Deadlock and Lock Freedom in the Linear π-Calculus

Deadlock and lock freedom in the linear π-calculus

Luca Padovani

Dipartimento di Informatica, Università di Torino, Italy

padovani@di.unito.it

Abstract

We study two refinements of the linear π-calculus that ensure *deadlock freedom* (the absence of stable states with pending linear communications) and *lock freedom* (the eventual completion of pending linear communications). The main feature of both type systems is a new form of channel polymorphism that affects their accuracy in a significant way: they are the first of their kind that can deal with recursive processes connected by cyclic networks.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation—Parallelism and concurrency

Keywords linear π-calculus, deadlock, types, polymorphism

1. Introduction

The linear π-calculus [27] is a resource-aware model of communicating processes that distinguishes between unlimited and linear channels. While unlimited channels can be used without restrictions, linear channels are meant to be used for *exactly one* communication. This intrinsic limitation is rewarded by several benefits, including specialized behavioral equivalences for reasoning about communication optimizations, the efficient implementation of linear channels, and the fact that communications on linear channels enjoy desirable properties such as determinism and confluence. The value of these benefits is amplified given that a significant fraction of channels in several actual systems happen to be linear.

From an operational standpoint, the linear π-calculus only guarantees that well-typed processes never communicate twice on the same linear channel. In practice, one may be interested in stronger guarantees, such as *deadlock freedom* [24] – the absence of stable states with pending communications on linear channels – or *lock freedom* [22, 26] – the possibility to complete pending communications on linear channels. A paradigmatic example of deadlock is illustrated by the process

$$a?(x).b!\langle x\rangle \mid b?(y).a!\langle y\rangle \tag{1}$$

where the left subprocess forwards on b the message x received from a, and the right subprocess forwards on a the message y received from b. The process (1) is well typed when a and b are linear channels, but none of the pending communications on a and b can complete because of the mutual dependencies between corresponding inputs and outputs. An example of lock which is not a deadlock is illustrated by the process

$$c!\langle a\rangle \mid *c?(x).c!\langle x\rangle \mid a!\langle 1984\rangle \tag{2}$$

where there is a pending output operation on the rightmost occurrence of channel a, whereas the leftmost occurrence of a is repeatedly sent over the unlimited channel c. Also this process is well typed when a is a linear channel, even though the pending communication cannot be completed because a is never actually used for an input operation. Note that (2), unlike (1), reduces forever.

In this work we propose two refinements of the linear π-calculus such that well-typed processes are (dead)lock free. The techniques we put forward have been inspired by previous ideas presented in [22–24, 26], except that by narrowing the focus on the linear π-calculus we obtain type systems that are not just technically simpler but also more accurate with respect to relevant processes.

In a nutshell, we consider channel types of the form $p^{\iota}[t]^{n}_{m}$ where the *polarity* p (either input ? or output !), the type t of messages, and the *multiplicity* ι (unlimited ω or linear 1), are just like in the linear π-calculus. In addition, we annotate channel types with two numbers, a *level* n and m *tickets*. *Levels* enforce an order on the use of channels: when a process owns two or more channels at the same time, channels with lower level must be used *before* channels with higher level. In particular, an input $u?(x).P$ is well typed if u has lower level than all the channels occurring in P and an output $u!\langle v\rangle$ is well typed if u has lower level than v. This mechanism makes (1) ill typed: it is not possible to assign two levels h and k to a and b, for the structure of the process requires the simultaneous satisfiability of the two constraints $h < k$ and $k < h$. *Tickets* limit the number of travels that channels can do: $u!\langle v\rangle$ is well typed if v has at least one ticket; each time a channel is sent as a message, one ticket is removed from its type; a channel with no tickets cannot travel and must be used directly for performing a communication. This mechanism makes (2) ill typed, because a is sent on c infinitely many times and so it would ideally need infinitely many tickets.

The technique described thus far prevents deadlocks (if we just consider the constraints on levels) and locks (if we also consider the constraints on tickets). Unfortunately, it fails to type most recursive processes. For example, the usual encoding of the factorial below where, for convenience, we have annotated linear names with their level, is ill typed:

$$\begin{array}{l} *\mathit{fact}?(x, y^{h}). \\ \quad \texttt{if } x = 0 \texttt{ then } y^{h}!\langle 1\rangle \\ \quad \texttt{else } (\nu a^{k})(\mathit{fact}!\langle x-1, a^{k}\rangle \mid a^{k}?(z).y^{h}!\langle x \times z\rangle) \end{array} \tag{3}$$

Since the newly created channel a is used in the same position as y in the recursive invocation of fact, we are led into thinking that a and y should have the same type hence the same level $h = k$. This clashes with the input on a that blocks the output on y, requiring

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM 978-1-4503-2886-9…$15.00.
http://dx.doi.org/10.1145/2603088.2603116

$k < h$. We see a symmetric phenomenon in

$$*stream?(x, y^h).(\nu a^k)(y^h!\langle x, a^k\rangle \mid stream!\langle x+1, a^k\rangle) \quad (4)$$

which generates a stream of integers. Here too a and y are used in the same position and should have the same level $h = k$, but the output $y!\langle x, a\rangle$ also requires $h < k$.

We need some way to overcome these difficulties. Let us digress for a moment from *fact* and *stream* and consider the processes

$$F_1 \stackrel{\text{def}}{=} c?(x^h).y^k!\langle x^h\rangle \qquad \text{and} \qquad F_2 \stackrel{\text{def}}{=} c?(x^h, y^k).y^k!\langle x^h\rangle$$

both of which forward a linear channel x received from c to the linear channel y. In both cases it must be $k < h$, but there is a key difference between F_1 and F_2: the level of x in F_1 has a fixed lower bound $k+1$ because y is free, while the level of x in F_2 is arbitrary, provided that $k < h$, because y is bound. Rephrased in technical jargon, c is *monomorphic* in F_1 and *polymorphic* in F_2 with respect to levels. The fact that a channel like c is monomorphic or polymorphic depends on the presence or absence of free linear channels in the continuation that follows the input on c. Normally this information is not inferrable solely from the type of c, but it turns out that we can easily approximate it in the linear π-calculus, where replicated processes cannot have free linear channels because it is not known how many times they will run. So, unlimited channels that are *always* used for replicated inputs (*i.e.*, *replicated channels* [27]), are polymorphic. This is a very convenient circumstance, because replicated channels are the primary mechanism for implementing recursion.

Indeed, going back to *fact* and *stream*, we see that they are replicated channels, that is they are polymorphic. This means that the mismatches between the levels h of y and k of a in (3) and (4) can be compensated by this form of polymorphism and these processes can be declared well typed.

The interplay between recursion and polymorphism leads to a technical problem, though. Recall that in (4) there are two occurrences of a, one in $stream!\langle x+1, a\rangle$ having the same type as y but higher level and another in $y!\langle x, a\rangle$ with a similar type, but opposite (input) polarity. Overall we realize that the type t of y should satisfy the equations

$$t = \mathtt{!}[\mathtt{int} \times t_1]^1 \qquad \begin{array}{l} t_1 = \mathtt{?}[\mathtt{int} \times t_2]^2 \\ t_2 = \mathtt{?}[\mathtt{int} \times t_3]^3 \\ \quad\vdots \end{array} \qquad (5)$$

where t_1 would be the type of a in $y!\langle x, a\rangle$ (for the sake of readability we omit tickets). The problem is that t is not a regular type and therefore cannot be finitely represented by means of the well-known μ notation [11]. To recover a finite representation for t, we "relativize" levels. In particular, we postulate that t should satisfy the equations

$$t = \mathtt{!}[\mathtt{int} \times s]^1 \qquad s = \mathtt{?}[\mathtt{int} \times s]^1$$

and that the type of messages sent on a channel of type t is $\mathtt{int} \times s$ (as written in t) with levels shifted up by 1 (the level of t), namely $\mathtt{int} \times \mathtt{?}[\mathtt{int} \times s]^2$. In turn, the type of messages sent on a channel of type $\mathtt{?}[\mathtt{int} \times s]^2$ is $\mathtt{int} \times \mathtt{?}[\mathtt{int} \times s]^3$, and so on and so forth. Note that $\mathtt{int}$ is not affected by the shifting because it is not a linear type and that overall the channel types obtained by such shifting are the same $t_1, t_2, \ldots$ that we initially guessed in (5), except that now t and s are regular hence finitely representable.

Summary of contributions. We strengthen the notion of *linearity* in the linear π-calculus by defining two type systems ensuring the absence of deadlocks and locks involving linear channels, namely that linear channels are used *exactly* once as opposed to *at most* once. We exploit the features of the linear π-calculus to devise a form of channel polymorphism that allows us to deal with recursive processes also in cyclic network topologies, that is in configurations where two (or more) processes are mutually engaged in communications on different channels. These configurations, while common in the implementation of parallel algorithms and session-based networks, cannot be dealt with existing type systems for the generic π-calculus [22–24].

