9

Type-Based Analysis of Linear Communications

Luca Padovani

Dipartimento di Informatica, Università di Torino, Italy

Abstract

This chapter presents a tool called Hypha for the type-based analysis of processes that communicate on linear channels. We describe the specification language used to model the systems under analysis (Section 9.1) followed by the typing rules on which the tool is based in order to verify two properties of systems, *deadlock freedom* and *lock freedom* (Section 9.2). In the final part of the chaper we illustrate the expressiveness and the limitations of the tool discussing a number of examples inspired by representative communication patterns using in parallel computing (Section 9.3) and then discuss closely related work (Section 9.4). The tool can be downloaded from the author's home page, the type system has been described by Padovani [18] and the corresponding reconstruction algorithms by Padovani *et al.* [19, 20].

9.1 Language

The Hypha specification language is a mildly sugared variant of the linear π-calculus [16] whose grammar is shown in Table 9.1. It makes use of booleans, integers, an infinite set $\mathbb{X}$ of names, and comprises *expressions* and *processes*. The syntax shown here is somewhat simplified and tailored to the modeling of the examples discussed in this chapter. Hypha supports other forms that may be useful in the description of protocols with branching points and provide convenient syntactic sugar on top of those given in Table 9.1. The Hypha specification language is appropriate for modeling concurrent processes that exchange messages on private (or *session*) channels [10].

Table 9.1 Syntax of `Hypha` input language (partial)

Notation	b	$\in$	$\{\texttt{true}, \texttt{false}\}$	Booleans
	h, k, m, n	$\in$	$\mathbb{Z}$	Integers
	x, y, z, u, v	$\in$	$\mathbb{X}$	Names
Expression	e	$::=$	b	Boolean constant
		$\mid$	n	Integer constant
		$\mid$	u	Name
		$\mid$	$\cdots$	
Process	P, Q	$::=$	$\texttt{\{ \}}$	Idle process
		$\mid$	$u\texttt{!(}e_1, \dots, e_n\texttt{)}$	Output
		$\mid$	$u\texttt{?(}x_1, \dots, x_n\texttt{).}P$	Input
		$\mid$	$\texttt{*}u\texttt{?(}x_1, \dots, x_n\texttt{).}P$	Replicated input
		$\mid$	$\texttt{new}\ u_1, \dots, u_n\ \texttt{in}\ P$	Channel creation
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional execution
		$\mid$	$P \texttt{ \| } Q$	Parallel composition
		$\mid$	$\texttt{\{}\,P\,\texttt{\}}$	Grouping

For simplicity, in the provided syntax expressions are limited to values and comprise booleans, integers, and names. In the examples we will also make use of a few binary operators (such as +) and relations (such as <). Processes comprise the usual terms of the π-calculus. The term `{ }` models the idle process that performs no actions. The term $u\texttt{!(}e_1, \dots, e_n\texttt{)}$ models a process that outputs the tuple $\texttt{(}e_1, \dots, e_n\texttt{)}$ on the channel u. We omit the parentheses when n is 1 and write, for example, $u\texttt{!}n$ in place of $u\texttt{!(}n\texttt{)}$. We consider two forms of input processes. The term $u\texttt{?(}x_1, \dots, x_n\texttt{).}P$ models an *ephemeral input process* that waits for *one* message from u, which is supposed to be an n-tuple, and then executes P where x_i is replaced by the i-th component of the tuple. The term $\texttt{*}u\texttt{?(}x_1, \dots, x_n\texttt{).}P$ models a *persistent input process* (also called *service*) that waits for an arbitrary number of messages. Each time a message is received, a new copy of P (with the variables x_i suitably instantiated) is spawned and the service makes itself available again for further receptions. The term $\texttt{new}\ u_1, \dots, u_n\ \texttt{in}\ P$ models a process creating new channels $u_1, \dots, u_n$ with scope P. As usual in the π-calculus, the scope of a channel may broaden as a result of communications (scope extrusion). The terms $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ and $P \mid Q$ respectively model conditional and parallel processes P and Q. Finally, $\texttt{\{}P\texttt{\}}$ represents the same process as P and is useful to disambiguate the way processes are

```
*fibo?(n,c).
 if n ≤ 0 then c!1
 else new a,b in { fibo!(n-1,a) | fibo!(n-2,b) | a?(x).b?(y).c!(x+y) }
```

Listing 9.1 Modeling of the recursive Fibonacci function.

grouped. The notions of *free* and *bound names* of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are as expected.

Example 1 (recursive Fibonacci function)**.** Listing 9.1 shows the modeling of a service that computes the n-th number in the Fibonacci sequence. The service waits for invocations on channel `fibo`, each invocation consisting of the number n and a channel c on which the n-th Fibonacci number will be sent. The body of the service closely follows the familiar structure of the recursive definition of the Fibonacci sequence. When $n \leq 0$ the answer is immediately sent over c. When $n > 0$, two new channels a and b are created, the service invokes itself twice to compute the $(n-1)$-th and $(n-2)$-th numbers in the sequence, and then the sum of the two partial results is sent over c. ■

As usual, the operational semantics of the language is defined in terms of a *structural congruence relation*, which identifies terms that are meant to have the same semantics, and a *reduction relation* that defines the proper computation steps. We omit the formal definition of structural congruence, which is essentially the same as in the π-calculus and includes commutativity and associativity laws for parallel composition and the usual laws for shrinking and extending the scope of channels. The second one is the least relation defined by the rules in Table 9.2 and closed by structural congruence and under the following *reduction contexts*:

Reduction context $\quad C \quad ::= \quad [\,] \quad | \quad C \mid P \quad | \quad \mathtt{new}\ u_1,\dots,u_n\ \mathtt{in}\ C$

The fully-fledged formalization of the language also includes an evaluation relation for compound expressions [18].

To formulate the properties enforced by our typing discipline we introduce a few predicates that describe the pending communications of a process

Table 9.2 Operational semantics of processes

$u!(e_1,\dots,e_n) \mid u?(x_1,\dots,x_n).P$	$\longrightarrow$	$P\{e_1/x_1\}\cdots\{e_n/x_n\}$
$u!(e_1,\dots,e_n) \mid {*u?(x_1,\dots,x_n).P}$	$\longrightarrow$	$P\{e_1/x_1\}\cdots\{e_n/x_n\} \mid {*u?(x_1,\dots,x_n).P}$
$\mathtt{if\ true\ then}\ P\ \mathtt{else}\ Q$	$\longrightarrow$	P
$\mathtt{if\ false\ then}\ P\ \mathtt{else}\ Q$	$\longrightarrow$	Q

