Types for Deadlock-Free Higher-Order Programs

Luca Padovani(✉) and Luca Novara

Dipartimento di Informatica, Università di Torino, Torino, Italy
luca.padovani@di.unito.it

Abstract. Type systems for communicating processes are typically studied using abstract models – *e.g.*, *process algebras* – that distill the communication behavior of programs but overlook their structure in terms of functions, methods, objects, modules. It is not always obvious how to apply these type systems to structured programming languages. In this work we port a recently developed type system that ensures *deadlock freedom* in the π-calculus to a higher-order language.

1 Introduction

In this article we develop a type system that guarantees well-typed programs that communicate over channels to be free from deadlocks. Type systems ensuring this property already exist [7,8,10], but they all use the π-calculus as the reference language. This choice overlooks some aspects of concrete programming languages, like the fact that programs are structured into compartmentalized blocks (*e.g.*, functions) within which only the local structure of the program (the body of a function) is visible to the type system, and little if anything is know about the exterior of the block (the callers of the function). The structure of programs may hinder some kinds of analysis: for example, the type systems in [7,8,10] enforce an ordering of communication events and to do so they take advantage of the nature of π-calculus processes, where programs are flat sequences of communication actions. How do we reason on such ordering when the execution order is dictated by the reduction strategy of the language rather than by the syntax of programs, or when events occur within a function, and nothing is known about the events that are supposed to occur after the function terminates? We answer these questions by porting the type system in [10] to a higher-order functional language.

To illustrate the key ideas of the approach, let us consider the program

$$\langle \texttt{send}\ a\ (\texttt{recv}\ b)\rangle \mid \langle \texttt{send}\ b\ (\texttt{recv}\ a)\rangle \tag{1.1}$$

consisting of two parallel threads. The thread on the left is trying to send the message received from channel b on channel a; the thread on the right is trying to do the opposite. The communications on a and b are mutually dependent, and the program is a deadlock. The basic idea used in [10] and derived from [7,8] for detecting deadlocks is to assign each channel a number – which we call *level* – and to verify that channels are used in order according to their levels. In (1.1) this mechanism requires b to have smaller level than a in the leftmost thread, and a to have a smaller level than b in the rightmost thread. No level assignment can simultaneously satisfy both constraints. In order to perform these checks with a type system, the first step is to attach levels to

S. Graf and M. Viswanathan (Eds.): FORTE 2015, LNCS 9039, pp. 3–18, 2015.
DOI: 10.1007/978-3-319-19195-9_1

channel types. We therefore assign the types $![\mathsf{int}]^m$ and $?[\mathsf{int}]^n$ respectively to a and b in the leftmost thread of (1.1), and $?[\mathsf{int}]^m$ and $![\mathsf{int}]^n$ to the same channels in the rightmost thread of (1.1). Crucially, distinct occurrences of the same channel have types with opposite polarities (input ? and output !) and equal level. We can also think of the assignments $\mathtt{send} : \forall\iota.![\mathsf{int}]^\iota \to \mathsf{int} \to \mathsf{unit}$ and $\mathtt{recv} : \forall\iota.?[\mathsf{int}]^\iota \to \mathsf{int}$ for the communication primitives, where we allow polymorphism on channel levels. In this case, the application $\mathtt{send}\,a\ (\mathtt{recv}\,b)$ consists of two subexpressions, the partial application $\mathtt{send}\,a$ having type $\mathsf{int} \to \mathsf{unit}$ and its argument $\mathtt{recv}\,b$ having type int. Neither of these types hints at the I/O operations performed in these expressions, let alone at the levels of the channels involved. To recover this information we pair types with *effects* [1]: the effect of an expression is an abstract description of the operations performed during its evaluation. In our case, we take as effect the level of channels used for I/O operations, or $\bot$ in the case of pure expressions that perform no I/O. So, the judgment

$$b : ?[\mathsf{int}]^n \vdash \mathtt{recv}\,b : \mathsf{int} \,\&\, n$$

states that $\mathtt{recv}\,b$ is an expression of type int whose evaluation performs an I/O operation on a channel with level n. As usual, function types are decorated with a *latent effect* saying what happens when the function is applied to its argument. So,

$$a : ![\mathsf{int}]^m \vdash \mathtt{send}\,a : \mathsf{int} \to^m \mathsf{unit} \,\&\, \bot$$

states that $\mathtt{send}\,a$ is a function that, applied to an argument of type int, produces a result of type unit and, in doing so, performs an I/O operation on a channel with level m. By itself, $\mathtt{send}\,a$ is a pure expression whose evaluation performs no I/O operations, hence the effect $\bot$. Effects help us detecting dangerous expressions: in a *call-by-value* language an application $e_1 e_2$ evaluates e_1 first, then e_2, and finally the body of the function resulting from e_1. Therefore, the channels used in e_1 must have smaller level than those occurring in e_2 and the channels used in e_2 must have smaller level than those occurring in the body of e_1. In the specific case of $\mathtt{send}\,a\ (\mathtt{recv}\,b)$ we have $\bot < n$ for the first condition, which is trivially satisfied, and $n < m$ for the second one. Since the same reasoning on $\mathtt{send}\,b\ (\mathtt{recv}\,a)$ also requires the symmetric condition ($m < n$), we detect that the parallel composition of the two threads in (1.1) is ill typed, as desired.

It turns out that the information given by latent effects in function types is not sufficient for spotting some deadlocks. To see why, consider the function

$$f \stackrel{\text{def}}{=} \lambda x.(\mathtt{send}\,a\,x;\ \mathtt{send}\,b\,x)$$

which sends its argument x on both a and b and where ; denotes sequential composition. The level of a (say m) should be smaller than the level of b (say n), for a is used before b (we assume that communication is synchronous and that $\mathtt{send}$ is a potentially blocking operation). The question is, what is the latent effect that decorates the type of f, of the form $\mathsf{int} \to^h \mathsf{unit}$? Consider the two obvious possibilities: if we take $h = m$, then

$$\langle \mathtt{recv}\,a \rangle \mid \langle f\ 3;\ \mathtt{recv}\,b \rangle \tag{1.2}$$

is well typed because the effect m of $f\ 3$ is smaller than the level of b in $\mathtt{recv}\,b$, which agrees with the fact that $f\ 3$ is evaluated *before* $\mathtt{recv}\,b$; if we take $h = n$, then

$$\langle \mathtt{recv}\,a;\ f\ 3 \rangle \mid \langle \mathtt{recv}\,b \rangle \tag{1.3}$$

is well typed for similar reasons. This is unfortunate because both (1.3) and (1.2) reduce to a deadlock. To flag both of them as ill typed, we must refine the type of f to $\mathtt{int} \rightarrow^{m,n} \mathtt{unit}$ where we distinguish the smallest level of the channels that *occur* in the body of f (that is m) from the greatest level of the channels that *are used* by f when f is applied to an argument (that is n). The first annotation gives information on the channels in the function's closure, while the second annotation is the function's latent effect, as before. So (1.2) is ill typed because the effect of $f\ 3$ is the same as the level of b in $\mathtt{recv}\ b$ and (1.3) is ill typed because the effect of $\mathtt{recv}\ a$ is the same as the level of f in $f\ 3$.

