A Hybrid Type System for Lock-Freedom of Mobile Processes

Naoki Kobayashi[1] and Davide Sangiorgi[2]

[1] Tohoku University
[2] Università di Bologna

Abstract. 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.

1 Introduction

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 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 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). 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 [1, 11, 12, 21, 22]. An automatic verification tool, TyPiCal [10], has been implemented based on Kobayashi's system [12]. 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.

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 analysis: *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 behaviours (even very simple ones), which are more easily dealt with by other methods such as model checking.

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 [13]), 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:

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

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 [12], 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 [12].

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 [13], 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. 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:

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

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 [4] and bisimulation, (and also in accordance with other definitions of similar forms of liveness for π-calculus such as [21]).

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. 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 implented 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).

2 Target Language

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:

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

Here, χ is either $\circ$ or $\bullet$, and $\widetilde{w}$ abbreviates a possibly empty sequence $w_1, \ldots, w_n$. The constructs are the standard ones of the polyadic π-calculus: nil, output and input prefixes, parallel composition, replication ($*P$ behaves like infinitely many copies of P running in parallel), restriction, and a conditional. The only difference is the annotation χ in prefixes, which 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 succeed then a set of prefixes that will eventually succeed is marked, see Section 5.) We call a prefix *marked* if its annotation is $\circ$. We usually omit the $\bullet$ annotation, thus for example $a(x).P$ stands for $a^{\bullet}(x).\, P$. As usual, restriction and input prefix are binders. A *closed* process has no free variables. 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.

Typing The type systems that we will propose are defined on top of the simply-typed π-calculus ($\mathtt{ST}$). 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 standard typing rules are omitted.

Operational Semantics We use 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, an output action $(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]$, or an input action $a[\widetilde{b}]$. See Appendix A for the definition of the transition relation. We write $\xrightarrow{\tau}{}^{*}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$; we write $P \xrightarrow{\tau}$ and $P \xrightarrow{\tau}{}^{*}$ if there is 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. $\Gamma \vdash_{\mathtt{ST}} P \xrightarrow{\eta} \Gamma' \vdash_{\mathtt{ST}} P'$ holds if: (1) $\Gamma \vdash_{\mathtt{ST}} P$; and (2) 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 $\Gamma' = \Gamma, \widetilde{b} : \widetilde{\mathtt{S}}$. 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 $\Gamma_i \vdash_{\mathtt{ST}} P_i \xrightarrow{\eta_{i+1}} \Gamma_{i+1} \vdash_{\mathtt{ST}} P_{i+1}$.

Deadlock-Freedom and Lock-Freedom A prefix is *at top level* if it is not underneath another input/output prefix or underneath a replication.

Definition 1 (deadlock-freedom) *P is* deadlock-free *if, whenever $P \xrightarrow{\tau}{}^{*} Q$ and Q has at least one marked prefix at top level, then $Q \xrightarrow{\tau}$.*

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

Definition 2 ((weak) lock-freedom) A tagged process P is *successful* if whenever $P \xrightarrow{\tau}^* Q$ then $Q \xrightarrow{\tau}^* \xrightarrow{\tau^\Box}$. 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.

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 [3, 11] for a formal definition of strong fairness in the π-calculus). See Appendix A.2 for a note on the difference between weak and strong lock-freedom.

Definition 3 (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 $\xrightarrow{\tau^\Box}$.*

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 [12] with hybrid rules appealing to deadlock/termination/confluence analyses.

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 [12] 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 in Appendix B.)

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 is guaranteed to succeed (in other words, its co-action will be provided by the environment) if it becomes ready for execution (i.e., it is at top-level). 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. 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.

To understand how this kind of judgment 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:

$$a\!:\!\sharp_{?^0_0}[\mathtt{Bool}], b\!:\!\sharp_{!^1_\infty}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x]$$
$$a\!:\!\sharp_{!^1_\infty}[\mathtt{Bool}], b\!:\!\sharp_{?^0_0}[\mathtt{Bool}] \vdash_{\mathtt{LT}} b^\circ(x).\,\overline{a}[x]$$

For the entire process, we can simply combine both type environments by combining usages pointwise:

$$a\!:\!\sharp_{?^0_0\,\mid\,!^1_\infty}[\mathtt{Bool}], b\!:\!\sharp_{!^1_\infty\,\mid\,?^0_0}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x] \mid b^\circ(x).\,\overline{a}[x]$$

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:

$$a\!:\!\sharp_{?^0_0\,\mid\,!^0_0}[\mathtt{Bool}], b\!:\!\sharp_{!^1_1\,\mid\,?^1_1}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,\overline{b}[x] \mid \overline{a}[\mathtt{true}].\,b^\circ(x)$$

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 1 in Appendix B summarizes the syntax of usages and types, and typing rules of Kobayashi's lock-freedom type system [12].