P with respect to some channel a. We use the obvious extension of bound names we have introduced for processes to reduction contexts:

$$\begin{aligned}
\mathsf{in}(a,P) &\overset{\text{def}}{\iff} P = C[a?\texttt{(}x_1,\dots,x_n\texttt{)}\,\texttt{.}\,Q] \land a \notin \mathsf{bn}(C)\\
{}^*\mathsf{in}(a,P) &\overset{\text{def}}{\iff} P = C[{}^*a?\texttt{(}x_1,\dots,x_n\texttt{)}\,\texttt{.}\,Q] \land a \notin \mathsf{bn}(C)\\
\mathsf{out}(a,P) &\overset{\text{def}}{\iff} P = C[a!\texttt{(}e_1,\dots,e_n\texttt{)}] \land a \notin \mathsf{bn}(C)\\
\mathsf{sync}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \lor {}^*\mathsf{in}(a,P)) \land \mathsf{out}(a,P)\\
\mathsf{wait}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \lor \mathsf{out}(a,P)) \land \neg\mathsf{sync}(a,P)
\end{aligned}$$

In words, $\mathsf{in}(a,P)$ holds if there is a sub-process Q within P that is waiting for a message on channel a. Note that, by definition of reduction context, the input cannot be guarded by other actions. The condition $a \notin \mathsf{bn}(C)$ implies that a is not captured by a binder in C, *i.e.* it occurs free in P. The predicates $\mathsf{out}(a,P)$ and ${}^*\mathsf{in}(a,P)$ are similar, but they regard outputs and persistent inputs, respectively. Therefore, when $\mathsf{in}(a,P)$ holds it means that there is a pending ephemeral input on a and when $\mathsf{out}(a,P)$ holds it means that there is a pending output on a. Then, $\mathsf{sync}(a,P)$ means that there are pending input/output operations on a, but a synchronization on a is *immediately* possible. On the contrary, $\mathsf{wait}(a,P)$ means that there is a pending output or a pending ephemeral input on a, but no immediate synchronization on a is possible. Note the asymmetry in the way pending inputs and outputs trigger the wait predicate. We do not interpret ${}^*\mathsf{in}(a,P)$ as a pending input operation, meaning that we do not require a persistent input process to run infinitely often. At the same time, *any* pending output triggers the wait predicate, even when the output represents a service invocation.

We say that a process P is deadlock free if every residual of P that cannot reduce further contains no pending communications. Formally:

Definition 9.1. P is *deadlock free* if whenever $P \longrightarrow^* \texttt{new}\, c_1,\dots,c_n\, \texttt{in}\, Q \not\longrightarrow$ we have $\neg\mathsf{wait}(a,Q)$ for every a.

We say that a process P is lock free if every residual Q of P in which there are pending communications can reduce further to a state in which such operations complete. Formally:

Definition 9.2. P is *lock free* if whenever $P \longrightarrow^* \texttt{new}\, c_1,\dots,c_n\, \texttt{in}\, Q$ and $\mathsf{wait}(a,Q)$ there is R such that $Q \longrightarrow^* R$ and $\mathsf{sync}(a,R)$.

In Definitions 9.1 and 9.2, it is important to universally quantifiy over the topmost channel restrictions in a residual of P so that the notion of (dead)lock freedom for P concerns both free and bound channels of P.

It is easy to prove that lock freedom implies deadlock freedom [19]. On the other hand, there exist deadlock-free processes that are not lock free, as shown in the example below.

Example 2 (deadlock-free, locked process)**.** The process

```
new c in { *forever?(x).forever!x | forever!c | c!42 }
```

is deadlock free but not lock free. Indeed, the process reduces forever, but no input operation is ever performed on the c channel. As a result, the pending output on c cannot be completed. ■

9.2 Type System

In this section we describe a type system that enforces the properties introduced in Section 9.1: well-typed processes are guaranteed to be (dead)lock free. The tool Hypha then implements a type reconstruction algorithm for this type system and finds a typing derivation for a given process, provided there is one. Note that, while the type reconstruction algorithm is complete with respect to the type system, the type system itself is not complete with respect to (dead)lock freedom: there exist (dead)lock free processes that are ill typed according to the type system. In fact, it is undecidable in general to establish whether a π-calculus process is (dead)lock free, hence the type system is necessarily conservative.

Polarities, qualifiers, and types are defined by the following grammar:

$$\begin{array}{rrcl}
\textbf{Polarity} & p, q & \subseteq & \{?, !\} \\
\textbf{Qualifier} & \mathfrak{q} & ::= & * \mid {}^{h}_{k} \\
\textbf{Type} & t, s & ::= & \texttt{bool} \mid \texttt{int} \mid \kappa[\bar{t}]^{\mathfrak{q}} \mid \alpha \mid \mu\alpha\,\texttt{.}\,t
\end{array}$$

Types comprise the base types `bool` and `int` of boolean and integer values, channel types $\kappa[\bar{t}]^{\mathfrak{q}}$, and the usual forms α and $\mu\alpha\,\texttt{.}\,t$ for representing recursive types. A channel type $\kappa[\bar{t}]^{\mathfrak{q}}$ consists of:

- A *polarity* p specifying the operations allowed on the channel: $\emptyset$ means none, $\{?\}$ means input, $\{!\}$ means output, and $\{?, !\}$ means both. We will abbreviate $\{?, !\}$ with # and $\{?\}$ and $\{!\}$ with ? and !, respectively.
- A sequence $t_1, \ldots, t_n$ of types, abbreviated as $\bar{t}$, specifying that each message exchanged over the channel is an n-tuple of values where the i-th value has type t_i.
- A *qualifier* $\mathfrak{q}$ specifying how many times the channel can or must be used according to its polarity. The qualifier * means that the channel can

be used any number of times. A qualifier of the form $^{h}_{k}$ means that the channel can only be used once. In this case, h and k are respectively the *level* and the *tickets* associated with the channel: channels with smaller levels must be used *before* channels with greater levels; a channel with k tickets can be sent as a message on another channel at most k times.

We require that, in a recursive type $\mu\alpha\texttt{.}t$, the type variable α can only occur guarded by a channel type constructor. For example, $\mu\alpha\texttt{.}\alpha$ is illegal while $\mu\alpha\texttt{.?[}\alpha\texttt{]}^{*}$ is allowed. 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 [4]. In particular, we have $\mu\alpha\texttt{.}t = t\{\mu\alpha\texttt{.}t/\alpha\}$.

Qualifiers distinguish *service channels* (with qualifier *) from *linear channels* (with qualifiers of the form h, k). Service channels are used for modeling persistent services, such as `fibo` in Listing 9.1 and `forever` in Example 2. Linear channels are used for modeling private communications between pairs of processes. Examples of linear channels are a and b in Listing 9.1 and c in Example 2. The fact that a linear channel can be used for one communication only is not a limitation in practice. Structured private conversations made of arbitrarily many communications can be encoded using a continuation-passing style [5, 12]. We will see several examples of this technique at work in the rest of the chapter. On the other hand, knowing that a channel is linear provides some guarantees on the fact that the channel will not be discarded without being used. This is a necessary (although not sufficient) condition for guaranteeing that communications on linear channels eventually occur.

