Q *terminates), and* f *a function from*

the types of Ter *to those of* ST*. If* $\Theta \vdash_{\mathtt{Ter}} P$ *is* f*-admissible then* P *is robustly terminating under* $f(\Theta)$*.*

PROOF. Let σ be a closing substitution for $f(\Theta)$. it suffices to show that $\sigma f(\Theta) \vdash_{\mathtt{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} P'$ implies that P' is terminating. By the definition of f-admissibility, there exists Θ' such that $\Theta' \vdash_{\mathtt{Ter}} P'$. By the assumption that Ter guarantees termination, P' is terminating. □

If $\Theta \vdash_{\mathtt{Ter}} P$ and $f(\Theta) \vdash_{\mathtt{ST}} P$, and provided that the definition of f is compositional, then f-admissibility normally follows from a Subject-Reduction theorem for Ter and injectivity of f on $\mathtt{CTypes}(\Theta)$ (that, we recall, is the set of channel types used in Θ).

LEMMA 5.3. *Given a type system* Ter*, and a function* f *from the types of* Ter *to those of* ST *(and mapping* Bool *onto* Bool*), suppose* f *and* Ter *satisfy the following conditions:*

(1) *whenever* $\Theta \vdash_{\mathtt{Ter}} P$ *also* $f(\Theta) \vdash_{ST} P$*;*

(2) *whenever* $\Theta \vdash_{\mathtt{Ter}} P$*, with* Θ *closed, and* $P \xrightarrow{\eta} P'$ *and, furthermore, when* η *is an input, all names received are fresh (i.e., these names do not appear in* Θ*), then there is* Θ' *closed s.t.* $\Theta' \vdash_{\mathtt{Ter}} P'$ *with* $\mathtt{CTypes}(\Theta') \subseteq \mathtt{CTypes}(\Theta)$*. Moreover, in the case of input with fresh names, say* $\eta = a[\widetilde{v}]$*, it should be* $f(\Theta)(a) = \sharp[f(\Theta')(\widetilde{v})]$ *and* $\Theta(p) = \Theta'(p)$ *for all names* $p \notin \{a, \widetilde{v}\}$*.*

(3) *whenever* $\Theta \vdash_{\mathtt{Ter}} P$ *and* $\Theta(p) = \Theta(q)$ *also* $\Theta \vdash_{\mathtt{Ter}} [q \mapsto p]P$*;*

Then for any Θ *and* P*, if* f *is injective on* $\mathtt{CTypes}(\Theta)$ *then* $\Theta \vdash_{\mathtt{Ter}} P$ *is* f*-admissible.*

In the lemma, the first condition ensures us that f converts a valid judgment in Ter into one valid in ST. The second condition is a Subject-Reduction property for Ter on transitions; $\mathtt{CTypes}(\Theta') \subseteq \mathtt{CTypes}(\Theta)$ essentially ensures that the types of fresh names received in an input or emitted in an output along a channel a can be deduced from the type of a. The third condition says that Ter maintains typability under substitution of names with names of the same type. In the conclusions, the injectivity condition on f is only on the initial type environment for P. It does not affect other environments that appear in the derivation of $\Theta \vdash_{\mathtt{Ter}} P$; therefore the types of the restricted names of P need not be subject to the condition.

PROOF. We prove that $\Theta \vdash_{\mathtt{Ter}} P$ is f-admissible. First, by condition (1) of the lemma, both $\Theta \vdash_{\mathtt{Ter}} P$ and $f(\Theta) \vdash_{\mathtt{ST}} P$ hold.

Consider now a substitution σ that is closing for $f(\Theta)$. We have $\sigma f(\Theta) \vdash_{\mathtt{ST}} \sigma P$. The substitution σ replaces each variable x in $f(\Theta)$ with either a value that is defined in $f(\Theta)$ with the same type as x, or with a fresh name. Since f is injective, the same property holds if σ is applied to Θ, therefore using also condition (3) of the lemma, we also have $\sigma\Theta \vdash_{\mathtt{Ter}} \sigma P$. (Note that if σ replaces x with a fresh name then $f(\Theta)(x)$ is a channel type and therefore also $\Theta(x)$ is a channel type, by the definition of f and its injectivity.)

We now show that whenever $\Theta \vdash_{\mathtt{Ter}} P$, with Θ closed and f injective on $\mathtt{CTypes}(\Theta)$, and $f(\Theta) \vdash_{\mathtt{ST}} P \xrightarrow{\eta} P'$, then there is Θ' closed with $\Theta' \vdash_{\mathtt{Ter}} P'$ and f injective on $\mathtt{CTypes}(\Theta')$. This would ensure us the remaining condition for f-admissibility (on the typability of all typed derivatives of a closed process).

If η is not an input, then this follows by condition (2) of the lemma. Suppose now η is an input, say $a[v]$ and v is a name (the case of monadic input is simpler to explain, the general case of polyadic input is however similar). If v is fresh then assertion follows from condition (2) of the lemma as before. Suppose now v appears in Θ, and let w be a fresh name. We also have $f(\Theta) \vdash_{\mathtt{ST}} P \xrightarrow{a[w]} P''$, for some P'' with $P' = [v \mapsto w]P''$. Again by condition (2) of the lemma we deduce that there is Θ' s.t. $\Theta' \vdash_{\mathtt{Ter}} P''$ with $\mathtt{CTypes}(\Theta') \subseteq \mathtt{CTypes}(\Theta)$. Now, if $f(\Theta)(a) = \sharp[T]$, then T must also be the type of v in $f(\Theta)$ (because we have $f(\Theta) \vdash_{\mathtt{ST}} P \xrightarrow{a[v]} P'$) and, since it must be $f(\Theta)(a) = \sharp[f(\Theta')(w)]$, type T is also the type of w in $f(\Theta')$. Further, since v does not appear in the names of the input $a[w]$, the type of v is also T in $f(\Theta')$. By the injectivity of f, we deduce that the types of v and w are the same in Θ'. We can therefore apply condition (3) of the lemma and infer $\Theta' \vdash_{\mathtt{Ter}} [v \mapsto w]P'' = P'$. □

Lemma 5.3 is applicable to the system for termination in [Sangiorgi 2006]. This system uses ST types together with some syntactic conditions on processes; it is straightforward to put these syntactic conditions into the type system, obtaining a refinement of ST that satisfies the hypothesis of the lemma. Lemma 5.3 is also applicable and to all but one of the four type systems in [Deng and Sangiorgi 2006], where the function f in the lemma can be taken to be the *Erase* function mentioned earlier in the section that strips off levels and other termination information. As an example, we discuss the type system Lev that we also used for Example 5.1. Condition (1) holds: function f just strips off the levels, hence $f(\Theta) \vdash_{\mathtt{ST}} P$ follows from the fact that the Lev system is a refinement of the ST system and as such already encompassing all type checks in ST. For condition (2), if P is well-typed under Θ in Lev, and P makes a transition that represents an internal movement or an output where only free channels or booleans are sent, then in the transition the set of free channels does not increase and therefore the derivative process remains well-typed under Θ. If the action is an input $a[\widetilde{v}]$ where $\widetilde{v}$ are fresh channels and $\Theta(a)$ is $\sharp^n[\widetilde{T}]$, then the derivative will be typed in an environment Θ' that extends Θ with the types $\widetilde{v}:\widetilde{T}$ for the newly arrived channels. Moreover, the types for the $\widetilde{v}$ can be deduced from that of a, thus the *Erase* function f satisfies $f(\Theta)(a) = f(\Theta')(a) = \sharp[f(\Theta')(\widetilde{v})]$. The case of an output in which some of the channels emitted are bound, that is, they are extruded channels, is similar, as the extruded names are fresh and their type is determined by that of the channel along which the output occurs. Finally, condition (3) of the lemma holds because the type checks in Lev on names simply involve looking up the type of a name in the type environment; the identity of the names is irrelevant, and may be modified as long as the type remains the same.

An exception for Lemma 5.3 is the system PO of [Deng and Sangiorgi 2006], with partial orders. We discuss refinements of the lemma that can handle PO and the system of [Yoshida et al. 2004] in the next section.

5.3 Discussions and refinements

Injectivity in Lemma 5.3. The main constraint in Lemma 5.3 is the injectivity of f. This says that the channel types that appear in Θ (that is, the types of the free

names of P and, recursively, of the names that can be communicated along them) should be the same whenever the corresponding simple types are the same.

This requirement may be demanding when the processes have many free names with the same simple type, as the termination analysis may need to distinguish some of them. For instance, in a CCS-like process, where all names have the same type, the injectivity condition on f would amount to requiring that all free names should have the same termination type (whereas restricted names can have arbitrary type). Thus we would be unable to distinguish the process $*a.\overline{b} \mid \overline{a}$, which is robustly terminating, from the process $*a.\overline{a} \mid \overline{a}$, which is non-terminating, as the name a and b have the same simple type. (The type system with levels `Lev`, mentioned above, recognizes $*a.\overline{b} \mid \overline{a}$ as terminating, by assigning to name a a level greater than that of b, and in doing so it indeed violates the injectivity condition.)

However, as shown by the example in (1), what makes robust termination harder than termination is channel aliasing on inputs, occurring when a process receives channels that it already possessed. We can thus improve Lemma 5.3 by requiring a milder form of injectivity for f. Let $\mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P)$ be the set of the channel types which are assigned to the variables of P in a typing derivation of $\Theta \vdash_{\mathtt{Ter}} P$ (assuming that such derivation is unique). We replace the injectivity condition of Lemma 5.3 with the following:

$$\begin{array}{c}\text{for all } T \in \mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P) \cap \mathtt{CTypes}(\Theta) \text{ and } S \in \mathtt{CTypes}(\Theta),\\ \text{if } f(T) = f(S) \text{ then } S = T.\end{array} \tag{2}$$

This is weaker because usually $\mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P)$ will be significantly smaller than $\mathtt{CTypes}(\Theta)$. For instance, if P is a CCS-like process, then $\mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P)$ is always empty, for any Θ. Further, a variable need not be taken into account when computing $\mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P)$ if no aliasing on that variable is possible (that is, after instantiation, the variable cannot become equal to another name in the process). In dialects of the π-calculus such as πI [Sangiorgi 1996], aliasing is forbidden altogether since only fresh names can be transmitted, hence $\mathcal{OT}(\Theta \vdash_{\mathtt{Ter}} P)$ is always empty. In general, any technique for computing the aliasing set of a variable (the set of names with which the variable could be instantiated), such as control flow analysis and abstract interpretation [Bodei et al. 1998; Feret 2005], can be helpful to further improve (2).

Another way of weakening the injectivity condition on $\mathtt{CTypes}(\Theta)$ of Lemma 5.3 is to impose a distinction on the types of free names of a process that "accidentally" have the same simple types. This could be achieved in various ways. An example is to adopt named forms of types, as for instance in Milner's *sorting system* [Milner 1993], where types have a name and type equality is given by name equality. Milner's sorting systems is indeed the "by-name" equivalent of the "structural" `ST` system. Using a sorting, names with the same simple type can be distinguished by giving different names to their types. There is in fact a *most precise sorting* for any process P; that is, a sorting environment in which two names have the type only if this is *necessary* for the typing of the process (therefore the two names must have the same type in any sorting environment in which P is typable). Computing the most precise sorting can be done in polynomial time, using a variant of the algorithm for type inference in `ST`. All results and examples shown in this paper using `ST` as a base typing can be transplanted to the sorting system.

Another possibility, equivalent to adopting a sorting, is to add dummy components to the values exchanged on certain channels (for instance, in the previous example of $*a.\overline{b} \mid \overline{a}$, we could take b as a name along which *pairs* of unit values are exchanged). However, when the robust termination analysis is applied to a subcomponent P of a larger system, a type distinction on two names a and b that is needed for the robust termination of P might be forbidden by usages of the names in other processes (for instance, both names could appear in outputs along the same channel, in which case, unless the type of this channel is polymorphic, a and b must have the same type). For these situations, we discuss in Appendix D a modification of the type systems in Deng and Sangiorgi [2006], where levels are replaced by *intervals*.

Substitutions in Lemma 5.3. Another possible source of failure in Lemma 5.3 is the substitution condition (3). This fails on the system `PO` of [Deng and Sangiorgi 2006], with the partial orders, because legal substitutions in `PO` must respect, besides types, also the partial order. Condition (3) also fails in Yoshida, Berger, and Honda's type system for termination [Yoshida et al. 2004], as it makes use of graph types with linearity information, and on linear types only a limited form of substitution holds. For this problem, the condition on aliasing mentioned earlier can again be useful. For instance, in languages without aliasing such as πI condition (3) can be dropped, together with the requirements in the final sentence of condition (2) ("Moreover, in the case ..."). Thus Lemma 5.3 is applicable to the system in Yoshida et al. [2004], which is formalized on a variant of πI. Besides via the control of aliasing, another way of applying Lemma 5.3 to the system `PO` is to require, in condition (3) of the lemma and in its conclusion, that the environment Θ is undecorated. Here, if a type T does not contain partial order requirements, then T is *undecorated*. Similarly, an environment Θ is undecorated if all its types (i.e., $\mathtt{CTypes}(\Theta)$) are undecorated. This maintains the typability of Example 3.11. (Indeed, the names with a decorated type are often just a few and restricted, hence they do not appear in the initial type environment.)

6. IMPLEMENTATION

We have implemented the new weak lock-freedom analysis as a feature of TyPiCal Version 1.6.0 [Kobayashi 2005b]. TyPiCal takes as an input a program written in the π-calculus (extended with data structures such as pairs and lists), and marks all input/output prefixes that are guaranteed to succeed. The strong lock-freedom analysis has not been implemented yet.