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.

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 transition that conforms to the base type system ST.

Definition 4 (robust termination) *A process P is* terminating *if there is no infinite internal transition sequence $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \cdots$. An (open) process P is* robustly terminating *under Γ, written $\Gamma \models_{\mathtt{RTer}} P$, if $\Gamma \vdash_{\mathit{ST}} P$, and for every closing substitution σ for Γ and for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\sigma\Gamma \vdash_{\mathit{ST}} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is terminating.*

We say that Δ is closed if $\mathit{dom}(\Delta) \cap \mathcal{V} = \emptyset$. We write $\mathit{rel}(\Delta)$ intuitively to mean that each capability in Δ is guaranteed by a corresponding obligation; and $\mathit{ob}_!(\mathtt{L})$ for the level of the obligation to send a message: again, precise definitions are in Appendix B.

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 $\Delta \xrightarrow{\eta} \Delta'$ (see Appendix B) expresses the increase/decrease of capabilities/obligations in Δ by the transition η. For example, $a:\sharp_{?^0_\infty}[\sharp_{!^1_\infty}[\mathtt{Bool}]] \xrightarrow{a[b]} a:\sharp_{\mathbf{0}}[\sharp_{!^1_\infty}[\mathtt{Bool}]], b:\sharp_{!^1_\infty}[\mathtt{Bool}]$ holds (where the usage $\mathbf{0}$ indicates that the channel cannot be used at all). Thus, $a:\sharp_{?^0_\infty}[\sharp_{!^1_\infty}[\mathtt{Bool}]] \models_{\mathtt{RD}} P$ 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.

Definition 5 (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 $\mathit{rel}(\Delta)$, then: (i) P is deadlock-free; (ii) If $\mathit{ob}_!(\Delta(a)) \neq \infty$, then either $P \xrightarrow{(\nu\widetilde{c})\,\overline{a}[\widetilde{b}]}$ or $P \xrightarrow{\tau}$; and (iii) If $\mathit{ob}_?(\Delta(a)) \neq \infty$ 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.*

Partial confluence means that any τ-transition commutes with any other transitions. To define the partial confluence, we assume that each prefix is uniquely labeled (as in [3]), and extend the transition relation to $\xrightarrow{\eta,S}$ where S is the set of the labels of the prefixes involved in the transition: see [15]. Robust confluence indicates partial confluence after any sequence of transition that conforms to the base type system ST.

Definition 6 (robust confluence) *A process P is* partially confluent, *if whenever $P_1 \xleftarrow{\tau,S_1} P \xrightarrow{\eta,S_2} P_2$, either $\eta = \tau \wedge S_1 = S_2$, or $P_1 \xrightarrow{\eta,S_2}\equiv\xleftarrow{\tau,S_1} P_2$. A process*

P is robustly confluent *under Γ, written $\Gamma \models_{\mathtt{RConf}} P$, if $\Gamma \vdash_{ST} P$ and for any closing substitution σ that respects Γ and for any Q, k, and $\eta_1, \cdots \eta_k$ such that $\sigma\Gamma \vdash_{ST} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} Q$, the derivative Q is partially confluent.*

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.

We have proved that robust deadlock-freedom is guaranteed by Kobayashi's type system for deadlock-freedom [13]. 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 $a : \sharp_{!^0_\infty}[\mathtt{Bool}]$, 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 [14] and race-freedom [20]; other static analysis methods such as model checking could also be used. Verification of robust termination is discussed in Section 4.

3.3 Hybrid Typing Rules

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

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad Er(\Delta) \models_{\mathtt{RTer}} P \qquad nocap(\Delta)}{\Delta \vdash_{\mathtt{LT}} P} \quad \text{(LT-Hyb)}$$

$$\frac{\Delta \models_{\mathtt{RD}} P \qquad Er(\Delta) \models_{\mathtt{RTer}} P \qquad Er(\Delta) \models_{\mathtt{RConf}} P \qquad nocap(\Delta)}{\Delta \vdash_{\mathtt{SLT}} P} \quad \text{(SLT-Hyb)}$$

Here, $Er(\Delta)$ is the simple type environment obtained from Δ by removing all usage annotations. The condition $nocap(\Delta)$ 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 precise definition of $nocap(\Delta)$, given in Appendix B, is subtle; for nested channel types, the nocap condition depends on whether a channel is used for input or output. For example, $nocap(\sharp_{?^0_\infty}[\sharp_{!^0_\infty}[\,]])$ holds but $nocap(\sharp_{!^0_\infty}[\sharp_{!^0_\infty}[\,]])$ does not. 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 4 in Appendix A.2 for a 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 Appendix B and LT-Hyb, and write $\Delta \vdash_{\mathtt{SLT}} P$ if it is derivable by using SLT-Hyb instead of LT-Hyb. The theorem below states the soundness of the type systems. Its proof is non-trivial because of the presence of the hybrid rules; for instance, conditions such as $nocap(\Delta)$ are not preserved by transitions, so in the proof we had to refine and extend the type systems. See the extended version [15].