The level of a linear channel measures the urgency with which the channel must be used: the lower the level is, the sooner the channel must be used. We extend this notion from linear channels to arbitrary types. 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 $n \in \mathbb{Z}$:

$$|t| \overset{\text{def}}{=} \begin{cases} \bot & \text{if } t = p\texttt{[}s\texttt{]}^{*} \text{ and } ? \in p \\ n & \text{if } t = p\texttt{[}s\texttt{]}^{n}_{m} \text{ and } p \neq \emptyset \\ \top & \text{otherwise} \end{cases} \tag{9.1}$$

According to this definition, service channels with input capability have the lowest level $\bot$ (first equation). This way we guarantee input receptiveness of services, for the use of a service channel with input capability cannot be postponed by any means. Base values, service channels with output capability, and linear channels with no capabilities have the highest level $\top$ (last equation) because they do not affect (dead)lock freedom in any way. Linear channels with non-empty polarity must be used according to their level (second equation). 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 $\$_k^h$ to *shift* levels and tickets: $\$_k^h\, t$ is the same as t except when t is a linear channel. In this case, the level/tickets in t are transposed by h and k respectively. Formally:

$$\$_k^h\, t \overset{\text{def}}{=} \begin{cases} p[s]_{m+k}^{n+h} & \text{if } t = p[s]_m^n \text{ and } p \neq \emptyset \\ t & \text{otherwise} \end{cases} \tag{9.2}$$

Note that positive/negative shifting of levels corresponds to decreasing/ increasing the urgency with which a value of a given type must be used.

The type system makes use of *type environments* Γ to keep track of the type of names occurring free in processes. A type environment is a finite map 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 $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. We also need a more general way of composing type environments that takes into account the level and tickets of linear channel types and the fact that we can split channel types by distributing different capabilities to different processes. Following [16], we define a partial operator + between types, thus:

$$\begin{array}{rll} t + t = t & & \text{if } t \text{ is unlimited} \\ p[t]^* + q[t]^* = (p \cup q)[t]^* & & \\ p[t]_h^n + q[t]_k^n = (p \cup q)[t]_{h+k}^n & & \text{if } p \cap q = \emptyset \end{array} \tag{9.3}$$

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

$$\begin{array}{rll} \Gamma + \Gamma' \overset{\text{def}}{=} \Gamma, \Gamma' & & \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset \\ (\Gamma, u : t) + (\Gamma', u : s) \overset{\text{def}}{=} (\Gamma + \Gamma'), u : t + s & & \end{array} \tag{9.4}$$

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 let $|\Gamma|$ denote the lowest level of the types in the range of Γ, that is

$$|\Gamma| \overset{\text{def}}{=} \min\{|\Gamma(u)| \mid u \in \mathsf{dom}(\Gamma)\} \tag{9.5}$$

We say that Γ is unlimited if $|\Gamma| = \top$.

We now turn our attention to the typing rules, which are meant to enforce the following properties of channels:

1. a service channel with input capability must be used by a replicated input process (we refer to this condition as *input receptiveness*);
2. a linear channel cannot be discarded until both its input and output capabilities have been used (we refer to this condition as *linearity*);
3. an operation on a linear channel cannot block channels with lower or equal level (with linearity, this condition guarantees *deadlock freedom*);
4. the use of a linear channel cannot be postponed forever (with linearity and deadlock freedom, this condition guarantees *lock freedom*).

The typing rules allow us to derive judgments of the form $\Gamma \vdash e : t$, stating that e is well typed in the environment Γ and has type t, and of the form $\Gamma \vdash_k P$, stating that P is well typed in the environment Γ. The parameter $k \in \{0, 1\}$ intuitively represents the "cost" for sending a channel over another channel: each output operation consumes k tickets from the channels being sent. The type system is designed in such a way that a well typed, closed process P is guaranteed to be deadlock free if $\emptyset \vdash_0 P$ and lock free if $\emptyset \vdash_1 P$.

Expressions. Because the model has an extremely simple expression language, the corresponding typing rules, shown below, are fairly obvious and extend easily to more complex expressions:

$$\begin{array}{lll} \text{[T-BOOL]} & \text{[T-INT]} & \text{[T-NAME]} \\ \emptyset \vdash b : \mathtt{bool} & \emptyset \vdash n : \mathtt{int} & u : t \vdash u : t \end{array}$$

The important remark concerning these rules is that the type environment used for typing an expression e only contains associations for the free names occurring in e. This makes sure that no linear or relevant resource (namely, channels that must be used at least once) is left unused. Later on we will discuss a structural rule that allows us to discard resources whose use is not necessary in order to enforce (dead)lock freedom.

Idle and grouped processes. Rule [T-IDLE] states that the idle process is well typed in the empty environment only:

$$\text{[T-IDLE]} \qquad \emptyset \vdash_k \texttt{\{ \}}$$

For example, the judgment $a : \texttt{![int]}^n_m \vdash_k \texttt{\{ \}}$ is not derivable, because the linear channel a is supposed to be used for one output operation, whereas the process `{ }` does nothing. This typing rule illustrates a key trait of the type system, making sure that linear channels with pending capabilities are not discarded. If this were not the case, one could write processes like

$$\texttt{new}\ a\ \texttt{in}\ a\texttt{?}(x)$$

which are stuck waiting for messages that will never arrive.

The typing rule for a grouped process is simple and does not enforce any constraint other than typability of the process itself:

$$[\textsc{t-group}] \qquad \frac{\Gamma \vdash_k P}{\Gamma \vdash_k \texttt{\{}\, P\, \texttt{\}}}$$

Outputs. Rule [T-OUT] is used for typing output operations on linear channels:

$$[\textsc{t-out}] \qquad u : \texttt{![}t\texttt{]}^m_n + v : \$^n_k\, t \vdash_k u\texttt{!}v \qquad 0 < |t|$$

First of all, the channel u being used for the output must indeed have capability `!`. The type of the message v must be t (as specified in the type of the channel u) except that its level is shifted by m and its tickets are shifted by k. The shifting of the level means that the level of t in $\texttt{![}t\texttt{]}^m_n$ is relative to m. This, together with the side condition $0 < |t|$, makes sure that the level of v (the channel being sent as a message) is strictly greater than the level of u. The shifting of the tickets in t accounts for the fact that, by sending v as a message, one ticket from v is consumed. Note that this is necessary only in the judgments for lock freedom ($k = 1$). Below are a few examples:

- The judgment $a : \texttt{![?[int]}^1_0\texttt{]}^2_0, b : \texttt{?[int]}^3_1 \vdash_1 a\texttt{!}b$ is derivable because $\texttt{?[int]}^3_1 = \$^2_1\,\texttt{?[int]}^1_0$. 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 $a : \texttt{![![int]}^1_0\texttt{]}^0_0, b : \texttt{![int]}^1_0 \vdash_1 a\texttt{!}b$ is not derivable because b has no tickets and so it cannot travel on a.
- Let $t = \texttt{?[}t\texttt{]}^0_0$ and observe that $\texttt{\#[}t\texttt{]}^1_0 = \texttt{![}t\texttt{]}^1_0 + \texttt{?[}t\texttt{]}^1_0$. The judgment $a : \texttt{\#[}t\texttt{]}^1_0 \vdash_0 a\texttt{!}a$ is not derivable, despite the message a has the “right” type $\texttt{?[}t\texttt{]}^1_0 = \$^1_0\, t$, because the condition $0 < |t| = 0$ is not satisfied. A process like $a\texttt{!}a$ is deadlocked because the occurrence of a that is meant to be used for matching this output operation is the very message sent on a itself.