Figure 8 shows a sample input program for TyPiCal, corresponding to the process in Example 3.8. An output process $\overline{a}[v]$ is written as $a!v$, and an input process $a(x).P$ is written as $a?x.P$. and Figure 9 is the output produced by the program. Input and output operations that are guaranteed to succeed are marked by `??` and `!!` respectively.

The original type system for lock-freedom (reviewed in Section 3.1) had been implemented already [Kobayashi 2005a; 2006]. A major challenge in the implementation of the new system was to automate verification of the robust termination property. We have modified the type systems of Deng and Sangiorgi [2006], so that the resulting systems can guarantee robust termination, and also so to make them

```
(new fact_it in
   *fact?x.(let n=fst(x) in let reply=snd(x) in
      fact_it!(n, (1, reply)))
 | *fact_it?x.(let n=fst(x) in
    let acc = fst(snd(x)) in let reply=snd(snd(x)) in
     if n=0 then reply!acc
     else fact_it!(n - 1,(acc * n,reply))))
| *(new r in fact!(n, r) | r?result.print!result)
```

Fig. 8. A sample input for TyPiCal

```
(new fact_it in
   *fact?x.(let n=fst(x) in let reply=snd(x) in
      fact_it!(n, (1, reply)))
 | *fact_it?x.(let n=fst(x) in
    let acc = fst(snd(x)) in let reply=snd(snd(x)) in
     if n=0 then reply!acc

     else fact_it!(n - 1,(acc * n,reply))))
| *(new r in fact!!(n, r) | r??result.print!result)
```

Fig. 9. The output produced by TyPiCal

more suited for automatic verification (e.g., using heuristic and incomplete algorithms when the original ones were NP-complete). The implementation of robust termination analysis in TyPiCal and its difference from [Deng and Sangiorgi 2006] are summarized as follows.

—As summarized in Section 5.1, in all the four type systems of Deng and Sangiorgi [2006], level information assigned to each channel type plays a central role in guaranteeing termination. In the TyPiCal implementation, a level variable is attached to each channel type, and constraints on the level variables are generated and solved. For robust termination, we have also added an extra requirement for the injectivity of f (recall Theorem 5.2 and Lemma 5.3).

—The second type system of Deng and Sangiorgi [2006] allows a process of the form $*c(x).(\cdots\overline{p}[v]\cdots)$ either if c has a greater level than p, or if c and p have the same level and v is always smaller than x with respect to the order on natural numbers. This feature can be used for typing primitive recursion. In the TyPiCal implementation, the size change relation between arguments of channels (e.g., x and v above) is generated, and then the consistency of the size change relation is checked using a size change termination library [Ben-Amram and Lee 2007]. Thanks to this extension, the resulting type system is more expressive than the original type system [Deng and Sangiorgi 2006]; For example, we can handle non-primitive recursion such as an Ackermann function server.

—The third type system of Deng and Sangiorgi [2006] is NP-complete [Demangeon et al. 2008]. Thus, we use a heuristic, incomplete algorithm to handle it.

—The fourth type system of Deng and Sangiorgi [2006] allows a process of the form $*c(y).(\cdots\overline{p}[v]\cdots)$ either if c has a greater level than p, or if c and p have the same level, and c is greater than p with respect to a certain partial order on

	termination analysis	lock-freedom analysis	lock-freedom analysis (auto)
factorial	0.01 sec	0.02 sec	0.02 sec
broadcast	0.01 sec	0.05 sec	0.13 sec
btree	0.02 sec	5.47 sec	10.62 sec
stable	0.01sec	0.11 sec	0.22 sec
eventchan	0.03 sec	0.20 sec	0.62 sec

Table I. Analysis time (measured on a machine with Intel Pentium 1.2GHz and 500MB memory)

channels. We have implemented a separate analysis to infer the channel creation order, and use it as the partial order.

We have carried out preliminary experiments to test the feasibility of our lock-freedom analysis. Table I summarizes the result. "factorial," "broadcast," and "btree" are the examples discussed in Section 3.4. "stable" is a variation of the symbol table example taken from Deng and Sangiorgi [2006]. "eventchan" is an implementation of event channels, which was originally a sample program of Pict [Pierce and Turner 2000], and rewritten for TyPiCal. Those programs are available in the distribution of TyPiCal [Kobayashi 2005b].

All the programs have been verified successfully. The second column shows running times for robust termination analysis only. The third column shows those for the whole (weak) lock-freedom analysis of programs having annotations on where the bybrid rule should be applied (i.e., the result of running TyPiCal with "-wl" option). The rightmost column shows running times for lock-freedom analysis of programs without the annotations (i.e., the result of running TyPiCal with "-wlauto" option). Given non-annotated programs, TyPiCal with "-wlauto" option first performs deadlock-freedom analysis and lock-freedom analysis (without using the hybrid rule). By comparing the results, TyPiCal heuristically inserts annotations on where the hybrid rule should be applied. It then re-run lockfreedom analysis for the annotated programs. Thus, the current "-wlauto" mode is 2–3 times slower than the "-wl" mode. As can be seen in the table, the new components (dealing with termination) run fast; most of the analysis time is spent by the other components (dealing with deadlock- and lock-freedom). We have also tested robust termination analysis for all the examples given in Deng and Sangiorgi [2006], and confirmed that they were verified successfully.

7. DISCUSSIONS

This section informally discusses further extensions of our type system. We also describe some ideas for using model checkers to verify robust deadlock-freedom.

7.1 Relaxing Robust Termination/Confluence

One of the main advantages of our hybrid rules is that deadlock-freedom, termination, and confluence are required only locally, for the processes on which the hybrid rules are applied. The requirement may be, however, still too demanding. For example, consider a process:

$$(\nu f)\,(\overline{f}[\mathtt{rnd}(), a] \mid *f(n, r).\,(\mathbf{if}\ n = 0\ \mathbf{then}\ \overline{r}\ \mathbf{else}\ \overline{f}[n-1, r] \mid P)).$$

Suppose that P does not read from f. The process will eventually send a message on a, no matter whether P diverges. Our hybrid rules are, however, applicable only when P is also terminating (and partially confluent, in the case of SLT-Hyb).

To overcome the limitation above, we can modify the definitions of robust deadlock-freedom/termination/confluence, so that only marked actions are taken into account. We write $\xrightarrow{\tau^\circ}$ for the τ-transition on a marked prefix or an if-expression. The definitions of robust deadlock-freedom and termination can be modified as follows.

Definition 7.1 *(robust $\circ$-deadlock-freedom).* The relation $\Delta \models_{\mathtt{RD}\circ} P$ is the largest relation such that $\Delta \models_{\mathtt{RD}\circ} P$ implies all of the following conditions.

(1) If Δ is closed and $rel(\Delta)$, then:
   —If P has a marked prefix at top level, then $P \xrightarrow{\tau^\circ}$.
   —If $ob_!(\Delta(a)) \neq \infty$, then either $P \xrightarrow{(\nu \widetilde{c})\,\overline{a}[\widetilde{b}]}$ or $P \xrightarrow{\tau^\circ}$.
   —If $ob_?(\Delta(a)) \neq \infty$, then either $P \xrightarrow{a[\widetilde{b}]}$ or $P \xrightarrow{\tau^\circ}$.
(2) If $[v \mapsto a]\Delta$ is well-defined, then $[v \mapsto a]\Delta \models_{\mathtt{RD}\circ} [v \mapsto a]P$.
(3) If $P \xrightarrow{\eta} P'$ and, furthermore, when η is an input, all names received are fresh, then $\Delta \xrightarrow{\eta} \Delta'$ and $\Delta' \models_{\mathtt{RD}} P'$ for some Δ'.

We say that P is *robustly $\circ$-deadlock-free* under Δ if $\Delta \models_{\mathtt{RD}\circ} P$ holds.

Definition 7.2 *robust $\circ$-termination.* A process P is *$\circ$-terminating* if there is no infinite transition sequence of the form $P \xrightarrow{\tau^\circ} P_1 \xrightarrow{\tau^\circ} P_2 \xrightarrow{\tau^\circ} \cdots$. An (open) process P is *robustly $\circ$-terminating under* Γ, written $\Gamma \models_{\mathtt{RTer}\circ} P$, if $\Gamma \vdash_{\mathtt{ST}} P$, and for every closing substitution σ for Γ and for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\sigma\Gamma \vdash_{\mathtt{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is $\circ$-terminating.

By using the robust $\circ$-deadlock-freedom and termination, the hybrid rule LT-Hyb can be modified as follows.

$$\frac{\Delta \models_{\mathtt{RD}\circ} P \qquad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}\circ} P \qquad \mathit{nocap}(\Delta)}{\Delta \vdash_{\mathtt{LT}} P} \qquad \text{(LT-HybE)}$$

A similar modification is possible for the rule SLT-Hyb for strong lock-freedom.

It is not difficult to adopt verification methods of robust deadlock-freedom, termination, and confluence to the corresponding robust $\circ$ conditions. For robust $\circ$-deadlock-freedom, we can modify Kobayashi's type system for deadlock-freedom [Kobayashi 2006], so that a prefix is marked if and only if its capability level is finite. For robust $\circ$-termination, we can first perform program slicing to eliminate communications that do not affect marked actions, and then apply robust termination analysis.

7.2 Relaxing the *nocap* condition

The present side condition $\mathit{nocap}(\Delta)$ for LT-Hyb is sometimes too restrictive for local reasoning. For example, consider $\mathit{Client} \mid \mathit{Server}_1 \mid \mathit{Server}_2$, where *Client* sends a request to Server_1, which consults Server_2 to answer the request. Then, we have

to apply LT-Hyb to $Server_1 \mid Server_2$ rather than $Server_1$ alone, since $Server_1$'s type environment would contain a capability to consult $Server_2$.

One approach to relaxing (or eliminating, actually) the *nocap* condition is to impose a stronger requirement on robust deadlock-freedom. We modify the definition of $\Delta \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]} \Delta'$ as follows.

$$\frac{U \xrightarrow{!} U' \qquad \Delta, \widetilde{c}:\widetilde{\mathtt{L}_c} \leq \Delta' \mid \widetilde{b}:\uparrow\widetilde{\mathtt{L}} \qquad rel(\widetilde{\mathtt{L}_c})}{\Delta, a:\sharp_U[\widetilde{\mathtt{L}}] \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]} \Delta', a:\sharp_{U'}[\widetilde{\mathtt{L}}]}$$

The only change is in the second premise, where $\uparrow$ is applied to $\widetilde{\mathtt{L}}$. This ensures that the level of an obligation is decreased by one whenever it is passed through channels. For example,

$$a:\sharp_{?\,\infty}^{?\,0}[\sharp_{!\,\infty}^{!\,1}[\mathtt{Bool}]], b:\sharp_{!\,\infty}^{!\,2}[\mathtt{Bool}] \xrightarrow{\overline{a}[b]} a:\sharp_{\mathbf{0}}[\sharp_{!\,\infty}^{!\,1}[\mathtt{Bool}]]$$

hold, but

$$a:\sharp_{?\,\infty}^{?\,0}[\sharp_{!\,\infty}^{!\,1}[\mathtt{Bool}]], b:\sharp_{!\,\infty}^{!\,1}[\mathtt{Bool}] \xrightarrow{\overline{a}[b]} a:\sharp_{\mathbf{0}}[\sharp_{!\,\infty}^{!\,1}[\mathtt{Bool}]]$$

does not.

We strengthen robust deadlock-freedom and robust termination as follows.

Definition 7.3 *robust strong $\circ$-deadlock-freedom.* The relation $\Delta \models_{\mathtt{SRD}\circ} P$ is the largest relation such that $\Delta \models_{\mathtt{SRD}\circ} P$ implies all of the following conditions.

(1) If Δ is closed and P has a marked prefix at top-level, then one of the following conditions holds:
   —$P \xrightarrow{\tau^\circ}$
   —$cap_?(\Delta(a)) < ob_!(\Delta(a))$ and $P \xrightarrow{a^\circ[\widetilde{b}]}$
   —$cap_!(\Delta(a)) < ob_?(\Delta(a))$ and $P \xrightarrow{(\nu\widetilde{c})\,\overline{a}^\circ[\widetilde{b}]}$
(2) If Δ is closed and $ob_!(\Delta(a)) \neq \infty$, then one of the following conditions holds:
   —$P \xrightarrow{\tau^\circ}$
   —$P \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}$
   —$cap_?(\Delta(d)) < ob_!(\Delta(a))$ and $P \xrightarrow{d^\circ[\widetilde{b}]}$
   —$cap_!(\Delta(d)) < ob_!(\Delta(a))$ and $P \xrightarrow{(\nu\widetilde{c})\,\overline{d}^\circ[\widetilde{b}]}$
(3) If Δ is closed and $ob_?(\Delta(a)) \neq \infty$, then one of the following conditions holds:
   —$P \xrightarrow{\tau^\circ}$
   —$P \xrightarrow{a[\widetilde{b}]}$
   —$cap_?(\Delta(d)) < ob_?(\Delta(a))$ and $P \xrightarrow{d^\circ[\widetilde{b}]}$
   —$cap_!(\Delta(d)) < ob_?(\Delta(a))$ and $P \xrightarrow{(\nu\widetilde{c})\,\overline{d}^\circ[\widetilde{b}]}$
(4) If $[v \mapsto a]\Delta$ is well-defined, then $[v \mapsto a]\Delta \models_{\mathtt{SRD}\circ} [v \mapsto a]P$.
(5) If $P \xrightarrow{\eta} P'$ and, furthermore, when η is an input, all names received are fresh, then $\Delta \xrightarrow{\eta} \Delta'$ and $\Delta' \models_{\mathtt{SRD}\circ} P'$ for some Δ'.

