Deadlock Analysis of Unbounded Process Networks

Elena Giachino[1], Naoki Kobayashi[2], and Cosimo Laneve[1]

[1] Dept. of Computer Science and Egineering, University of Bologna – INRIA FOCUS, Italy
[2] Dept. of Computer Science, University of Tokyo, Japan

Abstract. Deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes is extremely complex and solutions either give imprecise answers or do not scale. To enable the analysis of such programs, (1) we define an algorithm for detecting deadlocks of a basic model featuring recursion and fresh name generation: the *lam programs*, and (2) we design a type system for value passing CCS that returns lam programs. As a byproduct of these two techniques, we have an algorithm that is more powerful than previous ones and that can be easily integrated in the current release of TyPiCal, a type-based analyser for pi-calculus.

1 Introduction

Deadlock-freedom of concurrent programs has been largely investigated in the literature [2,4,1,11,18,19]. The proposed algorithms automatically detect deadlocks by building graphs of dependencies (a, b) between resources, meaning that the release of a resource referenced by a depends on the release of the resource referenced by b. The absence of cycles in the graphs entails deadlock freedom. When programs have infinite states, in order to ensure termination, current algorithms use finite approximate models that are excerpted from the dependency graphs. The cases that are particularly critical are those of programs that create networks with an arbitrary number of nodes.

To illustrate the issue, consider the following pi-calculus-like process that computes the factorial:

```
Fact(n,r,s) =  if n=0 then r?m. s!m
               else  new t in (r?m. t!(m*n)) | Fact(n-1,t,s)
```

Here, `r?m` waits to receive a value for `m` on `r`, and `s!m` sends the value `m` on `s`. The expression **new** `t in P` creates a fresh communication channel `t` and executes `P`. If the above code is invoked with `r!1 | Fact(n,r,s)`, then there will be a synchronisation between `r!1` and the input `r?m` in the body of `Fact(n,r,s)`. In turn, this may delegate the computation of the factorial to another process in parallel by means of a subsequent synchronisation on a new channel `t`. That is, in order to compute the factorial of `n`, `Fact` builds a network of $n+1$ nodes, where node `i` takes as input a value `m` and outputs `m*i`. Due to the inability of statically reasoning about unbounded structures, the current analysers usually return false positives when fed with `Fact`. For example, this is the case of `TyPiCal` [12,11], a tool developed for pi-calculus. (In particular, `TyPiCal` fails to recognise that there is no circularity in the dependencies among r, s, and t.)

In this paper we develop a technique to enable the deadlock analysis of processes with arbitrary networks of nodes. Instead of reasoning on finite approximations of such processes, we associate them with terms of a basic recursive model, called *lam* – for *deadLock Analysis Model* –, which collects dependencies and features recursion and dynamic name creation [5,6]. For example, the lam function corresponding to `Fact` is

$$\texttt{fact}(a_1, a_2, a_3, a_4) = (a_2, a_3) + (\nu\, a_5, a_6)((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4))$$

where (a_2, a_3) displays the dependency between the actions `r?m` and `s!m` and (a_2, a_5) the one between `r?m` and `t!(m*n)`. The function `fact` is defined operationally by unfolding the recursive invocations; see Section 3. The unfolding of $\texttt{fact}(a_1, a_2, a_3, a_4)$ yields the following sequence of abstract states (bound names in the definition of `fact` are replaced by fresh ones in the unfoldings).

$$\begin{aligned}
&\texttt{fact}(a_1, a_2, a_3, a_4) \longrightarrow (a_2, a_3) + ((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4))\\
&\quad\longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, \texttt{fact}(a_7, a_8, a_3, a_4)\\
&\quad\longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_3)\\
&\qquad\quad +(a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_{10}) \,\&\, \texttt{fact}(a_9, a_{10}, a_3, a_4)\\
&\quad\longrightarrow \cdots
\end{aligned}$$

While the model of `fact` is not finite-state, in Section 4 we demonstrate that it is decidable whether the computations of a lam program will ever produce a circular dependency. In our previous work [5,6], the decidability was established only for a restricted subset of lams.

We then define a type system that associates lams to processes. Using the type system, for example, the lam program `fact` can be extracted from the factorial process `Fact`. For the sake of simplicity, we address the (*asynchronous*) *value passing CCS* [15], a simpler calculus than pi-calculus, because it is already adequate to demonstrate the power of our lam-based approach. The syntax, semantics, and examples of value passing CCS are in Section 5; the type system is defined in Section 6. As a byproduct of the above techniques, our system is powerful enough to detect deadlocks of programs that create networks with arbitrary numbers of processes. It is also worth to notice that our system admits type inference and can be easily extended to pi-calculus. We discuss the differences of our techniques with respect to the other ones in the literature in Section 7 where we also deliver some concluding remark.

2 Preliminaries

We use an infinite set $\mathscr{A}$ of (*level*) *names*, ranged over by $a, b, c, \cdots$. A relation on a set A of names, denoted $\mathtt{R}, \mathtt{R}', \cdots$, is an element of $\mathscr{P}(A \times A)$, where $\mathscr{P}(\cdot)$ is the standard powerset operator and $\cdot \times \cdot$ is the cartesian product. Let

– $\mathtt{R}^+$ be the *transitive closure* of $\mathtt{R}$.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that $\mathtt{R}_i \subseteq \mathtt{R}'^{+}_j$.

– $(a_0, a_1), \cdots, (a_{n-1}, a_n) \in \{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ if and only if there is $\mathtt{R}_i$ such that $(a_0, a_1), \cdots, (a_{n-1}, a_n) \in \mathtt{R}_i$.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \,\&\, \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \overset{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \le i \le m \text{ and } 1 \le j \le n\}$.

We use $\mathscr{R}, \mathscr{R}', \cdots$ to range over $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ and $\{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$, which are elements of $\mathscr{P}(\mathscr{P}(A \times A))$.

Definition 1. *A relation* $\mathtt{R}$ ***has a circularity*** *if* $(a, a) \in \mathtt{R}^+$ *for some* a. *A set of relations* $\mathscr{R}$ ***has a circularity*** *if there is* $\mathtt{R} \in \mathscr{R}$ *that has a circularity.*

For instance $\big\{\{(a,b),(b,c)\},\ \{(a,b),(c,b),(d,b),(b,c)\},\ \{(e,d),(d,c)\},\ \{(e,d)\}\big\}$ has a circularity because the second element of the set does.

3 The Language of Lams

In addition to the set of (*level*) *names*, we will also use *function names*, ranged over by $\mathtt{f}$, $\mathtt{g}$, $\mathtt{h}$, $\cdots$. A sequence of names is denoted by $\widetilde{a}$ and, with an abuse of notation, we also use $\widetilde{a}$ to address the *set of names* in the sequence.

A *lam program* is a pair $(\mathscr{L}, \mathtt{L})$, where $\mathscr{L}$ is a *finite set* of *function definitions* $\mathtt{f}(\widetilde{a}) = \mathtt{L}_\mathtt{f}$, with $\widetilde{a}$ and $\mathtt{L}_\mathtt{f}$ being the *formal parameters* and the *body* of $\mathtt{f}$, and $\mathtt{L}$ is the *main lam*. The syntax of the function bodies and the main lam is

$$\mathtt{L} \quad ::= \quad \mathtt{0} \ \mid\ (a,b) \ \mid\ \mathtt{f}(\widetilde{a}) \ \mid\ \mathtt{L} \,\&\, \mathtt{L} \ \mid\ \mathtt{L} + \mathtt{L} \ \mid\ (\nu\, a)\mathtt{L}$$

The lam $\mathtt{0}$ enforces no dependency, the lam (a,b) enforces the dependency (a,b), while $\mathtt{f}(\widetilde{a})$ represents a function invocation. The composite lam $\mathtt{L} \,\&\, \mathtt{L}'$ enforces the dependencies of $\mathtt{L}$ *and* of $\mathtt{L}'$, while $\mathtt{L} + \mathtt{L}'$ nondeterministically enforces the dependencies of $\mathtt{L}$ *or* of $\mathtt{L}'$, $(\nu\, a)\mathtt{L}$ creates a fresh name a and enforces the dependencies of $\mathtt{L}$ that may use a. Whenever parentheses are omitted, the operation "&" has precedence over "+". We will shorten $\mathtt{L}_1 \,\&\, \cdots \,\&\, \mathtt{L}_n$ into $\&_{i\in 1..n}\mathtt{L}_i$ and $(\nu\, a_1)\cdots(\nu\, a_n)\mathtt{L}$ into $(\nu\, a_1\cdots a_n)\mathtt{L}$. Function definitions $\mathtt{f}(\widetilde{a}) = \mathtt{L}_\mathtt{f}$ and $(\nu\, a)\mathtt{L}$ are *binders* of $\widetilde{a}$ in $\mathtt{L}_\mathtt{f}$ and of a in $\mathtt{L}$, respectively, and the corresponding occurrences of $\widetilde{a}$ in $\mathtt{L}_\mathtt{f}$ and of a in $\mathtt{L}$ are called *bound*. A name x in $\mathtt{L}$ is *free* if it is not underneath a $(\nu\, a)$ (similarly for function definitions). Let $var(\mathtt{L})$ be the set of free names in $\mathtt{L}$.