- The judgment $a : \texttt{![?[int]}^*\texttt{]}^0_0, b : \texttt{?[int]}^* \vdash_0 a!b$ is not derivable because $0 \not< |\texttt{?[int]}^*| = \bot$. A service channel with input capability such as b cannot be sent as a message to guarantee input receptiveness.

Rule [T-OUT*] is used for typing outputs on unlimited channels.

$$[\text{T-OUT*}] \qquad u : \texttt{![}t\texttt{]}^* + v : \$^n_k\, t \vdash_k u!v \qquad \bot < |t|$$

There are two key differences between [T-OUT] and [T-OUT*]. First, the condition $\bot < |t|$, where t is the type of v means that only unlimited channels with input capability cannot be communicated. Second, the type of v does not need to match exactly the type t in the channel type of u, 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 often use this feature in the extended examples in the second half of this chapter.

Both [T-OUT] and [T-OUT*] generalize easily to an arbitrary number of message arguments. As an example, the former rule can be generalized as follows:

$$u : \texttt{?[}t_1, \dots, t_h\texttt{]}^m_n + v_1 : \$^m_k\, t_1 + \cdots + v_h : \$^m_k\, t_h \vdash_k u!(v_1, \dots, v_h) \qquad 0 < |t_i|$$

Inputs. Rule [T-IN] is used for typing linear input operations:

$$[\text{T-IN}] \qquad \frac{\Gamma, x : \$^n_0\, t \vdash_k P}{\Gamma + u : \texttt{?[}t\texttt{]}^n_m \vdash_k u?(x).P} \qquad n < |\Gamma|$$

The channel u must have type $\texttt{?[}t\texttt{]}^n_m$ and 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. The level of the type of x is shifted by n, consistently with the relative interpretation of levels of message types that we have already discussed for [T-OUT]. The tickets of u are irrelevant since u is used for an input operation, not as the content of a message. The condition $n < |\Gamma|$ ensures that the input on u does not block operations on other channels with equal or lower level. In particular, Γ cannot contain service channels with input capability. Below are some typical examples of ill-typed processes that violate this condition:

- $a : \texttt{?[int]}^1_0, b : \texttt{![int]}^0_0 \vdash_k a?(x).b!x$ is not derivable because $1 \not< 0$: the input on a blocks the output on b, but b has lower level than a;

- $a : \texttt{\#[int]}_0^h \vdash_k a\texttt{?(}x\texttt{)}.a\texttt{!}x$ 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 [16].
- $a : \texttt{?[int]}_0^h, c : \texttt{?[int]}^* \vdash_k a\texttt{?(}x\texttt{)}.{*c}\texttt{?(}y\texttt{)}$ is not derivable because $|\texttt{?[int]}^*| = \bot$. To guarantee input receptiveness, we require that replicated inputs cannot be guarded by other operations.

Rule [T-IN*] is used for typing replicated input operations corresponding to persistent services:

$$[\text{T-IN*}] \qquad \frac{\Gamma, x : t \vdash_k P}{\Gamma + u : \texttt{?[}t\texttt{]}^* \vdash_k {*u}\texttt{?(}x\texttt{)}.P} \qquad \top \le |\Gamma|$$

This rule differs from [T-IN] in three important ways. First of all, u must be a service channel with input capability. Second, the side condition $\top \le |\Gamma|$ makes sure that the environment Γ used for typing the body of the service is unlimited. This is because the service can be invoked an arbitrary number of times, hence its body cannot contain linear resources. Third, it may be the case that $u \in \mathsf{dom}(\Gamma)$, because $\texttt{?[}t\texttt{]}^* + \texttt{![}t\texttt{]}^* = \texttt{\#[}t\texttt{]}^*$ according to (9.3) and $\texttt{![}t\texttt{]}^*$ is unlimited. This means that services may recursively invoke themselves. We use this feature in several examples, including Example 1.

As for [T-OUT] and [T-OUT*], both [T-IN] and [T-IN*] can be easily generalized to handle arbitrary tuples of message arguments.

Conditional and parallel processes. The typing rule for conditional processes is shown below:

$$[\text{T-IF}] \qquad \frac{\Gamma_1 \vdash e : \texttt{bool} \qquad \Gamma_2 \vdash_k P \qquad \Gamma_2 \vdash_k Q}{\Gamma_1 + \Gamma_2 \vdash_k \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q}$$

As usual, the condition must have type `bool` and + is used for combining the type environments used in different parts of the process. Note that both branches must be typable using the same type environment, meaning that the linear channels occurring in P and Q must be used in the same order. For example, the judgment

$$a : \texttt{?[int]}_0^1, b : \texttt{?[int]}_0^2 \vdash_k \texttt{if}\ e\ \texttt{then}\ a\texttt{?(}x\texttt{)}.b\texttt{?(}y\texttt{)}\ \texttt{else}\ b\texttt{?(}y\texttt{)}.a\texttt{?(}x\texttt{)}$$

is not derivable because the `else` branch uses the two linear channels a and b in an order not allowed by their levels.

The typing rule for parallel compositions is shown below:

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

Because of the definition of the + operator, which combines the types of linear channels only provided that such channels have the same level, the order in which linear channels are used in the branches of the parallel composition must be consistent. For example, the judgment

$$a : \#[\mathtt{int}]_0^1, b : \#[\mathtt{int}]_0^2 \vdash_k a?(x).b!x \mid b?(y).a!y$$

cannot be derived because the second branch violates the side condition of [T-IN] requiring b to have a strictly smaller level than a. Indeed, the whole process is deadlocked.

Channel creation. Restrictions can be used to introduce both linear and service channels. In the former case, the typing rule is

$$[\text{T-NEW}] \qquad \frac{\Gamma, a : p[t]_m^n \vdash_k P}{\Gamma \vdash_k \mathtt{new}\ a\ \mathtt{in}\ P} \qquad p \in \{\mathbb{0}, \#\}$$

Note that the rule "guesses" the right level and number of tickets that are necessary for typing P. The polarity is either #, meaning that a must be used for both one input and one output operation in P, or $\mathbb{0}$, meaning that a is a depleted channel that is not supposed to be used at all in P. The reason why the typing rule accounts for this possibility is purely technical and is necessary to prove that process reductions preserve typing [18].