In the following, we define a core multithreaded functional language with communication primitives (Section 2), we present a basic type and effect system, extend it to address recursive programs, and state its properties (Section 3). Finally, we briefly discuss closely related work and a few extensions (Section 4). *Proofs and additional material can be found in long version of the paper, on the first author's home page.*

2 Language Syntax and Semantics

In defining our language, we assume a synchronous communication model based on linear channels. This assumption limits the range of systems that we can model. However, asynchronous and structured communications can be encoded using linear channels: this has been shown to be the case for binary sessions [5] and for multiparty sessions to a large extent [10, technical report].

We use a countable set of *variables* x, y, $\dots$, a countable set of *channels* a, b, $\dots$, and a set of constants $\mathtt{k}$. *Names* u, $\dots$ are either variables or channels. We consider a language of *expressions* and *processes* as defined below:

$$e \ ::= \ \mathtt{k} \ \big|\ u \ \big|\ \lambda x.e \ \big|\ ee \qquad\qquad P,Q \ ::= \ \langle e \rangle \ \big|\ (\nu a)P \ \big|\ P \mid Q$$

Expressions comprise constants $\mathtt{k}$, names u, abstractions $\lambda x.e$, and applications $e_1 e_2$. We write _ for unused/fresh variables. Constants include the unitary value $()$, the integer numbers m, n, $\dots$, as well as the primitives $\mathtt{fix}$, $\mathtt{fork}$, $\mathtt{new}$, $\mathtt{send}$, $\mathtt{recv}$ whose semantics will be explained shortly. Processes are either threads $\langle e \rangle$, or the restriction $(\nu a)P$ of a channel a with scope P, or the parallel composition $P \mid Q$ of processes.

The notions of free and bound names are as expected, given that the only binders are λ's and ν's. We identify terms modulo renaming of bound names and we write $\mathsf{fn}(e)$ (respectively, $\mathsf{fn}(P)$) for the set of names occurring free in e (respectively, in P).

The reduction semantics of the language is given by two relations, one for expressions, another for processes. We adopt a *call-by-value* reduction strategy, for which we need to define *reduction contexts* $\mathscr{E}$, $\dots$ and *values* v, w, $\dots$ respectively as:

$$\mathscr{E} \ ::= \ [\,] \ \big|\ \mathscr{E}e \ \big|\ \mathsf{v}\mathscr{E} \qquad\qquad \mathsf{v},\mathsf{w} \ ::= \ \mathtt{k} \ \big|\ a \ \big|\ \lambda x.e \ \big|\ \mathtt{send}\,\mathsf{v}$$

The reduction relation $\longrightarrow$ for expressions is defined by standard rules

$$(\lambda x.e)\mathsf{v} \longrightarrow e\{\mathsf{v}/x\} \qquad\qquad \mathtt{fix}\,\lambda x.e \longrightarrow e\{\mathtt{fix}\,\lambda x.e/x\}$$

and closed under reduction contexts. As usual, $e\{e'/x\}$ denotes the capture-avoiding substitution of e' for the free occurrences of x in e.

Table 1. Reduction semantics of expressions and processes

$$\langle \mathscr{E}[\texttt{send}\ a\ \mathsf{v}]\rangle \mid \langle \mathscr{E}'[\texttt{recv}\ a]\rangle \xrightarrow{a} \langle \mathscr{E}[\texttt{()}]\rangle \mid \langle \mathscr{E}'[\mathsf{v}]\rangle \qquad \langle \mathscr{E}[\texttt{fork}\ \mathsf{v}]\rangle \xrightarrow{\tau} \langle \mathscr{E}[\texttt{()}]\rangle \mid \langle \mathsf{v}\texttt{()}\rangle$$

$$\frac{}{\langle \mathscr{E}[\texttt{new()}]\rangle \xrightarrow{\tau} (\nu a)\langle \mathscr{E}[a]\rangle}\ a \notin \mathsf{fn}(\mathscr{E}) \qquad \frac{e \longrightarrow e'}{\langle e\rangle \xrightarrow{\tau} \langle e'\rangle}$$

$$\frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \frac{P \xrightarrow{\ell} Q}{(\nu a)P \xrightarrow{\ell} (\nu a)Q}\ \ell \neq a \qquad \frac{P \xrightarrow{a} Q}{(\nu a)P \xrightarrow{\tau} Q} \qquad \frac{P \equiv\xrightarrow{\ell}\equiv Q}{P \xrightarrow{\ell} Q}$$

The reduction relation of processes (Table 1) has *labels* $\ell, \dots$ that are either a channel name a, signalling that a communication has occurred on a, or the special symbol τ denoting any other reduction. There are four base reductions for processes: a communication occurs between two threads when one is willing to send a message v on a channel a and the other is waiting for a message from the same channel; a thread that contains a subexpression `fork` v spawns a new thread that evaluates v`()`; a thread that contains a subexpression `new()` creates a new channel; the reduction of an expression causes a corresponding τ-labeled reduction of the thread in which it occurs. Reduction for processes is then closed under parallel compositions, restrictions, and structural congruence. The restriction of a disappears as soon as a communication on a occurs: in our model channels are *linear* and can be used for one communication only; structured forms of communication can be encoded on top of this simple model (see Example 2 and [5]). Structural congruence is defined by the standard rules rearranging parallel compositions and channel restrictions, where $\langle\texttt{()}\rangle$ plays the role of the inert process.

We conclude this section with two programs written using a slightly richer language equipped with `let` bindings, conditionals, and a few additional operators. All these constructs either have well-known encodings or can be easily accommodated.

Example 1 (parallel Fibonacci function). The `fibo` function below computes the n-th number in the Fibonacci sequence and sends the result on a channel `c`:

```
fix λfibo.λn.λc.if n ≤ 1 then send c n
                else let a = new() and b = new() in
                     (fork λ_.fibo (n - 1) a);
                     (fork λ_.fibo (n - 2) b);
                     send c (recv a + recv b)
```

The fresh channels `a` and `b` are used to collect the results from the recursive, parallel invocations of `fibo`. Note that expressions are intertwined with I/O operations. It is relevant to ask whether this version of `fibo` is deadlock free, namely if it is able to reduce until a result is computed without blocking indefinitely on an I/O operation. ■

Example 2 (signal pipe). In this example we implement a function `pipe` that forwards signals received from an input stream `x` to an output stream `y`:

```
let cont = λx.let c = new() in (fork λ_.send x c); c in
let pipe = fix λpipe.λx.λy.pipe (recv x) (cont y)
```

Note that this pipe is only capable of forwarding handshaking signals. A more interesting pipe transmitting actual data can be realized by considering data types such as records and sums [5]. The simplified realization we consider here suffices to illustrate a relevant family of recursive functions that interleave actions on different channels.