In the syntax of $\mathtt{L}$, the operations "&" and "+" are associative, commutative with $\mathtt{0}$ being the identity on &, and definitions and lams are equal up-to alpha renaming of bound names. Namely, if $a \notin var(\mathtt{L})$, the following axioms hold:

$$(\nu\, a)\mathtt{L} = \mathtt{L} \qquad (\nu\, a)\mathtt{L}' \,\&\, \mathtt{L} = (\nu\, a)(\mathtt{L}' \,\&\, \mathtt{L}) \qquad (\nu\, a)\mathtt{L}' + \mathtt{L} = (\nu\, a)(\mathtt{L}' + \mathtt{L})$$

Additionally, when $\mathtt{V}$ ranges over lams that do not contain function invocations, the following axioms hold:

$$\mathtt{V} \,\&\, \mathtt{V} = \mathtt{V} \qquad \mathtt{V} + \mathtt{V} = \mathtt{V} \qquad \mathtt{V} \,\&\, (\mathtt{L}' + \mathtt{L}'') = \mathtt{V} \,\&\, \mathtt{L}' + \mathtt{V} \,\&\, \mathtt{L}''$$

These axioms permit to rewrite a lam without function invocations as a *collection* (operation +) *of relations* (elements of a relation are gathered by the operation &). Let $\equiv$ be the least congruence containing the above axioms. They are restricted to terms $\mathtt{V}$ that do not contain function invocations. In fact, $\mathtt{f}(\widetilde{d})\&((a,b)+(b,c)) \neq (\mathtt{f}(\widetilde{d})\&(a,b))+(\mathtt{f}(\widetilde{d})\&(b,c))$ because the evaluation of the two lams (see below) may produce terms with different names.

Definition 2. *A lam* $\mathtt{V}$ *is in* ***normal form****, denoted* $\mathtt{nf}(\mathtt{V})$*, if* $\mathtt{V} = (\nu\widetilde{a})(\mathtt{V}_1 + \cdots + \mathtt{V}_n)$*, where* $\mathtt{V}_1, \cdots, \mathtt{V}_n$ *are dependencies only composed with* &.

Proposition 1. *For every* $\mathtt{V}$*, there is* $\mathtt{nf}(\mathtt{V})$ *such that* $\mathtt{V} \equiv \mathtt{nf}(\mathtt{V})$.

In the rest of the paper, we always assume lam programs $(\mathscr{L}, \mathtt{L})$ to be *well formed*.

Definition 3. *A lam program* $(\mathscr{L}, \mathtt{L})$ *is **well formed** if* (1) *function definitions in* $\mathscr{L}$ *have pairwise different function names and all function names occurring in the function bodies and* $\mathtt{L}$ *are defined;* (2) *the arity of function invocations occurring anywhere in the program matches the arity of the corresponding function definition;* (3) *every function definition in* $\mathscr{L}$ *has shape* $\mathtt{f}(\widetilde{a}) = (\nu\,\widetilde{c})\mathtt{L}_\mathtt{f}$*, where* $\mathtt{L}_\mathtt{f}$ *does not contain any* ν*-binder and* $var(\mathtt{L}_\mathtt{f}) \subseteq \widetilde{a} \cup \widetilde{c}$*.*

Operational semantics. Let a *lam context*, noted $\mathfrak{L}[\,]$, be a term derived by the following syntax:

$$\mathfrak{L}[\,] \quad ::= \quad [\,] \quad | \quad \mathtt{L} \,\&\, \mathfrak{L}[\,] \quad | \quad \mathtt{L} + \mathfrak{L}[\,]$$

As usual $\mathfrak{L}[\mathtt{L}]$ is the lam where the hole of $\mathfrak{L}[\,]$ is replaced by $\mathtt{L}$. According to the syntax, lam contexts have no ν-binder; that is, the hassle of name captures is avoided. The operational semantics of a program $(\mathscr{L}, \mathtt{L})$ is a transition system where *states* are lams, the *transition relation* is the least one satisfying the rule

$$\text{(RED)}\quad \frac{\mathtt{f}(\widetilde{a}) = (\nu\,\widetilde{c})\mathtt{L}_\mathtt{f} \in \mathscr{L} \qquad \widetilde{c}' \text{ are fresh} \qquad \mathtt{L}_\mathtt{f}[\widetilde{c}'/\widetilde{c}][\widetilde{a}'/\widetilde{a}] = \mathtt{L}'_\mathtt{f}}{\mathfrak{L}[\mathtt{f}(\widetilde{a}')] \longrightarrow \mathfrak{L}[\mathtt{L}'_\mathtt{f}]}$$

and the initial state is the lam $\mathtt{L}'$ such that $\mathtt{L} \equiv (\nu\,\widetilde{c})\mathtt{L}'$ and $\mathtt{L}'$ does not contain any ν-binder. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

By (RED), a lam $\mathtt{L}$ is evaluated by successively replacing function invocations with the corresponding lam instances. Name creation is handled by replacing bound names of function bodies with fresh names. For example, if $\mathtt{f}(a) = (\nu\, c)((a, c) \,\&\, \mathtt{f}(c))$ and $\mathtt{f}(a')$ occurs in the main lam, then $\mathtt{f}(a')$ is replaced by $(a', c') \,\&\, \mathtt{f}(c')$, where c' is a fresh name.

Let us discuss some examples.

1. $(\{\mathtt{f}(a, b, c) = (a, b) \,\&\, \mathtt{g}(b, c) + (b, c),\ \mathtt{g}(d, e) = (d, e) + (e, d)\},\ \mathtt{f}(a, b, c))$. Then

$$\begin{aligned}\mathtt{f}(a, b, c) &\longrightarrow (a, b) \,\&\, \mathtt{g}(b, c) + (b, c) \longrightarrow (a, b) \,\&\, ((b, c) + (c, b)) + (b, c)\\ &\longrightarrow (a, b) \,\&\, (b, c) + (a, b) \,\&\, (c, b) + (b, c)\end{aligned}$$

The lam in the final state *does not contain function invocations*. This is because the above program is not recursive. Additionally, the evaluation of $\mathtt{f}(a, b, c)$ *has not created names*. This is because the bodies of $\mathtt{f}$ and $\mathtt{g}$ do not contain ν-binders.

2. $(\{\mathtt{f}'(a) = (\nu\, b)(a, b) \,\&\, \mathtt{f}'(b)\}\ ,\ \mathtt{f}'(a_0))$. Then

$$\begin{aligned}\mathtt{f}'(a_0) &\longrightarrow\ (a_0, a_1) \,\&\, \mathtt{f}'(a_1) \longrightarrow (a_0, a_1) \,\&\, (a_1, a_2) \,\&\, \mathtt{f}'(a_2)\\ &\longrightarrow^n (a_0, a_1) \,\&\, \cdots \,\&\, (a_{n+1}, a_{n+2}) \,\&\, \mathtt{f}'(a_{n+2})\end{aligned}$$

In this case, *because of the* $(\nu\, b)$ *binder*, the lam grows in the number of dependencies as the evaluation progresses.