Theorem 1 (lock-freedom). *If $\emptyset \vdash_{\mathtt{LT}} P$, then P is (weakly) lock-free. If $\emptyset \vdash_{\mathtt{SLT}} P$, then P is strongly lock-free.*

Example 1. Consider the following processes.

$$\begin{aligned} \mathit{Clients} &\stackrel{\text{def}}{=} *(\nu r_1)\,(\overline{\mathit{fact}}^{\circ}[\mathtt{rnd}(), r_1] \,|\, r_1{}^{\circ}(x).\,\mathbf{0}) \\ \mathit{Server} &\stackrel{\text{def}}{=} (\nu \mathit{fact\_it})\,(*\mathit{fact}\,(n,r).\,\overline{\mathit{fact\_it}}[n,1,r] \\ &\qquad\quad |\, *\mathit{fact\_it}\,(n,x,r).\,\mathbf{if}\ n=0\ \mathbf{then}\ \overline{r}[x]\ \mathbf{else}\ \overline{\mathit{fact\_it}}[n-1, x\times n, r]) \end{aligned}$$

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. *Client* consists of infinitely many copies of the process that creates a fresh channel r_1 for receiving a reply, sends a request $[\mathtt{rnd}(), r_1]$ (where $\mathtt{rnd}()$ creates a random number) and then waits for the result on r_1.

Let Δ be $\mathit{fact} : \sharp_{*?^0_\infty}[\mathtt{Nat}, \sharp_{!^1_\infty}[\mathtt{Nat}]]$. Then, we have $\Delta \models_{\mathtt{RD}} \mathit{Server}$, $\mathit{Er}(\Delta) \models_{\mathtt{RTer}} \mathit{Server}$, and $\mathit{Er}(\Delta) \models_{\mathtt{RConf}} \mathit{Server}$ with $\mathit{nocap}(\Delta)$. Thus, by using SLT-Hyb, we obtain $\Delta \vdash_{\mathtt{SLT}} \mathit{Server}$. From this judgment and $\mathit{fact} : \sharp_{*!^\infty_0}[\mathtt{Nat}, \sharp_{!^1_\infty}[\mathtt{Nat}]] \vdash_{\mathtt{SLT}} \mathit{Clients}$, we obtain: $\emptyset \vdash_{\mathtt{SLT}} (\nu \mathit{fact})\,(\mathit{Server} \,|\, \mathit{Clients})$. 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.   See Appendix B.3 for futher examples.

Remark 1. The present side condition $\mathit{nocap}(\Delta)$ for LT-Hyb is sometimes too restrictive for local reasoning. For example, consider $C \,|\, S_1 \,|\, S_2$, where C sends a request to S_1, which consults S_2 to answer the request. Then, we have to apply LT-Hyb to $S_1 \,|\, S_2$ rather than S_1 alone, since S_1 makes an assumption that S_2 will answer. See [15] for an extension to relax the *nocap* condition.

4 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* guarantees that termination is maintained when the process interacts with its environment. Termination is strictly weaker than robust termination. Consider for instance the term $P \stackrel{\text{def}}{=} \overline{c}[b] \,|\, c(x).(\overline{x} \,|\, *a.\overline{x})$. The process P has one reduction only, and therefore it is terminating. It is indeed typable in the simplest of the type systems in [9]. 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 $\overline{c}[b] \,|\, \overline{a} \,|\, *a.\overline{a}$.

We define some abstract conditions with which a type system for termination also guarantees robust termination; we then discuss refinements of the conditions. We denote by $\mathtt{Ter}$ a generic type system for termination, and with $\Theta \vdash_{\mathtt{Ter}} P$ a judgment in $\mathtt{Ter}$. We recall that $\mathtt{ST}$ indicates the types and the type systems of the simply-typed π-calculus (Section 2).

Definition 7 *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 $f(\Theta) \vdash_{ST} P$ hold and, for all closing $f(\Theta)$-substitutions σ, whenever $\sigma f(\Theta) \vdash_{ST} \sigma P \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_k} P'$, there is 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 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 $f(\Theta)$.*

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 subject-reduction for Ter and injectivity of f on the set of channel types used in Θ. See Appendix C.

5 Implementation

We have implemented the new weak lock-freedom analysis as a feature of TyPiCal Version 1.6.0 [10]. TyPiCal takes as an input a program written in the π-calculus, and marks all input/output prefixes that are guaranteed to succeed.