We say that P is *robustly and strongly* $\circ$*-deadlock-free* under Δ if $\Delta \models_{\mathtt{SRD}\circ} P$ holds.

Definition 7.4 *robust strong* $\circ$*-termination.* A transition is *marked* if it is an input, output, or τ-transition on a marked prefix or if it is a reduction on an if-expression. A process P is *strongly* $\circ$*-terminating* if there is no infinite internal sequence of marked (input, output, or τ) transitions. An (open) process P is *robustly and strongly* $\circ$*-terminating under* Γ, written $\Gamma \models_{\mathtt{RSTer}\circ} P$, if $\Gamma \vdash_{\mathtt{ST}} P$, and for every closing substitution σ for Γ and for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\sigma\Gamma \vdash_{\mathtt{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is strongly $\circ$-terminating.

We conjecture that the following hybrid rule is sound.

$$\frac{\Delta \models_{\mathtt{SRD}\circ} P \qquad \mathit{Erase}(\Delta) \models_{\mathtt{RSTer}\circ} P}{\Delta \vdash_{\mathtt{LT}} P} \qquad \text{(LT-HybE2)}$$

7.3 Using Model Checkers for Robust Deadlock-Freedom

In Section 3.2, we mentioned that types systems, notably Kobayashi's one [Kobayashi 2006] can be used for verification of robust deadlock-freedom. In certain special cases, however, we can appeal to model checkers. This is an important advantage since type systems for deadlock-freedom usually ignore value-dependent behaviors. For example, Kobayashi's type system [Kobayashi 2006] cannot verify the robust deadlock-freedom of:

$$(\mathbf{if}\ x > 0\ \mathbf{then}\ \overline{a}^{\circ}\ \mathbf{else}\ \mathbf{0}) \mid (\mathbf{if}\ x > 0\ \mathbf{then}\ a^{\circ}\ \mathbf{else}\ \mathbf{0})$$

On the other hand, model checkers can verify it instantly.

We consider here Δ is of the form $a : \sharp_U[\,]$ where U is of the following restricted form.

$$U ::= \mathbf{0} \mid !^t_\infty.U \mid ?^t_\infty.U$$

In this case, the verification problem of $\Delta \models_{\mathtt{RD}} P$ can be reduced to the ordinary model checking problem $P \models \mathit{u2l}(a, U) \wedge \mathit{OnlyA}$ in modal μ-calculus, where $\mathit{u2l}(a, U)$ is given by:

$$\begin{array}{l}
\mathit{u2l}(a, \mathbf{0}) = \nu X.(\neg\langle a\rangle \wedge \neg\langle\overline{a}\rangle \wedge [\tau]X) \\
\mathit{u2l}(a, !^t_\infty.U) = \\
\quad \begin{cases} \nu X.(\neg\langle a\rangle \wedge ([\overline{a}]\mathit{u2l}(a, U)) \wedge [\tau]X) \\ \qquad\qquad (\text{if } t = \infty) \\ \nu X.(\neg\langle a\rangle \wedge ([\overline{a}]\mathit{u2l}(a, U)) \wedge [\tau]X \wedge (\langle\overline{a}\rangle \vee \langle\tau\rangle)) \\ \qquad\qquad (\text{if } t \neq \infty) \end{cases} \\
\mathit{u2l}(a, ?^t_\infty.U) = \\
\quad \begin{cases} \nu X.(\neg\langle\overline{a}\rangle \wedge ([a]\mathit{u2l}(a, U)) \wedge [\tau]X) \\ \qquad\qquad (\text{if } t = \infty) \\ \nu X.(\neg\langle\overline{a}\rangle \wedge ([a]\mathit{u2l}(a, U)) \wedge [\tau]X \wedge (\langle a\rangle \vee \langle\tau\rangle)) \\ \qquad\qquad (\text{if } t \neq \infty) \end{cases}
\end{array}$$

$\mathit{u2l}(a, \mathbf{0})$ means that a is used for neither input nor output. $\mathit{u2l}(a, !^\infty_\infty.U)$ means that a can first be used only for output, and after output, a must be used according to U. If $t \neq \infty$, $\mathit{u2l}(a, !^t_\infty.U)$ also requires (by the subformula $\langle\overline{a}\rangle \vee \langle\tau\rangle$) that the process must use a for output unless it diverges.

OnlyA, which means that the process never performs an input or an output on names other than a, is

$$\nu X.(\wedge_{b\in\mathcal{L}\setminus\{a\}}(\neg\langle b\rangle \wedge \neg\langle\overline{b}\rangle) \wedge [a]X \wedge [\overline{a}]X \wedge [\tau]X).$$

It is not difficult to extend the above translation for a type environment with multiple names: $a_1 : \sharp_{U_1}[\,], \ldots, a_n : \sharp_{U_n}[\,]$. To deal with a more general case, we need to use logics for mobile processes [Caires and Cardelli 2003; Dam 1996].

As for model checking tools, there are some for mobile process calculi [Hugo Vieira and Viegas 2005]. For some restricted case, we may also be able to use other model checking tools such as SPIN [Holzmann 2003].

8. RELATED WORK

Several type systems for lock-freedom (sometimes referred to by different names) have been already proposed [Kobayashi 2002; 2005a; Yoshida 2002; Acciai and Boreale 2008; Sangiorgi 1999; Yoshida et al. 2004]. Our type system substantially improves the expressiveness of previous type systems; for instance, it can handle non-trivial recursive structures (e.g., the binary trees as in Example 3.11), and value-dependent behaviors. This is possible through a parameterization that appeals to other analyzers, in particular those for deadlock freedom (so that more powerful analyzers make the lock-freedom type system more powerful too). Most of the previous type systems [Kobayashi 2002; 2005a; Yoshida 2002; Yoshida et al. 2004; Sangiorgi 1999] do not handle recursion (such as those given in Section 3.4) well: if a channel is passed as an argument of a recursive call, lock-freedom on that channel is not guaranteed. Acciai and Boreale [2008] recently proposed a type system that can handle a limited form of recursion, but does not seem to work for non-trivial recursive structures like the binary tree Example 3.11, and imperative structures such as locks and reference cells. In Acciai and Boreale's type system, reasoning about termination is *hardwired* into the type system for lock-freedom. In contrast, our type system is parameterized by termination analysis, so that we can incorporate any other techniques for proving termination (in fact, in the implementation, we have already incorporated the technique based on size change graphs [Ben-Amram and Lee 2007]). Yoshida, Berger, and Honda's type system [Yoshida et al. 2004] can guarantee termination and a form of lock-freedom for encodings of simply-typed λ-terms. Our type system can also guarantee lock-freedom of those processes, using [Sangiorgi 2006] or [Yoshida et al. 2004] for the robust-termination analysis (and the extension of the `DT` type system in [Kobayashi 2006]). As already mentioned, the system [Yoshida et al. 2004] cannot handle recursion well. Another important point is that none of the previous type systems for lock-freedom, except Kobayashi's one [2005a], has been implemented. In fact, most of the type systems classify channels into a few usage patterns, and prepare separate typing rules for each of the usage patterns. Thus, verification based on those type systems would not be possible without heavy program annotations.

Type systems for deadlock-freedom have been studied extensively [Kobayashi 2006; Suenaga and Kobayashi 2007; Boyapati et al. 2002]. As already mentioned, deadlock-freedom is weaker than lock-freedom, so that those type systems alone cannot be used for lock-freedom analysis. For example, the divergent process ob-

tained by replacing $\overline{fact\_it}[n-1, x \times n, r]$ in Example 3.8 with $\overline{fact\_it}[n, x \times n, r]$ is deadlock-free.

The idea of reducing verification of lock-freedom to verification of robust termination is a reminiscence of Cook et al.'s work on reducing verification of liveness properties to that of fair termination [Cook et al. 2007]. The target language of their work is a sequential, imperative language and is quite different from our language, which is concurrent and allows dynamic creation of communication channels and threads. The used techniques are also quite different; they use model checking while we use types. It is not clear whether their technique can be effectively used for verification of lock-freedom in our language.

In general, model checking can be used for verification of lock-freedom. The current model checking technology does not seem, however, mature enough for automatic verification of liveness properties of concurrent programs that have infinite states and create threads and channels dynamically.

There are a number of methods for proving termination of programs, and they have been extensively studied in the context of term rewriting systems and sequential programs. The point of parameterizing our type system for lock-freedom by the robust termination property was to reuse those techniques for termination verification, instead of developing a sophisticated type system that can reason about both termination and deadlock within the single type system.

Demangeon et al. [2008] discuss the complexity of type inference problems for variants of Deng and Sangiorgi's type systems [2006]. In particular, they show that the third and fourth type systems of Deng and Sangiorgi [2006] are NP-complete and propose variants of them that admit polynomial-time type inference algorithms, at the price of reducing the expressiveness in certain cases (e.g., the binary tree example cannot be handled). Our current termination analysis algorithm in TyPiCal makes use of heuristic, incomplete algorithms, based on the original ones in [Deng and Sangiorgi 2006] and which further integrate [Deng and Sangiorgi 2006] with the size-change termination analysis [Ben-Amram and Lee 2007].

Parameterized, or hybrid, type systems of this kind presented in this paper are fairly rare in the literature, mainly due to the difficulties in combining the analyses. For instance, in Leroy's modular module system [Leroy 2000] a type system for module is presented that is parametric on the type system used for the core language. This is quite different from ours, as the judgments of the two type systems are similar and, most important, the world on which the two type systems operate—modules and core languages—are stratified, hence clearly separated. Among the approaches to combining type systems with other verification methods for concurrent programs, the closest to ours is probably Chaki et al. [2002], where a type system is used to extract CCS processes as abstract models of the π-calculus, and then a model checker verifies such models. In our case, by contrast, the parameterization in the typing rules makes the different analyses closely intertwined and makes it possible local applications of the parameterized analyses. Caires [2007] recently proposed a generic type system for the π-calculus, whose judgment is defined semantically; thus, the type system can be freely combined with other verification methods. It is however generally difficult to develop a completely semantic type system for complex properties like lock-freedom. Our approach (where robust

deadlock-freedom/termination/confluence are semantically defined) is a mixture of the syntactic and semantic approaches to defining type systems.

9. CONCLUSION AND FUTURE WORK

We have proposed a hybrid type system for lock-freedom. Unlike the previous type systems for lock-freedom, our type system can handle non-trivial recursive communication structures and can be fully automated. The key development was the special rules LT-Hyb and SLT-Hyb for combining four different analyses: lock-freedom, robust deadlock-freedom, robust termination, and robust confluence analyses. The rules allow local reasoning about deadlock-freedom, termination and confluence, thus avoiding application of those analyses to the whole program. We have also introduced the notion of robust termination, and presented a generic method for strengthening type systems for termination to guarantee robust termination.

The proposed verification framework has been implemented as an extension of TyPiCal and tested for non-trivial programs such as symbol tables and concurrent binary tree search.

An interesting direction for future work would be more integration with other verification techniques in TyPiCal program analysis tool, to take full advantage of our hybrid, parametrized type system. For example, since our type system is parameterized by verification methods for robust termination and deadlock-freedom, we can possibly use model checking techniques for proving termination [Cook et al. 2007] and deadlock-freedom (recall the discussion in Section 7). Since type-based analysis seems in general more efficient but inaccurate, a typical combination would be to first apply type-based analyses and then use model checking in case programs cannot be verified using types.

Future work also includes an application of the new lock-freedom analysis to dependency analyses, such as information flow analysis and program slicing [Honda and Yoshida 2007; Honda et al. 2000; Kobayashi 2005a]. To see why lock-freedom analysis is related to information flow analysis, consider an input process: $a(x).\overline{public}[\texttt{"Succeeded!"}]$. Note that it leaks information about whether or not the communication on a succeeds through channel *public*. Therefore, if it is unknown whether a communication on a high security channel a succeeds, only communications on high security channels are allowed after that communication, which are too restrictive. (In a sequential language, it corresponds to the restriction that once a high-security variable is accessed, only high-security computation is allowed afterwards). Thus, the previous type systems for information flow analysis of concurrent programs [Honda et al. 2000; Kobayashi 2005a] have been built on top of some form of type systems for (weak) lock-freedom. Information flow analysis can be made more accurate by replacing the underlying type systems for lock-freedom with ours. Resource usage analysis [Kobayashi et al. 2006] is also built on top of lock-freedom analysis; hence it can benefit from the lock-freedom analysis in this paper.

Acknowledgment

This work was partially supported by Kakenhi 20240001 and 17300003, and the European Project "HATS" (contract number 231620). We would like to thank

Eijiro Sumii for discussions on this work, Luca Aceto, Xavier Leroy, and Benjamin Pierce for pointers to relevant work, and anonymous referees for useful comments. We would also like to thank Roberto Bruni and Maurizio Gabbrielli for comments on a draft of this paper.

REFERENCES

Acciai, L. and Boreale, M. 2008. Responsiveness in process calculi. *Theor. Comput. Sci. 409,* 1, 59–93.

Ben-Amram, A. M. and Lee, C. S. 2007. Program termination analysis in polynomial time. *ACM Trans. Prog. Lang. Syst. 29,* 1 (Article 5).