3. $(\{\mathtt{f}''(a) = (\nu\, b)(a, b) + (a, b) \,\&\, \mathtt{f}''(b)\},\ \mathtt{f}''(a_0))$. Then

$$\begin{aligned}\mathtt{f}''(a_0) &\longrightarrow\ (a_0, a_1) + (a_0, a_1) \,\&\, \mathtt{f}''(a_1)\\ &\longrightarrow\ (a_0, a_1) + (a_0, a_1) \,\&\, (a_1, a_2) + (a_0, a_1) \,\&\, (a_1, a_2) \,\&\, \mathtt{f}''(a_2)\\ &\longrightarrow^n (a_0, a_1) + \cdots + (a_0, a_1) \,\&\, \cdots \,\&\, (a_{n+1}, a_{n+2}) \,\&\, \mathtt{f}''(a_{n+2})\end{aligned}$$

In this case, the lam grows in the number of "+"-terms, which in turn become larger and larger as the evaluation progresses.

Flattening and circularities. Lams represent elements of the set $\mathscr{P}(\mathscr{P}(\mathscr{A} \times \mathscr{A}))$. This property is displayed by the following flattening function.

Let $\mathscr{L}$ be a set of function definitions and let $I(\cdot)$, called *flattening*, be a function on lams that (i) maps function name $\mathtt{f}$ defined in $\mathscr{L}$ to elements of $\mathscr{P}(\mathscr{P}(A \times A))$ and (ii) is defined on lams as follows

$$I(\mathbb{0}) = \{\varnothing\}, \qquad I((a,b)) = \{\{(a,b)\}\}, \qquad I(\mathtt{L} \,\&\, \mathtt{L}') = I(\mathtt{L}) \,\&\, I(\mathtt{L}'),$$

$$I(\mathtt{L} + \mathtt{L}') = I(\mathtt{L}) \cup I(\mathtt{L}'), \qquad I((\nu\, a)\mathtt{L}) = I(\mathtt{L})[^{a'}/_{a}] \text{ with } a' \text{ fresh},$$

$$I(\mathtt{f}(\widetilde{c})) = I(\mathtt{f})[^{\widetilde{c}}/_{\widetilde{a}}] \text{ (where } \widetilde{a} \text{ are the formal parameters of } \mathtt{f}).$$

Note that $I(\mathtt{L})$ is unique up to a renaming of names that do not occur free in $\mathtt{L}$. Let $I^{\perp}$ be the map such that, for every $\mathtt{f}$ defined in $\mathscr{L}$, $I^{\perp}(\mathtt{f}) = \{\varnothing\}$. For example, let $\mathscr{L}$ defines $\mathtt{f}$ and $\mathtt{g}$ and let

$$I(\mathtt{f}) = \{\{(a,b),(b,c)\}\} \qquad I(\mathtt{g}) = \{\{(b,a)\}\}$$
$$\mathtt{L}'' = \mathtt{f}(a,b,c) + (a,b) \,\&\, \mathtt{g}(b,c) \,\&\, \mathtt{f}(d,b,c) + \mathtt{g}(d,e) \,\&\, (d,c) + (e,d).$$

Then

$$I(\mathtt{L}'') = \{\{(a,b),(b,c)\},\ \{(a,b),(c,b),(d,b),(b,c)\},\ \{(e,d),(d,c)\},\ \{(e,d)\}\}$$
$$I^{\perp}(\mathtt{L}'') = \{\varnothing, \{(a,b)\}, \{(d,c)\}, \{(e,d)\}\}\,.$$

Definition 4. *A lam* $\mathtt{L}$ ***has a circularity*** *if* $I^{\perp}(\mathtt{L})$ *has a circularity. A lam program* $(\mathscr{L}, \mathtt{L})$ ***has a circularity*** *if there is* $\mathtt{L} \longrightarrow^{*} \mathtt{L}'$ *and* $\mathtt{L}'$ *has a circularity.*

The property of "having a circularity" is preserved by $\equiv$ while the "absence of circularities" of a composite lam can be carried to its components.

Proposition 2. *1. if* $\mathtt{L} \equiv \mathtt{L}'$ *then* $\mathtt{L}$ *has a circularity if and only if* $\mathtt{L}'$ *has a circularity;*
2. $\mathtt{L} \,\&\, \mathtt{L}'$ *has no circularity implies both* $\mathtt{L}$ *and* $\mathtt{L}'$ *have no circularity (similarly for* $\mathtt{L} + \mathtt{L}'$ *and for* $(\nu\, a)\mathtt{L}$*).*

4 The Decision Algorithm for Detecting Circularities

In this section we assume a lam program $(\mathscr{L}, \mathtt{L})$ such that pairwise different function definitions in $\mathscr{L}$ have disjoint formal parameters. Without loss of generality, we assume that $\mathtt{L}$ does not contain any ν-binder.

Fixpoint definition of the interpretation function. The basic item of our algorithm is the computation of lam functions' interpretation. This computation is performed by means of a standard fixpoint technique that is detailed below.

Let A be the set of formal parameters of definitions in $\mathscr{L}$ and let $\varkappa$ be a special name that does not occur in $(\mathscr{L}, \mathtt{L})$. We use the domain $\left(\mathscr{P}(\mathscr{P}(A \cup \{\varkappa\} \times A \cup \{\varkappa\})),\ \subseteq\right)$ which is a *finite* lattice [3].

Definition 5. *Let* $\mathtt{f}_i(\widetilde{a_i}) = (\nu\widetilde{c_i})\mathtt{L}_i$*, with* $i \in 1..n$*, be the function definitions in* $\mathscr{L}$*. The family of flattening functions* $I^{(k)}_{\mathscr{L}} : \{\mathtt{f}_1, \cdots, \mathtt{f}_n\} \to \mathscr{P}(\mathscr{P}(A \cup \{\varkappa\} \times A \cup \{\varkappa\}))$ *is defined as follows*

$$I^{(0)}_{\mathscr{L}}(\mathtt{f}_i) = \{\varnothing\} \qquad I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_i) = \{\mathtt{proj}_{\widetilde{a_i}}(R^{+}) \mid R \in I^{(k)}_{\mathscr{L}}(\mathtt{L}_i)\}$$

where $\mathtt{proj}_{\widetilde{a}}(R) \stackrel{def}{=} \{(a,b) \mid (a,b) \in R \text{ and } a,b \in \widetilde{a}\} \cup \{(\varkappa,\varkappa) \mid (c,c) \in R \text{ and } c \notin \widetilde{a}\}$.

We notice that $I^{(0)}_{\mathscr{L}}$ is the function $I^{\perp}$ of the previous section.

Proposition 3. *Let* $\mathtt{f}(\widetilde{a}) = (\nu\widetilde{c})\mathtt{L}_{\mathtt{f}} \in \mathscr{L}$. *(i) For every* k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \in \mathscr{P}(\mathscr{P}((\widetilde{a} \cup \{\varkappa\}) \times (\widetilde{a} \cup \{\varkappa\})))$. *(ii) For every* k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \Subset I^{(k+1)}_{\mathscr{L}}(\mathtt{f})$.

Since, for every k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}_i)$ ranges over a finite lattice, by the fixpoint theory [3], there exists m such that $I^{(m)}_{\mathscr{L}}$ is a fixpoint, namely $I^{(m)}_{\mathscr{L}} \approx I^{(m+1)}_{\mathscr{L}}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let $I_{\mathscr{L}}$, called the *interpretation function* (of a lam), be the least fixpoint $I^{(m)}_{\mathscr{L}}$.

Example 1. For example, let $\mathscr{L}$ be the factorial function in Section 1. Then

$$I^{(0)}_{\mathscr{L}}(\mathtt{fact}) = \{\emptyset\} \qquad I^{(1)}_{\mathscr{L}}(\mathtt{fact}) = \{\{(a_2,a_3)\},\emptyset\} \qquad I^{(2)}_{\mathscr{L}}(\mathtt{fact}) = \{\{(a_2,a_3)\},\emptyset\}$$

That is, in this case, $I_{\mathscr{L}} = I^{(1)}_{\mathscr{L}}$. □

Theorem 1. *A lam program* $(\mathscr{L},\mathtt{L})$ *has a circularity if and only if* $I_{\mathscr{L}}(\mathtt{L})$ *has a circularity.*

For example, let $\mathscr{L}$ be the factorial function in Section 1 and let $\mathtt{L} = (a_3,a_2)\ \&\ \mathtt{fact}(a_1,a_2,a_3.a_4)$. From Example 1, we have $I_{\mathscr{L}}(\mathtt{fact}) = \{\{(a_2,a_3)\},\emptyset\}$. Since $I_{\mathscr{L}}(\mathtt{L})$ has a circularity, by Theorem 1, there is $\mathtt{L} \longrightarrow^* \mathtt{L}'$ such that $I^{\perp}(\mathtt{L}')$ has a circularity. In fact it displays a circularity after the first transition:

$$\mathtt{L} \longrightarrow (a_3,a_2)\ \&\ ((a_2,a_3) + ((a_2,a_5)\ \&\ \mathtt{fact}(a_5,a_6,a_3,a_4)))\ .$$

5 Value-Passing CCS

In the present and next sections, we apply the foregoing theory of lams to refine Kobayashi's type system for deadlock-freedom of concurrent programs [11]. In his type system, the deadlock-freedom is guaranteed by a combination of *usages*, which are a kind of behavioral types capturing channel-wise communication behaviors, and *capability/obligation levels*, which are natural numbers capturing inter-channel dependencies (like "a message is output on x only if a message is received along y"). By replacing numbers with (lam) level names, we can achieve a more precise analysis of deadlock-freedom because of the algorithm in Section 4. The original type system in [11] is for the pi-calculus [16], but for the sake of simplicity, we consider a variant of the value-passing CCS [15], which is sufficient for demonstrating the power of our lam-based approach.