Outline. In Section 2 we quickly review syntax and semantics of the π-calculus and give formal definitions of deadlock and lock freedom. The type systems are described in Section 3 and illustrated on several examples that highlight the features of our approach, particularly in presence of cyclic network topologies. Section 4 contains a more detailed and technical comparison with related work and particularly with [22–24]. Section 5 concludes and hints at ongoing and future developments. Additional technical material, extended examples, and proofs of the results can be found in the associated technical report [31].

2. Language

We use integer numbers $h, k, \ldots, m, n, \ldots$, a countable set of *variables* $x, y, \ldots$, and a countable set of *channels* $a, b, \ldots$; *names* $u, v, \ldots$ are either channels or variables. *Expressions* $e, \ldots$ and *processes* $P, Q, \ldots$ are defined by:

$$\begin{array}{rl} e ::= & n \mid u \mid (e, e) \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \\ P ::= & \mathbf{0} \mid u?(x).P \mid *u?(x).P \mid u!\langle e\rangle \mid (P \mid Q) \mid (\nu a)P \mid \\ & \mathtt{let}\ x, y = e\ \mathtt{in}\ P \mid \mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}} \end{array}$$

Expressions are integers, names, pairs of expressions, or expressions injected using either $\mathtt{inl}$ or $\mathtt{inr}$. *Values* $\mathsf{v}, \mathsf{w}, \ldots$ are expressions without variables. Processes are the usual terms of the π-calculus enriched with pattern matching for pairs and injected values. In particular, $\mathtt{let}\ x, y = e\ \mathtt{in}\ P$ deconstructs the value of e, which must be a pair, and continues as P where x and y have been respectively replaced by the first and second component of the pair. A process $\mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ evaluates e, which must result into a value $(i\ \mathsf{v})$ where $i \in \{\mathtt{inl}, \mathtt{inr}\}$, and continues as P_i where x_i has been replaced by v. Binders are as expected, in particular $\mathtt{let}\ x, y = e\ \mathtt{in}\ P$, binds both x and y in P, and $\mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ binds x_i in P_i. The notions of *free* and *bound names* of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are defined consequently. In the following we write $(\nu\tilde{a})$ for sequences of restrictions and we use polyadic inputs $u?(x_1, \ldots, x_n)$ as an abbreviation for monadic input of (possibly nested) pairs followed by (possibly nested) pair deconstructions using $\mathtt{let}$'s. For example, $u?(x, y).P$ abbreviates $u?(z).\mathtt{let}\ x, y = z\ \mathtt{in}\ P$ for some fresh z.

The operational semantics is defined by a combination of *structural congruence* $\equiv$ and a *reduction relation* $\rightarrow$. The former is the same as in the π-calculus, except that we do *not* include the law $*P \equiv P \mid *P$ because we treat replicated inputs in a special way. Reduction is defined by the rules below

$$\begin{array}{rl} a!\langle\mathsf{v}\rangle \mid a?(x).P & \rightarrow P\{\mathsf{v}/x\} \\ a!\langle\mathsf{v}\rangle \mid *a?(x).P & \rightarrow P\{\mathsf{v}/x\} \mid *a?(x).P \\ \mathtt{let}\ x, y = \mathsf{v}, \mathsf{w}\ \mathtt{in}\ P & \rightarrow P\{\mathsf{v}, \mathsf{w}/x, y\} \\ \mathtt{case}\ k\ \mathsf{v}\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}} & \rightarrow P_k\{\mathsf{v}/x_k\} \end{array}$$

and closed by *reduction contexts* $\mathscr{C} ::= [\,] \mid (\mathscr{C} \mid P) \mid (\nu a)\mathscr{C}$ and structural congruence. The rules are unremarkable and $P\{\mathsf{v}/x\}$ is the usual capture-avoiding substitution of v in place of the free occurrences of x in P. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$ and we say that P is *stable*, notation $P \not\rightarrow$, if there is no Q such that $P \rightarrow Q$.

To formulate deadlock and lock freedom, we need a few predicates that describe the pending communications of a process P

with respect to some channel a:

$$\begin{aligned}
\mathsf{in}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[a?(x).Q] \wedge a \notin \mathsf{bn}(\mathscr{C})\\
{*}\mathsf{in}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[{*}a?(x).Q] \wedge a \notin \mathsf{bn}(\mathscr{C})\\
\mathsf{out}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[a!\langle e\rangle] \wedge a \notin \mathsf{bn}(\mathscr{C})\\
\mathsf{sync}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \vee {*}\mathsf{in}(a,P)) \wedge \mathsf{out}(a,P)\\
\mathsf{wait}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \vee \mathsf{out}(a,P)) \wedge \neg\mathsf{sync}(a,P)
\end{aligned}$$

In words, $\mathsf{in}(a,P)$ holds if there is a sub-process Q within P that is waiting for a message from a (it is similar to the *live* predicate in [6]). Note that, by definition of reduction context, the input cannot guarded by other actions. The condition $a \notin \mathsf{bn}(\mathscr{C})$ implies that a occurs free in P. The predicates $\mathsf{out}(a,P)$ and ${*}\mathsf{in}(a,P)$ are similar, but they regard outputs and replicated inputs, respectively. Therefore, when $\mathsf{in}(a,P)$ holds it means that there is a pending non-replicated input on a and when $\mathsf{out}(a,P)$ holds it means that there is a pending output operation on a. This discussion explains the $\mathsf{sync}(a,P)$ and $\mathsf{wait}(a,P)$ predicates: the first one denotes the fact that there are pending input/output operations on a, but a synchronization on a is *immediately* possible; the second one denotes the fact that there is a pending output or a pending non-replicated input on a, but no immediate synchronization on a is possible. There is an asymmetry in the way pending inputs and outputs trigger the wait predicate. In particular, we do not interpret ${*}\mathsf{in}(a,P)$ as a pending input operation, meaning that we do not require a replicated input process to run infinitely often. At the same time, *any* pending output triggers the wait predicate, even when the output regards an unlimited channel. This is because such outputs may carry linear channels and, in order to guarantee deadlock freedom, we must be sure that *all* outputs, including those on unlimited channels, can be completed and the messages delivered. For this reason we treat unlimited channels as replicated channels and enforce input receptiveness for them. On the contrary, replicated inputs, which can be though of as persistently available servers, do not trigger the wait predicate because, by definition, they are not allowed to have free linear channels (the type system will enforce this property).

We define deadlock freedom as the absence of stable states with pending communications and lock freedom as the ability to eventually complete any pending communication. Formally:

Definition 2.1 (deadlock [24] and lock freedom [26])**.**

1. We say that P is *deadlock free* if, for every Q such that $P \to^* (\nu\tilde{a})Q \not\to$, we have $\neg\mathsf{wait}(a,Q)$ for every a.
2. We say that P is *lock free* if, for every Q such that $P \to^* (\nu\tilde{a})Q$ and $\mathsf{wait}(a,Q)$, there exists R such that $Q \to^* R$ and $\mathsf{sync}(a,R)$.

For example, both $(\nu a)a!\langle 1984\rangle$ and $(\nu a)a?(x).P$ are deadlocks, whereas $(\nu a){*}a?(x).P$ is lock free. In principle, we should also require that `let` and `case` processes are always able to reduce and that expressions in top-level outputs are always evaluated. These properties are already guaranteed (for closed processes) by conventional type systems for the π-calculus, so we keep Definition 2.1 focused on pending communications. Note that lock freedom implies deadlock freedom, but not viceversa (process (2) is deadlock free but not lock free).