Bidinger, P. and Compagnoni, A. B. 2009. Pict correctness revisited. *Theor. Comput. Sci. 410,* 2-3, 114–127.

Bodei, C., Degano, P., Nielson, F., and Nielson, H. R. 1998. Control flow analysis for the pi-calculus. In *Proceedings of CONCUR'98*, D. Sangiorgi and R. de Simone, Eds. LNCS, vol. 1466. Springer-Verlag, 84–98.

Boreale, M., Nicola, R. D., and Pugliese, R. 1999. Basic observables for processes. *Inf. Comput. 149,* 1, 77–98.

Boyapati, C., Lee, R., and Rinard, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA 2002*. ACM Press, 211–230.

Brinksma, E., Rensink, A., and Volger, W. 1995. Fair testing. In *Proceedings of CONCUR 1995*. LNCS, vol. 962. Springer-Verlag, 313–327.

Caires, L. 2007. Logical semantics of types for concurrency. In *Proceedings of CALCO 2007*. LNCS, vol. 4624. Springer-Verlag, 16–35.

Caires, L. and Cardelli, L. 2003. A spatial logic for concurrency (part I). *Info. Comput. 186,* 2, 194–235.

Chaki, S., Rajamani, S., and Rehof, J. 2002. Types as models: Model checking message-passing programs. In *Proc. of POPL*. ACM Press, 45–57.

Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y. 2007. Proving that programs eventually do something good. In *Proc. of POPL*. ACM Press, 265–276.

Cook, B., Podelski, A., and Rybalchenko, A. 2007. Proving thread termination. In *Proc. of PLDI*. ACM Press, 320–330.

Dam, M. 1996. Model checking mobile processes. *Info. Comput. 129,* 1, 35–51.

Demangeon, R., Hirschkoff, D., Kobayashi, N., and Sangiorgi, D. 2008. On the complexity of termination inference for processes. In *Proceedings of TGC 2007*. LNCS, vol. 4912. Springer-Verlag, 140–155.

Deng, Y. and Sangiorgi, D. 2006. Ensuring termination by typability. *Info. Comput. 204,* 7, 1045–1082.

Feret, J. 2005. Abstract interpretation of mobile systems. *J. Log. Algebr. Program. 63,* 1, 59–130.

Holzmann, G. J. 2003. *The SPIN Model Checker: Premier and Reference Manual*. Addison-Wesley.

Honda, K., Vasconcelos, V., and Yoshida, N. 2000. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*. LNCS, vol. 1782. Springer-Verlag, 180–199.

Honda, K. and Yoshida, N. 2007. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst. 29,* 6.

Hugo Vieira, L. C. and Viegas, R. 2005. The spatial logic model checker user's manual v1.0. TR-DI/FCT/UNL-05, `http://ctp.di.fct.unl.pt/SLMC/`.

Jones, C. 1993. A π-calculus semantics for an object-based design notation. In *Proceedings of CONCUR'93*. LNCS, vol. 715. Springer-Verlag, 158–172.

Kobayashi, N. 2002. A type system for lock-free processes. *Info. Comput. 177*, 122–159.

Kobayashi, N. 2005a. Type-based information flow analysis for the pi-calculus. *Acta Informatica 42,* 4-5, 291–347.

Kobayashi, N. 2005b. TyPiCal: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.

Kobayashi, N. 2006. A new type system for deadlock-free processes. In *Proceedings of CONCUR 2006*. LNCS, vol. 4137. Springer-Verlag, 233–247.

Kobayashi, N., Pierce, B. C., and Turner, D. N. 1999. Linearity and the pi-calculus. *ACM Trans. Prog. Lang. Syst. 21,* 5, 914–947.

Kobayashi, N., Suenaga, K., and Wischik, L. 2006. Resource usage analysis for the pi-calculus. *Logical Methods in Computer Science 2,* 3:4, 1–42.

Leroy, X. 2000. A modular module system. *J. Funct. Program. 10,* 3, 269–303.

Milner, R. 1993. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Springer-Verlag.

Natarajan, V. and Cleaveland, R. 1995. Divergence and fair testing. In *Proceedings of ICALP'95*. LNCS, vol. 944. Springer-Verlag, 648–659.

Pierce, B. C. and Turner, D. N. 2000. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 455–494.

Sangiorgi, D. 1996. π-calculus, internal mobility and agent-passing calculi. *Theor. Comput. Sci. 167,* 2, 235–274.

Sangiorgi, D. 1999. The name discipline of uniform receptiveness. *Theor. Comput. Sci. 221,* 1-2, 457–493.

Sangiorgi, D. 2006. Termination of processes. *Math. Struct. Comput. Sci. 16,* 1, 1–39.

Sangiorgi, D. and Walker, D. 2001. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

Suenaga, K. and Kobayashi, N. 2007. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Proceedings of ESOP 2007*. LNCS, vol. 4421. Springer-Verlag, 490–504.

Terauchi, T. and Aiken, A. 2008. A capability calculus for concurrency and determinism. *ACM Trans. Prog. Lang. Syst. 30,* 5.

Yoshida, N. 2002. Type-based liveness guarantee in the presence of nontermination and nondeterminism. Tech. Rep. 2002-20, MSC Technical Report, University of Leicester. April.

Yoshida, N., Berger, M., and Honda, K. 2004. Strong normalisation in the pi-calculus. *Info. Comput. 191,* 2, 145–202.

A. PROOF OF TYPE PRESERVATION (LEMMA 4.6)

We first prove properties of the predicate $nocap_m$ defined in Definition 4.3.

LEMMA A.1. *(1) $nocap_{\mathbf{0}}(\mathtt{L})$ holds for any $\mathtt{L}$.*

(2) Suppose $\mathtt{L}_1 \,|\, \mathtt{L}_2$ is well-defined. If $nocap_m(\mathtt{L}_1)$ and $nocap_m(\mathtt{L}_2)$, then $nocap_m(\mathtt{L}_1 \,|\, \mathtt{L}_2)$.

(3) If $nocap_m(\mathtt{L})$ and $\mathtt{L} \le \mathtt{L}_1 \,|\, \mathtt{L}_2$, then $nocap_m(\mathtt{L}_1)$.

(4) Suppose $\mathtt{L}_1 \,|\, \mathtt{L}_2$ is well-defined. If $noob(\mathtt{L}_1)$, then $nocap_{Modes(\mathtt{L}_1)}(\mathtt{L}_2)$ holds.

(5) If $nocap_{m_1}(\mathtt{L})$ and $nocap_{m_2}(\mathtt{L})$, then $nocap_{m_1 \sqcap m_2}(\mathtt{L})$.

PROOF. Since the other properties follow immediately from the definition, we show only the 4th property. The case where $\mathtt{L}_1 = \mathtt{Bool}$ is trivial. Suppose $\mathtt{L}_1 = \sharp_{U_1}[\widetilde{\mathtt{L}}]$. In this case, $\mathtt{L}_2 = \sharp_{U_2}[\widetilde{\mathtt{L}}]$. Let $m = Modes(\mathtt{L}_1)$. Since $noob(\mathtt{L}_1)$, we have:

$$!?_\epsilon \le m \qquad m \le ! \Rightarrow nocap(\widetilde{\mathtt{L}}) \qquad m \le ? \Rightarrow noob(\widetilde{\mathtt{L}})$$

So, we obtain $nocap_m(\mathtt{L}_2)$ as required. □

LEMMA A.2. *Suppose $nocap_\Lambda(\Delta)$ holds. If $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$ and $enabled(\Lambda, \Delta, \eta)$, then $nocap_{\Lambda'}(\Delta')$ holds.*

PROOF. The proof proceeds by case analysis on η.

—Case $\eta = \tau$:
In this case, we have either $\langle \Lambda', \Delta' \rangle = \langle \Lambda, \Delta \rangle$, or:

$$\Lambda' = \Lambda \qquad U \xrightarrow{\tau} U'$$
$$\Delta = \Delta_1, a : \sharp_U[\widetilde{\mathtt{L}}] \qquad \Delta' = \Delta_1, a : \sharp_{U'}[\widetilde{\mathtt{L}}]$$

The former case is trivial. In the latter case, by the last condition, (i) $nocap(U)$ implies $nocap(U')$ and (ii) $mode(U', \alpha)$ implies $mode(U, \alpha)$. Thus, $nocap_m(\sharp_U[\widetilde{\mathtt{L}}])$ implies $nocap_m(\sharp_{U'}[\widetilde{\mathtt{L}}])$. By the definition of $nocap_\Lambda(\Delta)$, $nocap_\Lambda(\Delta')$ follows immediately from $nocap_\Lambda(\Delta)$.

—Case $\eta = a[\widetilde{b}]$: In this case, we have:

$$\Lambda' = \Lambda \qquad U \xrightarrow{?} U'$$
$$\Delta = \Delta_1, a : \sharp_U[\widetilde{\mathtt{L}}] \qquad \Delta' = \Delta_1 \,|\, \widetilde{b} : \widetilde{\mathtt{L}}, a : \sharp_{U'}[\widetilde{\mathtt{L}}]$$

By the condition $nocap_\Lambda(\Delta)$ and $U \xrightarrow{?} U'$, we have

$$nocap_\Lambda(\Delta_1, a : \sharp_{U'}[\widetilde{\mathtt{L}}]).$$

Moreover, since $\Lambda(a) \le !$, we also have $nocap(\widetilde{\mathtt{L}})$, which implies $nocap_\Lambda(\widetilde{b} : \widetilde{\mathtt{L}})$. Therefore, by using Lemma A.1(2), we get $nocap_\Lambda(\Delta')$ as required.

—Case $\eta = (\nu \widetilde{c})\, \overline{a}[\widetilde{b}]$: In this case, we have:

$$\Lambda(a) \le ? \qquad U \xrightarrow{!} U'$$
$$\Delta = \Delta_1, a : \sharp_U[\widetilde{\mathtt{L}}] \qquad \Delta' = \Delta_1', a : \sharp_{U'}[\widetilde{\mathtt{L}}]$$
$$\Delta_1, \widetilde{c} : \widetilde{\mathtt{L}_c} \le \Delta_1' \,|\, (\widetilde{b} : \widetilde{\mathtt{L}}) \qquad \Lambda' = \Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap Modes(\widetilde{b} : \widetilde{\mathtt{L}})$$

From $\Lambda(a) \leq ?_\epsilon$ and $nocap_\Lambda(\Delta(a))$, we get $noob(\widetilde{\mathtt{L}})$. By the condition $U \xrightarrow{!} U'$ and $nocap_\Lambda(\Delta)$, we have $nocap_\Lambda(a : \sharp_{U'}[\widetilde{\mathtt{L}}])$. Since $a \notin \widetilde{b}$ (note that we do not have recursive types), $Modes(\widetilde{b} : \widetilde{\mathtt{L}})(x) = \mathbf{0}$. Therefore, we have $\Lambda(a) = \Lambda'(a)$, which implies $nocap_{\Lambda'}(a : \sharp_{U'}[\widetilde{\mathtt{L}}])$. Thus, it remains to show $nocap_{\Lambda'}(\Delta_1')$. By Lemma A.1(5), it suffices to show:

$$nocap_{\Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\}}(\Delta_1') \qquad nocap_{Modes(\widetilde{b} : \widetilde{\mathtt{L}})}(\Delta_1').$$

By using Lemma A.1(1), we get $nocap_{\Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\}}(\widetilde{c} : \widetilde{\mathtt{L}_c})$. Combining it with the fact $nocap_\Lambda(\Delta_1)$, we obtain $nocap_{\Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\}}(\Delta_1, \widetilde{c} : \widetilde{\mathtt{L}_c})$. Thus, by using Lemma A.1(3), we obtain $nocap_{\Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\}}(\Delta_1')$.

It remains only to show $nocap_{Modes(\widetilde{b} : \widetilde{\mathtt{L}})}(\Delta_1')$. For $d \notin \{\widetilde{b}\}$, we have $Modes(\widetilde{b} : \widetilde{\mathtt{L}})(d) = \mathbf{0}$, so that $nocap_{Modes(\widetilde{b} : \widetilde{\mathtt{L}})(d)}(\Delta_1'(d))$ follows from Lemma A.1(1). For b_i, $nocap_{Modes(\mathtt{L}_i)}(\Delta_1'(b_i))$ follows from $noob(\widetilde{\mathtt{L}})$ and Lemma A.1(4).

□

Definition A.3. We write $\langle \Lambda, \Delta \rangle \leq \langle \Lambda', \Delta' \rangle$ when $\Lambda' \leq \Lambda$ and $\Delta \leq \Delta'$.

LEMMA A.4. *If* $\langle \Lambda_1, \Delta_1 \rangle \leq \langle \Lambda_1', \Delta_1' \rangle \xrightarrow{\eta} \langle \Lambda_2', \Delta_2' \rangle$ *and* $enabled(\Lambda_1, \Delta_1, l)$*, then there exist* Λ_2 *and* Δ_2 *such that* $\langle \Lambda_1, \Delta_1 \rangle \xrightarrow{\eta} \langle \Lambda_2, \Delta_2 \rangle \leq \langle \Lambda_2', \Delta_2' \rangle$.

PROOF. We first note that $U_1 \leq U_1' \xrightarrow{\alpha} U_2'$ implies that there exists U_2 such that $U_1 \xrightarrow{\alpha} U_2 \leq U_2'$. Therefore, the case for $\eta = \tau$ follows immediately.

