The Impact of Linearity Information on the Performance of TyCO

Francisco Martins

Department of Mathematics, University of Azores, Portugal

Luís Lopes

Department of Computer Science, Faculty of Sciences, University of Porto, Portugal.

Vasco T. Vasconcelos

Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal.

Abstract

We describe a linear channel inference system for the TyCO programming language, where channel usage is tracked through method invocations as well as procedure calls. We then apply linear channel information to optimize code generation for a multithreaded runtime system. The impact in terms of speed and space is analyzed.

1 Introduction

Modern compilers rely on type information for code generation. Message passing concurrent languages base their computation model in two abstractions: processes, representing arbitrary computations, and channels, used for processes to exchange messages. For these kind of languages, knowledge of the usage of channels is crucial for efficient code generation: code size is reduced, tests are avoided, less heap is allocated and thus garbage collection is performed less often. This has an obvious impact on performance. Moreover, due to hardware limitations, type driven optimizations can make the difference between being and not being able to run a program.

In the realm of channel-based concurrent (π-based) programming languages there are different kinds of information that may be used for efficient code generation. For example, the Pict compiler crucially relies on the fact that a replicated process is the only input on a given channel, and that it appears prior to any message on the channel [9]. Another example uses *receptive channels* [11]: if we know that at any time there is a receptor on a given channel, then the heap space allocated for the receptor can be reused. Furthermore, the code for reduction may be simplified since no checks are required on the state of the channel.

Channel *linearity* information allows important optimizations to be performed [4,6,9]. Linear channels—channels that may be used exactly once for output and exactly once for input—are of special interest since they encompass the important case of synchronization channels, being pervasive, for example, in functional style constructs.

This paper describes a type inference algorithm that computes how many times (zero, one, many) each channel is used in a given program written in the TyCO programming language [14,15], and describes the experimental results obtained with the TyCO compiler [10]. The type inference system is an extension of that of Igarashi and Kobayashi, allowing for mutually recursive definitions.

The outline of the paper is as follows. The next section briefly introduces the TyCO programming language and its process calculus. Section 3 presents a linear type assignment system, and the following section a linear type inference system. Section 6 describes and assesses the performance increment resulting from the optimization of linear channels in the TyCO compiler and virtual machine. The last section compares our system with that of Igarashi and Kobayashi [3], and points to further work.

2 The TyCO language and its calculus

The TyCO programming language is an object-based concurrent programming language based on a calculus with the same name [13], featuring a predicative polymorphic type system.

An example

We start with a program that produces consecutive prime numbers using the algorithm of Eratosthenes. We assume a procedure Ints that produces consecutive integer values on some output stream, starting from 2. The integers are fed into a series of *sieves*, each with its own *grain*. A sieve of grain n filters all numbers that are multiple of n, forwarding the remaining numbers to the next sieve in the chain. Parameters to Sieve are the input stream, the grain, and the output stream. Here is a possible definition:

```
Sieve (inStream, grain, outStream) =
  inStream ? (n) :
    if n % grain /= 0
    then
      outStream ! [n] ;
```

$$\begin{array}{rcl} P & ::= & a\,!\,[\vec{v}] \quad | \quad a\,?\,\{l_i(\vec{x}_i) = P_i\}_{i\in I} \quad | \quad P \mid Q \quad | \quad \mathsf{0} \quad | \quad \mathsf{new}\ x\ P \quad | \\ & & X[\vec{v}] \quad | \quad \mathsf{def}_{i\in I}\ X_i(\vec{x}_i) = P_i\ \mathsf{in}\ Q \end{array}$$

Fig. 1. Syntax of the TyCO process calculus

```
    Sieve [inStream, grain, outStream]
  else
    Sieve [inStream, grain, outStream]
```

An invariant of the program says that sieves are ordered by their grain, the one with the smallest grain being closer to the source of integers. The last sieve in this chain is special, we call it a Sink. If a number (say n) ever reaches the last sieve, it must be a prime. The Sink then outputs the number, creates a new sink, and *becomes* a regular Sieve of grain n, reading from wherever the Sink used to read, and writing into the newly created sink.

```
Sink (inStream) =
  inStream ? (n) :
    io ! puti [n] ;
    new newSieve
    Sink [newSieve] |
    Sieve [inStream, n, newSieve]
```

The example highlights a feature unusual on most object-oriented programming languages: the ability to change the behavior of objects half-way through computation, essentially, the *become* operation of the actor model [2]:

```
  Sink (inStream, ...) = ... Sieve [inStream, ...]
```

The only restriction is that channel inStream in both Sink and Sieve share the same type: a stream of integers, in this case (more on types in the next section). To put all this code into work we need to instantiate a copy of Ints, and another of Sink, connected by a new channel that we decided to call aStream. The program that writes on the output consecutive prime numbers, ad eternum, is then:

```
  new aStream Ints [aStream] | Sink [aStream]
```

Syntax

We briefly introduce the TyCO process calculus that lies at the heart of programming language with the same name, while, at the same time, explain the program above. Assume a countable set of (channel) *names*, a set of *labels*, and a countable set of *procedure identifiers*. We denote names, labels, and procedure identifiers, respectively, by letters a, b, v, x, y, by letter l, and by letters X, Y, Z. The syntax of process expressions is given by the grammar in figure 1.