Since linear channels are consumed after communication, each signal includes a *continuation channel* on which the subsequent signals in the stream will be sent/received. In particular, `cont x` sends a fresh continuation `c` on `x` and returns `c`, so that `c` can be used for subsequent communications, while `pipe x y` sends a fresh continuation on `y` after it has received a continuation from `x`, and then repeats this behavior on the continuations. The program below connects two pipes:

```
3   let a = new() and b = new() in
4     (fork λ_.pipe a b); (fork λ_.pipe b (cont a))
```

Even if the two pipes realize a cyclic network, we will see in Section 3 that this program is well typed and therefore deadlock free. Forgetting `cont` on line 4 or not forking the `send` on line 1, however, produces a deadlock. ■

3 Type and Effect System

We present the features of the type system gradually, in three steps: we start with a monomorphic system (Section 3.1), then we introduce level polymorphism required by Examples 1 and 2 (Section 3.2), and finally recursive types required by Example 2 (Section 3.3). We end the section studying the properties of the type system (Section 3.4).

3.1 Core Types

Let $\mathbb{L} \overset{\text{def}}{=} \mathbb{Z} \cup \{\bot, \top\}$ be the set of *channel levels* ordered in the obvious way ($\bot < n < \top$ for every $n \in \mathbb{Z}$); we use ρ, σ, ... to range over $\mathbb{L}$ and we write $\rho \sqcap \sigma$ (respectively, $\rho \sqcup \sigma$) for the *minimum* (respectively, the *maximum*) of ρ and σ. *Polarities* p, q, ... are non-empty subsets of $\{?, !\}$; we abbreviate $\{?\}$ and $\{!\}$ with ? and ! respectively, and $\{?, !\}$ with #. *Types* t, s, ... are defined by

$$t, s \;\; ::= \;\; \mathsf{B} \;\big|\; p[t]^n \;\big|\; t \rightarrow^{\rho,\sigma} s$$

where *basic types* B, ... include unit and int. The type $p[t]^n$ denotes a channel with polarity p and level n. The polarity describes the operations allowed on the channel: ? means input, ! means output, and # means both input and output. Channels are linear resources: they can be used once according to each element in their polarity. The type $t \rightarrow^{\rho,\sigma} s$ denotes a function with domain t and range s. The function has level ρ (its closure contains channels with level ρ or greater) and, when applied, it uses channels with level σ or smaller. If $\rho = \top$, the function has no channels in its closure; if $\sigma = \bot$, the function uses no channels when applied. We write $\rightarrow$ as an abbreviation for $\rightarrow^{\top,\bot}$, so $\rightarrow$ denotes pure functions not containing and not using any channel.

Recall from Section 1 that levels are meant to impose an order on the use of channels: roughly, the lower the level of a channel, the sooner the channel must be used. We extend the notion of level from channel types to arbitrary types: basic types have level $\top$ because there is no need to use them as far as deadlock freedom is concerned; the level of functions is written in their type. Formally, the level of t, written $|t|$, is defined as:

$$|\mathsf{B}| \stackrel{\text{def}}{=} \top \qquad |p[t]^n| \stackrel{\text{def}}{=} n \qquad |t \rightarrow^{\rho,\sigma} s| \stackrel{\text{def}}{=} \rho \tag{3.1}$$

Levels can be used to distinguish *linear types*, denoting values (such as channels) that *must* be used to guarantee deadlock freedom, from *unlimited types*, denoting values that have no effect on deadlock freedom and *may* be disregarded. We say that t is *linear* if $|t| \in \mathbb{Z}$; we say that t is *unlimited*, written $\mathsf{un}(t)$, if $|t| = \top$.

Below are the type schemes of the constants that we consider. Some constants have many types (constraints are on the right); we write $\mathsf{types}(\mathtt{k})$ for the *set of* types of $\mathtt{k}$.

$() : \mathsf{unit}$	$\mathtt{fix} : (t \rightarrow t) \rightarrow t$	$\mathtt{new} : \mathsf{unit} \rightarrow \#[t]^n$	$n < \|t\|$
$n : \mathsf{int}$	$\mathtt{fork} : (\mathsf{unit} \rightarrow^{\rho,\sigma} \mathsf{unit}) \rightarrow \mathsf{unit}$	$\mathtt{recv} : ?[t]^n \rightarrow^{\top,n} t$	$n < \|t\|$
		$\mathtt{send} : ![t]^n \rightarrow t \rightarrow^{n,n} \mathsf{unit}$	$n < \|t\|$

The type of $()$, of the numbers, and of `fix` are ordinary. The primitive `new` creates a fresh channel with the full set # of polarities and arbitrary level n. The primitive `recv` takes a channel of type $?[t]^n$, blocks until a message is received, and returns the message. The primitive itself contains no free channels in its closure (hence the level $\top$) because the only channel it manipulates is its argument. The latent effect is the level of the channel, as expected. The primitive `send` takes a channel of type $![t]^n$, a message of type t, and sends the message on the channel. Note that the partial application `send` a is a function whose level and latent effect are both the level of a. Note also that in `new`, `recv`, and `send` the level of the message must be greater than the level of the channel: since levels are used to enforce an order on the use of channels, this condition follows from the observation that a message cannot be used until *after* it has been received, namely after the channel on which it travels has been used. Finally, `fork` accepts a thunk with arbitrary level ρ and latent effect σ and spawns the thunk into an independent thread (see Table 1). Note that `fork` is a pure function with no latent effect, regardless of the level and latent effect of the thunk. This phenomenon is called *effect masking* [1], whereby the effect of evaluating an expression becomes unobservable: in our case, `fork` discharges effects because the thunk runs in parallel with the code executing the `fork`.