The original type system for lock-freedom (reviewed in Section 3.1) had been implemented already [12, 13]. 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 [9], so that the resulting systems can guarantee robust termination, and also so to make them more suited for automatic verification (e.g., using heuristic and incomplete algorithms when the original ones were NP-complete). We also integrated them with a termination analysis based on size-change graphs [2]. See the extended version for details.

We have applied the implementation to non-trivial programs (including the examples in Section 3 and Appendix B.3), and verified them fully automatically (without any type annotations). See [15] for the benchmark results.

6 Related Work

Several type systems for lock-freedom (sometimes referred to by different names) have been already proposed [1, 11, 12, 18, 21, 22]. 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 4), 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).

Another important point is that none of the previous type systems for lock-freedom, except Kobayashi's one [12], 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. 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 obtained by replacing $\overline{fact\_it}[n-1, x \times n, r]$ in Example 1 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 [7]. 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.

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.

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 [16] 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 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. [6], 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 make the different analyses closely intertwined and make it possible local applications of the parameterized analyses. Caires [5] 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.

7 Conclusion

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 allows 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.

Acknowledgment

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

References

1. L. Acciai and M. Boreale. Responsiveness in process calculi. In *Proc. of 11th Annual Asian Computing Science Conference (ASIAN 2006)*, LNCS, 2006.
2. A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Prog. Lang. Syst.*, 29(1 (Article 5)), 2007.
3. P. Bidinger and A. B. Compagnoni. Pict correctness revisited. In *Proceeds of FMOODS 2007*, volume 4468 of *LNCS*, pages 206–220. Springer-Verlag, 2007.
4. E. Brinksma, A. Rensink, and W. Volger. Fair testing. In *Proceedings of CONCUR 1995*, volume 962 of *LNCS*, pages 313–327. Springer-Verlag, 1995.
5. L. Caires. Logical semantics of types for concurrency. In *Proceedings of CALCO 2007*, volume 4624 of *LNCS*, pages 16–35. Springer-Verlag, 2007.
6. S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. of POPL*, pages 45–57, 2002.
7. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *Proc. of POPL*, pages 265–276, 2007.
8. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *Proc. of PLDI*, 2007.
9. Y. Deng and D. Sangiorgi. Ensuring termination by typability. *Info. Comput.*, 204(7):1045–1082, 2006.
10. N. Kobayashi. TyPiCal: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.
11. N. Kobayashi. A type system for lock-free processes. *Info. Comput.*, 177:122–159, 2002.
12. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

13. N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer-Verlag, 2006.
14. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Prog. Lang. Syst.*, 21(5):914–947, 1999.
15. N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. An extended version. `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hybrid.pdf`, 2008.
16. X. Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.
17. D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theor. Comput. Sci.*, 167(2):235–274, 1996.
18. D. Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.
19. D. Sangiorgi. Termination of processes. *Math. Struct. Comput. Sci.*, 16(1):1–39, 2006.
20. T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. In *Proceedings of CONCUR 2006*, volume 4137 of *LNCS*, pages 218–232. Springer-Verlag, 2006.
21. N. Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. Technical Report 2002-20, MSC Technical Report, University of Leicester, April 2002.
22. N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi-calculus. *Info. Comput.*, 191(2):145–202, 2004.

Appendix

A Appendix for Section 2

A.1 Transition Rules

$$\overline{a}^{\chi}[\widetilde{b}].\,P \xrightarrow{\overline{a}[\widetilde{b}]} P \qquad a^{\chi}(\widetilde{y}).\,P \xrightarrow{a[\widetilde{b}]} [\widetilde{y}\mapsto\widetilde{b}]P \qquad *a^{\chi}(\widetilde{y}).\,P \xrightarrow{a[\widetilde{b}]} *a^{\chi}(\widetilde{y}).\,P \mid [\widetilde{y}\mapsto\widetilde{b}]P$$

$$\textbf{if } \mathtt{true} \textbf{ then } P \textbf{ else } Q \xrightarrow{\tau} P \qquad \textbf{if } \mathtt{false} \textbf{ then } P \textbf{ else } Q \xrightarrow{\tau} Q$$

$$\frac{P \xrightarrow{\eta} Q \qquad \mathbf{BN}(\eta)\cap\mathbf{FN}(R)=\emptyset}{P \mid R \xrightarrow{\eta} Q \mid R} \qquad \frac{P \xrightarrow{(\nu\widetilde{c})\,\overline{d}[\widetilde{b}]} Q \qquad a\in\{\widetilde{b}\}\setminus\{d,\widetilde{c}\}}{(\nu a)\,P \xrightarrow{(\nu a,\widetilde{c})\,\overline{d}[\widetilde{b}]} Q}$$

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

$$\frac{P \xrightarrow{\eta} Q \qquad a\notin\mathbf{FN}(\eta)\cup\mathbf{BN}(\eta)}{(\nu a)\,P \xrightarrow{\eta} (\nu a)\,Q} \qquad \frac{*P \mid P \xrightarrow{\eta} Q \qquad P \text{ is not an input process}}{*P \xrightarrow{\eta} Q}$$

A.2 On the Difference between Weak and Strong Lock-Freedom

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 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 (inspired by [8]) shows the difference between weak lock-freedom and strong lock-freedom. Consider the following process P:

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

There are two servers, which are listening on f and g respectively. The server 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 reaches 0.