Our *value-passing CCS* uses several disjoint countable sets of names: in addition to level names, there are *integer and channel names*, ranged over by $x, y, z, \cdots$, *process names*, ranged over by A, B, $\cdots$, and *usage names*, ranged over by α, β, $\cdots$. A *value-passing CCS program* is a pair $(\mathscr{D}, P)$, where $\mathscr{D}$ is a *finite set* of *process name definitions* $A(\widetilde{a};\widetilde{x}) = P_A$, with $\widetilde{a};\widetilde{x}$ and P_A respectively being the *formal parameters* and the *body* of A, and P is the *main process*.

The syntax of processes P_A and P is shown in Figure 1. A process can be the inert process $\mathbb{0}$, a message $x!e$ sent on a name x that carries (the value of) an expression e, an input $x?y.P$ that consumes a message $x!v$ and behaves like $P[^v/_y]$, a parallel composition of processes $P \mid Q$, a conditional **if** e **then** P **else** Q that evaluates e and behaves either

$$
\begin{array}{l}
P \text{ (processes) } ::= \mathbb{0} \mid x!e \mid x?y.P \mid (P \mid Q) \mid \textbf{if } e \textbf{ then } P \textbf{ else } Q \mid (\nu \widetilde{a}; x : \mathtt{T})P \mid A(\widetilde{a}; \widetilde{e}) \\
e \text{ (expressions) } ::= x \mid v \mid e_1 \textbf{ op } e_2 \\
\mathtt{T} \text{ (types) } ::= \textbf{int} \mid U \\
U \text{ (usages) } ::= \mathbf{0} \mid !^{a_1}_{a_2} \mid ?^{a_1}_{a_2}.U \mid (U_1|U_2) \mid \alpha \mid \mu\alpha.U
\end{array}
$$

Fig. 1. The Syntax of value-passing CCS

like P or like Q depending on whether the value is $\neq 0$ (*true*) or $= 0$ (*false*), a restriction $(\nu\widetilde{a}; x : \mathtt{T})P$ that behaves like P except that communications on x with the external environment are prohibited, an invocation $A(\widetilde{a}; \widetilde{e})$ of the process corresponding to A.

An expression e can be a name x, an integer value v, or a generic binary operation on integers v **op** v', where **op** ranges over a set including the usual operators like $+$, $\leq$, etc. Integer expressions without names (*constant expressions*) may be evaluated to an integer value (the definition of the evaluation of constant expressions is omitted). Let $[\![e]\!]$ be the evaluation of a constant expression e ($[\![e]\!]$ is undefined when the integer expression e contains integer names). Let also $[\![x]\!] = x$ when x is a non-integer name.

We defer the explanation of the meaning of types $\mathtt{T}$ (and usages U) until Section 6. It is just for the sake of simplicity that processes are annotated with types and level names. They do not affect the operational semantics of processes, and can be automatically inferred by using an inference algorithm similar to those in [11,10].

Similarly to lams, $A(\widetilde{a}; \widetilde{x}) = P_A$ and $(\nu\widetilde{a}; x : \mathtt{T})P$ are *binders* of $\widetilde{a}; \widetilde{x}$ in P_A and of $\widetilde{a}, x$ in P, respectively. We use the standard notions of alpha-equivalence, free and bound names of processes and, with an abuse of notation, we let $var(P)$ be the free names in P. In process name definitions $A(\widetilde{a}; \widetilde{x}) = P_A$, we always assume that $var(P_A) \subseteq \widetilde{a}, \widetilde{x}$.

Definition 6. *The* ***structural equivalence*** $\equiv$ *on processes is the least congruence containing alpha-conversion of bound names, commutativity and associativity of* $\mid$ *with identity* $\mathbb{0}$*, and closed under the following rule:*

$$((\nu\widetilde{a}; x : T)P) \mid Q \equiv (\nu\widetilde{a}; x : T)(P \mid Q) \qquad \widetilde{a}, x \notin var(Q)\ .$$

The ***operational semantics*** *of a program* $(\mathscr{D}, P)$ *is a transition system where the states are processes, the initial state is P, and the* ***transition relation*** $\longrightarrow_{\mathscr{D}}$ *is the least one closed under the following rules:*

$$
\text{(R-Com)}\ \frac{[\![e]\!] = v}{x!e \mid x?y.P \longrightarrow_{\mathscr{D}} P[^{v}/_{y}]}
\qquad
\text{(R-Par)}\ \frac{P \longrightarrow_{\mathscr{D}} P'}{P \mid Q \longrightarrow_{\mathscr{D}} P' \mid Q}
\qquad
\text{(R-New)}\ \frac{P \longrightarrow_{\mathscr{D}} Q}{(\nu\widetilde{a}; x : T)P \longrightarrow_{\mathscr{D}} (\nu\widetilde{a}; x : T)Q}
$$

$$
\text{(R-IfT)}\ \frac{[\![e]\!] \neq 0}{\textbf{if } e \textbf{ then } P \textbf{ else } Q \longrightarrow_{\mathscr{D}} P}
\qquad
\text{(R-IfF)}\ \frac{[\![e]\!] = 0}{\textbf{if } e \textbf{ then } P \textbf{ else } Q \longrightarrow_{\mathscr{D}} Q}
\qquad
\text{(R-Call)}\ \frac{[\![\widetilde{e}]\!] = \widetilde{v} \quad A(\widetilde{a}; \widetilde{x}) = P \in \mathscr{D}}{A(\widetilde{a'}; \widetilde{e}) \longrightarrow_{\mathscr{D}} P[^{\widetilde{a'}}/_{\widetilde{a}}][^{\widetilde{v}}/_{\widetilde{x}}]}
$$

$$
\text{(R-Cong)}\ \frac{P \equiv P' \quad P' \longrightarrow_{\mathscr{D}} Q' \quad Q' \equiv Q}{P \longrightarrow_{\mathscr{D}} Q}
$$

We often omit the subscript of $\longrightarrow_{\mathscr{D}}$ *when it is clear from the context. We write* $\longrightarrow^*$ *for the reflexive and transitive closure of* $\longrightarrow$*.*

The deadlock-freedom of a process P, which is the basic property that we will verify, means that P does not get stuck into a state where there is a message or an input. The formal definition is below.