The typing rule for introducing a service channel is essentially the same:

$$[\text{T-NEW*}] \qquad \frac{\Gamma, a : \#[t]^* \vdash_k P}{\Gamma \vdash_k \mathtt{new}\ a\ \mathtt{in}\ P}$$

Unlike linear channels, the capability of restricted unlimited channels is always #. Since an unlimited channel a with input capability must be used (*cf.* [T-IN*]), this guarantees that there is always a service waiting for invocations on a. On the other hand, a service channel with output capability does not have to be used (*cf.* (9.1)), therefore imposing that the capability of a is # does not mandate invocations on a.

Unused resources. The following structural rule provides a limited form of weakening whereby it is possible to add unused resources in a type environment, provided that these resources have an unlimited type:

$$[\textsc{t-weak}] \qquad \frac{\Gamma \vdash_k P}{\Gamma + \Gamma' \vdash_k P} \qquad \top \leq |\Gamma'|$$

For example, both $a : \emptyset\texttt{[int]}^m_n \vdash_k \texttt{\{ \}}$ and $x : \texttt{int} \vdash_k \texttt{\{ \}}$ are derivable, because the type environments only contain resources that impose no usage and therefore can be discarded using [T-WEAK]. On the contrary, neither $a : \texttt{![int]}^m_n \vdash_k \texttt{\{ \}}$ nor $a : \texttt{?[int]}^* \vdash_k$ are derivable.

The type system refines the one for the linear π-calculus [16], hence all the properties of the linear π-calculus (partial confluence, linear usage of channels, etc.) are still guaranteed. The added value is that the type system also guarantees deadlock/lock freedom.

Theorem 9.3. *The following properties hold:*

1. *If $\emptyset \vdash_0 P$, then P is deadlock free.*
2. *If $\emptyset \vdash_1 P$, then P is lock free.*

Example 3 (recursive Fibonacci function)**.** Let P be the process shown in Listing 9.1. Then it is possible to derive

$$\texttt{fibo} : \texttt{\#[int, ![int]}^0_0\texttt{]}^* \vdash_k P$$

if and only if $k = 0$. It is not possible to find a derivation for $k = 1$ since the type system cannot establish an upper bound to the time after which the continuation channel c will be used in an invocation `fibo!(n,c)`. In fact, such upper bound depends on n and on the fact that the recursion of the `fibo` service is well-founded This latter property requires a kind of analysis that eludes the capabilities of the type system. ■

9.3 Extended Examples

9.3.1 Fibonacci Stream Network

In this section we discuss an alternative modeling of system that computes the sequence of Fibonacci numbers and that is an example of *stream network*, that is a network of communicating processes that exchange infinite

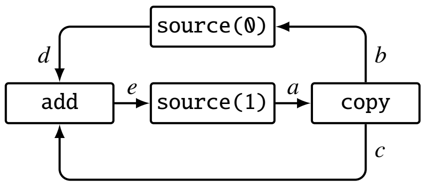

Figure 9.1 Graphical representation of the Fibonacci stream network [8, 22].

sequences (streams) of messages. Figure 9.1 depicts the Fibonacci stream network [8, 22] where the boxes represent processing units and the arrows represent communication channels.

Each `source`(n) process sends n on the outgoing channel followed by each message received from the incoming channel. The `copy` process forwards each received message on each of its two outgoing channels. Finally, `add` sends on the outgoing channel the sum of corresponding messages received from the two incoming channels. Overall, it is easy to see that the stream of messages flowing on channel a corresponds to the Fibonacci sequence $1, 1, 2, 3, 5, \ldots$.

The modeling of the Fibonacci stream network in `Hypha`'s input language is shown in Listing 9.2. There is a service for each of the boxes in Figure 9.1, with `source` that makes use of an auxiliary service `link` that acts as a persistent message forwarder. The network itself is created on line 5, where the services are invoked and connected by the channels a through e. The most distinctive aspect of the modeling is the use of continuation passing for the representation of message streams: each channel that connects two combinators is in fact a linear channel (a channel that is meant to be used for one communication only); whenever a message is exchanged on the channel, the payload is paired with a fresh (linear) channel on which the subsequent message will be exchanged. This pattern can be clearly observed in the definitions of `link`, `add`, and `copy`. To improve readability, hereafter we write $\bar{x}$ for a channel name that is meant to represent the continuation of x.

```
{ *link?(x,y).x?(v,x̄).source!(v,x̄,y)
| *source?(n,x,y).new ȳ in { y!(n,ȳ) | link!(x,ȳ) }
| *add?(x,y,z).x?(v,x̄).y?(w,ȳ).new z̄ in { z!(v+w,z̄) | add!(x̄,ȳ,z̄) }
| *copy?(x,y,z).x?(v,x̄).new ȳ,z̄ in { y!(v,ȳ) | z!(v,z̄) | copy!(x̄,ȳ,z̄) }
| source!(1,e,a) | copy!(a,b,c) | source!(0,b,d) | add!(d,c,e) }
```

Listing 9.2 Term representation of the Fibonacci stream network [8, 22].

Hypha infers the following types for the channels used in the system

$$\begin{aligned}
\texttt{source} &: \#[\texttt{int}, ?[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{2}]^{2}_{1}, ![\texttt{int}, \mu\beta.?[\texttt{int},\beta]^{3}_{1}]^{0}_{0}]^{*}\\
\texttt{link} &: \#[?[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{2}]^{0}_{0}, ![\texttt{int}, \mu\beta.?[\texttt{int},\beta]^{3}_{1}]^{1}_{1}]^{*}\\
\texttt{add} &: \#[?[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{1}]^{0}_{0}, ?[\texttt{int}, \mu\beta.?[\texttt{int},\beta]^{3}_{1}]^{1}_{0},\\
&\qquad ![\texttt{int}, \mu\gamma.?[\texttt{int},\gamma]^{3}_{2}]^{2}_{0}]^{*}\\
\texttt{copy} &: \#[?[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{1}]^{0}_{0},\\
&\qquad ![\texttt{int}, \mu\beta.?[\texttt{int},\beta]^{3}_{2}]^{2}_{0}, ![\texttt{int}, \mu\gamma.?[\texttt{int},\gamma]^{3}_{1}]^{1}_{0}]^{*}\\
a, d &: \#[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{1}]^{0}_{2}\\
b, e &: \#[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{2}]^{2}_{3}\\
c &: \#[\texttt{int}, \mu\alpha.?[\texttt{int},\alpha]^{3}_{1}]^{1}_{2}
\end{aligned}$$

confirming that the system is well typed and therefore lock free. In particular, Theorem 9.3(2) allows us to deduce that every number in the sequence of Fibonacci is computed in finite time. We make some observations concerning the inferred channel types: first, Hypha correctly distinguishes between the channels representing services (such as copy and source) from the linear channels that connect them (such as *a* and *b*). Second, the levels associated with linear channels give hints concerning the order of synchronizations in the system. The synchronizations on *a* and *d* (with level 0) happen first, followed by that on *c* (level 1), and then by that on *e* (level 2). Note however, that the total order on levels does not necessarily reflect the partial order that represents dependencies between channels. For example, *b* has a strictly greater level than *c* and yet the synchronizations on these two channels may happen in any order. Concerning the ticket annotations, note that all linear channels require at least 2 tickets because they are used to connect 2 services. For example, *a* connects source(1) and copy. By contrast, *b* and *e* need one more ticket because they are also forwarded by source to link.