Another example of the difference between weak and strong lock-freedom is the process in Example 4 of Appendix B.3. In fact, using our type systems, we can prove weak lock-freedom of the process, but not its strong lock-freedom.

B Appendix for Section 3

B.1 Appendix for Section 3.1

Figure 1 summarizes (a slightly simplified version of) Kobayashi's type system for lock-freedom. See the extended version for futher explanation [15]. A tutorial paper on type systems for the π-calculus is found at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

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. The usage $U_1 \,|\, U_2$ 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.

The definition of the *subusage relation* $U_1 \leq U_2$ (which is used in the definition of subtyping) is omitted. It means that U_1 represents a more liberal usage of channels, so that a channel of usage U_1 may be used also according to U_2. For example, $!^{t_1}_{t_2} \leq !^{t'_1}_{t'_2}$ holds if $t'_1 \leq t_1$ and $t_2 \leq t'_2$. In the figure, the subusage relation is used to define the relations $\mathbf{Top}(\mathtt{L})$ and $\mathtt{L}_1 \leq \mathtt{L2}$. The former means that a name of type $\mathtt{L}$ need not be used, and the latter means that a name of type $\mathtt{L}_1$ may be used as that of type $\mathtt{L}_2$. We also omit the definition of *rel*(U), which is used in the rule T-New. It means that U is consistent in the sense that for each capability in U, there is a corresponding obligation. We write *rel*$(\mathtt{L})$ if $\mathtt{L}$ is a channel type $\sharp_U[\widetilde{\mathtt{L}}]$ and *rel*(U). We write *rel*(Δ) if *rel*$(\Delta(v))$ for every $v \in dom(\Delta)$. Please refer to [15] for the definitions.

Here are some intuitions on typing rules.

In the rule LT-In, the type environment $v : \sharp_{?^0_t}[\widetilde{\mathtt{L}}];\Delta$ 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, by using LT-In, we can obtain $a : \sharp_{?^0_2.!^1_\infty}[\mathtt{Bool}], b : \sharp_{!^3_\infty}[\mathtt{Bool}] \vdash_{\mathtt{LT}} a^\circ(x).\,P$. 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: $v : \sharp_{!^0_t}[\widetilde{\mathtt{L}}];(\Delta_1 \,|\, \widetilde{w} : \uparrow\widetilde{\mathtt{L}})$ captures the condition that v is first used for output. The part $\widetilde{w} : \uparrow\widetilde{\mathtt{L}}$ 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

Syntax:

$$U \text{ (usages)} ::= \mathbf{0} \mid \alpha^{t_1}_{t_2}.U \mid (U_1 \,|\, U_2) \mid *U \qquad \alpha \text{ (actions)} ::= ? \mid ! \qquad t \text{ (levels)} \in \mathbf{Nat} \cup \{\infty\}$$

$$\mathtt{L} \text{ (usage types)} ::= \mathtt{Bool} \mid \sharp_U[\widetilde{\mathtt{L}}] \qquad \Delta \text{ (type environments)} ::= v_1 : \mathtt{L}_1, \ldots, v_n : \mathtt{L}_n$$

Operations:

$$\uparrow^t \mathbf{0} = \mathbf{0} \qquad \uparrow^t \alpha^{t_1}_{t_2}.U = \alpha^{\mathbf{max}(t,t_1)}_{t_2}.U \qquad \uparrow^t (U_1 \,|\, U_2) = \uparrow^t U_1 \,|\, \uparrow^t U_2 \qquad \uparrow^t (*U) = *\uparrow^t U$$

$$\uparrow \mathbf{0} = \mathbf{0} \qquad \uparrow \alpha^{t_1}_{t_2}.U = \alpha^{t_1+1}_{t_2}.U \qquad \uparrow (U_1 \,|\, U_2) = \uparrow U_1 \,|\, \uparrow U_2 \qquad \uparrow (*U) = *\uparrow U$$