—Case $\eta = a[\widetilde{b}]$: In this case, we have:

$$\Delta_1' = \Delta_{11}', a : \sharp_{U_1'}[\widetilde{\mathtt{L}}] \qquad \Delta_2' = \Delta_{11}' \mid \widetilde{b} : \widetilde{\mathtt{L}}, a : \sharp_{U_2'}[\widetilde{\mathtt{L}}]$$
$$U_1' \xrightarrow{?} U_2' \qquad \Lambda_2' = \Lambda_1'$$

By the condition $\Delta_1 \leq \Delta_1'$, we also have:

$$\Delta_1 = \Delta_{11}, a : \sharp_{U_1}[\widetilde{\mathtt{L}}] \qquad \Delta_{11} \leq \Delta_{11}' \qquad U_1 \leq U_1'$$

By the condition $U_1 \leq U_1' \xrightarrow{?} U_2'$, there exists U_2 such that $U_1 \xrightarrow{?} U_2 \leq U_2'$. The required result holds for $\Lambda_2 = \Lambda_1$ and $\Delta_2 = \Delta_{11} \mid \widetilde{b} : \widetilde{\mathtt{L}}, a : \sharp_{U_2}[\widetilde{\mathtt{L}}]$. Note that $\Delta_{11} \mid \widetilde{b} : \widetilde{\mathtt{L}}$ is well-defined by the assumption $enabled(\Lambda_1, \Delta_1, l)$.

—Case $\eta = (\nu \widetilde{c})\, \overline{a}[\widetilde{b}]$: In this case, we have:

$$\Delta_1' = \Delta_{11}', a : \sharp_{U_1'}[\widetilde{\mathtt{L}}] \qquad \Delta_2' = \Delta_{21}', a : \sharp_{U_2'}[\widetilde{\mathtt{L}}] \qquad \Delta_{11}', \widetilde{c} : \widetilde{\mathtt{L}_c} \leq \Delta_{21}' \mid \widetilde{b} : \widetilde{\mathtt{L}}$$
$$U_1' \xrightarrow{!} U_2' \qquad \Lambda_2' = \Lambda_1'\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap Modes(\widetilde{b} : \widetilde{\mathtt{L}})$$

By the condition $\Delta_1 \leq \Delta_1'$, we also have:

$$\Delta_1 = \Delta_{11}, a : \sharp_{U_1}[\widetilde{\mathtt{L}}] \;\; \Delta_{11} \leq \Delta_{11}' \;\; U_1 \leq U_1'$$

By the condition $U_1 \leq U_1' \xrightarrow{!} U_2'$, there exists U_2 such that $U_1 \xrightarrow{!} U_2 \leq U_2'$. Let $\Delta_2 = \Delta_{21}', a : \sharp_{U_2}[\widetilde{\mathtt{L}}]$ and $\Lambda_2 = \Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} + Modes(\widetilde{b} : \widetilde{\mathtt{L}})$. Then, by using the

fact $\Delta_{11}, \widetilde{c}:\widetilde{\mathtt{L}_c} \leq \Delta'_{11}, \widetilde{c}:\widetilde{\mathtt{L}_c} \leq \Delta'_{21} \mid \widetilde{b}:\widetilde{\mathtt{L}}$, we get:

$$\langle \Lambda_1, \Delta_1 \rangle \xrightarrow{\eta} \langle \Lambda_2, \Delta_2 \rangle.$$

We also have $\Lambda'_2 \leq \Lambda_2$ and $\Delta_2 \leq \Delta'_2$ as required.

□

LEMMA A.5 SUBSTITUTION LEMMA. *Suppose that* $\Delta \mid a:\mathtt{L}$ *is well-defined. If* $\Delta, x:\mathtt{L} \vdash^{\perp}_{\mathtt{LT}} P$, *then* $\Delta \mid a:\mathtt{L} \vdash^{\perp}_{\mathtt{LT}} [x \mapsto a]P$.

PROOF. Induction on derivation of $\Delta, a:\mathtt{L} \vdash_J P$. □

LEMMA A.6. *If* $\langle \Lambda, (\Delta, d:\sharp_U[\sigma]) \rangle \xrightarrow{\eta} (\Lambda', \Delta')$ *and* $d \in \mathbf{FN}(\eta) \setminus \mathbf{SN}(\eta)$, *then there exists* Λ'' *such that* $\langle \Lambda, \Delta \rangle \xrightarrow{(\nu d)\,\eta} (\Lambda'', \Delta')$ *and* $\Lambda' \leq \Lambda''$.

PROOF. By the definition of the transition relation for type environments, we have:

$$\begin{array}{c} \eta = (\nu \widetilde{c})\,\overline{a}[\widetilde{b}] \qquad U_1 \xrightarrow{!} U_1' \qquad \Delta = \Delta_1, a:\sharp_{U_1}[\widetilde{\mathtt{L}}] \\ \Delta_1, d:\sharp_U[\sigma], \widetilde{c}:\widetilde{\mathtt{L}_c} \leq \Delta' \mid \widetilde{b}:\widetilde{\mathtt{L}} \qquad \Lambda' = \Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \end{array}$$

Let $\Lambda'' = \Lambda\{d \mapsto \mathbf{0}, \widetilde{c} \mapsto \widetilde{\mathbf{0}}\} + \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}})$. Then, we have $\langle \Lambda, \Delta \rangle \xrightarrow{(\nu d)\,\eta} \langle \Lambda'', \Delta' \rangle$ and $\Lambda' \leq \Lambda''$ as required. □

LEMMA A.7. *If* $\langle \Lambda, \Delta \rangle \xrightarrow{(\nu \widetilde{c})\,\overline{a}[\widetilde{b}]} \langle \Lambda', \Delta' \rangle$, *then there exists* Δ'' *such that* $\mathit{Modes}(\Delta) \leq \mathit{Modes}(\Delta' \setminus \{\widetilde{c}\})$ *and* $\Delta'' \leq \Delta'$ *with* $\langle \Lambda, \Delta \rangle \xrightarrow{(\nu \widetilde{c})\,\overline{a}[\widetilde{b}]} \langle \Lambda', \Delta'' \rangle$.

PROOF. $\mathit{Modes}(\Delta)(v) \leq \mathit{Modes}(\Delta')$ fails only if $\mathit{Modes}(\Delta)(v) = \alpha_\epsilon$ and $\mathit{Modes}(\Delta')(v) = \alpha_{\mathtt{o}}$. Let $\Delta''(v)$ be the type obtained from $\Delta'(v)$ by replacing all finite obligation levels with ∞ for such v, and $\Delta''(v) = \Delta'(v)$ for other v. Then, Δ'' satisfies the required conditions. □

PROOF LEMMA 4.6. Double induction on the derivation of transition $P \xrightarrow{\eta} Q$ and the derivation of $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$. (In other words, well-founded induction on the pair of the derivation trees for $P \xrightarrow{\eta} Q$ and $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$.)

Case analysis on the last rule used for deriving $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$.

—Case ELT-HYB: By the typing rule, we have:

$$\Delta \models_{\mathtt{RD}} P \quad \mathit{Erase}(\Delta) \models_{\mathtt{RTer}} P \quad \mathit{nocap}_{\Lambda}(\Delta)$$

By the definition of $\models_{\mathtt{RD}}$ and $\mathit{enabled}(\Lambda, \Delta, \eta)$, there exists Δ' such that $\Delta \xrightarrow{\eta} \Delta'$ and $\Delta' \models_{\mathtt{RD}} Q$. Moreover, there exists Λ' such that $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$. By the definition of $\models_{\mathtt{RTer}}$ and $P \xrightarrow{\eta} Q$, we have $\mathit{Erase}(\Delta') \models_{\mathtt{RTer}} Q$. By Lemma A.2, we also have $\mathit{nocap}_{\Lambda'}(\Delta')$. Thus, we get $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} Q$ by using ELT-HYB.

—Case ELT-WEAK: By the typing rule, we have:

$$\Delta_1 \vdash^{\Lambda_1}_{\mathtt{LT}} P \quad \langle \Lambda, \Delta \rangle \leq \langle \Lambda_1, \Delta_1 \rangle$$

The assumption $\mathit{enabled}(\Lambda, \Delta, l)$ and the above conditions imply $\mathit{enabled}(\Lambda_1, \Delta_1, l)$. By the induction hypothesis, there must exist Λ'_1 and

Δ_1' such that $\langle \Lambda_1, \Delta_1 \rangle \xrightarrow{\eta} \langle \Lambda_1', \Delta_1' \rangle$ and $\Delta_1' \vdash_{\mathtt{LT}}^{\Lambda_1'} Q$. By Lemma A.4, there exist Λ' and Δ' such that $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle \leq \langle \Lambda_1', \Delta_1' \rangle$. Thus, by using ELT-Weak, we get $\Delta' \vdash_{\mathtt{LT}}^{\Lambda'} Q$ and $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$ as required.

—Case ELT-Out: In this case, we have:

$$P = \overline{a}^{\chi}[\widetilde{b}].\, Q \qquad \eta = \overline{a}^{t_c}[\widetilde{b}] \qquad \Delta = a : \sharp_{!^0_{t_c}}[\widetilde{\mathtt{L}}]; (\Delta_1 \,|\, \widetilde{b} : \uparrow\widetilde{\mathtt{L}})$$
$$\Delta_1 \vdash_{\mathtt{LT}} Q \qquad \Lambda = \bot$$

Let $\Delta' = \Delta_1 \,|\, a : \sharp_{\mathbf{0}}[\widetilde{\mathtt{L}}]$ and $\Lambda' = \Lambda + \mathit{Modes}(\widetilde{b} : \widetilde{\mathtt{L}}) = \bot$. Then, we have $\langle \Lambda, \Delta \rangle \xrightarrow{\overline{a}[\widetilde{b}]} \langle \Lambda', \Delta' \rangle$ and $\Delta' \vdash_{\mathtt{LT}}^{\Lambda'} Q$ as required.

—Case ELT-In: In this case, we have:

$$P = a^{\chi}(\widetilde{y}).\, P_1 \qquad \eta = a[\widetilde{b}] \qquad Q = [\widetilde{y} \mapsto \widetilde{b}]P_1$$
$$\Lambda = \bot \qquad \Delta = a : \sharp_{?^0_{t_c}}[\widetilde{\mathtt{L}}]; \Delta_1 \qquad \Delta_1, \widetilde{y} : \widetilde{\mathtt{L}} \vdash_{\mathtt{LT}} P_1$$

By Lemma A.5, we have $\Delta_1 \,|\, (\widetilde{b} : \widetilde{\mathtt{L}}) \vdash_{\mathtt{LT}} Q$. (Note that $\Delta_1 \,|\, (\widetilde{b} : \widetilde{\mathtt{L}})$ is well-defined since $\mathit{enabled}(\Lambda, \Delta, l)$ holds.) Let Δ' be $\Delta_1 \,|\, \widetilde{b} : \widetilde{\mathtt{L}}$ if $a \in \mathit{dom}(\Delta_1)$ and $\Delta_1 \,|\, \widetilde{b} : \widetilde{\mathtt{L}} \,|\, a : \sharp_{\mathbf{0}}[\widetilde{\mathtt{L}}]$ otherwise. Let Λ' be $\bot$. Then, we have $\Delta' \vdash_{\mathtt{LT}}^{\Lambda'} Q$ and $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$ as required.

—Case ELT-Par: We have:

$$P = P_1 \,|\, P_2 \qquad \Delta = \Delta_1 \,|\, \Delta_2 \qquad \Delta_1 \vdash_{\mathtt{LT}}^{\Lambda_1} P_1 \qquad \Delta_2 \vdash_{\mathtt{LT}}^{\Lambda_2} P_2$$
$$\Lambda_2 \leq \mathit{Modes}(\Delta_1) \qquad \Lambda_1 \leq \mathit{Modes}(\Delta_2) \qquad \Lambda = \Lambda_1 \sqcup \Lambda_2$$

We perform case analysis on the rule used for deriving $P \xrightarrow{\eta} Q$.

—Case Tr-ParL: In this case, we have:

$$Q = P_1' \,|\, P_2 \qquad P_1 \xrightarrow{\eta} P_1'$$

By the induction hypothesis, we have

$$\langle \Lambda_1, \Delta_1 \rangle \xrightarrow{\eta} \langle \Lambda_1', \Delta_1' \rangle \qquad \Delta_1' \vdash_{\mathtt{LT}}^{\Lambda_1'} P_1'$$

for some Λ_1' and Δ_1'. Let Λ_2' be $\Lambda_2\{\widetilde{c} \mapsto \widetilde{!?_{\circ}}\}$ if $\eta = (\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$ and Λ_2' be Λ_2 otherwise. If $\eta = (\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$, then without loss of generality, we can assume that $\widetilde{c}$ does not appear in P_2, so that $\Delta_2 \vdash_{\mathtt{LT}}^{\Lambda_2'} P_2$ holds. Let $\Delta' = \Delta_1' \,|\, \Delta_2$ and $\Lambda' = \Lambda_1' \sqcup \Lambda_2'$. We need to show $\Delta' \vdash_{\mathtt{LT}}^{\Lambda'} P_1' \,|\, P_2$ and $\langle \Lambda, \Delta \rangle \xrightarrow{\eta} \langle \Lambda', \Delta' \rangle$.
$\Delta' \vdash_{\mathtt{LT}}^{\Lambda'} P_1' \,|\, P_2$ follows if we show $\Lambda_1' \leq \mathit{Modes}(\Delta_2)$ and $\Lambda_2' \leq \mathit{Modes}(\Delta_1')$.