Example 2.2. Many parallel algorithms (Jacobi and Gauss-Seidel, leader election, vertex coloring, just to mention a few) use batteries of processes that iteratively communicate with their neighbors. To maximize parallelism, communication is *full-duplex*, that is processes simultaneously send messages to each other. For instance, the process

$$Node_{\mathrm{A}} \overset{\text{def}}{=} {*}c_{\mathrm{A}}?(x,y).(\nu a)(x!\langle a\rangle \mid y?(z).c_{\mathrm{A}}!\langle a,z\rangle)$$

uses a channel x for sending messages to, and another channel y for receiving messages from, a neighbor process. Each message sent on x carries a payload (omitted) as well as a continuation channel a with which $Node_{\mathrm{A}}$ communicates with its neighbor at the next iteration. Symmetrically, each message received from y contains the neighbor's payload (omitted) and continuation z. The use of fresh continuations at each iteration makes sure that messages are received in the desired order. An alternative modeling using *half-duplex* communication is

$$Node_{\mathrm{B}} \overset{\text{def}}{=} {*}c_{\mathrm{B}}?(x,y).y?(z).(\nu a)(x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a,z\rangle)$$

where the process first waits for the message from its neighbor, and only then sends its own information. It may then be relevant to understand whether a given configuration consisting of mixed nodes, some of type A and others of type B, is lock free. Below we represent three of them

$$\begin{aligned}
L_1(\mathrm{X}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,e\rangle\\
L_2(\mathrm{X},\mathrm{Y}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,f\rangle \mid c_{\mathrm{Y}}!\langle f,e\rangle\\
L_3(\mathrm{X},\mathrm{Y},\mathrm{Z}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,f\rangle \mid c_{\mathrm{Y}}!\langle g,e\rangle \mid c_{\mathrm{Z}}!\langle f,g\rangle
\end{aligned}$$

where each of X, Y, and Z can be either A and B to yield a different configuration. For example, we have

$$\begin{aligned}
L_2(\mathrm{B},\mathrm{B}) \to^* Node_{\mathrm{A}} \mid Node_{\mathrm{B}} &\mid f?(z).(\nu a)(e!\langle a\rangle \mid c_{\mathrm{B}}!\langle a,z\rangle)\\
&\mid e?(z).(\nu b)(f!\langle a\rangle \mid c_{\mathrm{B}}!\langle b,z\rangle) \not\to
\end{aligned}$$

and the final configuration satisfies the in predicate but not out for both e and f, namely $L_2(\mathrm{B},\mathrm{B})$ is *not* lock free. We will see later that a given configuration L_i is lock free if and only if at least one of its nodes is of type A. ■

3. Type system

Notation. *Polarities* $p, q, \ldots$ are subsets of $\{?, !\}$. We abbreviate $\{?\}$ with $?$, $\{!\}$ with $!$, and $\{?, !\}$ with $\#$ and we say that $\emptyset$ and $\#$ are *even polarities*. *Multiplicities* $\iota, \ldots$ are either 1 or ω. We also use a countable set of *type variables* $\alpha, \ldots$. *Types* $t, s, \ldots$ are defined by the grammar below:

$$t ::= \texttt{int} \mid \alpha \mid t \times s \mid t \oplus s \mid p^{\iota}[t]^{n}_{m} \mid \mu\alpha.t$$

The types `int`, $t \times s$, and $t \oplus s$ respectively denote integers, pairs inhabited by values (v, w) where v has type t and w has type s, and the disjoint sum of t and s inhabited by values (`inl` v) when v has type t or (`inr` w) when w has type s. The type $p^{\iota}[t]^{n}_{m}$ denotes a channel to be used with polarity p and multiplicity ι for exchanging messages of type t. The polarity determines the operations allowed on the channel: $\emptyset$ means none, $?$ means input, $!$ means output, and $\#$ means both. The multiplicity determines how many times the channel can or must be used: 1 means that the channel must be used exactly once (for each element in p), while ω means that the channel can be used any number of times. The numbers n and m are respectively the *level* and the *tickets* of the channel and are used only when the channel is a linear one. As we have anticipated in Section 1, levels are used for imposing an order on the usage of channels, so that an action on a channel with high level cannot block another action on a channel with lower level. For example an input on a channel with level 4 can block an input on a channel with level 5, but not viceversa. The tickets of a linear channel limit the number of times the channel can be sent in a message. For example, a channel with 3 tickets can be sent *at most* three times in a message, but it can be used at any time for an input/output operation according to its polarity. A linear channel with 0 tickets cannot be sent in a message. Tickets are always non-negative. We omit level and tickets when $\iota = \omega$ and the multiplicity when it is 1. Also, we often omit tickets when there is none. We use type variables and μ's for building recursive types. Notions of free and bound type variables are as expected.

Contractiveness. For simplicity, we forbid non-contractive types in which recursion variables are not guarded by a channel type. For example, $\mu\alpha.\alpha$ and $\mu\alpha.(\alpha \times \alpha)$ are illegal while $\mu\alpha.p^\iota[\alpha]^n_m$ is allowed. In this way, we will be able to define functions by induction on the structure of types, when such functions do not recur within channel types. Contractiveness can be weakened without compromising the results that follow, but some functions on types must be defined more carefully. The formal definition of contractiveness can be found in [31]. We identify two types modulo renaming of bound type variables and if they have the same infinite unfolding, that is if they denote the same (regular) tree [11]. In particular, we have $\mu\alpha.t = t\{\mu\alpha.t/\alpha\}$.

Levels and tickets. The level of a channel gives a measure to the urgency with which the channel must be used: the lower the level is, the sooner the channel must be used. This measure can be extended from channels to arbitrary types, the idea being that the level of a type is the lowest of the levels of the channel types occurring therein. To compute the level of a type, we define an auxiliary function $|\cdot|$ such that $|t|$ is an element of $\mathbb{Z} \cup \{\bot, \top\}$ where $\bot < n < \top$ for every $n \in \mathbb{Z}$. The function is defined inductively thus:

$$|t| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } t = p^\omega[s] \text{ and } ? \in p \\ n & \text{if } t = p[s]^n_m \text{ and } p \neq \emptyset \\ \min\{|t_1|, |t_2|\} & \text{if } t = t_1 \times t_2 \text{ or } t = t_1 \oplus t_2 \\ \top & \text{otherwise} \end{cases} \tag{6}$$

Intuitively, unlimited channels with input polarity have the lowest level $\bot$ (first equation) because their use cannot be postponed by any means to guarantee input receptiveness. Numbers, unlimited channels with only output polarity, and linear channels with empty polarity have the highest level $\top$ (last equation) because they do not affect (dead)lock freedom in any way. Linear channels with pending operations must be used according to their level (second equation). The level of compound types is the minimum of the levels of the component types (third equation). For instance, $|\,![t]^3_1 \times ?[s]^2_4| = \min\{|\,![t]^3_1|, |?[s]^2_4|\} = \min\{3, 2\} = 2$.

We say that a (value with) type t is *unlimited* if $|t| = \top$, that it is *linear* if $|t| \in \mathbb{Z}$, that it is *relevant* if $|t| = \bot$.

We define another auxiliary function $\$^h_k$ to *shift* levels and tickets: $\$^h_k\, t$ has the same structure of t, except that all levels/tickets in the topmost linear channel types of t have been transposed by h and k respectively. Formally:

$$\$^h_k\, t \stackrel{\text{def}}{=} \begin{cases} p[s]^{n+h}_{m+k} & \text{if } t = p[s]^n_m \text{ and } p \neq \emptyset \\ (\$^h_k\, t_1) \times (\$^h_k\, t_2) & \text{if } t = t_1 \times t_2 \\ (\$^h_k\, t_1) \oplus (\$^h_k\, t_2) & \text{if } t = t_1 \oplus t_2 \\ t & \text{otherwise} \end{cases} \tag{7}$$

As an example, we have $\$^1_1\,(\texttt{int} \times ?[\texttt{int}]^3) = (\$^1_1\,\texttt{int}) \times (\$^1_1\,?[\texttt{int}]^3) = \texttt{int} \times ?[\texttt{int}]^4_1$. Note that positive/negative shifting of levels respectively corresponds to decreasing/increasing the urgency with which a value of a given type must be used. We write $\$^n$ for $\$^n_0$.

A key property used in the proof of subject reduction is that shifting behaves like the identity on unlimited types:

Lemma 3.1. *If t is unlimited, then $\$^h_k\, t = t$ for any h, k.*

Monomorphic and polymorphic channels. A distinguishing feature of our type systems is that message types are "relative" to the level of the channels on which they travel. We use shifting to compute the absolute type of messages that can be sent on a channel. For example, a channel whose type is $![s]^h$ accepts messages of type $\$^h\, s$, that is the type s (within the type of the channel) where all top-level channel types have their level shifted by h (the level of the channel). As we have seen in Section 1, this feature allows us to work with finite representations of recursive types having infinitely many different levels. For example, a recursive type that satisfies the equation $t = ![t]^1$ denotes a linear channel on which it is possible to send a message of type $![t]^2$. In turn, on such a channel it is possible to send a message of type $![t]^3$, and so on and so forth.

Shifting is also used to implement polymorphism. For example, an unlimited (*i.e.*, polymorphic) channel of type $!^\omega[s]$ accepts messages of type $\$^h\, s$ *for any* h. So, for linear (*i.e.*, monomorphic) channels, the shifting is fixed and determined by the level of the channel, while for unlimited (*i.e.*, polymorphic) channels, the shifting is arbitrary.

We are aware that this way of realizing polymorphism, as opposed to *e.g.* using "level variables", is sometimes more restrictive than necessary. For example, a channel of type $!^\omega[?[\texttt{int}]^1 \times ![\texttt{int}]^0]$ accepts messages of type

$$\$^h\,(?[\texttt{int}]^1 \times ![\texttt{int}]^0) = ?[\texttt{int}]^{h+1} \times ![\texttt{int}]^h$$

for every h, but not messages of type $?[\texttt{int}]^{h+2} \times ![\texttt{int}]^h$ where the first component of the pair has been shifted by $h+1$ and the second component of the pair only by h. A message of the latter type could be safely sent on the unlimited channel c appearing in the process F_2 of Section 1. The type system for deadlock freedom allows more flexible forms of polymorphism (see Section 5).