Definition 7 (deadlock-freedom). *A program $(\mathscr{D}, P)$ is **deadlock-free** if the following condition holds: whenever $P \rightarrow^* P'$ and either (i) $P' \equiv (\nu \widetilde{a}_1; x_1 : T_1) \cdots (\nu \widetilde{a}_k; x_k : T_k)(x!v \mid Q)$, or (ii) $P' \equiv (\nu \widetilde{a}_1; x_1 : T_1) \cdots (\nu \widetilde{a}_k; x_k : T_k)(x?y.Q_1 \mid Q_2)$, then there exists R such that $P' \rightarrow R$.*

Example 2 (The dining philosophers). Consider the program consisting of the process definitions

$$\begin{array}{l} \mathit{Phils}(a_1, a_2, a_3, a_4; n : \mathbf{int}, \mathit{fork}_1 : U_1, \mathit{fork}_2 : U_2) = \\ \quad \mathbf{if}\ n = 1\ \mathbf{then}\ \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)\ \mathbf{else} \\ \qquad (\nu a_5, a_6; \mathit{fork}_3 : U_3 \mid U_3 \mid !^{a_5}_{a_6})(\ \ \mathit{Phils}(a_1, a_2, a_5, a_6; n-1, \mathit{fork}_1, \mathit{fork}_3) \\ \qquad\qquad\qquad \mid\ \ \mathit{Phil}(a_5, a_6, a_3, a_4; \mathit{fork}_3, \mathit{fork}_2)\ \ \mid\ \ \mathit{fork}_3!1\ \ ) \end{array}$$

$$\begin{array}{l} \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1 : U_1, \mathit{fork}_2 : U_2) = \\ \quad \mathit{fork}_1?x_1.\mathit{fork}_2?x_2.(\,\mathit{fork}_1!x_1 \mid \mathit{fork}_2!x_2 \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)\,) \end{array}$$

and of the main process P:

$$\begin{array}{l} (\nu a_1, a_2; \mathit{fork}_1 : U_1 \mid U_1 \mid !^{a_1}_{a_2})(\nu a_3, a_4; \mathit{fork}_2 : U_2 \mid U_2 \mid !^{a_1}_{a_2}) \\ (\,\mathit{Phils}(a_1, a_2, a_3, a_4; m, \mathit{fork}_1, \mathit{fork}_2) \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2) \mid \mathit{fork}_1!1 \mid \mathit{fork}_2!1\,) \end{array}$$

Here, $U_1 = \mu\alpha.?^{a_2}_{a_1}.(!^{a_1}_{a_2} \mid \alpha)$, $U_2 = \mu\alpha.?^{a_4}_{a_3}.(!^{a_3}_{a_4} \mid \alpha)$, and $U_3 = \mu\alpha.?^{a_6}_{a_5}.(!^{a_5}_{a_6} \mid \alpha)$, but please ignore the types for the moment. Every philosopher $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)$ grabs the two forks fork_1 and fork_2 in this order, releases the forks, and repeats the same behavior. The main process creates a ring consisting of $m + 1$ philosophers, where only one of the philosophers grabs the forks in the opposite order to avoid deadlock. This program is indeed deadlock-free in our definition. On the other hand, if we replace $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)$ with $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_2, \mathit{fork}_1)$ in the main process, then the resulting process is not deadlock-free. □

The dining philosophers example is a paradigmatic case of the power of the analysis described in the next section. This example cannot be type-checked in Kobayashi's previous type system [11]: see Remark 1 in Section 6.

6 The Deadlock Freedom Analysis of Value-Passing CCS

We now explain the syntax of types in Figure 1. A type is either **int** or a usage. The former is used to type *integer* names; the latter is used to type *channel* names [11,9]. A usage describes how a channel can be used for input and output. The usage $\mathbf{0}$ describes a channel that cannot be used, $!^{a_1}_{a_2}$ describes a channel that is used for output, $?^{a_1}_{a_2}.U$ describes a channel that is first used for input and then used according to U, and $U \mid U'$ describes a channel that is used according to U and U', possibly in parallel. For example, in process $x!2 \mid x?z.y!z$, y has the usage $!^{a_1}_{a_2}$ (please, ignore the subscript and superscript for the moment), and x has the usage $!^{a_3}_{a_4} \mid ?^{a_5}_{a_6}.\mathbf{0}$. The usage $\mu\alpha.U$ describes a channel that is used recursively according to $U[^{\mu\alpha.U}/_{\alpha}]$. The operation $\mu\alpha.-$ is a binder and we use the standard notions of alpha-equivalence, free and bound usage names. For example, $\mu\alpha.!^{a_1}_{a_2}.\alpha$ describes a channel that can be sequentially used for output an arbitrary number of times; $\mu\alpha.?^{a_1}_{a_2}.!^{a_3}_{a_4}.\alpha$ describes a channel that should be used for input and output alternately. We often omit a trailing $\mathbf{0}$ and just write $?^{a_1}_{a_1}$ for $?^{a_1}_{a_1}.\mathbf{0}$.

The superscripts and subscripts of ? and ! are level names of lams (recall Section 3), and are used to control the causal dependencies between communications [11]. The superscript, called an *obligation level*, describes the degree of the obligation to use the channel for the specified operation. The subscript, called a *capability level*, describes the degree of the capability to use the channel for the specified operation (and successfully find a partner of the communication).

In order to detect deadlocks we consider the following two conditions:

1. If a process has an obligation of level a, then it can exercise only capabilities of level *a' less than a* before fulfilling the obligation. This corresponds to a dependency (a', a). For example, if x has type $?^{a_1}_{a_2}$ and y has type $!^{a_3}_{a_4}$, then the process $x?u.y!u$ has lam (a_2, a_3).

2. The whole usage of each channel must be consistent, in the sense that if there is a capability of level a to perform an input (respectively, a message), there must be a corresponding obligation of level a to perform a corresponding message (respectively, input). For example, the usage $!^{a_1}_{a_2} \mid ?^{a_2}_{a_1}$ is consistent, but neither $!^{a_1}_{a_2} \mid ?^{a_1}_{a_2}$ nor $!^{a_1}_{a_2}$ is.

To see how the constraints above guide our deadlock analysis, consider the (deadlocked) process: $x?u.y!u \mid y?u.x!u$. Because of condition 2 above, the usage of x and y must be of the form $?^{a_1}_{a_2} \mid !^{a_2}_{a_1}$ and $?^{a_3}_{a_4} \mid !^{a_4}_{a_3}$ respectively. Due to 1, we derive (a_2, a_4) for $x?u.y!u$, and (a_4, a_2) for $y?u.x!u$. Hence the process is deadlocked because the lam $(a_2, a_4) \& (a_4, a_2)$ has a circularity. On the other hand, for the process $x?u.y!u \mid y?u.\mathbf{0} \mid x!u$, we derive the lam (a_2, a_4), which has no circularity. Indeed, this last process is not deadlocked. While we use lams to detect deadlocks, Kobayashi [11] used natural numbers for obligation/capability levels.

As explained above, usages describe the channel-wise behavior of a process, and they form a tiny process calculus. The usage reduction relation $U \rightsquigarrow U'$ defined below means that if a channel of usage U is used for a communication, the channel may be used according to U' afterwards.

Definition 8. *Let = be the least congruence on usages containing alpha-conversion of bound names, commutativity and associativity of $\mid$ with identity $\mathbf{0}$, and closed under the following rule:*

(UC-Mu)

$$\mu\alpha.U \;=\; U[^{\mu\alpha.U}/_{\alpha}]$$

The reduction relation $U \rightsquigarrow U'$ is the least relation closed under the rules:

(UR-Com)
$$!^{a_1}_{a_2} \mid ?^{a_3}_{a_4}.U \rightsquigarrow U$$

(UR-Par)
$$\frac{U_1 \rightsquigarrow U_1'}{U_1 \mid U_2 \rightsquigarrow U_1' \mid U_2}$$

(UR-Cong)
$$\frac{U_1 = U_1' \quad U_1' \rightsquigarrow U_2' \quad U_2' = U_2}{U_1 \rightsquigarrow U_2}$$

As usual, we let $\rightsquigarrow^*$ be the reflexive and transitive closure of $\rightsquigarrow$.

The following relation $rel(U)$ guarantees the condition 2 on capabilities and obligations above, that each capability must be accompanied by a corresponding obligation. This must hold during the whole computation, hence the definition below. The predicate $rel(U)$ is computable because it may be reduced to Petri Nets reachability (see [10] for the details about the encoding).