We now turn to the typing rules. A *type environment* Γ is a finite map $u_1 : t_1, \dots, u_n : t_n$ from names to types. We write $\emptyset$ for the empty type environment, $\mathsf{dom}(\Gamma)$ for the domain of Γ, and $\Gamma(u)$ for the type associated with u in Γ; we write Γ_1, Γ_2 for the union of Γ_1 and Γ_2 when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We also need a more flexible way of combining type environments. In particular, we make sure that every channel is used linearly by distributing different polarities of a channel to different parts of the program. To this aim, following [9], we define a partial *combination* operator $+$ between types:

$$\begin{aligned} t + t &\stackrel{\text{def}}{=} t && \text{if } \mathsf{un}(t) \\ p[t]^n + q[t]^n &\stackrel{\text{def}}{=} (p \cup q)[t]^n && \text{if } p \cap q = \emptyset \end{aligned} \tag{3.2}$$

that we extend to type environments, thus:

$$\begin{aligned} \Gamma + \Gamma' &\stackrel{\text{def}}{=} \Gamma, \Gamma' && \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset \\ (\Gamma, u : t) + (\Gamma', u : s) &\stackrel{\text{def}}{=} (\Gamma + \Gamma'), u : t + s \end{aligned} \tag{3.3}$$

For example, we have $(x : \mathsf{int}, a : ![\mathsf{int}]^n) + (a : ?[\mathsf{int}]^n) = x : \mathsf{int}, a : \#[\mathsf{int}]^n$, so we might have some part of the program that (possibly) uses a variable x of type int along with channel a for sending an integer and another part of the program that uses the same channel a but this time for receiving an integer. The first part of the program would be typed in the environment $x : \mathsf{int}, a : ![\mathsf{int}]^n$ and the second one in the environment $a : ?[\mathsf{int}]^n$. Overall, the two parts would be typed in the environment $x : \mathsf{int}, a : \#[\mathsf{int}]^n$ indicating that a is used for both sending *and* receiving an integer.

We extend the function $|\cdot|$ to type environments so that $|\Gamma| \overset{\text{def}}{=} \bigsqcap_{u \in \mathsf{dom}(\Gamma)} |\Gamma(u)|$ with the convention that $|\emptyset| = \top$; we write $\mathsf{un}(\Gamma)$ if $|\Gamma| = \top$.

Table 2. Core typing rules for expressions and processes

Typing of expressions

[T-NAME]
$$\frac{}{\Gamma, u : t \vdash u : t \,\&\, \bot} \ \mathsf{un}(\Gamma)$$

[T-CONST]
$$\frac{}{\Gamma \vdash \mathtt{k} : t \,\&\, \bot} \ \begin{array}{l}\mathsf{un}(\Gamma)\\ t \in \mathsf{types}(\mathtt{k})\end{array}$$

[T-FUN]
$$\frac{\Gamma, x : t \vdash e : s \,\&\, \rho}{\Gamma \vdash \lambda x.e : t \rightarrow^{|\Gamma|,\rho} s \,\&\, \bot}$$

[T-APP]
$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho,\sigma} s \,\&\, \tau_1 \qquad \Gamma_2 \vdash e_2 : t \,\&\, \tau_2}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \,\&\, \sigma \sqcup \tau_1 \sqcup \tau_2} \ \begin{array}{l}\tau_1 < |\Gamma_2|\\ \tau_2 < \rho\end{array}$$

Typing of processes

[T-THREAD]
$$\frac{\Gamma \vdash e : \mathsf{unit} \,\&\, \rho}{\Gamma \vdash \langle e \rangle}$$

[T-PAR]
$$\frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}$$

[T-NEW]
$$\frac{\Gamma, a : \#[t]^n \vdash P}{\Gamma \vdash (\nu a)P}$$

We are now ready to discuss the core typing rules, shown in Table 2. Judgments of the form $\Gamma \vdash e : t \,\&\, \rho$ denote that e is well typed in Γ, it has type t and effect ρ; judgments of the form $\Gamma \vdash P$ simply denote that P is well typed in Γ.

Axioms [T-NAME] and [T-CONST] are unremarkable: as in all substructural type systems the unused part of the type environment must be unlimited. Names and constants have no effect ($\bot$); they are evaluated expressions that do not use (but may contain) channels.

In rule [T-FUN], the effect ρ caused by evaluating the body of the function becomes the latent effect in the arrow type of the function and the function itself has no effect. The level of the function is determined by that of the environment Γ in which the function is typed. Intuitively, the names in Γ are stored in the *closure* of the function; if any of these names is a channel, then we must be sure that the function is eventually used (*i.e.*, applied) to guarantee deadlock freedom. In fact, $|\Gamma|$ gives a slightly more precise information, since it records the smallest level of all channels that occur in the body of the function. We have seen in Section 1 why this information is useful. A few examples:

- the identity function $\lambda x.x$ has type $\mathsf{int} \rightarrow^{\top,\bot} \mathsf{int}$ in any unlimited environment;
- the function $\lambda\_.a$ has type $\mathsf{unit} \rightarrow^{n,\bot} ![\mathsf{int}]^n$ in the environment $a : ![\mathsf{int}]^n$; it contains channel a with level n in its closure (whence the level n in the arrow), but it does not use a for input/output (whence the latent effect $\bot$); it is nonetheless well typed because a, which is a linear value, is returned as result;
- the function $\lambda x.\mathtt{send}\ x\ 3$ has type $![\mathsf{int}]^n \rightarrow^{\top,n} \mathsf{unit}$; it has no channels in its closure but it performs an output on the channel it receives as argument;

– the function $\lambda x.(\texttt{recv}\ a + x)$ has type $\mathsf{int} \rightarrow^{n,n} \mathsf{int}$ in the environment $a : ?[\mathsf{int}]^n$; note that neither the domain nor the codomain of the function mention any channel, so the fact that the function has a channel in its closure (and that it performs some I/O) can only be inferred from the annotations on the arrow;
– the function $\lambda x.\texttt{send}\ x\ (\texttt{recv}\ a)$ has type $![\mathsf{int}]^{n+1} \rightarrow^{n,n+1} \mathsf{unit}$ in the environment $a : ![\mathsf{int}]^n$; it contains channel a with level n in its closure and performs input/output operations on channels with level $n+1$ (or smaller) when applied.

Rule [T-APP] deals with applications $e_1 e_2$. The first thing to notice is the type environments in the premises for e_1 and e_2. Normally, these are exactly the same as the type environment used for the whole application. In our setting, however, we want to distribute polarities in such a way that each channel is used for exactly one communication. For this reason, the type environment $\Gamma_1 + \Gamma_2$ in the conclusion is the combination of the type environments in the premises. Regarding effects, τ_i is the effect caused by the evaluation of e_i. As expected, e_1 must result in a function of type $t \rightarrow^{\rho,\sigma} s$ and e_2 in a value of type t. The evaluation of e_1 and e_2 may however involve blocking I/O operations on channels, and the two side conditions make sure that no deadlock can arise. To better understand them, recall that reduction is *call-by-value* and applications $e_1 e_2$ are evaluated *sequentially from left to right*. Now, the condition $\tau_1 < |\Gamma_2|$ makes sure that any I/O operation performed during the evaluation of e_1 involves only channels whose level is smaller than that of the channels occurring free in e_2 (the free channels of e_2 must necessarily be in Γ_2). This is enough to guarantee that the functional part of the application can be fully evaluated without blocking on operations concerning channels that occur *later* in the program. In principle, this condition should be paired with the symmetric one $\tau_2 < |\Gamma_1|$ making sure that any I/O operation performed during the evaluation of the argument does not involve channels that occur in the functional part. However, when the argument is being evaluated, we know that the functional part has already been reduced a value (see the definition of reduction contexts in Section 2). Therefore, the only really critical condition to check is that no channels involved in I/O operations during the evaluation of e_2 occur in the *value* of e_1. This is expressed by the condition $\tau_2 < \rho$, where ρ is the level of the functional part. Note that, when e_1 is an abstraction, by rule [T-FUN] ρ coincides with $|\Gamma_1|$, but in general ρ may be greater than $|\Gamma_1|$, so the condition $\tau_2 < \rho$ gives better accuracy. The effect of the whole application $e_1 e_2$ is, as expected, the combination of the effects of evaluating e_1, e_2, and the latent effect of the function being applied. In our case the "combination" is the greatest level of any channel involved in the application. Below are some examples:

– $(\lambda x.x)\ a$ is well typed, because both $\lambda x.x$ and a are pure expressions whose effect is $\bot$, hence the two side conditions of [T-APP] are trivially satisfied;
– $(\lambda x.x)\ (\texttt{recv}\ a)$ is well typed in the environment $a : ?[\mathsf{int}]^n$: the effect of $\texttt{recv}\ a$ is n (the level of a) which is smaller than the level $\top$ of the function;
– $\texttt{send}\ a\ (\texttt{recv}\ a)$ is ill typed in the environment $a : \#[\mathsf{int}]^n$ because the effect of evaluating $\texttt{recv}\ a$, namely n, is the same as the level of $\texttt{send}\ a$;
– $(\texttt{recv}\ a)\ (\texttt{recv}\ b)$ is well typed in the environment $a : ?[\mathsf{int} \rightarrow \mathsf{int}]^0, b : ?[\mathsf{int}]^1$. The effect of the argument is 1, which is *not* smaller than the level of the environment $a : ?[\mathsf{int} \rightarrow \mathsf{int}]^0$ used for typing the function. However, 1 is smaller than $\top$, which

is the level of the *result* of the evaluation of the functional part of the application. This application would be illegal had we used the side condition $\tau_2 < |\Gamma_1|$ in [T-APP].

The typing rules for processes are standard: [T-PAR] splits contexts for typing the processes in parallel, [T-NEW] introduces a new channel in the environment, and [T-THREAD] types threads. The effect of threads is ignored: effects are used to prevent circular dependencies between channels used within the *sequential* parts of the program (*i.e.*, within expressions); circular dependencies that arise between *parallel* threads are indirectly detected by the fact that each occurrence of a channel is typed with the same level (see the discussion of (1.1) in Section 1).

3.2 Level Polymorphism

Looking back at Example 1, we notice that `fibo n c` may generate two recursive calls with two corresponding fresh channels `a` and `b`. Since the `send` operation on `c` is blocked by `recv` operations on `a` and `b` (line 5), the level of `a` and `b` must be smaller than that of `c`. Also, since expressions are evaluated left-to-right and `recv a + recv b` is syntactic sugar for the application `(+) (recv a) (recv b)`, the level of `a` must be smaller than that of `b`. Thus, to declare `fibo` well typed, we must allow different occurrences of `fibo` to be applied to channels with different levels. Even more critically, this form of level polymorphism of `fibo` is necessary *within* the definition of `fibo` itself, so it is an instance of *polymorphic recursion* [1].

The core typing rules in Table 2 do not support level polymorphism. Following the previous discussion on `fibo`, the idea is to realize level polymorphism by *shifting* levels in types. We define level shifting as a type operator $\Uparrow^n$, thus:

$$\Uparrow^n \mathsf{B} \stackrel{\text{def}}{=} \mathsf{B} \qquad \Uparrow^n p[t]^m \stackrel{\text{def}}{=} p[\Uparrow^n t]^{n+m} \qquad \Uparrow^n (t \rightarrow^{\rho,\sigma} s) \stackrel{\text{def}}{=} \Uparrow^n t \rightarrow^{n+\rho,n+\sigma} \Uparrow^n s \tag{3.4}$$

where $+$ is extended from integers to levels so that $n+\top=\top$ and $n+\bot=\bot$. The effect of $\Uparrow^n t$ is to shift all the finite level annotations in t by n, leaving $\top$ and $\bot$ unchanged.

Now, we have to understand in which cases we can use a value of type $\Uparrow^n t$ where one of type t is expected. More specifically, when a value of type $\Uparrow^n t$ can be passed to a function expecting an argument of type t. This is possible if the function has level $\top$. We express this form of level polymorphism with an additional typing rule for applications:

[T-APP-POLY]
$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\top,\sigma} s \,\&\, \tau_1 \qquad \Gamma_2 \vdash e_2 : \Uparrow^n t \,\&\, \tau_2}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : \Uparrow^n s \,\&\, (n+\sigma) \sqcup \tau_1 \sqcup \tau_2} \quad \begin{array}{l}\tau_1 < |\Gamma_2| \\ \tau_2 < \top\end{array}$$

This rule admits an arbitrary mismatch n between the level the argument expected by the function and that of the argument supplied to the function. The type of the application and the latent effect are consequently shifted by the same amount n.

Soundness of [T-APP-POLY] can be intuitively explained as follows: a function with level $\top$ has no channels in its closure. Therefore, the only channels possibly manipulated by the function are those contained in the argument to which the function is applied or channels created within the function itself. Then, the fact that the argument has level

$n+k$ rather than level k is completely irrelevant. Conversely, if the function has channels in its closure, then the absolute level of the argument might have to satisfy specific ordering constraints with respect to these channels (recall the two side conditions in [T-APP]). Since level polymorphism is a key distinguishing feature of our type system, and one that accounts for much of its expressiveness, we elaborate more on this intuition using an example. Consider the term

$$\mathtt{fwd} \overset{\text{def}}{=} \lambda x.\lambda y.\mathtt{send}\; y\; (\mathtt{recv}\; x)$$

which forwards on y the message received from x. The derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{y : ![\mathsf{int}]^1 \vdash \mathtt{send}\; y : \mathsf{int} \to^{1,1} \mathsf{unit} \,\&\, \bot}\,[\text{T-APP}] \qquad \dfrac{\vdots}{x : ?[\mathsf{int}]^0 \vdash \mathtt{recv}\; x : \mathsf{int} \,\&\, 0}\,[\text{T-APP}]}{x : ?[\mathsf{int}]^0, y : ![\mathsf{int}]^1 \vdash \mathtt{send}\; y\; (\mathtt{recv}\; x) : \mathsf{unit} \,\&\, 1}\,[\text{T-APP}]}{x : ?[\mathsf{int}]^0 \vdash \lambda y.\mathtt{send}\; y\; (\mathtt{recv}\; x) : ![\mathsf{int}]^1 \to^{0,1} \mathsf{unit} \,\&\, \bot}\,[\text{T-FUN}]}{\vdash \mathtt{fwd} : ?[\mathsf{int}]^0 \to ![\mathsf{int}]^1 \to^{0,1} \mathsf{unit} \,\&\, \bot}\,[\text{T-FUN}]$$