Type environments. We check that processes are well typed in *type environments* $\Gamma, \dots$, which are finite maps from names to types written $u_1 : t_1, \dots, u_n : t_n$. We write $\mathsf{dom}(\Gamma)$ for the *domain* of Γ, namely the set of names for which there is an association in Γ, and Γ, Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. In general we need a more flexible way of composing type environments, taking into account the linearity and relevance of types and the fact that we can split channel types by distributing different polarities to different processes. Following [27], we define a partial composition operator $+$ between types, thus:

$$\begin{array}{rcll} t + t &=& t & \text{if } t \text{ is unlimited} \\ p^\omega[t] + q^\omega[t] &=& (p \cup q)^\omega[t] & \\ p[s]^n_h + q[s]^n_k &=& (p \cup q)[s]^n_{h+k} & \text{if } p \cap q = \emptyset \end{array} \tag{8}$$

Informally, unlimited types compose with themselves without restrictions. The composition of two unlimited/relevant channel types has the union of their polarities. Two linear channel types can be composed only if they have the same level and disjoint polarities, and the composition has the union of their polarities and the sum of their tickets. We extend the partial operator $+$ to type environments:

$$\begin{array}{rcll} \Gamma + \Gamma' &\stackrel{\text{def}}{=}& \Gamma, \Gamma' & \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset \\ (\Gamma, u : t) + (\Gamma', u : s) &\stackrel{\text{def}}{=}& (\Gamma + \Gamma'), u : t + s & \end{array} \tag{9}$$

Note that $\Gamma + \Gamma'$ is undefined if there is $u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma')$ such that $\Gamma(u) + \Gamma'(u)$ is undefined and that $\mathsf{dom}(\Gamma + \Gamma') = \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma')$. We write $\mathsf{un}(\Gamma)$ if all the types in the range of Γ are unlimited. We let $|\Gamma|$ denote the lowest level of the types in the range of Γ, that is $|\Gamma| = \min\{|\Gamma(u)| \mid u \in \mathsf{dom}(\Gamma)\}$.

Typing rules. The typing rules for expressions and processes are presented in Table 1. A judgment $\Gamma \vdash e : t$ denotes that e is well typed and has type t in Γ and a judgment $\Gamma \vdash_k P$ denotes that P is well typed in Γ. The only difference between the type systems for deadlock and lock freedom is the "cost" k of travels for linear channels, which is 0 for deadlock freedom and 1 for lock freedom. This parameter affects only the rules for outputs, and is always propagated unchanged by all rules.

Rules for expressions as well as [T-IDLE], [T-PAR], [T-LET], and [T-CASE] are standard for the linear π-calculus, with the usual split-

Expressions

$$\text{[T-CONST]}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash n : \mathtt{int}} \qquad \text{[T-NAME]}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, u : t \vdash u : t} \qquad \text{[T-PAIR]}\ \frac{\Gamma \vdash e : t \qquad \Gamma' \vdash e' : s}{\Gamma + \Gamma' \vdash (e, e') : t \times s}$$

$$\text{[T-INL]}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{inl}\ e : t \oplus s} \qquad \text{[T-INR]}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inr}\ e : t \oplus s}$$

Processes

$$\text{[T-IN]}\ \frac{\Gamma, x : \$^{n}\, t \vdash_k P \qquad n < |\Gamma|}{\Gamma + u : \mathtt{?}[t]^{n}_{m} \vdash_k u?(x).P} \qquad \text{[T-OUT]}\ \frac{\Gamma \vdash e : \$^{n}_{k}\, t \qquad 0 < |t|}{\Gamma + u : \mathtt{!}[t]^{n}_{m} \vdash_k u!\langle e\rangle}$$

$$\text{[T-IN*]}\ \frac{\Gamma, x : t \vdash_k P \qquad \mathsf{un}(\Gamma)}{\Gamma + u : \mathtt{?}^{\omega}[t] \vdash_k *u?(x).P} \qquad \text{[T-OUT*]}\ \frac{\Gamma \vdash e : \$^{n}_{k}\, t \qquad \bot < |t|}{\Gamma + u : \mathtt{!}^{\omega}[t] \vdash_k u!\langle e\rangle}$$

$$\text{[T-NEW]}\ \frac{\Gamma, a : p^{\iota}[t]^{n}_{m} \vdash_k P \qquad p \text{ even}}{\Gamma \vdash_k (\nu a)P} \qquad \text{[T-PAR]}\ \frac{\Gamma_i \vdash_k P_i\,^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash_k P_1 \mid P_2}$$

$$\text{[T-IDLE]}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_k \mathbf{0}} \qquad \text{[T-LET]}\ \frac{\Gamma \vdash e : t \times s \qquad \Gamma', x : t, y : s \vdash_k P}{\Gamma + \Gamma' \vdash_k \mathtt{let}\ x, y = e\ \mathtt{in}\ P}$$

$$\text{[T-CASE]}\ \frac{\Gamma \vdash e : t \oplus s \qquad \Gamma', x : t \vdash_k P \qquad \Gamma', y : s \vdash_k Q}{\Gamma + \Gamma' \vdash_k \mathtt{case}\ e\ \{\mathtt{inl}\ x \Rightarrow P, \mathtt{inr}\ y \Rightarrow Q\}}$$

Table 1. Typing rules for expressions and processes.

ting of environments and the requirement that unused environments in axioms must be unlimited.

Rule [T-IN] is used for typing linear inputs $u?(x).P$, where u must have type $\mathtt{?}[t]^{n}_{m}$. The continuation P is typed in an environment where the input polarity of u has been removed and the received message x has been added. Note that the type of x is not just t, but t shifted by n, consistently with the relative interpretation of levels of message types. The tickets are irrelevant since u is used for an input operation, not as the content of a message. The condition $n < |\Gamma|$ verifies that the input on u does not block operations on other channels with equal or lower level. In particular, Γ cannot contain relevant channels. Below are some typical examples of ill-typed processes that violate this condition:

- $a : \mathtt{?}[\mathtt{int}]^{1}, b : \mathtt{!}[\mathtt{int}]^{0} \vdash_k a?(x).b!\langle x\rangle$ is not derivable because $1 \not< 0$: the input on a blocks the output on b, but b has lower level than a;
- $a : \#[\mathtt{int}]^{h} \vdash_k a?(x).a!\langle x\rangle$ is a degenerate case of the previous example, where the input on a blocks the very output that should synchronize with it. Note that this process is well typed in the traditional linear π-calculus because $\#[\mathtt{int}] = \mathtt{?}[\mathtt{int}] + \mathtt{!}[\mathtt{int}]$;
- $a : \mathtt{?}[\mathtt{int}]^{h}, c : \mathtt{?}^{\omega}[\mathtt{int}] \vdash_k a?(x).{*c?(y)}$ is not derivable because $|\mathtt{?}^{\omega}[\mathtt{int}]| = \bot$. To guarantee input receptiveness, we require that replicated inputs cannot be guarded by other operations.

Rule [T-OUT] is used for typing linear outputs on channels of type $\mathtt{!}[t]^{n}_{m}$. The type of the message e must be t (as specified in the type of the channel u) shifted by n again since t is relative to the level of u. The tickets are shifted by 1, but only for lock freedom ($k = 1$). The meaning is that, by sending e as a message, one ticket from every linear channel type in the type of e is consumed. This prevents the degenerate phenomenon of infinite delegation that we have seen in (2). The condition $0 < |t|$ verifies that the level of the message is greater than that of the channel on which it travels. Below are a few examples:

- The judgment $a : \mathtt{!}[\mathtt{?}[\mathtt{int}]^{1}_{0}]^{2}_{0}, b : \mathtt{?}[\mathtt{int}]^{3}_{1} \vdash_1 a!\langle b\rangle$ is derivable because $\mathtt{?}[\mathtt{int}]^{3}_{1} = \$^{2}_{1}\, \mathtt{?}[\mathtt{int}]^{1}_{0}$. Note in particular that the channel to be sent on a must have no tickets, which is in fact what happens to b after 1 ticket is consumed from its type before it travels on a.
- The judgment $a : \mathtt{?}[\mathtt{!}[\mathtt{int}]^{2}_{0}]^{0}_{0}, b : \mathtt{!}[\mathtt{!}[\mathtt{int}]^{1}_{0}]^{1}_{0} \vdash_1 a?(x).b!\langle x\rangle$ is not derivable, because x received from a has no tickets so it cannot travel on b.
- Let $t = \mu\alpha.\mathtt{?}[\alpha]^{0}$ and observe that $\#[t]^{1} = \mathtt{!}[t]^{1} + \mathtt{?}[t]^{1}$. The judgment $a : \#[t]^{1} \vdash_0 a!\langle a\rangle$ is not derivable, despite the message a has the "right" type $\mathtt{?}[t]^{1} = \$^{1}\, t$, because the condition $0 < |t| = 0$ is not satisfied. According to Definition 2.1, a process like $(\nu a)a!\langle a\rangle$ is deadlock (there are systems in which such process would actually be able to reduce, but in doing so it would generate memory leaks [4, 21]).