9.3.2 Full-Duplex and Half-Duplex Communications

Many parallel algorithms use batteries of processes arranged in a grid that iteratively update array elements and communicate with processes assigned to neighbor elements (Figure 9.2). Processes may communicate according to one out of two modalities: when communication is *full-duplex*, processes simultaneously send messages to each other; when communication is *half-duplex*, only one message travels between two processes at any

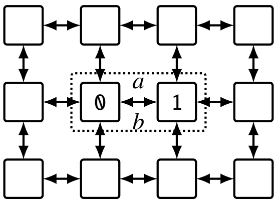

Figure 9.2 Graphical representation of a 4 × 3 bi-dimensional stencil.

moment in time. Correspondingly, we can model the dotted grid fragment in Figure 9.2 as

$$e\,!\,(0,a,b)\ \mid\ f\,!\,(1,b,a) \tag{9.6}$$

where e and f are service channels defined as either

$$*\texttt{full?}(n,x,y)\texttt{.new}\ \bar{x}\ \texttt{in}\ \{\ x\,!\,(n,\bar{x})\ \mid\ y?(m,\bar{y})\texttt{.full!}(n{+}m,\bar{x},\bar{y})\ \}$$

in case of full-duplex communication or as

$$*\texttt{half?}(n,x,y)\texttt{.}y?(m,\bar{y})\texttt{.new}\ \bar{x}\ \texttt{in}\ \{\ x\,!\,(n,\bar{x})\ \mid\ \texttt{half!}(n{+}m,\bar{x},\bar{y})\ \}$$

in case of half-duplex communication. In both cases, x is used for sending messages to, and y for receiving messages from, a neighbor process. Each message sent on x carries a payload n as well as a continuation channel $\bar{x}$ used for the communication at the next iteration. Symmetrically, each message received from y contains the neighbor's payload m and a continuation $\bar{y}$. The difference between `full` and `half` is that, in the latter case, the sender waits for the message from its neighbor before sending its own.

Overall there are 4 possible configurations of the system (9.6) obtained by instantiating e and f with either `full` or `half`. It is easy to see that a configuration is lock free as long as *at least* one of e or f is instantiated with `full`. Indeed, Hypha infers the types

$$a,b : \#[\texttt{int},\mu\alpha.?[\texttt{int},\alpha]^{1}_{1}]^{0}_{2}$$

when $e = f =$ `full` and the types

$$a : \#[\texttt{int},\mu\alpha.?[\texttt{int},\alpha]^{2}_{1}]^{1}_{2} \qquad b : \#[\texttt{int},\mu\alpha.?[\texttt{int},\alpha]^{2}_{1}]^{0}_{2}$$

when $e =$ `half` and $f =$ `full`. The case when $e =$ `full` and $f =$ `half` is symmetric, while the one when $e = f =$ `half` is ill typed for deadlock freedom and, therefore, for lock freedom as well.

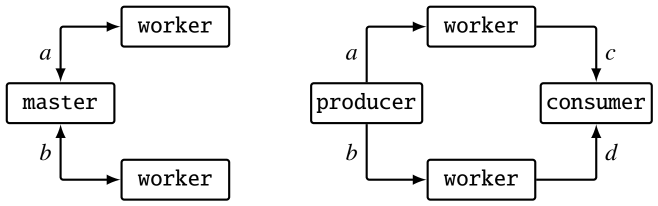

Figure 9.3 Master-worker (left) and producer-consumer (right).

9.3.3 Load Balancing

Figure 9.3 shows two network topologies aimed at taking advantage of parallelism by distributing multiple tasks to independent workers. They differ in that in the *master-worker* topology the same process that produces tasks is also the one that collects the results, whereas in the *producer-consumer* topology (sometimes called "farm") producer and consumer are different processes. The distinction between the two topologies has important consequences at the communication layer since the channels are bi-directional in the former network and uni-directional in the latter.

Listings 9.3 and 9.4 show the modeling of the network topologies in Figure 9.3, both of which are well-typed according to the lock freedom type system implemented in Hypha. For the master-worker network, Hypha infers the types

$$\begin{aligned} a &: \#[\mathtt{int}, \mu\alpha.\,![\mathtt{int}, ![\mathtt{int}, \alpha]^{2}_{1}]^{1}_{0}]^{0}_{2} \\ b &: \#[\mathtt{int}, \mu\alpha.\,![\mathtt{int}, ![\mathtt{int}, \alpha]^{2}_{1}]^{1}_{0}]^{1}_{2} \end{aligned}$$

```
{ *master?(n,x,y).
   new x̄,ȳ in { x!(n,x̄) | y!(n+1,ȳ) | x̄?(v,x̿).ȳ?(w,y̿).master!(n+2,x̿,y̿) }
| *worker?(n,z).z?(m,z̄).new z̿ in z̄!(m mod n,z̿).worker!(n,z̿)
| master!(0,a,b) | worker!(2,a) | worker!(3,b) }
```

Listing 9.3 Term representation of master-worker (half-duplex channels).

```
{ *producer?(n,x,y).new x̄,ȳ in { x!(n,x̄) | y!(n+1,ȳ) | producer!(n+2,x̄,ȳ) }
| *consumer?(x,y).x?(v,x̄).y?(w,ȳ).{ print!v | print!w | consumer!(x̄,ȳ) }
| *worker?(n,x,y).x?(m,x̄).new ȳ in { y!(m mod n,ȳ) | worker!(n,x̄,ȳ) }
| producer!(0,a,b) | worker!(2,a,c) | worker!(3,b,d) | consumer!(c,d) }
```

Listing 9.4 Term representation of producer-consumer.

which describe a communication protocol whereby the master sends a task (represented as an integer number) to each worker along with a continuation channel. By using this continuation channel, the worker will answer back with the processed task (again represented as an integer number) and another continuation that the master uses for starting another iteration.

For the producer-consumer network Hypha infers the types

$$\begin{array}{lllllll} a & : & \#[\mathtt{int}, \mu\alpha.?[\mathtt{int}, \alpha]^{2}_{1}]^{0}_{2} & \qquad & c & : & \#[\mathtt{int}, \mu\alpha.?[\mathtt{int}, \alpha]^{2}_{1}]^{1}_{2} \\ b & : & \#[\mathtt{int}, \mu\alpha.?[\mathtt{int}, \alpha]^{2}_{1}]^{1}_{2} & & d & : & \#[\mathtt{int}, \mu\alpha.?[\mathtt{int}, \alpha]^{2}_{1}]^{2}_{2} \end{array}$$

again confirming that the network is lock free.