$$\uparrow \mathtt{Bool} = \mathtt{Bool} \qquad \uparrow^t \mathtt{Bool} = \mathtt{Bool} \qquad *\mathtt{Bool} = \mathtt{Bool} \qquad \mathtt{Bool} \,|\, \mathtt{Bool} = \mathtt{Bool}$$

$$\uparrow(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{\uparrow U}[\widetilde{\mathtt{L}}] \qquad \uparrow^t(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{\uparrow^t U}[\widetilde{\mathtt{L}}] \qquad *(\sharp_U[\widetilde{\mathtt{L}}]) = \sharp_{*U}[\widetilde{\mathtt{L}}] \qquad \sharp_{U_1}[\widetilde{\mathtt{L}}] \,|\, \sharp_{U_2}[\widetilde{\mathtt{L}}] = \sharp_{U_1 \,|\, U_2}[\widetilde{\mathtt{L}}]$$

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

$$(\Delta_1 \,|\, \Delta_2)(v) = \Delta_1(v) \,|\, \Delta_2(v) \qquad (\Delta_1 \,|\, \Delta_2 \text{ is defined only if } \mathit{dom}(\Delta_1) = \mathit{dom}(\Delta_2))$$

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

Subtyping:

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

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

Typing:

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

$$\frac{}{\emptyset \vdash_{\mathtt{LT}} \mathbf{0}} \text{ (LT-Zero)} \qquad \frac{\Delta_1 \vdash_{\mathtt{LT}} P_1 \qquad \Delta_2 \vdash_{\mathtt{LT}} P_2}{\Delta_1 \,|\, \Delta_2 \vdash_{\mathtt{LT}} P_1 \,|\, P_2} \text{ (LT-Par)} \qquad \frac{\Delta' \vdash_{\mathtt{LT}} P \qquad \Delta \le \Delta'}{\Delta \vdash_{\mathtt{LT}} P} \text{ (LT-Weak)} \qquad \frac{\Delta \vdash_{\mathtt{LT}} P}{*\Delta \vdash_{\mathtt{LT}} *P} \text{ (LT-Rep)}$$

$$\frac{\Delta, a : \sharp_U[\widetilde{\mathtt{L}}] \vdash_{\mathtt{LT}} P \qquad \mathit{rel}(U)}{\Delta \vdash_{\mathtt{LT}} (\nu a)\, P} \text{ (LT-New)} \qquad \frac{\Delta \vdash_{\mathtt{LT}} P \qquad \Delta \vdash_{\mathtt{LT}} Q}{\Delta \,|\, (v : \mathtt{Bool}) \vdash_{\mathtt{LT}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \text{ (LT-If)}$$

Fig. 1. Kobayashi's type system for lock-freedom [12]

example, let Δ be:

$$a : \sharp_{?^{\infty}_{0} \mid !^{\infty}_{0}}[\sharp_{!^{2}_{\infty}}[\,]], b : \sharp_{?^{\infty}_{0} \mid !^{\infty}_{0}}[\sharp_{!^{1}_{\infty}}[\,]].$$

Then we can derive $\Delta \vdash_{\mathtt{LT}} a(x).\,\overline{b}[x]$, but neither $\Delta \vdash_{\mathtt{LT}} a(x).\,\overline{a}[x]$ nor $\Delta \vdash_{\mathtt{LT}} b(x).\,\overline{a}[x]$. This condition prevents a process from infinitely delegating obligations.

In the rule LT-New, the condition $rel(U)$ 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 T-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 $a : \sharp_{!^{1}_{0}}[\mathtt{Bool}] \vdash_{\mathtt{LT}} P$.

B.2 Appendix for Section 3.2

The realtion $\Delta \xrightarrow{\eta} \Delta'$ used in the definition of robust deadlock-freedom is defined by:

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

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

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

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

Here, the transition relation $U \xrightarrow{l_u} U'$ is defined as follows.

Definition 8 *The transition relation $U \xrightarrow{l_u} U'$ (where $l_u \in \{!, ?, \tau\}$) is the least relation closed under the following rules:*

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

$$\frac{U_1 \xrightarrow{l_u} U_1'}{U_1 \,|\, U_2 \xrightarrow{l_u} U_1' \,|\, U_2} \qquad \frac{U_2 \xrightarrow{l_u} U_2'}{U_1 \,|\, U_2 \xrightarrow{l_u} U_1 \,|\, U_2'}$$

$$\frac{U_1 \xrightarrow{!} U_1' \qquad U_2 \xrightarrow{?} U_2'}{U_1 \,|\, U_2 \xrightarrow{\tau} U_1' \,|\, U_2'} \qquad \frac{U_1 \xrightarrow{?} U_1' \qquad U_2 \xrightarrow{!} U_2'}{U_1 \,|\, U_2 \xrightarrow{\tau} U_1' \,|\, U_2'}$$

The condition $nocap(\Delta)$ used in Section 3.3 is defined as follows.