Definition 9. *U is **reliable**, written $rel(U)$, when the following conditions hold:*

1. *whenever $U \rightsquigarrow^* U'$ and $U' = !^{a_1}_{a_2} \mid U_1$, there are U_2 and U_3 such that $U_1 = ?^{a_2}_{a_3}.U_2 \mid U_3$ for some a_3; and*
2. *whenever $U \rightsquigarrow^* U'$ and $U' = ?^{a_1}_{a_2}.U_1 \mid U_2$, there is U_3 such that $U_2 = !^{a_2}_{a_3} \mid U_3$ for some a_3.*

The following type system uses *type environments*, ranged over $\Gamma, \Gamma', \cdots$, that map integer and channel names to types and process names to sequences $[\widetilde{a}; \widetilde{T}]$. When $x \notin dom(\Gamma)$, we write $\Gamma, x{:}T$ for the environment such that $(\Gamma, x{:}T)(x) = T$ and $(\Gamma, x{:}T)(y) = \Gamma(y)$, otherwise. The operation $\Gamma_1 \mid \Gamma_2$ is defined by:

$$(\Gamma_1 \mid \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \text{ and } x \notin dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \text{ and } x \notin dom(\Gamma_1) \\ [\widetilde{a}; \widetilde{T}] & \text{if } \Gamma_1(x) = \Gamma_2(x) = [\widetilde{a}; \widetilde{T}] \\ \mathbf{int} & \text{if } \Gamma_1(x) = \Gamma_2(x) = \mathbf{int} \\ U_1 \mid U_2 & \text{if } \Gamma_1(x) = U_1 \text{ and } \Gamma_2(x) = U_2 \end{cases}$$

The map $\Gamma_1 \mid \Gamma_2$ is undefined if, for some x, $(\Gamma_1 \mid \Gamma_2)(x)$ does not match any of the cases. Let $var(\Gamma) = \{a \mid \text{there is } x : \Gamma(x) = U \text{ and } a \in var(U)\}$.

There are three kinds of type judgments:

$\Gamma \vdash e : T$ – the expression e has type T in Γ;
$\Gamma \vdash P : \mathsf{L}$ – the process P has lam L in Γ;
$\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathsf{L})$ – the program $(\mathscr{D}, P)$ has lam program $(\mathscr{L}, \mathsf{L})$ in Γ.

As usual, $\Gamma \vdash e : T$ means that e evaluates to a value of type T under an environment that respects the type environment Γ. The judgment $\Gamma \vdash P : \mathsf{L}$ means that P uses each channel x according to $\Gamma(x)$, with the causal dependency as described by L. For example, $x{:}?^{a_1}_{a_2}, y{:}!^{a_3}_{a_4} \vdash x?u.y!u : (a_2, a_3)$ should hold.

The typing rules of value-passing CCS are defined in Figure 2, where we use the predicate $noact(\Gamma)$ and the function $ob(U)$ defined as follows:

$noact(\Gamma) = true$ if and only if, for every channel name $x \in dom(\Gamma)$, $\Gamma(x) = \mathbf{0}$;
$ob(\Gamma) = \bigcup_{x \in dom(\Gamma), \Gamma(x) = U} ob(U)$ where

$$ob(\mathbf{0}) = \emptyset \qquad ob(!^{a_1}_{a_2}) = \{a_1\} \qquad ob(?^{a_1}_{a_2}.U) = \{a_1\}$$
$$ob(U \mid U') = ob(U) \cup ob(U') \qquad ob(\mu\alpha.U) = ob(U[^{\mathbf{0}}/_{\alpha}])$$

The predicate $noact(\Gamma)$ is used for controlling weakening (as in linear type systems). For example, if we did not require $noact(\Gamma)$ in rule T-Zero, then we would obtain $x{:}?^{a_1}_{a_2}.\mathbf{0} \vdash \mathbb{0} : \mathbb{0}$. Then, by using T-In and T-Out, we would obtain: $x{:}?^{a_1}_{a_2}.\mathbf{0} \mid !^{a_2}_{a_1} \vdash \mathbb{0} \mid x!1 : \mathbb{0}$, and wrongly conclude that the output on x does not get stuck. It is worth to notice that, in the typing rules, we identify usages up to $=$.

Processes:

(T-Zero)
$$\frac{noact(\Gamma)}{\Gamma \vdash \mathbf{0} : \mathbf{0}}$$

(T-Out)
$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma, x{:}!^{a_1}_{a_2} \vdash x!e : \mathbf{0}}$$

(T-In)
$$\frac{\Gamma, x : U, y : \mathbf{int} \vdash P : \mathtt{L}}{\Gamma, x{:}?^{a_1}_{a_2}.U \vdash x?y.P : \mathtt{L} \mathbin{\&} (\&_{a \in ob(\Gamma)}(a_2, a))}$$

(T-Par)
$$\frac{\Gamma \vdash P : \mathtt{L} \qquad \Gamma' \vdash P' : \mathtt{L}'}{\Gamma \mid \Gamma' \vdash P \mid P' : \mathtt{L} \mathbin{\&} \mathtt{L}'}$$

(T-New)
$$\frac{\Gamma, x : U \vdash P : \mathtt{L} \qquad rel(U) \qquad \widetilde{a} \cap var(\Gamma) = \emptyset}{\Gamma \vdash (\nu \widetilde{a}; x : U)P : (\nu \widetilde{a})\mathtt{L}}$$

(T-If)
$$\frac{\Gamma \vdash e : \mathbf{int} \qquad \Gamma' \vdash P : \mathtt{L} \qquad \Gamma' \vdash P' : \mathtt{L}'}{\Gamma \mid \Gamma' \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ P' : \mathtt{L} + \mathtt{L}'}$$

(T-Call)
$$\frac{\Gamma(A) = [\widetilde{a}; \widetilde{\mathtt{T}}] \qquad |\widetilde{a}| = |\widetilde{a'}| \qquad \Gamma \vdash \widetilde{e} : \widetilde{\mathtt{T}}}{\Gamma \vdash A(\widetilde{a'}; \widetilde{e}) : \mathtt{f}_A(\widetilde{a'})}$$

Expressions:

(T-Int)
$$\frac{noact(\Gamma)}{\Gamma \vdash n : \mathbf{int}}$$

(T-Var)
$$\frac{noact(\Gamma)}{\Gamma, x : \mathtt{T} \vdash x : \mathtt{T}}$$

(T-Op)
$$\frac{\Gamma \vdash e : \mathbf{int}\ \Gamma \vdash e' : \mathbf{int}}{\Gamma \vdash e\ \mathbf{op}\ e' : \mathbf{int}}$$

(T-Seq)
$$\frac{(\Gamma_i \vdash e_i : T_i)^{i \in 1..n}}{\Gamma_1 \mid \cdots \mid \Gamma_n \vdash e_1, \ldots, e_n : T_1, \ldots, T_n}$$

Programs:

(T-Prog)
$$\frac{\begin{array}{c}\mathscr{D} = \bigcup_{i \in 1..n}\{A_i(\widetilde{a_i}; \widetilde{x_i} : \widetilde{\mathtt{T}}_i) = P_i\} \qquad \Gamma = (A_i : [\widetilde{a_i}; \widetilde{\mathtt{T}}_i])^{i \in 1..n} \\ (\Gamma, \widetilde{x_i} : \widetilde{\mathtt{T}}_i \vdash P_i : \mathtt{L}_i)^{i \in 1..n} \qquad \Gamma' \vdash P : \mathtt{L} \qquad \mathscr{L} = \bigcup_{i \in 1..n}\{\mathtt{f}_{A_i}(\widetilde{a_i}) = \mathtt{L}_i\}\end{array}}{\Gamma \mid \Gamma' \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})}$$

Fig. 2. The type system of value-passing CCS (we assume a function name $\mathtt{f}_A$ for every process name A)

A few key rules are discussed. Rule (T-In) is the unique one that introduces dependency pairs. In particular, the process $x?u.P$ will be typed with a lam that contains pairs (a_2, a), where a_2 is the capability of x and a is the obligation of every channel in P (because they are all causally dependent from x). Rule (T-Out) just records in the type environment that x is used for output. Rule (T-Par) types a parallel composition of processes by collecting the environments – operation "$\mid$" – (like in other linear type systems [13,9]) and the lams of the components. Rule (T-Call) types a process name invocation in terms of a (lam) function invocation and constrains the sequences of level names in the two invocations to have equal lengths ($|\widetilde{a}| = |\widetilde{a'}|$) and the types of expressions to match with the types in the process declaration.