does *not* depend on the absolute values 0 and 1, but only on the level of x being smaller than that of y, as required by the fact that the `send` operation on y is blocked by the `recv` operation on x. Now, consider an application $\mathtt{fwd}\; a$, where a has type $?[\mathsf{int}]^2$. The mismatch between the level of x (0) and that of a (2) is not critical, because all the levels in the derivation above can be *uniformly shifted up* by 2, yielding a derivation for

$$\vdash \mathtt{fwd} : ?[\mathsf{int}]^2 \to ![\mathsf{int}]^3 \to^{2,3} \mathsf{unit} \,\&\, \bot$$

This shifting is possible because `fwd` has no free channels in its body (indeed, it is typed in the empty environment). Therefore, using [T-APP-POLY], we can derive

$$a : ?[\mathsf{int}]^2 \vdash \mathtt{fwd}\; a : ![\mathsf{int}]^3 \to^{2,3} \mathsf{unit} \,\&\, \bot$$

Note that $(\mathtt{fwd}\; a)$ is a function having level 2. This means that $(\mathtt{fwd}\; a)$ is *not* level polymorphic and can only be applied, through [T-APP], to channels with level 3. If we allowed $(\mathtt{fwd}\; a)$ to be applied to a channel with level 2 using [T-APP-POLY] we could derive

$$a : \#[\mathsf{int}]^2 \vdash \mathtt{fwd}\; a\; a : \mathsf{unit} \,\&\, 2$$

which reduces to a deadlock.

Example 3. To show that the term in Example 1 is well typed, consider the environment

$$\Gamma \overset{\text{def}}{=} \mathtt{fibo} : \mathsf{int} \to ![\mathsf{int}]^0 \to^{\top,0} \mathsf{unit}, \mathtt{n} : \mathsf{int}, \mathtt{c} : ![\mathsf{int}]^0$$

In the proof derivation for the body of `fibo`, this environment is eventually enriched with the assignments $\mathtt{a} : \#[\mathsf{int}]^{-2}$ and $\mathtt{b} : \#[\mathsf{int}]^{-1}$. Now we can derive

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{fibo\ (n\ -\ 2)} : ![\mathsf{int}]^0 \to^{\top,0} \mathsf{unit} \,\&\, \bot}\,[\text{T-APP}] \qquad \dfrac{}{\mathtt{a} : ![\mathsf{int}]^{-2} \vdash \mathtt{a} : ![\mathsf{int}]^{-2} \,\&\, \bot}\,[\text{T-NAME}]}{\Gamma, \mathtt{a} : ![\mathsf{int}]^{-2} \vdash \mathtt{fibo\ (n\ -\ 2)\ a} : \mathsf{unit} \,\&\, -2}\,[\text{T-APP-POLY}]$$

where the application `fibo (n - 2) a` is well typed despite the fact that `fibo (n - 2)` expects an argument of type $![\mathsf{int}]^0$, while `a` has type $![\mathsf{int}]^{-2}$. A similar derivation can be obtained for `fibo (n - 1) b`, and the proof derivation can now be completed. ■

3.3 Recursive Types

Looking back at Example 2, we see that in a call `pipe x y` the channel `recv x` is used in the same position as `x`. Therefore, according to [T-APP-POLY], `recv x` must have the same type as `x`, up to some shifting of its level. Similarly, channel `c` is both sent on `y` and then used in the same position as `y`, suggesting that `c` must have the same type as `y`, again up to some shifting of its level. This means that we need recursive types in order to properly describe `x` and `y`.

Instead of adding explicit syntax for recursive types, we just consider the possibly infinite trees generated by the productions for t shown earlier. In light of this broader notion of types, the inductive definition of type level (3.1) is still well founded, but type shift (3.4) must be reinterpreted coinductively, because it has to operate on possibly infinite trees. The formalities, nonetheless, are well understood.

It is folklore that, whenever infinite types are *regular* (that is, when they are made of finitely many distinct subtrees), they admit finite representations either using type variables and the familiar μ notation, or using systems of type equations [4]. Unfortunately, a careful analysis of Example 2 suggests that – at least in principle – we also need *non-regular* types. To see why, let `a` and `c` be the channels to which `(recv x)` and `(cont y)` respectively evaluate on line 2 of the example. Now:

- `x` must have smaller level than `a` since `a` is received from `x` (*cf.* the types of `recv`).
- `y` must have smaller level than `c` since `c` is sent on `y` (*cf.* the types of `send`).
- `x` must have smaller level than `y` since `x` is used in the functional part of an application in which `y` occurs in the argument (*cf.* line 2 and [T-APP-POLY]).

Overall, in order to type `pipe` in Example 2 we should assign `x` and `y` the types t^n and s^n that respectively satisfy the equations

$$t^n = ?[t^{n+2}]^n \qquad s^n = ![t^{n+3}]^{n+1} \tag{3.5}$$

Unfortunately, these equations do not admit regular types as solutions. We recover typeability of `pipe` with regular types by introducing a new type constructor

$$t \ ::= \ \cdots \ \big| \ \lceil t \rceil^n$$

that wraps types with a pending shift: intuitively $\lceil t \rceil^n$ and $\Uparrow^n t$ denote the same type, except that in $\lceil t \rceil^n$ the shift $\Uparrow^n$ on t is pending. For example, $\lceil ?[\mathsf{int}]^0 \rceil^1$ and $\lceil ?[\mathsf{int}]^2 \rceil^{-1}$ are both possible wrappings of $?[\mathsf{int}]^1$, while $\mathsf{int} \to^{0,\perp} ![\mathsf{int}]^0$ is the unwrapping of $\lceil \mathsf{int} \to^{1,\perp} ![\mathsf{int}]^1 \rceil^{-1}$. To exclude meaningless infinite types such as $\lceil\lceil\lceil \cdots \rceil^n\rceil^n\rceil^n$ we impose a *contractiveness condition* requiring every infinite branch of a type to contain infinite occurrences of channel or arrow constructors. To see why wraps help finding regular representations for otherwise non-regular types, observe that the equations

$$t^n = ?[\lceil t^n \rceil^2]^n \qquad s^n = ![\lceil t^{n+1} \rceil^2]^{n+1} \tag{3.6}$$

denote – up to pending shifts – the same types as the ones in (3.5), with the key difference that (3.6) admit regular solutions and therefore finite representations. For example, t^n could be finitely represented as a familiar-looking $\mu\alpha.?[\lceil\alpha\rceil^2]^n$ term.