9.3.4 Sorting Networks

Figure 9.4 depicts an example of so-called *sorting network*, that is a network of communicating processes whose overall effect is that of sorting an input vector of fixed size, 6 in this case. The network is made of two different kinds of processes: *comparators* (the rectangular boxes) input two values and possibly swap them if the first happens to be larger than the second; *buffers* (the square boxes) simply forward the input value. The input values go through three identical phases; in each stage, the odd-indexed inputs and then the even-indexed inputs are compared to, and possibly swapped with, their successor.

The sorting network in Figure 9.4 is modeled in the linear π-calculus as shown in Listing 9.5. Note the use of auxiliary services odd and even corresponding to the two sub-phases of each phase and linked together by

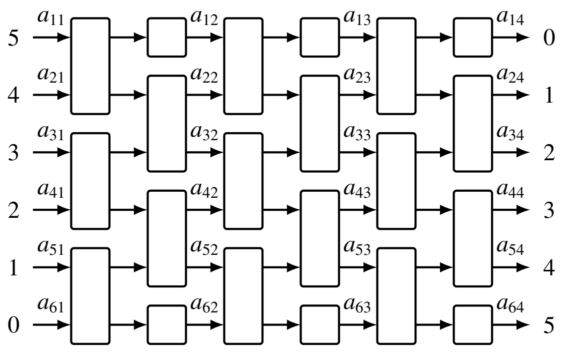

Figure 9.4 Graphical representation of an odd-even 6-input sorting network.

```
{ *compare?(x,y,l,h).
   new z in { x?(v).z!v | y?(w).z?(v).if v<w then l!v | h!w else l!w | h!v }
| *buffer?(x,y).x?(v).y!v
| *provide?(x,n).x!n
| *consume?(x,n).x?(v).print!(n,v)
| *even?(x1,x2,x3,x4,x5,x6,y1,y2,y3,y4,y5,y6).
   { compare!(x1,x2,y1,y2) | compare!(x3,x4,y3,y4) | compare!(x5,x6,y5,y6) }
| *odd?(x1,x2,x3,x4,x5,x6,y1,y2,y3,y4,y5,y6).
   { buffer!(x1,y1) | compare!(x2,x3,y2,y3)
   | buffer!(x6,y6) | compare!(x4,x5,y4,y5) }
| *phase?(x1,x2,x3,x4,x5,x6,y1,y2,y3,y4,y5,y6).new z1,z2,z3,z4,z5,z6 in
   { even!(x1,x2,x3,x4,x5,x6,z1,z2,z3,z4,z5,z6)
   | odd!(z1,z2,z3,z4,z5,z6,y1,y2,y3,y4,y5,y6) }
| phase!(a11,a21,a31,a41,a51,a61,a12,a22,a32,a42,a52,a62)
| phase!(a12,a22,a32,a42,a52,a62,a13,a23,a33,a43,a53,a63)
| phase!(a13,a23,a33,a43,a53,a63,a14,a24,a34,a44,a54,a64)
| provide!(a11,0) | provide!(a21,1) | provide!(a31,2)
| provide!(a41,3) | provide!(a51,4) | provide!(a61,5)
| consume!(a14,0) | consume!(a24,1) | consume!(a34,2)
| consume!(a44,3) | consume!(a54,4) | consume!(a64,5) }
```

Listing 9.5 Term representation of an odd-even 6-input sorting network.

restricted channels z_i. This network is well-typed and Hypha infers the types

$$\begin{array}{rcl}
\texttt{comparator} & : & \#[?[\texttt{int}]_0^0, ?[\texttt{int}]_0^0, ![\texttt{int}]_0^2, ![\texttt{int}]_0^2]^* \\
\texttt{buffer} & : & \#[?[\texttt{int}]_0^0, ![\texttt{int}]_0^2]^* \\
a_{ij} & : & \#[\texttt{int}]_2^{4(j-1)}
\end{array}$$

confirming that each value sent on a_{i1} is eventually received on some a_{j4}. More specifically, the level 12 assigned with the a_{j4} channels gives an upper bound to the number of synchronizations needed for producing the output.

Comparators input values in parallel (from the channels x and y) and perform an internal synchronization (on a private linear channel z) to join the results of the two receptions and output the results. Alternatively, one could model comparators in such a way that the receive operations on x and y are performed in a fixed order. The choice of a particular modeling affects the levels associated with the input channels, but not the typeability of the network as a whole. This is not the case for buffers: they are operationally irrelevant and are usually omitted in standard presentations of sorting networks. Their use in Listing 9.5 is key for the lock freedom analysis to succeed as they make sure that the levels of the channels connecting one phase to the next one remain aligned.

9.3.5 Ill-typed, Lock-free Process Networks

In general, the problem of verifying whether a π-calculus process is (dead)lock free is undecidable. For this reason, the type system on which `Hypha` is based is necessarily incomplete, in the sense that there exist processes satisfying Definitions 9.1 and 9.2 which are ill typed according to the type system described in Section 9.2. In this section, we discuss two representative examples of processes that cannot be handled by our type system. In all cases, the inability to find a typing derivation is tightly related to the fact that the type system uses integer numbers for reasoning on the dependencies between linear channels and such numbers measure the (abstract) moment of time at which the synchronization occurs on these channels.

Listing 9.6 shows a process that computes the sequence of prime numbers. The process is modeled after Eratosthenes' sieve: the `from` process emits the infinite stream of natural numbers starting from 2; the sequence goes through a growing pipeline of `filter`s, each filter removing those numbers of the sequence that happen to be a multiple of a given prime number m; if a number n manages to cross the entire pipeline and hits the emitter process `output`, then it is prime. In this case n is sent on `print` and a new filter removing the multiples of n is inserted at the end of the pipeline. `Hypha` is able to distinguish linear from service channels and to infer the type of messages exchanged therein, but the process is ill-typed for deadlock freedom even though it is deadlock free. The problem can be traced to the body of `filter`: when the received number n turns out to be a multiple of m, the number is simply discarded and no output is sent on y. So, the recursive invocation of `filter` on line 3 reuses the same output channel y that was received as input. Observe that $\bar{x}$ is received from x, meaning that the level of $\bar{x}$ must necessarily be greater than the level of x, and that the level of $\bar{x}$ must be strictly smaller than the level of y, since the input on performed on $\bar{x}$ at the next iteration of `filter` blocks the possible output on y. Given that the distribution of prime numbers is irregular, there is no upper bound to the number of inputs on x that may be necessary before the next output

```
1  { *from?(n,x).new x̄ in { x!(n,x̄) | from!(n+1,x̄) }
2  | *filter?(m,x,y).x?(n,x̄).{ if n mod m = 0 then filter!(n,x̄,y)
3                               else new ȳ in { y!(n,ȳ) | filter!(m,x̄,ȳ) } }
4  | *output?(x).x?(n,x̄).{ print!n | new y in { filter!(n,x̄,y) | output!y } }
5  | from!(2,a) | output!a }
```

Listing 9.6 Stream Network computing the sequence of prime numbers.

on y is guaranteed to be performed. In general, the type system can handle those cases in which communications occur following a regular pattern that is independent of the content of messages themselves.