Definition 9 (nocap) *We write $nocap(U)$ when all the capability levels in U are ∞, and write $noob(U)$ when all the obligation levels in U are ∞. The relations are extended to those on types by the following rules.*

$$\frac{}{nocap(\mathtt{Bool})} \qquad \frac{nocap(U) \qquad mode(U, ?) \Rightarrow nocap(\widetilde{\mathtt{L}}) \qquad mode(U, !) \Rightarrow noob(\widetilde{\mathtt{L}})}{nocap(\sharp_U[\widetilde{\mathtt{L}}])}$$

$$\frac{}{noob(\mathtt{Bool})} \qquad \frac{noob(U) \qquad mode(U, ?) \Rightarrow noob(\widetilde{\mathtt{L}}) \qquad mode(U, !) \Rightarrow nocap(\widetilde{\mathtt{L}})}{noob(\sharp_U[\widetilde{\mathtt{L}}])}$$

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

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

Example 2. $nocap(\sharp_{?^0_\infty}[\sharp_{!^0_\infty}[\,]])$ and $nocap(\sharp_{!^0_\infty}[\sharp_{!^\infty_0}[\,]])$ hold. $nocap(\sharp_{!^0_\infty}[\sharp_{!^0_\infty}[\,]])$ does not hold.

B.3 Further Examples

This section gives more complex examples.

Example 3. Consider the following process $\mathtt{BSystem}$.

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

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 receive messages, in the server `rec` and in the clients, are expected to succeed. The success of the receive operation 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, some of the channels in its list).

Let $\mathit{\Delta} = \mathtt{bcast}\,{:}\,\sharp_{*?^0_\infty}[\sharp_{!^1_\infty \mid !^1_\infty}[\,]\ \mathtt{List}]$. Then, we have:

$$\mathit{\Delta} \models_{\mathtt{RD}} \mathtt{BServer} \qquad \mathit{Er}(\mathit{\Delta}) \models_{\mathtt{RTer}} \mathtt{BServer} \qquad \mathit{Er}(\mathit{\Delta}) \models_{\mathtt{RConf}} \mathtt{BServer} \qquad \mathit{nocap}(\mathit{\Delta})$$

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

Example 4. This example shows a binary 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 2, 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. `TInit` 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. There is parallelism in the system: many requests can be rippling down the tree at the same time; in doing so, requests can even overtake each other.

Let $\mathit{\Delta}$ be $\mathtt{insert}\,{:}\,\sharp_{*?^0_\infty}[\mathtt{Nat}, \sharp_{!^1_\infty}[\,]], \mathtt{search}\,{:}\,\sharp_{*?^0_\infty}[\mathtt{Nat}, \sharp_{!^1_\infty}[\mathtt{Bool}]]$. Then, we have:

$$\mathit{\Delta} \models_{\mathtt{RD}} \mathtt{TInit} \quad \mathit{Er}(\mathit{\Delta}) \models_{\mathtt{RTer}} \mathtt{TInit} \quad \mathit{nocap}(\mathit{\Delta})$$

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

Note that SLT-Hyb is not applicable since `TInit` 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.

Example 5. Figure 3 shows a strongly lock-free implementation of binary search trees. The server `TInit'` receives requests along channel a one by one. A request