Processes of the form $a\,!\,l[\vec{v}]$ describe *messages*, where a is the channel through which the communication $l[\vec{v}]$ is sent, l is a label that selects a method in the target object, and $\vec{v}$ is the actual contents of the message. We allow label val to be omitted; so outStream ! [n] abbreviates outStream ! val [n].

Objects are described by processes of the form $a\,?\,M$, where a is the location of the object and M is its collection of methods. A method is of the form $l_i\,(\vec{x}_i) = P_i$, where l_i is its label (unique within the collection of methods), $\vec{x}_i$ represents the formal parameters, and P_i is the method body. Objects with a single method labeled with val may be abbreviated to $a\,?\,(\vec{x}) = P$, thus regaining the usual prefixes of the π-calculus.

The process $P \mid Q$ represents the parallel execution of P and Q. Inaction denotes a terminated process. *Scope restriction*, or channel declaration, is introduced by processes of the form new x P, suggesting x as a new channel visible only within P. Procedures are introduced with processes of the form $\mathsf{def}_{i \in I}\ X_i(\vec{x}_i) = P_i$ in Q, allowing for mutually recursive definitions. The program above should be understood as the process

def Ints (..) = ... Sieve (..) = ... Sink (..) = ... **in new** aStream ...

Core to the language is also the conditional construct, and expressions built from channels, base types (integers, booleans, strings, floats), and primitive operations on base types. The remaining constructs are translated at parsing time into the core (two of them are described below; for the full language refer to the language definition [14]). For example, the sequential composition operator is derived. The above piece of code

outStream ! [n] ; Sieve [..]

is translated into (the scope of ack extends as far to the right as possible)

new ack outStream ! [n, ack] | ack ? { done () = Sieve [..]}

where we expect the object at outStream to output a message ack ! done [] upon reception of a message. The colon syntax is used for this exact purpose. The above piece of code

inStream ? (n) : P

is an abbreviation to (again, the scope of the receptor extends as far to the right as possible)

inStream ? (n, r) = r ! done [] | P

thus regaining the usual synchronous prefixes of the π-calculus.

Notice that the semi-colon operator does not allow to compose two arbitrary processes, in contrast to the parallel composition: at the left of the semi-colon one can only have a message or a procedure call. This is the reason why we can't lift the recursive call to Sieve out of the if-then-else.

$$\text{COM} \quad a\,!\,l_j[\vec{v}] \mid a\,?\,\{l_i(\vec{x}_i) = P_i\}_{i\in I} \to \{\vec{v}/\vec{x}_j\}P_j$$

$$\text{CALL} \quad \mathsf{def}_{i\in I}\ X_i(\vec{x}_i) = P_i \ \mathsf{in}\ X_j[\vec{v}] \mid Q \to \mathsf{def}_{i\in I}\ X_i(\vec{x}_i) = P_i \ \mathsf{in}\ \{\vec{v}/\vec{x}_j\}P_j \mid Q$$

$$\text{RES}\frac{P \to Q}{\mathsf{new}\ x\ P \to \mathsf{new}\ x\ Q} \qquad \text{PAR}\frac{P \to Q}{P \mid R \to Q \mid R}$$

$$\text{DEF}\frac{P \to Q}{\mathsf{def}\ D\ \mathsf{in}\ P \to \mathsf{def}\ D\ \mathsf{in}\ Q} \qquad \text{STR}\frac{P \equiv R \quad R \to S \quad S \equiv Q}{P \to Q}$$

Fig. 2. Reduction relation

Reduction

The operational semantics of the calculus is presented following Milner [7]: a *congruence relation* (not shown) between processes simplifies the *reduction relation* introduced thereafter. The rules in figure 2 inductively define the *reduction relation*. COM is the *communication* rule between a message and an object. The resulting process is the method body P_j, selected by the label l_j, with its parameters $\vec{x}_j$ replaced by the arguments $\vec{v}$. CALL rule describes the replacement of a procedure call by its definition, performing the necessary substitution. Structural congruence is crucially used to bring processes into the form requested by the left-hand-side of axioms COM and CALL. The remaining rules allow reduction to happen within restriction, parallel composition, and definition. Rule STR brings structural congruence into reduction.

3 Linear type assignment system

This section introduces a type system allowing for reasoning about how many times channels are used during reduction. The type system for TyCO includes recursive types and predicative polymorphism (over procedure identifiers), which we omit for the sake of clarity.