We should remark that $\lceil t\rceil^n$ and $\Uparrow^n t$ are *different* types, even though the former is morally equivalent to the latter: wrapping is a type *constructor*, whereas shift is a type *operator*. Having introduced a new constructor, we must suitably extend the notions of type level (3.1) and type shift (3.4) we have defined earlier. We postulate

$$|\lceil t\rceil^n| \overset{\text{def}}{=} n+|t| \qquad\qquad \Uparrow^n\lceil t\rceil^m \overset{\text{def}}{=} \lceil\Uparrow^n t\rceil^m$$

in accordance with the fact that $\lceil\cdot\rceil^n$ denotes a pending shift by n (note that $|\cdot|$ extended to wrappings is well defined thanks to the contractiveness condition).

We also have to define introduction and elimination rules for wrappings. To this aim, we conceive two constants, `wrap` and `unwrap`, having the following type schemes:

$$\texttt{wrap} : \Uparrow^n t \to \lceil t\rceil^n \qquad \texttt{unwrap} : \lceil t\rceil^n \to \Uparrow^n t$$

We add `wrap` v to the value forms. Operationally, we want `wrap` and `unwrap` to annihilate each other. This is done by enriching reduction for expressions with the axiom

$$\texttt{unwrap}\ (\texttt{wrap}\ \mathsf{v}) \longrightarrow \mathsf{v}$$

Example 4. We suitably dress the code in Example 2 using `wrap` and `unwrap`:

```
let cont = λx.let c = new() in (fork λ_.send x (wrap c)); c in
let pipe = fix λpipe.λx.λy.pipe (unwrap (recv x)) (cont y)
```

and we are now able to find a typing derivation for it that uses regular types. In particular, we assign `cont` the type $s^n \to s^{n+2}$ and `pipe` the type $t^n \to s^n \to^{n,\top}$ `unit` where t^n and s^n are the types defined in (3.6). Note that `cont` is a pure function because its effects are masked by `fork` and that `pipe` has latent effect $\top$ since it loops performing `recv` operations on channels with increasing level. Because of the side conditions in [T-APP] and [T-APP-POLY], this means that `pipe` can only be used in tail position, which is precisely what happens above and in Example 2. ■

3.4 Properties

To formulate subject reduction, we must take into account that linear channels are *consumed* after communication (last but one reduction in Table 1). This means that when a process P communicates on some channel a, a must be removed from the type environment used for typing the residual of P. To this aim, we define a partial operation $\Gamma - \ell$ that removes ℓ from Γ, when ℓ is a channel. Formally:

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\ell} Q$, *then* $\Gamma - \ell \vdash Q$ *where* $\Gamma - \tau \overset{\text{def}}{=} \Gamma$ *and* $(\Gamma, a : \#[t]^n) - a \overset{\text{def}}{=} \Gamma$.

Note that $\Gamma - a$ is undefined if $a \notin \mathsf{dom}(\Gamma)$. This means that well-typed programs never attempt at using the same channel twice, namely that channels in well-typed programs are indeed *linear channels*. This property has important practical consequences, since it allows the efficient implementation (and deallocation) of channels [9].

Deadlock freedom means that *if* the program halts, then there must be no pending I/O operations. In our language, the only halted program without pending operations is (structurally equivalent to) $\langle\,()\,\rangle$. We can therefore define deadlock freedom thus:

Definition 1. *We say that P is* deadlock free *if $P \xrightarrow{\tau}{}^{*} Q \not\rightarrow$ implies $Q \equiv \langle\,()\,\rangle$.*

As usual, $\xrightarrow{\tau}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{\tau}$ and $Q \not\rightarrow$ means that Q is unable to reduce further. Now, every well-typed, closed process is free from deadlocks:

Theorem 2 (Soundness). *If $\emptyset \vdash P$, then P is deadlock free.*

Theorem 2 may look weaker than desirable, considering that every process P (even an ill-typed one) can be "fixed" and become part of a deadlock-free system if composed in parallel with the diverging thread $\langle \mathtt{fix}\ \lambda x.x\rangle$. It is not easy to state an interesting property of well-typed *partial programs* – programs that are well typed in uneven environments – or of *partial computations* – computations that have not reached a stable (*i.e.*, irreducible) state. One might think that well-typed programs eventually use all of their channels. This property is false in general, for two reasons. First, our type system does not ensure termination of well-typed expressions, so a thread like $\langle \mathtt{send}\ a\ (\mathtt{fix}\ \lambda x.x)\,\rangle$ never uses channel a, because the evaluation of the message diverges. Second, there are threads that continuously generate (or receive) new channels, so that the set of channels they own is never empty; this happens in Example 2. What we can prove is that, *assuming* that a well-typed program does not internally diverge, then *each* channel it owns is eventually used for a communication or is sent to the environment in a message. To formalize this property, we need a labeled transition system describing the interaction of programs with their environment. *Labels* $\pi, \ldots$ of transitions are defined by

$$\pi \ ::= \ \ell \ \big| \ a?e \ \big| \ a!\mathsf{v}$$

and the transition relation $\overset{\pi}{\longmapsto}$ extends reduction with the rules

$$\frac{a \notin \mathsf{bn}(\mathscr{C})}{\mathscr{C}[\mathtt{send}\ a\ \mathsf{v}] \overset{a!\mathsf{v}}{\longmapsto} \mathscr{C}[()]} \qquad \frac{a \notin \mathsf{bn}(\mathscr{C}) \qquad \mathsf{fn}(e) \cap \mathsf{bn}(\mathscr{C}) = \emptyset}{\mathscr{C}[\mathtt{recv}\ a] \overset{a?e}{\longmapsto} \mathscr{C}[e]}$$

where $\mathscr{C}$ ranges over *process contexts* $\mathscr{C} ::= \langle\mathscr{E}\rangle \mid (\mathscr{C} \mid P) \mid (P \mid \mathscr{C}) \mid (\nu a)\mathscr{C}$. Messages of input transitions have the form $a?e$ where e is an arbitrary expression instead of a value. This is just to allow a technically convenient formulation of Definition 2 below. We formalize the assumption concerning the absence of internal divergences as a property that we call *interactivity*. Interactivity is a property of *typed processes*, which we write as pairs $\Gamma \mathbin{⨾} P$, since the messages exchanged between a process and the environment in which it executes are not arbitrary in general.