—$\Lambda_1' \leq \mathit{Modes}(\Delta_2)$ follows immediately if $\eta = \tau$ or $\eta = a[\widetilde{b}]$. If $\eta = (\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$, then $\Lambda_1'(d) \leq \Lambda_1(d) \leq \mathit{Modes}(\Delta_2(d))$ for $d \in \mathit{dom}(\Lambda_1') \setminus \{\widetilde{c}\}$. For c_i, we can assume without loss of generality that $c_i \notin \mathit{dom}(\Delta_2)$, which implies $\mathit{Modes}(\Delta_2)(c_i) = \mathbf{0}$. Therefore, $\Lambda_1' \leq \mathit{Modes}(\Delta_2)$ holds.

—$\Lambda_2' = \Lambda_2 \leq \mathit{Modes}(\Delta_1) = \mathit{Modes}(\Delta_1')$ holds if $\eta = \tau$. If $\eta = (\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$, by Lemma A.7, we can also assume that $\mathit{Modes}(\Delta_1) \leq \mathit{Modes}(\Delta_1' \setminus \{\widetilde{c}\})$. So, $\Lambda_2' \leq \mathit{Modes}(\Delta_1')$ follows from

$$\Lambda_2\{\widetilde{c} : \widetilde{!?_{\circ}}\} \leq \mathit{Modes}(\Delta_1)\{\widetilde{c} : \widetilde{!?_{\circ}}\} \leq \mathit{Modes}(\Delta_1').$$

If $\eta = a[\widetilde{b}]$, then we have $\mathit{Modes}(\Delta_1) \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \leq \mathit{Modes}(\Delta_1')$. By the assumption $\mathit{enabled}(\Lambda, \Delta, l)$, we have $\Lambda_2 \leq \Lambda \leq \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}})$. From this and $\Lambda_2 \leq \mathit{Modes}(\Delta_1)$, we get $\Lambda_2 \leq \mathit{Modes}(\Delta_1) \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \leq \mathit{Modes}(\Delta_1')$.

It remains to show $\langle \Lambda, \Delta\rangle \xrightarrow{\eta} \langle \Lambda', \Delta'\rangle$. The case where $\eta = \tau$ or $\eta = a[\widetilde{b}]$ is trivial. Suppose $\eta = (\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$. By the condition $\langle \Lambda_1, \Delta_1\rangle \xrightarrow{\eta} \langle \Lambda_1', \Delta_1'\rangle$, we have:

$$\begin{array}{cccc} \Lambda_1' = \Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) & & U_1 \xrightarrow{!} U_1' & \Delta_1 = \Delta_{11}, a:\sharp_{U_1}[\widetilde{\mathtt{L}}] \\ \Delta_1' = \Delta_{11}', a:\sharp_{U_1'}[\widetilde{\mathtt{L}}] & & \Delta_{11}, \widetilde{c}:\widetilde{\mathtt{L}}_c \leq \Delta_{11}' \mid \widetilde{b}:\widetilde{\mathtt{L}} & \end{array}$$

We can assume without loss of generality that $\widetilde{c} \notin \mathit{dom}(\Delta_2)$ and $a \in \mathit{dom}(\Delta_2)$ (otherwise add $a:\sharp_{\mathbf{0}}[\widetilde{\mathtt{L}}]$ to Δ_2). So, $\Delta_2 = \Delta_{21}, a:\sharp_{U_2}[\widetilde{\mathtt{L}}]$ for some Δ_{21} and U_2. Then, we have $\Delta_{11} \mid \Delta_{21}, \widetilde{c}:\widetilde{\mathtt{L}}_c \leq (\Delta_{11}' \mid \Delta_{21}) \mid \widetilde{b}:\widetilde{\mathtt{L}}$. Since $\Lambda_2 \leq \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}})$, we also have:

$$\begin{array}{rcl} \Lambda' & = & \Lambda_1' \sqcup \Lambda_2' \\ & = & (\Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}})) \sqcup (\Lambda_2\{\widetilde{c} \mapsto \widetilde{!?_{\mathtt{o}}}\}) \\ & = & (\Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcup \Lambda_2\{\widetilde{c} \mapsto \widetilde{!?_{\mathtt{o}}}\}) \sqcap (\mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \sqcup \Lambda_2\{\widetilde{c} \mapsto \widetilde{!?_{\mathtt{o}}}\}) \\ & = & (\Lambda_1 \sqcup \Lambda_2)\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \\ & = & \Lambda\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}). \end{array}$$

Hence, we have $\langle \Lambda, \Delta\rangle \xrightarrow{\eta} \langle \Lambda', \Delta'\rangle$ as required.

—Case Tr-ParR: Similar to the case for Tr-ParL.

—Case Tr-ComL: In this case, we have:

$$\begin{array}{cc} P = P_1 \mid P_2 & Q = (\nu\widetilde{c})\,(P_1' \mid P_2') \\ P_1 \xrightarrow{(\nu\widetilde{c})\,\overline{a}[b]} P_1' & P_2 \xrightarrow{a[\widetilde{b}]} P_2' \end{array}$$

By the induction hypothesis, we have:

$$\begin{array}{cc} \langle \Lambda_1, \Delta_1\rangle \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]} \langle \Lambda_1', \Delta_1'\rangle & \Delta_1' \vdash^{\Lambda_1'}_{\mathtt{LT}} P_1' \\ \langle \Lambda_2, \Delta_2\rangle \xrightarrow{a[\widetilde{b}]} \langle \Lambda_2, \Delta_2'\rangle & \Delta_2' \vdash^{\Lambda_2}_{\mathtt{LT}} P_2' \end{array}$$

From the above conditions, we also obtain:

$$\begin{array}{ccc} \Delta_1 = \Delta_{11}, a:\sharp_{U_1}[\widetilde{\mathtt{L}}] & \Delta_1' = \Delta_{11}', a:\sharp_{U_1'}[\widetilde{\mathtt{L}}] & U_1 \xrightarrow{!} U_1' \\ \Delta_2 = \Delta_{21}, a:\sharp_{U_2}[\widetilde{\mathtt{L}}] & \Delta_2' = \Delta_{21} \mid \widetilde{b}:\widetilde{\mathtt{L}}, a:\sharp_{U_2'}[\widetilde{\mathtt{L}}] & U_2 \xrightarrow{?} U_2' \\ \Lambda_1' = \Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) & \Delta_{11}, \widetilde{c}:\widetilde{\mathtt{L}}_c \leq \Delta_{11}' \mid \widetilde{b}:\widetilde{\mathtt{L}} & \end{array}$$

Let $\Lambda_2' = \Lambda_2\{\widetilde{c} \mapsto \widetilde{!?_{\mathtt{o}}}\}$. Then, we can assume that $\widetilde{c}$ do not appear in P_2, so that $\Delta_2 \vdash^{\Lambda_2'}_{\mathtt{LT}} P_2$ and $\Delta_2' \vdash^{\Lambda_2'}_{\mathtt{LT}} P_2'$ hold. Let $\Delta'' = \Delta_{11}' \mid \Delta_{21} \mid \widetilde{b}:\widetilde{\mathtt{L}}, a:\sharp_{U_1' \mid U_2'}[\widetilde{\mathtt{L}}]$ and $\Lambda'' = \Lambda_1' \sqcup \Lambda_2'$. We first show $\Delta'' \vdash^{\Lambda''}_{\mathtt{LT}} P_1' \mid P_2'$, which will follow if we show $\Lambda_1' \leq \mathit{Modes}(\Delta_2')$ and $\Lambda_2' \leq \mathit{Modes}(\Delta_1')$. Without loss of generality, we can assume $\widetilde{c} \notin \mathit{dom}(\Delta_2)$. Therefore, by the conditions $\Lambda_1' = \Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}})$ and $\Lambda_1 \leq \mathit{Modes}(\Delta_2)$, we have

$$\Lambda_1' \leq \mathit{Modes}(\Delta_2) + \mathit{Modes}(\widetilde{b}:\widetilde{\mathtt{L}}) \leq \mathit{Modes}(\Delta_2').$$

By Lemma A.7, we can also assume $\mathit{Modes}(\Delta_1) \leq \mathit{Modes}(\Delta' \setminus \{\widetilde{c}\}$, so that we have:

$$\Lambda'_2 \;\leq\; \mathit{Modes}(\Delta_1)\{\widetilde{c} \mapsto \widetilde{!?_\circ}\} \leq \mathit{Modes}(\Delta'_1).$$

So, by using ELT-Par, we obtain $\Delta'' \vdash^{\Lambda''}_{\mathtt{LT}} P'_1 \,|\, P'_2$. By applying ELT-Weak, we obtain $\Delta_{11} \,|\, \Delta_{21}, a : \sharp_{U'_1 \,|\, U'_2}[\widetilde{\mathtt{L}}], \widetilde{c} : \widetilde{\mathtt{L}_c} \vdash^{\Lambda''}_{\mathtt{LT}} P'_1 \,|\, P'_2$. Let $\Delta' = \Delta_{11} \,|\, \Delta_{21}, a : \sharp_{U'_1 \,|\, U'_2}[\widetilde{\mathtt{L}}]$ and $\Lambda' = \Lambda''\{\widetilde{c} \mapsto \widetilde{!?_\circ}\}$. Then, by using ELT-New, we get $\Delta' \vdash^{\Lambda''\{\widetilde{c} \mapsto \widetilde{!?_\circ}\}}_{\mathtt{LT}} P'_1 \,|\, P'_2$. We get $\Delta' \vdash^{\Lambda}_{\mathtt{LT}} P'_1 \,|\, P'_2$ by using ELT-Weak, because for $d \notin \{\widetilde{c}\}$, we have:

$$\begin{array}{rl} \Lambda''(d) & \leq\; (\Lambda'_1 \sqcup \Lambda'_2)(d) \\ & \leq\; ((\Lambda_1\{\widetilde{c} \mapsto \widetilde{\mathbf{0}}\} \sqcap \mathit{Modes}(\widetilde{b} : \widetilde{\mathtt{L}})) \sqcup \Lambda_2)(d) \\ & \leq\; (\Lambda_1 \sqcup \Lambda_2)(d) \\ & \leq\; \Lambda(d). \end{array}$$

It remains to check $\langle \Lambda, \Delta \rangle \xrightarrow{\tau} \langle \Lambda, \Delta' \rangle$, which follows immediately from $U_1 \,|\, U_2 \xrightarrow{\tau} U'_1 \,|\, U'_2$.

—Case ELT-New: We have:

$$\begin{array}{ll} P = (\nu a)\, P_1 & \Delta, a : \sharp_U[\widetilde{\mathtt{L}}] \vdash^{\Lambda_1}_{\mathtt{LT}} P_1 \\ \mathit{rel}(U) & \Lambda_1\{a \mapsto !?_\circ\} = \Lambda \end{array}$$

We perform case analysis on the rule used for deriving $P \xrightarrow{\eta} Q$.

—Case Tr-Open: In this case, $\eta = (\nu a)\, \eta'$ and $P_1 \xrightarrow{\eta'} Q$. By the induction hypothesis, we have

$$\langle \Lambda_1, (\Delta, a : \sharp_U[\widetilde{\mathtt{L}}]) \rangle \xrightarrow{\eta} (\Lambda'_1, \Delta') \qquad \Delta' \vdash^{\Lambda'_1}_{\mathtt{LT}} Q$$

By Lemma A.6, there exists Λ' such that $(\Lambda, \Delta) \xrightarrow{(\nu a)\, \eta'} (\Lambda', \Delta')$ and $\Lambda'_1 \leq \Lambda'$. By using ELT-Weak, we obtain $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} Q$ as required.

—Case Tr-New: In this case, we have $Q = (\nu a)\, Q_1$ and $P_1 \xrightarrow{\eta} Q_1$ with $a \notin \mathbf{FN}(l) \cup \mathbf{BN}(l)$. By the induction hypothesis, we have:

$$\langle \Lambda_1, (\Delta, a : \sharp_U[\widetilde{\mathtt{L}}]) \rangle \xrightarrow{\eta} \langle \Lambda'_1, (\Delta', a : \sharp_{U'}[\widetilde{\mathtt{L}}]) \rangle \qquad \Delta', a : \sharp_{U'}[\widetilde{\mathtt{L}}] \vdash^{\Lambda'_1}_{\mathtt{LT}} Q_1$$

By the condition $a \notin \mathbf{FN}(l) \cup \mathbf{BN}(l)$, we have:

$$\langle \Lambda_1\{a \mapsto !?_\circ\}, \Delta \rangle \xrightarrow{\eta} \langle \Lambda'_1\{a \mapsto !?_\circ\}, \Delta' \rangle \qquad U \leq U'$$

From the last condition and $\mathit{rel}(U)$, we obtain $\mathit{rel}(U')$. So, by using ELT-New, we get: $\Delta' \vdash^{\Lambda'_1\{a \mapsto !?_\circ\}}_{\mathtt{LT}} Q$. The required result holds for $\Lambda' = \Lambda'_1\{a \mapsto !?_\circ\}$.

—Case ELT-Rep: In this case, $P \xrightarrow{\eta} Q$ must have been derived by using Tr-Rep or Tr-RIn. We show only the former case; the latter case is similar. We have:

$$P = *P_1 \qquad *P_1 \,|\, P_1 \xrightarrow{\eta} Q \qquad \Delta_1 \vdash^{\bot}_{\mathtt{LT}} P_1 \qquad \Delta = *\Delta_1 \qquad \Lambda = \bot$$

By using ELT-Rep and ELT-Par, we obtain $*\Delta_1 \,|\, \Delta_1 \vdash^{\bot}_{\mathtt{LT}} *P_1 \,|\, P_1$. Since $\Delta = *\Delta_1 \leq *\Delta_1 \,|\, \Delta_1$ holds, we get $\Delta \vdash^{\bot}_{\mathtt{LT}} *P_1 \,|\, P_1$. By the induction hypothesis, there exist Δ' and Λ' such that $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} Q$ and $\langle \Delta, \Lambda \rangle \xrightarrow{\eta} \langle \Delta', \Lambda' \rangle$.