Rule [T-IN*] is used for typing replicated inputs $*u?(x).P$. This rule differs from [T-IN] in three important ways. First of all, u must be an unlimited channel with input polarity. Second, the residual environment Γ must be unlimited, because the continuation P can be spawned an arbitrary number of times. Third, it may be the case that $u \in \mathsf{dom}(\Gamma)$, because $\mathtt{?}^{\omega}[t] + \mathtt{!}^{\omega}[t] = \#^{\omega}[t]$ according to (8) and $\mathtt{!}^{\omega}[t]$ is unlimited. This means that replicated input processes may invoke themselves, as we have seen in many examples.

Rule [T-OUT*] is used for outputs on unlimited channels. There are two key differences with respect to [T-OUT]. First, the condition $\bot < |t|$, where t is the type of e, implies that only relevant names cannot be communicated, but a process like $a!\langle a\rangle$ is well typed when a is unlimited, for example by giving the inner a type $\mu\alpha.\mathtt{!}^{\omega}[\alpha]$. Second, the type of the message need not match exactly the type t in the channel, but its level can be shifted by an arbitrary amount n. This is the technical realization of polymorphism. In particular, each distinct output on u can shift the type of the message by a different amount, therefore allowing polymorphic recursion. We will see this feature at work Example 3.7.

Rule [T-NEW] is used for restricting new (linear and unlimited) channels. Only channels with an even polarity can be restricted: either the channel comes with full polarity $\#$, or with no polarity $\emptyset$ (this is only useful for subject reduction).

Notation 3.2 (well-typed process)**.** We write:

1. $\vdash_{\text{DF}} P$ if $\emptyset \vdash_0 P$, and
2. $\vdash_{\text{LF}} P$ if $\emptyset \vdash_1 P$ and all levels in the derivation are non-negative.

Intuitively, posing $k = 0$ in the type system for deadlock freedom means that tickets are not consumed (and therefore are irrelevant) when a channel travels as a message. This is acceptable because infinite delegations (like in (2)) are not deadlocks. The type system for lock freedom requires a well-founded order on levels (hence the restriction to natural numbers for levels) and posing $k = 1$ means establishing an upper bound on delegations (hence the relevance of tickets).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a : \$^n_2\, s \vdash a : \$^n_2\, s \qquad 0 < m}{x : t, a : \$^n_2\, s \vdash_1 x!\langle a\rangle}\,[\text{T-OUT}] \qquad \dfrac{z : \$^m_1\, s, a : \$^m_1\, t \vdash (a,z) : \$^m_1\,(t \times s)}{c_{\text{B}} : \mathbf{!}^\omega[t \times s], z : \$^m_1\, s, a : \$^m_1\, t \vdash_1 c_{\text{B}}!\langle a, z\rangle}\,[\text{T-OUT*}]}{c_{\text{B}} : \mathbf{!}^\omega[t \times s], x : t, z : \$^m_1\, s, a : \#[\$^0_1\, s]^{m+n}_3 \vdash_1 x!\langle a\rangle \mid c_{\text{B}}!\langle a, z\rangle}\,[\text{T-PAR}]}{c_{\text{B}} : \mathbf{!}^\omega[t \times s], x : t, z : \$^m_1\, s \vdash_1 (\nu a)(x!\langle a\rangle \mid c_{\text{B}}!\langle a, z\rangle)}\,[\text{T-NEW}] \qquad m < n}{c_{\text{B}} : \mathbf{!}^\omega[t \times s], x : t, y : s \vdash_1 y?(z).(\nu a)(x!\langle a\rangle \mid c_{\text{B}}!\langle a, z\rangle)}\,[\text{T-IN}]}{c_{\text{B}} : \#^\omega[t \times s] \vdash_1 *c_{\text{B}}?(x,y).y?(z).(\nu a)(x!\langle a\rangle \mid c_{\text{B}}!\langle a, z\rangle)}\,[\text{T-IN*}]$$

Table 2. Typing derivation for Node_{B}.

Properties. The relative interpretation of levels makes it possible to shift whole derivations and preserve typing. More precisely, let $\$^n\,\Gamma$ be the environment obtained by shifting all the types in the range of Γ by n, that is $(\$^n\,\Gamma)(u) = \$^n\,\Gamma(u)$ for every $u \in \mathsf{dom}(\Gamma)$. We have:

Lemma 3.3 (shifting)**.** *The following properties hold:*

1. *If $\Gamma \vdash e : t$, then $\$^n\,\Gamma \vdash e : \$^n\, t$.*
2. *If $\Gamma \vdash_0 P$, then $\$^n\,\Gamma \vdash_0 P$.*
3. *If $\Gamma \vdash_1 P$ and $n \geq 0$, then $\$^n\,\Gamma \vdash_1 P$.*

Note that the converse of 3) does not hold. For example, the derivation for $u : \mathbf{!}[\mathtt{int}]^1_0 \vdash_1 (\nu a)(a!\langle 1984\rangle \mid a?(x).u!\langle x\rangle)$ relies on the fact that u has level 1, for otherwise it would not be possible to prefix $u!\langle x\rangle$ with an input operation on a.

Typing is preserved by reductions and the type environment may change as a consequence of communications on linear channels, just like in the linear π-calculus [27]:

Lemma 3.4 (subject reduction)**.** *If $\Gamma \vdash_k P$ and $P \to P'$, then $\Gamma' \vdash_k P'$ for some Γ'.*

The proof is essentially standard, except for the interesting case concerning the communication on unlimited (*i.e.*, polymorphic) channels sketched below. Consider the derivation

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{v} : \$^n\, t}{a : \mathbf{!}^\omega[t], \Gamma \vdash_k a!\langle \mathsf{v}\rangle} \qquad \dfrac{\Gamma', x : t \vdash_k P \quad \mathsf{un}(\Gamma')}{a : \mathbf{?}^\omega[t], \Gamma' \vdash_k *a?(x).P}}{a : \#^\omega[t], \Gamma + \Gamma' \vdash_k a!\langle \mathsf{v}\rangle \mid *a?(x).P}$$

and observe that the actual type $\$^n\, t$ of the message v being sent does *not* match exactly the type t expected by the receiver, but is shifted by some arbitrary amount n. In order to type the reduct $P\{\mathsf{v}/x\}$, we proceed like this. First of all we shift the derivation of $\Gamma', x : t \vdash_k P$ using Lemma 3.3, and obtain:

$$\$^n\,\Gamma', x : \$^n\, t \vdash_k P$$

Then, we observe that Γ' is unlimited because it is the type environment used for typing a replicated process. This means that $\$^n\,\Gamma' = \Gamma'$, since shifting is the identity on unlimited types (Lemma 3.1). Therefore we deduce:

$$\Gamma', x : \$^n\, t \vdash_k P$$

Now, the actual and expected types of the message v coincide and we know from the initial derivation that $\Gamma + \Gamma'$ is defined, so we can apply the conventional substitution lemma for the linear π-calculus [27] and conclude:

$$\Gamma + \Gamma' \vdash_k P\{\mathsf{v}/x\}$$

Since the type systems refine the one in [27], all the properties of the linear π-calculus (partial confluence, linear usage of channels, etc.) still hold. The added value is that the refined type systems guarantee deadlock/lock freedom.

Theorem 3.5 (soundness)**.** *The following properties hold:*

1. *If $\vdash_{\text{DF}} P$, then P is deadlock free.*
2. *If $\vdash_{\text{LF}} P$, then P is lock free.*

The proof of 1) is by contradiction, for the existence of a well-typed, stable process with pending input or output operations on linear channels would imply the existence of an infinite chain of channels with different levels, contradicting the fact that there are only finitely many distinct channels in the process. The proof of 2) does an induction on a *measure* that includes, among other information, level and tickets of the linear channel for which we are completing the communication (see Definition 2.1(2)). The induction is well founded if so is the domain of levels, whence the restriction to natural numbers for the levels in the type system for lock freedom. The details can be found in [31].

We conclude this section with a few examples showcasing the key features of our type systems. In particular, the recursive processes in Examples 3.6, 3.8, and 3.9 are representative of those configurations that are typeable thanks to our form of channel polymorphism, but not in similar type systems [22–24] (see Section 4 for a detailed discussion).