Example 3. We illustrate the type system in Figure 2 by typing two simple processes:

$$P = (\nu\, a_1, a_2; x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu\, a_3, a_4; y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3})(x?z.y!z \mid y?z.x!z)$$
$$Q = (\nu\, a_1, a_2; x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu\, a_3, a_4; y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3})(x?z.y!z \mid y?z.\mathbf{0} \mid x!1)$$

The proof tree of P is

$$\frac{\dfrac{\dfrac{y{:}!^{a_4}_{a_3},\ z : \mathbf{int} \vdash y!z : \mathbf{0}}{x{:}?^{a_1}_{a_2},\ y{:}!^{a_4}_{a_3} \vdash x?z.y!z : (a_2, a_4)} \qquad \dfrac{x{:}!^{a_2}_{a_1},\ z : \mathbf{int} \vdash x!z : \mathbf{0}}{x{:}!^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4} \vdash y?z.x!z : (a_4, a_2)}}{x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3} \vdash x?z.y!z \mid y?z.x!z : (a_2, a_4) \mathbin{\&} (a_4, a_2)}}{\emptyset \vdash P : (\nu\, a_1, a_2)(\nu\, a_3, a_4)((a_2, a_4) \mathbin{\&} (a_4, a_2))}$$

and we notice that the lam in the conclusion has a circularity (in fact, P is deadlocked). The typing of Q is

$$\dfrac{\dfrac{\dfrac{\dfrac{z:\mathbf{int}\vdash z:\mathbf{int}}{y{:}!^{a_4}_{a_3},\ z:\mathbf{int}\vdash y!z:\mathbb{0}}}{x{:}?^{a_1}_{a_2},\ y{:}!^{a_4}_{a_3}\vdash x?z.y!z:(a_2,a_4)} \qquad \dfrac{y:\mathbb{0},\ z:\mathbf{int}\vdash \mathbb{0}:\mathbb{0}}{y{:}?^{a_3}_{a_4}\vdash y?z.\mathbb{0}:\mathbb{0}} \qquad \dfrac{\varnothing\vdash 1:\mathbf{int}}{x{:}!^{a_2}_{a_1}\vdash x!1:\mathbb{0}}}{x{:}?^{a_1}_{a_2}\mid !^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4}\mid !^{a_4}_{a_3}\vdash x?z.y!z\mid y?z.\mathbb{0}\mid x!1:(a_2,a_4)}}{\varnothing\vdash Q:(\nu\, a_1,a_2)(\nu\, a_3,a_4)(a_2,a_4)}$$

The lam in the conclusion has no circularity. In fact, Q is not deadlocked. □

Example 3 also spots one difference between the type system in [11] and the one in Figure 2. Here the inter-channel dependencies check is performed *ex-post* by resorting to the lam algorithm in Section 4; in [11] this check is done *during* the type checking(/inference) and, for this reason, the process P is not typable in previous Kobayashi's type systems. In this case, the two analysers both recognize that P is deadlocked; Example 4 below discusses a case where the precision is different.

The following theorem states the soundness of our type system.

Theorem 2. *Let $\Gamma\vdash(\mathscr{D},P):(\mathscr{L},\mathtt{L})$ such that noact(Γ). If $(\mathscr{L},\mathtt{L})$ has no circularity then $(\mathscr{D},P)$ is deadlock-free.*

The following examples highlight the difference of the expressive power of the system in Figure 2 and the type system in [11].

Example 4. Let $(\mathscr{D},P)$ be the dining philosopher program in Example 2 and U_1 and U_2 be the usages defined therein. We have $\Gamma\vdash(\mathscr{D},P):(\mathscr{L},\mathtt{L})$ where

$$\begin{aligned}
\Gamma &= \mathit{Phils}:[a_1,a_2,a_3,a_4;\mathbf{int},U_1,U_2],\mathit{Phil}:[a_1,a_2,a_3,a_4;U_1,U_2]\\
\mathscr{L} &= \{\ \mathtt{f}_{\mathit{Phils}}(a_1,a_2,a_3,a_4)=\mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)\\
&\qquad\qquad\qquad\qquad +(\nu a_5,a_6)(\mathtt{f}_{\mathit{Phils}}(a_1,a_2,a_5,a_6)\ \&\ \mathtt{f}_{\mathit{Phil}}(a_5,a_6,a_3,a_4)),\\
&\qquad\ \mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)=(a_1,a_4)\ \&\ (a_3,a_1)\ \&\ (a_3,a_2)\ \&\ \mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)\ \}\\
\mathtt{L} &= (\nu a_1,a_2,a_3,a_4)(\mathtt{f}_{\mathit{Phils}}(a_1,a_2,a_3,a_4)\ \&\ \mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4))
\end{aligned}$$

For example, let

$$\begin{aligned}
P_1 &= \mathit{fork}_1?x_1.\mathit{fork}_2?x_2.(\,\mathit{fork}_1!x_1\ \mid \mathit{fork}_2!x_2\ \mid\ \mathit{Phil}(a_1,a_2,a_3,a_4;\mathit{fork}_1,\mathit{fork}_2)\,)\\
P_2 &= \mathit{fork}_2?x_2.(\,\mathit{fork}_1!x_1\ \mid \mathit{fork}_2!x_2\ \mid\ \mathit{Phil}(a_1,a_2,a_3,a_4;\mathit{fork}_1,\mathit{fork}_2)\,)\\
P_3 &= \mathit{fork}_1!x_1\ \mid \mathit{fork}_2!x_2\ \mid\ \mathit{Phil}(a_1,a_2,a_3,a_4;\mathit{fork}_1,\mathit{fork}_2)
\end{aligned}$$

Then the body P_1 of Phil is typed as follows:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Gamma_2,\mathit{fork}_1{:}!^{a_1}_{a_2}\vdash \mathit{fork}_1!x_1:\mathbb{0} \qquad \Gamma_2,\mathit{fork}_2{:}!^{a_3}_{a_4}\vdash \mathit{fork}_2!x_2:\mathbb{0}\\ \Gamma_2,\mathit{fork}_1:U_1,\mathit{fork}_2:U_2\vdash \mathit{Phil}(a_1,a_2,a_3,a_4;\mathit{fork}_1,\mathit{fork}_2):\mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)\end{array}}{\Gamma_2,\mathit{fork}_1{:}!^{a_1}_{a_2}\mid U_1,\mathit{fork}_2{:}!^{a_3}_{a_4}\mid U_2\vdash P_3:\mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)}}{\Gamma_1,\mathit{fork}_1{:}!^{a_1}_{a_2}\mid U_1,\mathit{fork}_2:U_2\vdash P_2:(a_3,a_1)\ \&\ (a_3,a_2)\ \&\ \mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)}}{\Gamma,\mathit{fork}_1:U_1,\mathit{fork}_2:U_2\vdash P_1:(a_1,a_4)\ \&\ (a_3,a_1)\ \&\ (a_3,a_2)\ \&\ \mathtt{f}_{\mathit{Phil}}(a_1,a_2,a_3,a_4)}$$

where $\Gamma_1=\Gamma,x_1:\mathbf{int}$, $\Gamma_2=\Gamma,x_2:\mathbf{int}$, $U_1=\mu\alpha.?^{a_2}_{a_1}.(!^{a_1}_{a_2}\mid\alpha)$ and $U_2=\mu\alpha.?^{a_4}_{a_3}.(!^{a_3}_{a_4}\mid\alpha)$. Because $(\mathscr{L},\mathtt{L})$ has no circularity, by Theorem 2, we can conclude that $(\mathscr{D},P)$ is deadlock-free. □

Remark 1. The dining philosopher program cannot be typed in Kobayashi's type system [11]. That is because his type system assigns obligation/capability levels to each input/output *statically*. Thus only a fixed number of levels (represented as natural numbers) can be used to type a process in his type system. Since the above process can create a network consisting of an arbitrary number of dining philosophers, we need an unbounded number of levels to type the process. (Kobayashi [11] introduced a heuristic to partially mitigate the restriction on the number of levels being fixed, but the heuristic does not work here.) A variant of the dining philosopher example has been discussed in [8]. Since the variant is designed so that a finite number of levels are sufficient, it is typed both in [11] and in our new type system.

Similarly to the dining philosopher program, the system in [11] returns a false positive for the process `Fact` in Section 1, while it is deadlock-free according to our new system. We detail the arguments in the next example.