The second example we consider is a process stream network (Figure 9.5) that computes the so-called Thue-Morse sequence, that is the sequence of binary digits $011010011001\cdots$ starting with 0 and obtained by appending the boolean complement of the sequence obtained thus far. The term representation of the process network (Listing 9.7) is modeled after its definition in terms of lazy streams [7] and makes use of a set of combinators some of which we have already used for the Fibonacci stream network (Section 9.3.1). The network is lock-free, as witnessed by the fact that the corresponding lazy stream definition can be shown to be productive [7], but also ill typed for deadlock freedom and hence for lock freedom as well. In this network the problematic combinator is `zip`, which interleaves on the output channel f the digits received from the input channels d and e hence producing messages on f at twice the rate at which they are consumed from d and e. This means that there is no fixed offset between the levels of d and e and that of f that could be dealt with by the typing rule [T-OUT*]. Note that this phenomenon does not manifest in the Fibonacci stream network (Figure 9.1) despite its seemingly similar topology. The key difference is that in the Fibonacci network the `add` process *combines* the messages received from the two input channels into a single message sent on the output channel. The example is interesting also

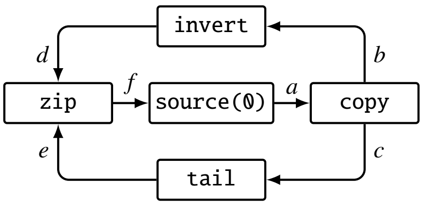

Figure 9.5 Stream network computing the Thue-Morse sequence.

```
{ *zip?(x,y,z).x?(u,x̄).new z̄ in { z!(u,z̄) | zip!(y,x̄,z̄) }
| *invert?(x,y).x?(u,x̄).new ȳ in { y!(1-u,ȳ) | invert!(x̄,ȳ) }
| *tail?(x,y).x?(_,x̄).link!(x̄,y)
| *link?(x,y).x?(v,x̄).source!(v,x̄,y)
| *copy?(x,y,z).x?(v,x̄).new ȳ,z̄ in { y!(v,ȳ) | z!(v,z̄) | copy!(x̄,ȳ,z̄) }
| *source?(n,x,y).new ȳ in { y!(n,ȳ) | link!(x,ȳ) }
| source!(0,f,a) | copy!(a,b,c) | invert!(b,d) | tail!(c,e) | zip!(d,e,f) }
```

Listing 9.7 Term representation of the stream network computing the Thue-Morse sequence.

because the impossibility to type the term is due to an excess of produced messages rather than the lack thereof, as it was with the sequence of prime numbers.

9.4 Related Work

Binary session type disciplines [12] provide intra-session guarantees of lock freedom but cannot enforce this property in general when multiple sessions are interleaved for types do not carry any information concerning the dependencies between different sessions. Some session type systems [1, 24] are designed in such a way that a plain session type discipline is sufficient to guarantee deadlock freedom. However, only networks with a tree-like communication topology are well typed. Among the examples we have considered, just the recursive Fibonacci (Listing 9.1) and the master-worker (Listing 9.3) fall in this category.

Multiparty session type disciplines [11, 12] extend (dead)lock freedom to sessions involving multiple processes. In these framework a *global type* 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. The global type is given explicitly by the system designer/programmer and a tool is then used to check the consistency of the global type against (a model of) the code that is meant to realize it. This top-down approach is complementary to the one we have pursued in this chapter: `Hypha` analyzes assemblies of processes knowing nothing about the intended communication topology. In general, global types have been designed 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. This has led to the study of hybrid approaches [2, 3] that keep track of the order in which different sessions interleave with the purpose of detecting mutual dependencies between sessions that could lead to (dead)locks.

The works most closely related to our own are those where each input/output operation described by a channel/session type is annotated with information that captures the dependencies between different channels/sessions. Such annotations come in the form of integer numbers as in our case, or as abstract events, or as combinations thereof. The original technique and the corresponding analysis tool `TyPiCal`, which our type system and `Hypha` are heavily inspired by, were described by Kobayashi [13–15].

These type systems and our own are uncomparable: on the one hand, in Kobayashi's works annotations can be used to reason about dependencies between arbitrary channels, whereas we focus on linear channels only. On the other hand, the form of level polymorphism allowed by rule [T-OUT*] enables the verification of cyclic networks of recursive processes (most of the examples we have examined in this chapter fall in this category) that cannot be successfully handled by Kobayashi's type systems [13–15]. A more recent work [9] improves the precision of the technique, although recursive types (hence recursive communication protocols) are not considered. The annotation-based technique has also been applied directly to binary [17, 23] and multiparty sessions [21].

It has been shown that the approaches imposing a tree-like communication topology [1, 24] are subsumed by those those annotating I/O actions in session types with dependency information [6].

Acknowledgments I'm grateful to the anonymous reviewers whose comments helped me improving both content and presentation of this chapter.

References

[1] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *Proceedings of COORDINATION'13*, LNCS 7890, pages 45–59. Springer, 2013.

[3] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26:238–302, 2016.

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

[5] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012.

[6] Ornela Dardha and Jorge A. Pérez. Comparing deadlock-free session typed processes. In *Proceedings of EXPRESS/SOS'15*, EPTCS 190, pages 1–15, 2015.

[7] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theoretical Computer Science*, 411(4-5):765–782, 2010.

[8] Marc Geilen and Twan Basten. Kahn process networks and a reactive extension. In *Handbook of Signal Processing Systems*, pages 1041–1081. Springer, 2013.

[9] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of CONCUR'14*, LNCS 8704, pages 63–77. Springer, 2014.

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

[11] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9, 2016.

[12] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49:3:1–3:36, 2016.

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

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

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

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

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

[18] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proceedings of CSL-LICS'14*, pages 72:1–72:10. ACM, 2014.

[19] Luca Padovani. Type Reconstruction for the Linear π-Calculus with Composite Regular Types. *Logical Methods in Computer Science*, 11:1–45, 2015.

[20] Luca Padovani, Tzu-Chun Chen, and Andrea Tosatto. Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi. In *Proceedings of COORDINATION'15*, LNCS 9037, pages 83–98. Springer, 2015.

[21] Luca Padovani, Vasco T. Vasconcelos, and Hugo Torres Vieira. Typing Liveness in Multiparty Communicating Systems. In *Proceedings of COORDINATION'14*, LNCS 8459, pages 147–162. Springer, 2014.

[22] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

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

[24] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2–3):384–418, 2014.