Example 3.6. Regarding the processes in Example 2.2, let

$$t \overset{\text{def}}{=} \mathbf{!}[s']^n_0 \quad \text{and} \quad s \overset{\text{def}}{=} \mathbf{?}[s']^m_0 \quad \text{and} \quad s' \overset{\text{def}}{=} \mu\alpha.\mathbf{?}[\alpha]^m_1$$

and observe that $s' = \$^0_1\, s = \mathbf{?}[\$^0_1\, s]^m_1$. For Node_{B} we obtain the typing derivation shown in Table 2 if and only if $0 < m < n$. An analogous derivation is obtained for Node_{A} if and only if $0 < n, m$. We deduce that $L_1(\text{A})$, $L_2(\text{A},\text{A})$, $L_2(\text{A},\text{B})$, and $L_2(\text{B},\text{A})$ are all well typed. Since these constraints on levels are the most general ones, we also deduce that there are no derivations for $L_1(\text{B})$ and $L_2(\text{B},\text{B})$, suggesting that these configurations are locked. ■

Example 3.7. Below is the encoding of the function that computes the n-th number in the sequence of Fibonacci. Instead of showing the typing derivation, which is conventional for the linear π-calculus, we just annotate each linear name occurring in the term with its level and leave it to the reader to verify that such annotations are consistent with the constraints expressed in the typing rules for deadlock freedom:

```
$*\mathit{fibo}?(x, y^0).$
  $\mathtt{if}\ x \leq 1\ \mathtt{then}\ y^0!\langle x\rangle$
  $\mathtt{else}\ (\nu a^{-1})(\nu b^{-2})\big(\mathit{fibo}!\langle x-1, a^{-1}\rangle \mid \mathit{fibo}!\langle x-2, b^{-2}\rangle \mid$
                    $b^{-2}?(z_2).a^{-1}?(z_1).y^0!\langle z_1 + z_2\rangle\big)$
```

Note the use of negative levels associated with the fresh continuation channels a and b, due to the fact that the inputs on these channels block the output on the continuation y. Observe also the role of polymorphic recursion in typing this process: the two recursive invocations of *fibo* are applied to continuation channels a and b with different levels. ■

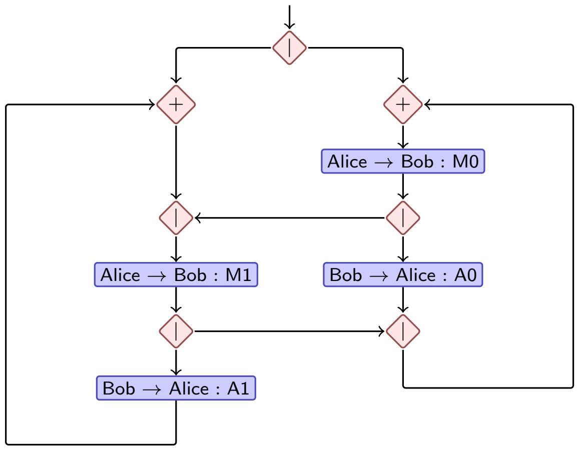

Figure 1. The protocol between Alice and Bob from [15].

Example 3.8. The following variation of the process $Node_{\mathrm{A}}$ gives an example of dynamic lock-free system (beware that in this and the next example, subscripts in names denote tickets; we use colors to ease readability):

$$\begin{aligned}&\mathit{Ring} \overset{\text{def}}{=} *c?(x_0^0, y_0^0).\\&\quad(\nu a_3^1)\big(x_0^0?(z_1^1).(\nu b_2^1)(c!\langle z_1^1, b_1^1\rangle \mid c!\langle b_1^1, a_1^1\rangle) \mid y_0^0!\langle a_2^1\rangle\big)\end{aligned}$$

Each message on c spawns two duplicates connected by a new channel b. As a consequence, the process $(\nu e)(\mathit{Ring} \mid c!\langle e, e\rangle)$ grows as a ring of processes, each sending a message to its neighbor, and the ring doubles its diameter each time a round of communications is completed. ■

Example 3.9. Figure 1 depicts a protocol between two interacting parties Alice and Bob taken from [15] in which Alice sends alternated messages M0 and M1 to Bob, and Bob answers with corresponding acknowledgments A0 and A1. The diagram describes communications (the rectangles) between the two parties and their dependencies (the arcs), combined with merging points (+ nodes), forks (| nodes with two outgoing arcs) and joins (| nodes with two incoming arcs). The particularity of this modeling of the protocol is that Alice concurrently waits for both A0 and A1 before sending the next corresponding Mi, but always alternating between M0 and M1 messages.

We model Bob as the process below, which is parametric in two channels x and y from which Bob respectively receives messages M0 and M1 along with the continuations on which he sends the corresponding Ai (we omit the actual payloads and focus on channels and their continuations):

$$\begin{aligned}&\mathsf{Bob} \overset{\text{def}}{=} *bob?(x^0, y^2).\\&\quad(\nu a_3^3 b_3^5)\big(x^0?(\bar{x}^1).(\bar{x}^1!\langle a_2^3\rangle \mid y^2?(\bar{y}^3).(\bar{y}^3!\langle b_2^5\rangle \mid bob!\langle a_1^3, b_1^5\rangle)))\big)\end{aligned}$$

We model Alice as two parallel threads Alice0 and Alice1, each handling messages and acknowledgments with matching number, so that Alice0 corresponds to the rightmost vertical flow in Figure 1, and Alice1 to the leftmost one. To respect the protocol and prevent that two messages with the same index are sent in a row, Alice uses a third channel z (unknown to Bob) with which she synchronizes the two threads:

$$\begin{aligned}&\mathsf{Alice0} \overset{\text{def}}{=} *alice0?(x^0, z^1).\\&\quad(\nu a_1^1 c_1^2)\big(x^0!\langle a_1^1\rangle \mid z^1!\langle c_1^2\rangle \mid a^1?(\bar{x}_1^3).c^2?(\bar{z}_1^4).alice0!\langle \bar{x}_1^3, \bar{z}_1^4\rangle\big)\end{aligned}$$

$$\begin{aligned}&\mathsf{Alice1} \overset{\text{def}}{=} *alice1?(y^2, z^1).\\&\quad(\nu b_1^3 c_3^4)\big(z^1?(\bar{z}^2).(y^2!\langle b_1^3\rangle \mid \bar{z}^2!\langle c_2^4\rangle \mid b^3?(\bar{y}_1^5).alice1!\langle \bar{y}_1^5, c_1^4\rangle))\big)\end{aligned}$$

The whole protocol is modeled as the term $(\nu ab)(\mathsf{Bob} \mid \mathsf{Alice0} \mid \mathsf{Alice1} \mid bob!\langle a, b\rangle \mid (\nu c)(alice0!\langle a, c\rangle \mid alice1!\langle b, c\rangle))$. Note the interleaving of possibly blocking actions on different channels in both Alice and Bob. ■

4. Related work

The articles [22–24, 26] have been a source of inspiration for our work so we devote most of this section to them. The discussion is quite technical and space constraints force us to assume some familiarity with these works.

Deadlock freedom. Kobayashi [24] defines a type system for ensuring deadlock freedom in the π-calculus. Channels types are pairs $[t]/U$ where t is the type of messages and U – called *usage* – is a term of a process algebra that includes prefixing and parallel composition and that specifies how the channel is accessed. For example, a channel of type $[\texttt{int}]/(!_n^m.?_k^h \mid ?_m^n.!_h^k)$ can be used concurrently by two processes, one that first sends and then receives an integer and the other that does the opposite. Actions like $!_n^m$ are annotated with an *obligation* m and a *capability* n to prevent mutual dependencies between channels: inputs on a channel with capability n can only block operations on other channels whose obligation is larger than n. The special value ∞ is allowed for denoting various forms of unlimited channels. There is a significant overlapping between [24] and our work: usages can describe linear channels as a particular case; linearized channels (those that can be accessed multiple times, but only sequentially) can be encoded using linear channels and continuation passing; levels have been directly inspired by obligations and capabilities and are used similarly. Let us now analyze the main differences.

First we observe that distinguishing between obligations and capabilities as opposed to having just a single level may improve the accuracy of the analysis in some cases. For example,

$$*c?(x, y).x?(z).y?(z) \mid *c?(x, y).y?(z).x?(z)$$

is ill typed in our type system but is typeable in [24] by giving both x and y obligation 1 and capability 0. The null capability means that, assuming that outputs on x and y are available in the environment immediately (capability 0), then either of these processes guarantees that both inputs will be performed after at most one reduction (obligation 1).