Example 5. Process `Fact` of Section 1 is written in the value passing CCS as follows.

$$\begin{array}{l}\texttt{Fact}(a_1, a_2, a_3, a_4; n : \mathbf{int}, r{:}?^{a_1}_{a_2}, s{:}!^{a_3}_{a_4}) = \\ \quad \mathbf{if}\ n = 0\ \mathbf{then}\ r?n.s!n\ \mathbf{else} \\ \qquad (\nu a_5, a_6; t{:}?^{a_5}_{a_6} \mid !^{a_6}_{a_5})(r?n.t!(m \times n) \mid \texttt{Fact}(a_5, a_6, a_3, a_4; n-1, t, s))\end{array}$$

Let $\Gamma = \texttt{Fact} : [a_1, a_2, a_3, a_4; \mathbf{int}, ?^{a_1}_{a_2}, !^{a_3}_{a_4}]$ and P be the body of the definition above. Then we have $\Gamma, n : \mathbf{int}, r{:}?^{a_1}_{a_2}, s{:}!^{a_3}_{a_4} \vdash P : \mathtt{L}$ for $\mathtt{L} = (a_2, a_3) + (\nu a_5, a_6)((a_2, a_6) \,\&\, \mathtt{f}_{\mathtt{Fact}}(a_5, a_6, a_3, a_4))$. Thus, we have: $\Gamma \vdash (\mathscr{D}, P'){:}(\mathscr{L}, \mathtt{L}')$ for:

$$\begin{array}{l}P' = (\nu a_1, a_2; r{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu a_3, a_4; s{:}?^{a_4}_{a_3} \mid !^{a_3}_{a_4})(r!1 \mid \texttt{Fact}(a_1, a_2, a_3, a_4; m, r, s) \mid s?x.\mathbf{0}) \\ \mathscr{L} = \{\mathtt{f}_{\mathtt{Fact}}(a_1, a_2, a_3, a_4) = \mathtt{L}\} \\ \mathtt{L}' = (\nu a_1, a_2, a_3, a_4)(\mathbb{0} \,\&\, \mathtt{f}_{\mathtt{Fact}}(a_1, a_2, a_3, a_4) \,\&\, \mathbb{0})\end{array}$$

where m is an integer constant. Since $(\mathscr{L}, \mathtt{L}')$ does not have a circularity, we can conclude that $(\mathscr{D}, P')$ is deadlock-free.

Type Inference. An *untyped* value-passing CCS program is a program where restrictions are $(\nu x)P$, process invocations are $A(\widetilde{e})$ and process definitions are $A(\widetilde{x}) = P$. Given an untyped value-passing CCS program $(\mathscr{D}, P)$, with $var(P) = \emptyset$, there is an inference algorithm to decide whether there exists a program $(\mathscr{D}', P')$ that coincides with $(\mathscr{D}, P)$, except for the type annotations, and such that $\Gamma \vdash (\mathscr{D}', P') : (\mathscr{L}, \mathtt{L})$. The algorithm is almost the same as that of the type system in [10] and, therefore, we do not re-describe it here. The only extra work compared with the previous algorithm is the lam program extraction, which is done using the rules in Figure 2. Finally, it suffices to analyze the extracted lams by using the fixpoint technique in Section 4.

Synchronous Value Passing CCS and pi Calculus. The type system above can be easily extended to the pi-calculus, where channel names can be passed around through other channels. To that end, we extend the syntax of types as follows.

$$\mathtt{T} ::= \mathbf{int} \mid \mathtt{ch}(\mathtt{T}, U).$$

The type $\mathtt{ch}(\mathtt{T}, U)$ describes a channel that is used according to the usage U, and $\mathtt{T}$ is the type of values passed along the channel. Only a slight change of the typing rules is sufficient, as summarized below.

(T-Out')
$$\frac{\Gamma \vdash e : \mathtt{T}}{\Gamma, x : \mathrm{ch}(\mathtt{T}, !^{a_1}_{a_2}) \vdash x!e : \&_{a \in ob(\Gamma)}(a_2, a)}$$

(T-In')
$$\frac{\Gamma, x : \mathrm{ch}(\mathtt{T}, U), y : \mathtt{T} \vdash P : \mathtt{L}}{\Gamma, x : \mathrm{ch}(\mathtt{T}, ?^{a_1}_{a_2}.U) \vdash x?y.P : \mathtt{L} \,\&\, (\&_{a \in ob(\Gamma)}(a_2, a))}$$

In particular, (T-Out') introduces dependencies between an output channel and the values sent along the channel. We notice that, in case of *synchronous value passing CCS* (as well as pi-calculus), where messages have continuations, rule (T-Out') also introduces dependency pairs between the capability of the channel and the obligations in the continuation.

7 Related Work and Conclusions

In this paper we have designed a new deadlock detection technique for the value-passing CCS (and for the pi-calculus) that enables the analysis of networks with arbitrary numbers of nodes. Our technique relies on a decidability result of a basic model featuring recursion and fresh name generation: the *lam programs*. This model has been introduced and studied in [5,6] for detecting deadlock of an object-oriented programming language [7], but the decidability was known only for a subset of lams where only linear recursion is allowed [6], and only approximate algorithms have been given for the full lam model.

The application of the lam model to deadlock-freedom of the value-passing CCS (and pi-calculus) is also new, and the resulting deadlock-freedom analysis significantly improves the previous deadlock-freedom analysis [11], as demonstrated through the dining philosopher example. In particular, Kobayashi's type system provides a mechanism for dealing with a limited form of unbounded dependency chains, but the mechanism is rather ad hoc and fragile with respect to a syntactic change. For example, while

```
Fib(n,r) =  if n<2 then r?n else new s in new t in
                (Fib!(n-1,s) | s?x.(Fib!(n-2,t)|t?y.r!(x+y))
```

is typable, the variation obtained by swapping new s in and new t in is untypable. Neither Fact nor the dining philosopher example are typable in [11]. More recently, in [17], Padovani has introduced another type system for deadlock-freedom, which has a better support than Kobayashi's one for reasoning about unbounded dependency chains, by using a form of polymorphism on levels. However, since the levels in his type system are also integers, neither the Fact example nor the dining philosopher example are typable. In addition, Padovani's type system cannot deal with non-linear channels, like the fork channels in the dining philosopher example. That said, our type system does not subsume Padovani's one, as our system does not support recursive types.

Like other type-based analyses, our method cannot reason about value-dependent behaviors. For example, consider the following process:

$$(\textbf{if } b \textbf{ then } x?z.y!z \textbf{ else } y!1 \mid x?z.) \mid (\textbf{if } b \textbf{ then } x!1 \mid y?z. \textbf{ else } y?z.x!z).$$

It is deadlock-free, but our type system would extract the lam expression: $((a_x, a_y) + \mathbb{0}) \,\&\, (\mathbb{0} + (a_y, a_x))$ (where a_x and a_y are the capability levels of the inputs on x and y respectively), detecting a (false) circular dependency.

The integration of `TyPiCal` with the deadlock detection technique of this paper is left for future work. We expect that we can extend our analysis to cover lock-freedom [8,17], too. To that end, we can require that a lam is not only circularity-free but is also *well founded*, and/or combine the deadlock-freedom analysis with the termination analysis, following the technique in [14].

Acknowledgments. This work was partially supported by JSPS Kakenhi 23220001 and by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. TOPLAS 28 (2006)
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program.: preventing data races and deadlocks. In: OOPSLA, pp. 211–230. ACM (2002)
3. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (2002)
4. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349. ACM (2003)
5. Giachino, E., Laneve, C.: A beginner's guide to the dead*l*ock *a*nalysis *m*odel. In: Palamidessi, C., Ryan, M.D. (eds.) TGC 2012. LNCS, vol. 8191, pp. 49–63. Springer, Heidelberg (2013)
6. Giachino, E., Laneve, C.: Deadlock detection in linear recursive programs. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 26–64. Springer, Heidelberg (2014)
7. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in ABS. In: Software and System Modeling (to appear, 2014)
8. Kobayashi, N.: A type system for lock-free processes. Information and Computation 177, 122–159 (2002)
9. Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K. (ed.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003)
10. Kobayashi, N.: Type-based information flow analysis for the pi-calculus. Acta Informatica 42(4-5), 291–347 (2005)
11. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
12. Kobayashi, N.: TyPiCal: Type-based static analyzer for the Pi-Calculus (2007), `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`
13. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems 21(5), 914–947 (1999)
14. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM TOPLAS 32(5) (2010)
15. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
16. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. and Comput. 100, 41–77 (1992)
17. Padovani, L.: Deadlock and Lock Freedom in the Linear π-Calculus. In: CSL-LICS 2014 (2014)
18. Suenaga, K.: Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 155–170. Springer, Heidelberg (2008)
19. Vasconcelos, V.T., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: PLACES. EPTCS, vol. 17, pp. 95–109 (2009)