$$
\begin{array}{l}
G \stackrel{\text{def}}{=} *\mathtt{newtree}(i,s).(\nu\mathtt{state})\Big(\overline{\mathtt{state}}[\mathbf{leaf}] \\
\quad \mid *i(n,r).\mathtt{state}(x). \qquad \texttt{/*** insertion ***/} \\
\qquad \mathbf{match}\ x\ \mathbf{with} \\
\qquad\quad \mathbf{leaf}\ \rightarrow \\
\qquad\qquad (\nu\mathtt{left\_i},\mathtt{left\_s},\mathtt{right\_i},\mathtt{right\_s}) \\
\qquad\qquad \Big(\overline{\mathtt{newtree}}[\mathtt{left\_i},\mathtt{left\_s}] \mid \overline{\mathtt{newtree}}[\mathtt{right\_i},\mathtt{right\_s}] \\
\qquad\qquad \mid \overline{\mathtt{state}}[\mathbf{node}(n,\mathtt{left\_i},\mathtt{left\_s},\mathtt{right\_i},\mathtt{right\_s})] \mid \overline{r}\Big) \\
\qquad \mid\mid \mathbf{node}(n_1,i_l,s_l,i_r,s_r)\ \rightarrow \\
\qquad\qquad \Big(\mathbf{if}\ n=n_1\ \mathbf{then}\ \overline{r}[]\ \mathbf{else\ if}\ n<n_1\ \mathbf{then}\ \overline{i_l}[n,r]\ \mathbf{else}\ \overline{i_r}[n,r] \\
\qquad\qquad \mid \overline{\mathtt{state}}[x]\Big) \\
\quad \mid *s(n,r).\mathtt{state}(x).\Big(\overline{\mathtt{state}}[x] \qquad \texttt{/*** search ***/} \\
\qquad \mid \mathbf{match}\ x\ \mathbf{with}\ \mathbf{leaf}\ \rightarrow\ \overline{r}[\mathtt{true}] \\
\qquad\qquad \mid\mid\ \mathbf{node}(n_1,i_l,s_l,i_r,s_r) \rightarrow \\
\qquad\qquad\quad \mathbf{if}\ n_1=n\ \mathbf{then}\ \overline{r}[\mathtt{false}]\ \mathbf{else\ if}\ n<n_1\ \mathbf{then}\ \overline{s_l}[n,r]\ \mathbf{else}\ \overline{s_r}[n,r]\Big)\Big) \\
\mathtt{TInit} \stackrel{\text{def}}{=} (\nu\mathtt{newtree})\,(G \mid \overline{\mathtt{newtree}}[\mathtt{insert},\mathtt{search}]) \\
\mathtt{Sys} \stackrel{\text{def}}{=} (\nu\mathtt{insert},\mathtt{search}) \\
\qquad (\mathtt{TInit} \mid *(\nu r_1)\,(\overline{\mathtt{insert}}^{\circ}[\mathtt{rnd}(),r_1] \mid r_1{}^{\circ}) \mid *(\nu r_2)\,(\overline{\mathtt{search}}^{\circ}[\mathtt{rnd}(),r_2] \mid r_2{}^{\circ}(x)))
\end{array}
$$

Fig. 2. A binary tree

is either of the form $\mathbf{insert}(n,r)$ or $\mathbf{search}(n,r)$. Unlike the system in Example 4, requests cannot be overtaken, although there is still parallelism (multiple requests can go down the tree simultaneously). $\mathtt{TInit}'$ is robustly confluent; note that the only τ-transitions inside $\mathtt{TInit}'$ are on channels `leaf`, `node`, `left`, and `right`, and that the first two of them are replicated input channels, and the others are linearized channels. Thus, we can derive

$$a : \sharp_{*?^1_\infty}[\mathtt{L}] \vdash_{\mathtt{SLT}} \mathtt{TInit}'$$

where

$$\mathtt{L} \stackrel{\text{def}}{=} \langle \mathbf{insert} : [\mathtt{Nat}, \sharp_{!^2_\infty}[\,]], \mathbf{search} : [\mathtt{Nat}, \sharp_{!^2_\infty}[\mathtt{Nat}]]\rangle.$$

Here, $\mathtt{L}$ 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:

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

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

C Appendix for Section 4

In this section, we show how f-admissibility follows from a subject reduction property for `Ter` and injectivity of f. We write $\mathtt{CTypes}(\Theta)$ for the set of channel types used in Θ.

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

$$
\begin{array}{l}
\mathtt{TInit}' \stackrel{\text{def}}{=} (\nu\mathtt{leaf},\mathtt{node})\,(G' \mid \overline{\mathtt{leaf}}^{\circ}[a])\\
\mathtt{Sys}' \stackrel{\text{def}}{=} (\nu a)\,(\mathtt{TInit}' \mid *(\nu r_1)\,(\overline{a}^{\circ}[\textbf{insert}(\mathtt{rnd}(),r_1)] \mid {r_1}^{\circ}) \mid *(\nu r_2)\,(\overline{a}^{\circ}[\textbf{search}(\mathtt{rnd}(),r_2)] \mid {r_2}^{\circ}(x)))
\end{array}
$$

Fig. 3. A strongly lock-free implementation of binary trees

Lemma 1. *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 $\eta = a[\widetilde{v}]$, 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; the remaining requirements, such as $\mathtt{CTypes}(\Theta') \subseteq \mathtt{CTypes}(\Theta)$, essentially ensure 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.

Lemma 1 is applicable to the system for termination in [19], and to all but one of the four type systems in [9] (the function f of Lemma 1 can be taken to be the function that strips off termination information).

In the full paper [15] we discuss some further improvements to the lemma, weakening the main constraints in it: first of all the injectivity of f, and also the substitution condition (3). This is done by appealing to methods for controlling the aliasing set of a variable (the set of names with which the variable could be instantiated) Note that there are dialects of the π-calculus, such as πI, where aliasing is forbidden altogether since only fresh names can be transmitted; in these languages both injectivity and condition (3) can be dropped. Thus the lemma becomes applicable to all type systems in [9], and to Yoshida, Berger, and Honda's system for termination [22].