The main distinguishing features of our approach, however, are polymorphism and the fact that levels have a relative interpretation in message types. To illustrate the effect of these features on the accuracy of our type systems, we revisit Example 2.2 in the setting of [24] assuming to extend usages with recursion (recursive usages are admitted in [24], but only for type reconstruction purposes). The following modeling of $L_2(\mathrm{A}, \mathrm{A})$ uses no explicit continuation passing and therefore is the most favorable to the type system in [24]:

$$*c?(x, y).(x!\langle 3\rangle \mid y?(z).c!\langle x, y\rangle) \mid c!\langle e, f\rangle \mid c!\langle f, e\rangle \qquad (10)$$

The process is tentatively typed using the assignments

$$\begin{aligned}e &: [\texttt{int}]/(\mu\alpha.!_n^m.\alpha \mid \mu\alpha.?_m^n.\alpha)\\ f &: [\texttt{int}]/(\mu\alpha.?_k^h.\alpha \mid \mu\alpha.!_h^k.\alpha)\end{aligned}$$

where the fact that obligations and capabilities are swapped in complementary actions of the usages of e and f is necessary to satisfy the *reliability* condition [24] which corresponds, in our setting, to the property that channel types with opposite polarities and same level can be combined together (last equation of (8)). The structure of (10) requires the simultaneous satisfiability of $k < m$ (there is an input on f with capability k that blocks e whose topmost obligation is m) and of $m < k$ (there is an input on e with capability m that blocks f whose topmost obligation is k), therefore (10) cannot be proven deadlock free with the type discipline in [24]. The problem is that obligations and capabilities, unlike levels, are statically associated with actions: e and f always have the same obligation and capability regardless of the number of unfoldings of their usages. By contrast, in Example 3.6 the channels e and f and their continuations are assigned an increasing sequence of levels. There are two features in [24] that partially overcome the limitations due to the static assignment of obligations and capabilities. The first one is that the behavioral nature of usages allows the typing of recursive processes involving one channel. For example, the following variant of $L_2(\mathrm{A}, \mathrm{B})$

$$*c?(y).(y!\langle\rangle \mid c!\langle y\rangle) \mid *d?(y).y?(z).d!\langle y\rangle \mid c!\langle e\rangle \mid d!\langle e\rangle$$

is well typed in [24]. The point is that in [24] an input process $u?(x).P$ is well typed if the capability of u is smaller than the obligation of all the channels in P *except* u *itself*. The second feature stems from the observation that the strict order $<$ used for comparing the capability of a channel u and the obligation of another channel v can be safely weakened to $\leq$ if u has been created later than v. This property, denoted by the relation $u \prec v$ in [24], can be inferred from the syntactic structure of processes and makes it possible to deal with processes like (3), where the input on the newer channel a is allowed to block operations on the older channel y despite the fact that a and y have the same type (hence the same obligations and capabilities). As noted in [24], the very same feature would also work using the opposite order $\succ$, but one would have to choose between either $\prec$ or $\succ$, since using their union would be unsound. The opposite order $\succ$ however arises naturally with continuation passing, as we have seen in (4) and in most of the examples throughout the paper.

In summary, the techniques based on static obligations and capabilities have difficulties handling systems where two or more recursive processes interleave actions on two or more channels. The use of the $\prec$ relation alleviates in part these limitations, when processes have a function-like behavior. The adaptation of our form of polymorphism to the type systems based on usages remains an open problem that we have not investigated in detail.

Usages provide more precise information on unlimited channels. For example, the lock-free modeling of the dining philosophers in [22] cannot be captured by our type system because it uses non-replicated unlimited channels for representing shared resources (the forks) accessed by competing processes (the philosophers). We are investigating whether it is possible to identify a subclass of unlimited channels used for modeling these configurations and that can be integrated within our typing discipline.

Lock freedom. Type systems for lock freedom have been presented in [22, 23]. These works rely on essentially the same types and usages already discussed for deadlock freedom, and so they are subject to the same problems that arise in (10). In particular, all the processes discussed in Examples 3.6, 3.8, 3.9, which are representative configurations mixing recursion and cyclic network topologies, are ill typed in [22–24].

The partial order $\prec$ is unsound and cannot be used to improve the accuracy of lock-free analysis in [22, 23]. This is reflected also in our type system for lock freedom, where negative levels are banned. Consequently, processes such as (3) or the one in Example 3.7 can be proved deadlock free but not lock free. Lock-free typability of (3) (restricted to natural numbers) and other recursive-like processes is recovered in [26], where it is observed that lock freedom can be characterized as the conjunction of deadlock freedom and *termination*. Such characterization turns out to be more accurate on recursive processes with function-like behaviors, partly because the type system for deadlock freedom imposes fewer constraints, partly thanks to the accuracy of the techniques for proving termination. In fact, the hybrid approach [26] is parametric on the specific techniques for verifying deadlock freedom or termination. This means that it suffers from the same shortcomings that we have discussed earlier for processes like (10) when instantiated with the deadlock free analysis of [24], but also that it should be applicable in our setting, in which case it would benefit from the better accuracy of our type system for deadlock freedom. By the way, the limit we impose on the number of times a channel can be sent in a message is akin to ensuring a form of termination (in fact, the eventual use of the channel for performing a communication) so in a sense our type system for lock freedom can already be seen as an instance of the hybrid framework described in [26].

Termination. The techniques used in [26] for ensuring termination are based on previous type systems studied by Deng and Sangiorgi [14]. Interestingly, also these type rely on *levels* associated with channels. There are two fundamental differences between levels in these and in our work. First, levels in [14] are associated with replicated channels, while in our case they matter only for linear ones. This is motivated by the two orthogonal purposes of the type systems in [14] and ours: in the former ones, the focus is on replicated channels since it is replication that may jeopardize termination, whereas in our type systems the focus is on linear channels, which are the critical ones as far as locks and deadlocks are concerned. The second difference is that levels in [14] work "the other way round" in the sense that a replicated process $*a?(x).P$ is well typed if the level of a is *greater than* the levels of the channels used in subject position within P. The idea is that recursive "calls" within P should involve "smaller" channels or channels with equal level but "smaller" messages. Interestingly, in the case of termination only the channels in subject position are considered in the typing rule for replication, whereas for (dead)lock freedom we enforce an ordering on *all* channels in the rule [T-IN]. Again, this difference is explained by the different purposes of the two type systems. In the case of termination, it is the act of sending of a message on a replicated channel that may yield divergences. In our type systems, the mere *occurrence* of a channel, no matter whether in subject or object position, may jeopardize (dead)lock freedom so the full extent of the condition in the rule [T-IN] is crucial in the proof of Theorem 3.5.

Session types. Session types [17, 18] are behavioral types akin to sequential usages in which each single input/output action is annotated with the type of a message. For example, the session type `![int].?[bool]` denotes a channel on which it is possible to first sent an integer, and then receive a boolean. *Duality* relates session types associated with corresponding endpoints of a binary session and generalizes the composition operator between linear types (8). For example, the session type above is dual of `?[int].![bool]`. In general, session type disciplines provide intra-session safety and progress guarantees, but fall short in assuring these properties at the inter-session level because channels are typed independently. There exist works relating communications on different sessions using *correspondence assertions* [3], although these cannot be used for preventing (dead)locks. In some works [6, 34] a plain session type discipline ensures deadlock freedom also when sessions are interleaved, but only because the syntax of (well-typed) processes prevents the modeling of cyclic network topologies.

Multiparty sessions and global types. A possibility for guaranteeing (dead)lock freedom to configurations where multiple processes are interacting is to extend the notion of session to multiple participants. This idea has inspired studies on *multipoint* [2] and *multiparty* [19] session types. In the particular case of multiparty session types, a global type [7, 15, 16, 19] is used to describe the *interactions* between participants of a session as opposed to the *actions* that participants perform on the channel of the session. For example, the interactions between the three processes that compose the system $L_3(\texttt{A}, \texttt{B}, \texttt{B})$ in Example 3.6 can be described by the recursive global type G_3 that satisfies the equation

$$G_3 = \texttt{A} \to \texttt{B}.\texttt{B} \to \texttt{C}.\texttt{C} \to \texttt{A}.G_3$$

which gives names A, B, and C to the three *participants* and specifies the order of their synchronizations. In general, global types also allow the specification of the type of the exchanged messages and include operators other than prefixing, such as choice and parallel composition. Global types that satisfy some well-formedness conditions are *projectable* into session types and processes that conform to these projections are guaranteed to be type safe, to implement the protocol described by the global type, and to be lock free. Therefore, global types are an effective way of extending the lock freedom property from binary to n-ary sessions (a similar property is conjectured also for multipoint session types in [2]).