—Cases ELT-If: Similar to the case for ELT-Rep.

□

We introduce a relation $\preceq$ on processes below. $\preceq$ is the least reflexive and transitive relation closed under the rule $E[(\nu a)\,P] \preceq (\nu a)\,E[P]$. Here, E ranges over the set of *evaluation contexts*, defined by:

$$E ::= [\,] \mid (E \mid P) \mid (P \mid E) \mid (\nu a)\,E$$

(Note that E does not contain $\langle [\,] \rangle^T$; so we disallow $\langle (\nu a)\,P \rangle^T \preceq (\nu a)\,\langle P \rangle^T$.)

Typing is also preserved by $\preceq$.

Lemma A.8. *If $\Delta \vdash^{\Lambda}_{\mathrm{LT}} P$ and $P \preceq P'$, then $\Delta \vdash^{\Lambda}_{\mathrm{LT}} P'$.*

Proof. This follows by straightforward induction on the derivation of $P \preceq Q$.
□

B. PROOF OF PROGRESS (LEMMA 4.7)

We extend the syntax of processes by adding explicitly typed processes $\langle P \rangle_{\Delta,\Lambda}$:

$$P ::= \cdots \mid \langle P \rangle_{\Delta,\Lambda}$$

The typing rule for $\langle P \rangle_{\Delta,\Lambda}$ is:

$$\frac{\Delta \vdash^{\Lambda}_{\mathrm{LT}} P}{\Delta \vdash^{\Lambda}_{\mathrm{LT}} \langle P \rangle_{\Delta,\Lambda}} \qquad \text{(LT-TProc)}$$

Lemma B.1. *If $nocap_{\Lambda}(\Delta)$, $rel(\Delta')$, and $\Delta' \vdash^{\Lambda'}_{L} E[\langle P \rangle_{\Delta,\Lambda}]$, then $rel(\Delta)$.*

Proof. We first note that if $\Delta' \vdash^{\Lambda'}_{L} E[\langle P \rangle_{\Delta,\Lambda}]$ then, $\Lambda(a) \leq \Lambda'(a)$ for any $a \in dom(\Delta) \cap dom(\Delta')$. To show the lemma, it suffices to show the following, stronger property.

> If (i) $nocap_{\Lambda}(\Delta)$, (ii) $rel(\Delta'(a))$ for every $a \in \{a \in dom(\Delta) \cap dom(\Delta') \mid \neg nocap(\Delta(a))\}$, and (iii) $\Delta' \vdash^{\Lambda'}_{\mathrm{LT}} E[\langle P \rangle_{\Delta,\Lambda}]$, then $rel(\Delta)$.

We show it by induction on derivation of $\Delta' \vdash^{\Lambda'}_{\mathrm{LT}} E[\langle P \rangle_{\Delta,\Lambda}]$, with case analysis on the last rule used. Since the other cases are trivial, we show only the case where the last rule is ELT-Par and $E = E_1 \mid Q$. In this case, we have:

$$\frac{\Delta'_1 \vdash^{\Lambda'_1}_{\mathrm{LT}} E_1[\langle P \rangle_{\Delta,\Lambda}] \qquad \Delta'_2 \vdash^{\Lambda'_2}_{\mathrm{LT}} Q \qquad \Lambda'_1 \leq Modes(\Delta'_2) \qquad \Lambda'_2 \leq Modes(\Delta'_1) \qquad \Lambda' = \Lambda'_1 \sqcup \Lambda'_2}$$

By the induction hypothesis, it suffices to show that $rel(\Delta'_1(a))$ holds for every $a \in \{a \in dom(\Delta) \cap dom(\Delta'_1) \mid \neg nocap(\Delta(a))\}$, Suppose $a \in \{a \in dom(\Delta) \cap dom(\Delta'_1) \mid \neg nocap(\Delta(a))\}$. Then, by the assumption $nocap_{\Lambda}(\Delta)$, it must be the case that $!?_{\epsilon} \leq \Lambda(a) \leq \Lambda'_1(a)$. By the condition $\Lambda'_1 \leq Modes(\Delta'_2)$, it must be the case that $noob(\Delta'_2(a))$. Thus, $rel(\Delta'_2(a))$ follows from the condition $rel(\Delta'(a))$. (Here, we have used the fact that if $rel(U_1 \mid U_2)$ and $noob(U_2)$, then $rel(U_1)$.) □

We write $\#(P)$ for the size of process P (i.e., the number of process constructors in P). The progress property (Lemma 4.7) follows as a corollary of the following lemma.

LEMMA B.2. *Suppose (i)* $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} E[\langle P\rangle_{\Delta,\Lambda}]$*, (ii)* $rel(\Delta')$*, and (iii)* $a \notin \mathbf{BN}(E[P])$*. Then,* $ob_!(\Delta(a)) = n(\neq \infty)$ *implies* $E[P] \xrightarrow{\tau}{}^{*(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}\!\!\longrightarrow$ *for some* $\widetilde{c}$ *and* $\widetilde{b}$*, and* $ob_?(\Delta(a)) = n$ *implies* $E[P] \xrightarrow{\tau}{}^{*}\xrightarrow{a[\widetilde{b}]}$ *for some* $\widetilde{b}$*.*

PROOF. The proof proceeds by well-founded induction on $(n, \#(P))$, where the well-founded order is defined by $(n,m) < (n',m') \iff (n < n') \vee (n = n' \wedge m < m')$. We perform case analysis on the structure of P. We show only the case for $ob_!(\Delta(a)) = n$; the other case is similar. Without loss of generality, we can assume that the last rule used for deriving $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$ is not ELT-WEAK, since if the last rule is ELT-WEAK, we can find Δ_1 and Λ'' such that $\Delta_1 \vdash^{\Lambda''}_{\mathtt{LT}} P$, $\Delta' \vdash^{\Lambda'}_{\mathtt{LT}} E[\langle P\rangle_{\Delta_1,\Lambda''}]$, and $ob_!(\Delta(a)) \le n$ holds. (Hence, more formally, the whole proof is by induction on $(n, \#(P), m)$, where m is the number of the last applications of ELT-WEAK for deriving $\Delta \vdash^{\Lambda}_{\mathtt{LT}} P$.) Note that the proof below is a little informal (e.g., in the treatment of contexts) and sketchy; Except for the case where $P = \langle P_1\rangle^T$, the proof is almost the same as the corresponding theorem for the previous type system [Kobayashi 2005a].

—Case $P = \langle P_1\rangle^T$: In this case, $\Delta \models_{\mathtt{RD}} P_1$, $\Delta \models_{\mathtt{RTer}} P_1$, and $nocap_\Lambda(\Delta)$. By Lemma B.1, we have $rel(\Delta)$. Hence, from Lemma 4.6 with $\Delta \models_{\mathtt{RD}} P_1$ and the conditions $\Delta \models_{\mathtt{RTer}} P$, we obtain $P_1 \xrightarrow{\tau}{}^{*(\nu\widetilde{c'})\,\overline{a}[\widetilde{b}]}\!\!\longrightarrow$. Thus, we have $E[P] \xrightarrow{\tau}{}^{*(\nu\widetilde{c})\,\overline{a}[b]}\!\!\longrightarrow$ as required.

—Case $P = \mathbf{0}$: This case cannot happen.

—Case $P = \overline{a_1}^{\chi}[\widetilde{d}].P_1$: If $a_1 = a$, then the result follows immediately. Suppose $a_1 \neq a$. By the typing rules, we have:

$$\Delta = a_1 : \sharp^{!0}_{t}[\widetilde{\mathtt{L}}];(\Delta_1 \mid \widetilde{d} : \uparrow\widetilde{\mathtt{L}}) \qquad \Delta_1 \vdash^{\perp}_{\mathtt{LT}} P_1 \qquad t < n$$

By the induction hypothesis (note that we can assume without loss of generality that a_1 is not bound in $E[P]$ since otherwise we can move the binder (νa_1) to the outermost place by using Lemma A.8 and remove it), we have $E[P] \xrightarrow{\tau}{}^{*} E_1[P] \xrightarrow{a_1[\widetilde{b}]}$, where P is not involved in the transitions. $E_1[P]$ must be of the form $E_2[P, a_1(\widetilde{y}).Q_1]$. Let $Q = E_3[P_1, [\widetilde{y} \mapsto \widetilde{b}]Q_1]$. (Here, we have extended evaluation contexts to those with multiple holes.) By Lemma 4.6 and the typing rules, we have:

$$\Delta'' \vdash^{\Lambda''}_{\mathtt{LT}} E_3[\langle P_1\rangle_{\Delta_1,\Lambda_1}, \langle [\widetilde{y} \mapsto \widetilde{b}]Q_1\rangle_{\Delta_2,\Lambda_2}] \qquad \langle \Delta', \Lambda'\rangle \xrightarrow{\tau}{}^{*} \langle \Delta'', \Lambda''\rangle$$

Moreover, $ob_!(\Delta_1(a)) \le n$ or $ob_!(\Delta_2(a)) \le n-1$ holds. In both cases, the result follows immediately from the induction hypothesis (note that $\#(P_1) < \#(P)$ in the former case).

—Case $P = a_1{}^{\chi}(\widetilde{y}).P_1$: Similar to the above case.

—Case $P = *P_1$: By the condition $\Delta \vdash^{\Lambda}_{J} P$, there must exist Δ_1 such that $\Delta_1 \vdash^{\perp}_{J} P_1$ and $\Delta \le *\Delta_1$. The latter condition implies $ob_!(\Delta_1(a)) \le n$. By $\Delta' \vdash^{\Lambda'}_{J} E[P \mid \langle P_1\rangle_{\Delta_1,\perp}]$ and the induction hypothesis, we get $E[P \mid P_1] \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}$. The required result $E[P] \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}$ is obtained by using TR-REP.

—Case P is $P_1 \mid P_2$, $(\nu c)\, P_1$, or **if** a **then** P_1 **else** P_2: Trivial by the induction hypothesis.

□

Proof Lemma 4.7. Suppose that Q is tagged and $\emptyset \vdash^{\Lambda}_{\mathtt{LT}} Q$. If the tagged process is inside $\langle\cdot\rangle^T$, i.e., if Q is of the form $E[\langle Q'\rangle^T]$, where Q is tagged, then $\Delta \models_{\mathtt{RD}} Q'$ and $\mathit{Erase}(\Delta) \models_{\mathtt{RTer}} Q'$ with $\mathit{nocap}_{\Lambda}\Delta$ for some Δ and Λ. The latter condition implies that $\mathit{rel}(\Delta)$. Thus, $Q' \stackrel{\tau}{\longrightarrow}^{*} \stackrel{\tau^{\square}}{\longrightarrow}$.

If the tagged process is not inside $\langle\cdot\rangle^T$, then Q must be of the form $E_1[(\nu a)\, E_2[a^{\square}(\widetilde{x}).\, Q']]$ or $E_1[(\nu a)\, E_2[\overline{a}^{\square}[\widetilde{v}].\, Q']]$. We show only the former case below, as the latter case is similar. By Lemma A.8 and the typing rules, we have:

$$a : \sharp_U[\widetilde{\mathrm{L}}] \vdash^{\Lambda}_{\mathtt{LT}} E_1[E_2[a^{\square}(\widetilde{x}).\, Q']] \qquad \mathit{rel}(U)$$

By the typing rules, it must be the case that $\mathit{cap}_?(U) \neq \infty$. By $\mathit{rel}(U)$, we get $\mathit{ob}_!(U) \neq \infty$. By Lemma B.2, we have $E_1[E_2[a^{\square}(\widetilde{x}).\, Q']] \stackrel{\tau}{\longrightarrow}^{*} \stackrel{\overline{a}[\widetilde{v}]}{\longrightarrow}$. Thus, we have $E_1[E_2[a^{\square}(\widetilde{x}).\, Q']] \stackrel{\tau}{\longrightarrow}^{*} \stackrel{\tau^{\square}}{\longrightarrow}$, which implies $P \stackrel{\tau}{\longrightarrow}^{*} \stackrel{\tau^{\square}}{\longrightarrow}$. □

C. PROOF OF THEOREM 4.2

Theorem 4.2 follows as a corollary of the following lemma, which is similar to Lemma B.2.