Definition 2 (Interactivity). *Interactivity is the largest predicate on well-typed processes such that $\Gamma \mathbin{⨾} P$* interactive *implies $\Gamma \vdash P$ and:*

1. *P has no infinite reduction $P \overset{\ell_1}{\longmapsto} P_1 \overset{\ell_2}{\longmapsto} P_2 \overset{\ell_3}{\longmapsto} \cdots$, and*
2. *if $P \overset{\ell}{\longmapsto} Q$, then $\Gamma - \ell \mathbin{⨾} Q$ is interactive, and*

3. *if* $P \xmapsto{a!\mathsf{v}} Q$ *and* $\Gamma = \Gamma', a : ![t]^n$, *then* $\Gamma'' \mathbin{⨟} Q$ *is interactive for some* $\Gamma'' \subseteq \Gamma'$, *and*
4. *if* $P \xmapsto{a?x} Q$ *and* $\Gamma = \Gamma', a : ?[t]^n$, *then* $\Gamma'' \mathbin{⨟} Q\{\mathsf{v}/x\}$ *is interactive for some* v *and* $\Gamma'' \supseteq \Gamma'$ *such that* $n < |\Gamma'' \setminus \Gamma'|$.

Clause (1) says that an interactive process does not internally diverge: it will eventually halt either because it terminates or because it needs interaction with the environment in which it executes. Clause (2) states that internal reductions preserve interactivity. Clause (3) states that a process with a pending output on a channel *a must* reduce to an interactive process after the output is performed. Finally, clause (4) states that a process with a pending input on a channel *a may* reduce to an interactive process after the input of a particular message v is performed. The definition looks demanding, but many conditions are direct consequences of Theorem 1. The really new requirements besides well typedness are *convergence* of P (1) and the *existence* of v (4). It is now possible to prove that well-typed, interactive processes eventually use their channels.

Theorem 3 (Interactivity). *Let* $\Gamma \mathbin{⨟} P$ *be an interactive process such that* $a \in \mathsf{fn}(P)$. *Then* $P \xmapsto{\pi_1} P_1 \xmapsto{\pi_2} \cdots \xmapsto{\pi_n} P_n$ *for some* $\pi_1, \dots, \pi_n$ *such that* $a \notin \mathsf{fn}(P_n)$.

4 Concluding Remarks

We have demonstrated the portability of a type system for deadlock freedom of π-calculus processes [10] to a higher-order language using an *effect system* [1]. We have shown that *effect masking* and *polymorphic recursion* are key ingredients of the type system (Examples 1 and 2), and also that latent effects must be paired with one more annotation – the function level. The approach may seem to hinder program modularity, since it requires storing levels in types and levels have global scope. In this respect, level polymorphism (Section 3.2) alleviates this shortcoming of levels by granting them a relative – rather than absolute – meaning at least for non-linear functions.

Other type systems for higher-order languages with session-based communication primitives have been recently investigated [6,14,2]. In addition to safety, types are used for estimating bounds in the size of message queues [6] and for detecting memory leaks [2]. Since binary sessions can be encoded using linear channels [5], our type system can address the same family of programs considered in these works with the advantage that, in our case, well-typed programs are guaranteed to be deadlock free also in presence of session interleaving. For instance, the `pipe` function in Example 2 interleaves communications on two different channels. The type system described by Wadler [14] is interesting because it guarantees deadlock freedom without resorting to any type annotation dedicated to this purpose. In his case the syntax of (well-typed) programs prevents the modeling of cyclic network topologies, which is a necessary condition for deadlocks. However, this also means that some useful program patterns cannot be modeled. For instance, the program in Example 2 is ill typed in [14].

The type system discussed in this paper lacks compelling features. *Structured data types* (records, sums) have been omitted for lack of space; an extended technical report [13] and previous works [11,10] show that they can be added without issues. The same goes for *non-linear channels* [10], possibly with the help of dedicated `accept` and `request` primitives as in [6]. *True polymorphism* (with level and type variables) has also been studied in the technical report [13]. Its impact on the overall type system is significant, especially because level and type constraints (those appearing as side conditions in the type schemes of constants, Section 3.1) must be promoted from the metatheory to the type system. The realization of level polymorphism as type shifting that we have adopted in this paper is an interesting compromise between impact and flexibility. Our type system can also be relaxed with *subtyping*: arrow types are contravariant in the level and covariant in the latent effect, whereas channel types are invariant in the level. Invariance of channel levels can be relaxed refining levels to *pairs* of numbers as done in [7,8]. This can also improve the accuracy of the type system in some cases, as discussed in [10] and [3]. It would be interesting to investigate which of these features are actually necessary for typing concrete functional programs using threads and communication/synchronization primitives.

Type reconstruction algorithms for similar type systems have been defined [11,12]. We are confident to say that they scale to type systems with arrow types and effects.

Acknowledgments. The authors are grateful to the reviewers for their detailed comments and useful suggestions. The first author has been supported by Ateneo/CSP project SALT, ICT COST Action IC1201 BETTY, and MIUR project CINA.

References

1. Amtoft, T., Nielson, F., Nielson, H.: Type and Effect Systems: Behaviours for Concurrency. Imperial College Press (1999)
2. Bono, V., Padovani, L., Tosatto, A.: Polymorphic Types for Leak Detection in a Session-Oriented Functional Language. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE 2013. LNCS, vol. 7892, pp. 83–98. Springer, Heidelberg (2013)
3. Carbone, M., Dardha, O., Montesi, F.: Progress as compositional lock-freedom. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 49–64. Springer, Heidelberg (2014)
4. Courcelle, B.: Fundamental properties of infinite trees. Theor. Comp. Sci. 25, 95–169 (1983)
5. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: PPDP 2012, pp. 139–150. ACM (2012)
6. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. 20(1), 19–50 (2010)
7. Kobayashi, N.: A type system for lock-free processes. Inf. and Comp. 177(2), 122–159 (2002)
8. 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)
9. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. 21(5), 914–947 (1999)
10. Padovani, L.: Deadlock and Lock Freedom in the Linear π-Calculus. In: CSL-LICS 2014, pp. 72:1–72:10. ACM (2014), http://hal.archives-ouvertes.fr/hal-00932356v2/
11. Padovani, L.: Type Reconstruction for the Linear π-Calculus with Composite and Equi-Recursive Types. In: Muscholl, A. (ed.) FOSSACS 2014. LNCS, vol. 8412, pp. 88–102. Springer, Heidelberg (2014)

12. Padovani, L., Chen, T.-C., Tosatto, A.: Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi. In: Holvoet, T., Viroli, M. (eds.) COORDINATION 2015. LNCS, vol. 9037, pp. 85–100. Springer, Heidelberg (2015)
13. Padovani, L., Novara, L.: Types for Deadlock-Free Higher-Order Concurrent Programs. Technical report, Università di Torino (2014), http://hal.inria.fr/hal-00954364
14. Wadler, P.: Propositions as sessions. In: ICFP 2012, pp. 273–286. ACM (2012)