One downside of global types is that they may make it harder to build and maintain systems *compositionally* (although some modularity and compositionality aspects have recently been addressed in [13] and [28]). For instance, $L_2(\texttt{A}, \texttt{B})$ and $L_3(\texttt{A}, \texttt{B}, \texttt{B})$ in Example 2.2 are assembled in a modular way using the same constituents, but the global types $G_2 = \texttt{A} \to \texttt{B}.\texttt{B} \to \texttt{A}.G_2$ and G_3 above that describe the interactions in these systems are quite different. It may also be harder to describe "unstructured" interactions from a global perspective. For instance, it takes the most sophisticated global type language available to date [15] to describe accurately the dependencies between Alice's threads in Example 3.9. For these reasons, global types are ideal for describing delimited interactions within sessions, but they cannot dispense completely from the need of interleaving different sessions, in which case they are unable to prevent (dead)locks.

These observations have motivated the study of mixed approaches [1, 9, 10] that keep track of the order in which different sessions interleave with the purpose of detecting and flagging mutual dependencies between sessions that could lead to (dead)locks. These mixed approaches turn out to be quite restrictive (*e.g.*, recursive processes can access one session only [10]) and require heavyweight and multi-layer type systems.

5. Conclusions

Type systems that enforce (dead)lock freedom of communicating processes inevitably rely on trade-offs, for such properties are undecidable [5] even under severe restrictions of the communication model (*e.g.*, for half-duplex communication with three or more processes [8]). These type systems essentially follow one of two approaches: some [22–24] favor compositionality and apply to generic process algebras, but have difficulties in dealing with recursive processes and cyclic network topologies; others can handle such configurations, but only within the scope of (multiparty) sessions and assuming that the communication protocol is explicitly described in advance by a global type. In the context of the linear π-calculus, our type systems conjugate the advantages of both approaches and do so with minimal machinery. Indeed, all previous type systems for (dead)lock freedom rely on rich behavioral types (usages [22–26], session types [29], global types [1, 9, 10, 15, 16, 19], conversation types [33]), and when they are able to deal with recursive processes it is because they take advantage of such richness (see Section 4), almost implying the necessity of rich types to ensure (dead)lock freedom. We have shown that this is not the case, at least for the linear π-calculus.

The linear π-calculus is an adequate model for many concrete communicating systems. In particular, it is the "assembly language" that lies beneath binary sessions [12, 25] and in [31] we show that many multiparty sessions too can be compiled into the linear π-calculus. So, one can establish (dead)lock freedom of a session-based process (also in presence of session interleaving) by compiling it into the linear π-calculus and by typing the resulting process using our type systems. When this approach is unfeasible – not all multiparty sessions can be encoded using linear channels – or undesirable – the structure of sessions may serve other important purposes – the technique described in this paper can still be used for reasoning on inter-session (dead)lock freedom: just like we annotate *channel types* with levels and tickets, it is possible to annotate the single *actions* within session and global types in the same way. This experiment was already attempted in [29, 33], but in these works the annotations are "static", much like obligations/capabilities, and so they suffer from issues analogous to those described in Section 4.

We conclude the paper discussing three lines of development.

Polymorphism. The form of channel polymorphism that lies at the core of our type systems allows the shifting of levels with maps of the shape $z \mapsto z + n$. This prevents, for example, typing an output $a!\langle x^0, y^1\rangle$ if a has type $!^\omega[?[\texttt{int}]^1 \times ![\texttt{int}]^3]$, because a expects x and y to have levels at distance 2, while they are at distance 1. In fact, the type system for deadlock freedom can be generalized to rational numbers for levels and *positive affine maps* for transforming them. In the example above, the map $z \mapsto \frac{1}{2}z - \frac{1}{2}$ turns the levels 1 and 3 in the type of a into 0 and 1, namely the levels of x and y. The type system for lock freedom is slightly less flexible, because its soundness proof requires levels to be part of a well-founded set.

Type reconstruction. As witnessed by the examples in this paper, it is unrealistic to assume that the programmer is capable of guessing the correct values of levels and tickets of linear channels in general. We are finalizing a complete type reconstruction algorithm for our type systems as an extension of existing reconstruction algorithms for the linear π-calculus [20, 30]. The algorithm consists of two phases: in the first one, it uses variables for representing unknown types, levels, and tickets and relies on a modified set of typing rules that *compute* constraints between such variables instead of of enforcing them. The second phase attempts at solving these constraints. Clearly, the most critical constraints are those concerning level and ticket variables, especially because of the shifting operator used in the rules [T-IN], [T-OUT], and [T-OUT*]. It turns out that, because of the nature of the shifting operator, the constraints concerning levels and tickets are always *linear*, hence their solvability coincides with that of an integer programming problem, for which there are well-known resolution methods.

Programming languages. Rule [T-IN] compares the level of a linear channel used for an input operation with that of all the channels that occur in the continuation of the process after the operation. This rule is not particularly realistic in a concrete (*i.e.* structured) programming language where input/output operations may occur within functions, methods, objects, modules, etc. In general, the structure of a program may prevent the type checker from knowing the behavior of the program beyond a certain horizon, which is in contrast with the rightmost premise of rule [T-IN]. In [32] we have shown how to adapt the type system for deadlock freedom to a higher-order concurrent programming language based on linear channels. The basic idea is to use an *effect system* to keep track of the channels occurring in, and used by, functions so as to enable a compositional analysis of programs.

Acknowledgments

The author is grateful to the anonymous CSL-LICS'14 reviewers for their comments and suggestions, to Mariangiola Dezani, Nobuko Yoshida, and Andrea Tosatto for their feedback on early drafts of the paper, and to Elena Giachino and Cosimo Laneve with whom he had insightful discussions on the topics of this paper. This research has been supported by ICT COST Action IC1201 BETTY, MIUR project CINA, and Ateneo/CSP project SALT.

References

[1] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proceedings of CONCUR'08*, LNCS 5201, pages 418–433, 2008.

[2] E. Bonelli and A. B. Compagnoni. Multipoint session types for a distributed calculus. In *Proceedings of TGC'07*, LNCS 4912, pages 240–256. Springer, 2008.

[3] E. Bonelli, A. B. Compagnoni, and E. L. Gunter. Correspondence assertions for process synchronization in concurrent communications. *Journal of Functional Programming*, 15(2):219–247, 2005.

[4] V. Bono and L. Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.

[5] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[6] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of CONCUR'10*, LNCS 6269, pages 222–236, 2010.

[7] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On Global Types and Multi-Party Sessions. *Logical Methods in Computer Science*, 8:1–45, 2012.

[8] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, 2005.

[9] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *Proceedings of COORDINATION'13*, LNCS 7890, pages 45–59. Springer, 2013.

[10] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, to appear.

[11] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[12] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012.

[13] R. Demangeon and K. Honda. Nested protocols in session types. In *Proceedings of CONCUR'12*, LNCS 7454, pages 272–286. Springer, 2012.

[14] Y. Deng and D. Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, 2006.

[15] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *Proceedings of ESOP'12*, LNCS 7211, pages 194–213. Springer, 2012.

[16] P.-M. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.

[17] K. Honda. Types for dyadic interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

[18] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.

[19] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of POPL'08*, pages 273–284. ACM, 2008.

[20] A. Igarashi and N. Kobayashi. Type reconstruction for linear -calculus with i/o subtyping. *Information and Computation*, 161(1):1–44, 2000.

[21] S. Jakšić and L. Padovani. Exception Handling for Copyless Messaging. *Science of Computer Programming*, 84:22–51, 2014.

[22] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

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

[24] N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of CONCUR'06*, LNCS 4137, pages 233–247. Springer, 2006.

[25] N. Kobayashi. Type systems for concurrent programs. Technical report, Tohoku University, 2007. Short version appeared in 10th Anniversary Colloquium of UNU/IIST, 2002.

[26] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Transactions on Programming Languages and Systems*, 32(5), 2010.

[27] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[28] F. Montesi and N. Yoshida. Compositional choreographies. In *Proceedings of CONCUR'13*, LNCS 8052, pages 425–439. Springer, 2013.

[29] L. Padovani. From Lock Freedom to Progress Using Session Types. In *Proceedings of PLACES'13*, EPTCS 137, pages 3–19, 2013.

[30] L. Padovani. Type reconstruction for the linear π-calculus with composite and equi-recursive types. In *Proceedings of FoSSaCS'14*, LNCS 8412, pages 88–102. Springer, 2014.

[31] L. Padovani. Deadlock and lock freedom in the linear π-calculus. Technical report, Università di Torino, 2014. Available at `http://hal.inria.fr/hal-00932356/`.

[32] L. Padovani and L. Novara. Types for deadlock-free higher-order concurrent programs. Technical report, Università di Torino, 2014. Available at `http://hal.inria.fr/hal-00954364/`.

[33] H. T. Vieira and V. T. Vasconcelos. Typing progress in communication-centred systems. In *Proceedings of COORDINATION'13*, LNCS 7890, pages 236–250. Springer, 2013.

[34] P. Wadler. Propositions as sessions. In *Proceedings of ICFP'12*, pages 273–286. ACM, 2012.