Lemma C.1. *Suppose (i)* $\Delta' \vdash^{\Lambda'}_{\mathtt{SLT}} E[\langle P\rangle_{\Delta,\Lambda}]$*, (ii)* $\mathit{rel}(\Delta')$*, and (iii)* $a \notin \mathbf{BN}(E[P])$*. If* $\mathit{ob}_!(\Delta(a)) = t \neq \infty$*, then in any full, strongly fair reduction sequence of* $E[P]$*, there is a process* Q *that satisfies* $Q \stackrel{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}{\longrightarrow}$ *for some* $\widetilde{c}$ *and* $\widetilde{b}$*. Similarly, if* $\mathit{ob}_?(\Delta(a)) = t \neq \infty$*, then in any full, strongly fair reduction sequence of* $E[P]$*, there is a process* Q *that satisfies* $Q \stackrel{a[\widetilde{b}]}{\longrightarrow}$ *for some* $\widetilde{b}$*.*

Proof. The proof proceeds in the same manner as that of Lemma B.2, by well-founded induction on $(t, \#(P))$, where the well-founded order is defined by $(n, m) < (n', m') \Longleftrightarrow (n < n') \vee (n = n' \wedge m < m')$. Since the other cases are similar to the proof of Lemma B.2, we show only the case for $P = \langle P_0\rangle^T$. In this case, by Lemma 4.6 with the conditions $\Delta \models_{\mathtt{RD}} P$ and $\mathit{Erase}(\Delta) \models_{\mathtt{RTer}} P$, there exists a reduction sequence $P_0 \stackrel{\tau,S}{\longrightarrow} P_1 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_n \stackrel{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}{\longrightarrow}$. Consider any full, strongly fair reduction sequence from $E[\langle P_0\rangle^T]$, and let $P_0 \stackrel{\eta_1,S'_1}{\longrightarrow} Q_1 \stackrel{\eta_2,S'_2}{\longrightarrow} Q_2 \stackrel{\eta_3,S'_3}{\longrightarrow} \cdots$ be the corresponding, local transition sequence of P_0. We shall show that there exists m such that $Q_m \stackrel{(\nu\widetilde{c}')\,\overline{a}[\widetilde{b}']}{\longrightarrow}$, by induction on n. The case where $n = 0$ is trivial. Suppose $n > 0$. Since P_0 is robustly confluent, the transition $\stackrel{\tau,S}{\longrightarrow}$ is continuously enabled until it occurs. Therefore, there must exist m such that $\stackrel{\eta_m,S'_m}{\longrightarrow} = \stackrel{\tau,S}{\longrightarrow}$. Moreover, there exists a transition sequence $P_1 \stackrel{\eta_1,S'_1}{\longrightarrow} R_1 \stackrel{\eta_2,S'_2}{\longrightarrow} \cdots \stackrel{\eta_{m-1},S'_{m-1}}{\longrightarrow} R_{m-1} \equiv Q_m$. Thus, there is a full, strongly fair reduction sequence

$$E[P_1] \stackrel{\tau}{\longrightarrow} E_1[R_1] \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} E_{m-1}[R_{m-1}] \stackrel{\tau}{\longrightarrow} E_m[R_m] \stackrel{\tau}{\longrightarrow} \cdots,$$

where $R_{m+k} \equiv Q_{m+k+1}$ for $k \geq 0$. By the induction hypothesis, there exists j such that $R_j \stackrel{(\nu\widetilde{c}')\,\overline{a}[\widetilde{b}']}{\longrightarrow}$. If $j \geq m$, then $Q_{j+1} \stackrel{(\nu\widetilde{c}')\,\overline{a}[\widetilde{b}']}{\longrightarrow}$ as required. If $j < m$ and

R_m cannot make an output transition on a, then there must exist i ($j < i \leq m$) such that S'_i contains the label of an output prefix on a. Thus, $Q_{i-1} \xrightarrow{(\nu \widetilde{c'})\,\overline{a}[\widetilde{b'}]}$ as required. □

Proof Theorem 4.2. Suppose that $\emptyset \vdash_{\mathtt{SLT}} P$ and $P \xrightarrow{\tau}{}^{*} Q$. It suffices to show (i) if $Q = E_1[(\nu a)\, E_2[a^{\circ}(\widetilde{x}).\, Q_1]]$, then $E_1[E_2[a^{\circ}(\widetilde{x}).\, Q_1]]$ is reduced to a process of the form $E[\overline{a}[\widetilde{v}].\, Q_2]$ in any full, strongly fair reduction sequence, and (ii) if $Q = E_1[(\nu a)\, E_2[\overline{a}^{\circ}[\widetilde{v}].\, Q_1]]$, then $E_1[E_2[\overline{a}^{\circ}[\widetilde{v}].\, Q_1]]$ is reduced to a process of the form $E[a(\widetilde{y}).\, Q_2]$ in any full, strongly fair reduction sequence. (Note that if the above conditions hold, any marked action will be enabled infinitely often.) We show only (i); the proof of (ii) is similar. Suppose $Q = E_1[(\nu a)\, E_2[a^{\circ}(\widetilde{x}).\, Q_1]]$. Then by Lemma 4.6, we have $\emptyset \vdash^{\perp}_{\mathtt{SLT}} Q$. By the typing rules, it must be the case that $a : \sharp_U[\widetilde{\mathtt{L}}] \vdash^{\Lambda}_{\mathtt{SLT}} E_1[E_2[a^{\circ}(\widetilde{x}).\, Q_1]]$ and $rel(U)$, which also implies $ob_!(U) \neq \infty$. Thus, by using Lemma C.1, $E_1[E_2[a^{\circ}(\widetilde{x}).\, Q_1]]$ must be reduced to a process of the form $E[\overline{a}[\widetilde{v}].\, Q_2]$ in any full, strongly fair reduction sequence. □

D. INTERVALS

We sketch here an extension of the type systems in [Deng and Sangiorgi 2006] that improves the expressiveness of their termination analysis (and hence also of the robust-termination analysis). We mainly explain the extension on the first of the type systems in [Deng and Sangiorgi 2006], namely the system of pure levels `Lev`; we are very brief on the others, as the modifications needed are similar.

The extension is obtained by replacing the levels of [Deng and Sangiorgi 2006] with *intervals*. An interval is written $[n, m]$, for $n \leq m$, and indicates a non-empty set of consecutive natural numbers. A type assignment $x : \sharp^{[n,m]}[V]$ intuitively means that x can be instantiated with any channel whose level is between n and m. Although in practice we may gain precision by maintaining levels for the types of the channels, for convenience of presentation we treat levels themselves as intervals; thus level n corresponds to the interval $[n, n]$.

We recall that the channel types are types that can be assigned to the channels, and the values types are the types that can be assigned to the values communicated along the channels. In an input $v(\widetilde{x})$ or an output $\overline{v}[\widetilde{w}]$ we call v the *subject* of the prefix.

Notations. We use μ to range over intervals. For intervals $\mu_1 = [n, m]$ and $\mu_2 = [r, s]$ we write $\mu_1 \subseteq \mu_2$ if $r \leq n$ and $m \leq s$; and $\mu_1 < \mu_2$ if $m \leq r$. If $\Theta(p) = \sharp^{\mu}[\widetilde{V}]$ then we call μ the *interval of p in Θ* (or simply the *interval of p*, if Θ is clear from the context).

The first type system. In the `Lev` type system each channel type is assigned a level. We replace the levels with the intervals. Thus the grammar of the types, called the *interval types*, is:

$$V \ ::= \ \mathtt{Bool} \ \Big| \ \sharp^{\mu}[\widetilde{V}] \ \text{types}$$

where μ is an interval. Judgments are of the form $\Theta \vdash^{\mu} P$. It is intended that $\Theta \vdash^{\mu} P$ should imply that for every active output $\overline{v}[\widetilde{w}]$ in P, the interval of v must be smaller than μ.

We write $V_1 \leq V_2$ if $V_1 = V_2$, or $V = \sharp^{\mu_1}[\widetilde{W}]$ and $V = \sharp^{\mu_2}[\widetilde{W}]$ with $\mu_1 \subseteq \mu_2$. We write $\Theta \vdash v : V$ if $\Theta(v) \leq V$. With these notations for the intervals and for the subtyping on the intervals, the rules can remain, notationally, the same as in `Lev` (of course, with intervals in place of levels). We report below the interesting rules, namely those for output, input, and replicated input:

$$\frac{\Theta(p) = \sharp^{\mu_2}[\widetilde{V}] \quad \Theta \vdash \widetilde{v} : \widetilde{V} \qquad \Theta \vdash^{\mu_1}_{\mathtt{Ter}} P \qquad \mu_2 < \mu_1}{\Theta \vdash^{\mu_1}_{\mathtt{Ter}} \overline{p}[\widetilde{v}].P} \tag{IT-Out}$$

$$\frac{\Theta(p) = \sharp^{\mu_2}[\widetilde{V}] \qquad \Theta, \widetilde{x} : \widetilde{V} \vdash^{\mu_1}_{\mathtt{Ter}} P}{\Theta \vdash^{\mu_1}_{\mathtt{Ter}} p(\widetilde{x}).P} \tag{IT-In}$$

$$\frac{\Theta(p) = \sharp^{\mu_2}[\widetilde{V}] \qquad \Theta, \widetilde{x} : \widetilde{V} \vdash^{\mu_2}_{\mathtt{Ter}} P}{\Theta \vdash^{\mu_1}_{\mathtt{Ter}} *p(\widetilde{x}).P} \tag{IT-RIn}$$

The resulting type system is strictly more expressive than the level system `Lev`. Any process typable in `Lev` is typable in our type system, by replacing each level n with interval $[n, n]$. On the other hand, the use of intervals in place of levels allows us to have some (limited) form of polymorphism with respect to the levels, so that a term such as

$$a(x).\mathbf{0} \mid \overline{a}[b] \mid \overline{a}[c] \mid *b.\overline{c}$$

is typable in our type system but not in [Deng and Sangiorgi 2006] (for typing the replication, b should have a level higher than c, which is impossible as both can instantiate x; with intervals it suffices to require that the interval for x contains those for b and c).

Further, we can take advantage of intervals in the conditions for robust termination. For instance, in (2) of Section 5.3, the type equality $S = T$ can be replaced by the subtyping requirement $S \leq T$. Other similar weakenings are possible in Lemma 5.3.

The following lemma is important. It shows that we can safely replace a variable with a channel whose interval is contained in that of the variable.

Lemma D.1. *If $\Theta, v : V' \vdash^{\mu} P$ and $V \leq V'$, then $\Theta, v : V \vdash^{\mu} P$.*

Proof. Induction on derivation of $\Theta, v : V' \vdash^{\mu}_{\mathtt{Ter}} P$. □

With the use of the lemma above, the proof of termination for the well-typed closed processes of the new system can be given along the lines of the corresponding theorem in system `Lev`.

The second type system. The system `Lev` allows nesting of inputs but forbids all forms of recursive inputs, that is, replications $*a(x).P$ with the body P having active outputs at a. The other type systems of [Deng and Sangiorgi 2006] allow us to relax this restriction. In the second type system, for instance, the body P can have active outputs $\overline{a}[v]$, but v must be provably smaller than x with respect to some pre-defined well founded ordering on values; thus the value received at the replicated input $a(x)$ is greater than the value emitted in any active output at a

that is underneath the replication. For instance, if the communicated values are integers, then this holds for $*a(x).\overline{a}[x-1]$. A mechanism is assumed for evaluating (possibly open) natural number expressions, which allows us to derive assertions such as $x-1 < x$, or $x-29+4*7 < x$. This evaluation mechanism is an orthogonal issue, independent from the type system.

In the corresponding type system with intervals, judgments are of the form $\Theta \vdash^{(\mu,\widetilde{x})} P$. It is intended that $\Theta \vdash^{(\mu,\widetilde{x})} P$ should imply that for an active output $\overline{v}[\widetilde{w}]$ in P, either (a) the interval of v is smaller than μ, or (b) the interval of v in Θ, say λ, is consecutive to μ (that is, if $\lambda = [n,m]$ and $\mu = [r,s]$ then $m = r$), but each component w_i of the tuple carried by v is provably smaller than the corresponding component x_i of $\widetilde{x}$. With this in mind, the rules are similar to those for the first type system previously discussed.

The third type system. The third type system of [Deng and Sangiorgi 2006] exploits some of the structure of the processes. Precisely, it takes into account sequences of inputs underneath a replication. In this way, intuitively, one can consider the *sum* of the levels of such inputs (rather than the level of a single input as in previous type systems), and then compare this against the active outputs in the body of the sequence. Call κ such a sequence of inputs, and P the body (i.e., the process underneath κ). We have to compare the weight of κ, written $wt(\kappa)$, against the weight of P, written $wt(P)$. In Deng and Sangiorgi [2006], where types have just levels, $wt(P)$ is the vector $\langle n_k, n_{k-1}, \cdots, n_1\rangle$, where each n_h represents the number of occurrences of outputs that are not underneath a replication and whose subject is a name of level h; then k is the highest level on which the process has non-zero output occurrences[5]. This definition of weight is extended to input patterns by taking into account the levels of all input subjects; i.e., if κ is $p_1(\widetilde{x_1}).\cdots.p_n(\widetilde{x_n})$, then $wt(\kappa)$ is the vectorial sum of all levels of the names p_h.

In our case, since we have intervals in place of pure levels, we have to be conservative. Thus $wt(\kappa)$ is the lowest possible sum given by the intervals (that is, we use the same vectorial sum as before but each interval $[n,m]$ of an input subject of κ contributes its infimum n), whereas $wt(P)$ is the highest possible sum given by the intervals (that is, each interval $[n,m]$ of the subject of an active output in P contributes its supremum m). Using ω to range over vectors, judgments are of the form $\Theta \vdash^{\omega} P$; it is intended that $\Theta \vdash^{\omega} P$ holds if $wt(P)$ is not greater than ω.

The fourth type system. The fourth type system of [Deng and Sangiorgi 2006] is the system `PO` discussed in Section 5. The use of partial orders on names is an orthogonal issue with respect to the choice of having type systems based on levels or on intervals, therefore we do not discuss it any further here.

[5]This definition makes sense in Deng and Sangiorgi [2006] where the type systems are formulated *à la Church*—each name is assigned a type a priori; in a formulation *à la Curry* the definition should be given with respect to a given typing derivation for P.