Uses and types

In order to record the number of times a channel has been used, Igarashi and Kobayashi introduce the concept of *uses*, that enables to keep track of channels usage both for input and for output [3]. There are three kinds of uses: 0, meaning that no communication is allowed on the channel; 1, meaning at most one communication—a linear channel; and ω describing an unbound number of communications on the channel.

Four operations on uses are useful to describe the type system. The *sum*, the *product*, the *least upper bound*, and the *supression* of uses, denoted re-

$$\begin{array}{lcll} \alpha & ::= & \{l_i : \vec{\rho}_i\}_{i \in I} \ \mid \ t & \text{(base types)} \\ \rho & ::= & \alpha^{(\kappa_1, \kappa_2)} & \text{(types)} \end{array}$$

Fig. 3. The grammar of types

spectively by $\kappa_1 + \kappa_2$, $\kappa_1 \times \kappa_2$, $\kappa_1 \sqcup \kappa_2$, and k^-, are defined as follows.

$\kappa_1 + \kappa_2$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\kappa_1 \times \kappa_2$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

$\kappa_1 \sqcup \kappa_2$	0	1	ω
0	0	1	ω
1	1	1	ω
ω	ω	ω	ω

κ^-	
0	undef
1	0
ω	ω

Assume a countable set of *type variables*, and let t range over the set. Types, annotated with uses, are described in figure 3. *Channel types* represent the type of an object with n methods labeled with l_i and parameters of types $\vec{\rho}_i$. To maintain a separate counting on the number of messages sent and received on a channel, we attach to each channel type a pair of uses (κ_1, κ_2), where κ_1 and κ_2 specify, respectively, the number of sends and receives recorded for the channel. Type variables are really not needed until type reconstruction (section 4). For the full language we must add the primitive types. Here are some of the types inferred by the TyCO compiler for the example in the previous section.

ack: $\{$done: $\}^{(1,1)}$

outStream: IntegerStream$^{(0,w)}$

Sieve: IntegerStream$^{(w,0)}$ Integer IntegerStream$^{(0,w)}$

where IntegerStream is the base type $\{$val: Integer $\{$done: $\}^{(1,1)}\}$.

Counting procedure calls

The def construct binds processes to procedure identifiers and allows for calls to these processes within its scope. In a process of the form $\mathsf{def}_{i \in I}\ X_i(\vec{x}_i) = P_i$ in Q, each procedure X_i may be called any number of times from any P_i or Q. For a process P to be typified correctly, the input and output uses of every (type of every) name in P must reflect, at least, its communication capabilities. If a name, say a, occurs free in a procedure $X_j\,(\vec{x}_j) = P_j$, it is not enough to consider only the usage of a within P_j. In fact, the usage of a depends also from the number of times that X_j is called within Q and within the remaining procedures. Our type systems and inference algorithm are parameterized on a function $\mathcal{U}$ that counts the number of times a procedure is called.

Definition 3.1 Let $D \overset{\text{def}}{=} (X_i(\vec{x}_i) = P_i)_{i \in I}$. A function $\mathcal{U}$ is a *call counting*

$$\textsc{Com}_a \quad a\,!\,l_j[\vec{v}] \mid a\,?\,\{l_i(\vec{x}_i) = P_i\}_{i\in I} \xrightarrow{a} \{\vec{v}/\vec{x}_j\}P_j$$

$$\textsc{Res}_\epsilon \frac{P \xrightarrow{x} R}{\mathsf{new}\ x : \alpha^{(\kappa_1,\kappa_2)} P \xrightarrow{\epsilon} \mathsf{new}\ x : \alpha^{(\kappa_1^-,\kappa_2^-)} R}$$

$$\textsc{Res}_l \frac{P \xrightarrow{\ell} R \qquad \ell \neq x}{\mathsf{new}\ x : \rho\ P \xrightarrow{\ell} \mathsf{new}\ x : \rho\ R}$$

Fig. 4. New rules for the reduction relation with uses

function if it satisfies the following requirements.

(i) $\mathcal{U}(X,D,Q) \geq \mathcal{U}(X,D,R)$, if $Q \to R$,

(ii) $\mathcal{U}(X,D,X_i[\vec{v}] \mid Q) = \begin{cases} 1 + \mathcal{U}(X,D,\{\vec{v}/\vec{x}_i\}P_i \mid Q) & \text{if } X = X_i, \\ \mathcal{U}(X,D,\{\vec{v}/\vec{x}_i\}P_i \mid Q) & \text{otherwise.} \end{cases}$

The first assertion states that the number of potential calls to a particular procedure cannot increase during reduction. The second assertion refers specifically to reductions that occur on a call: if the call is on X—the procedure that we are counting—then the number of calls decreases by 1, because X is called in P_i the same number of times in each side of equation, plus one more time in the call to $X[\vec{v}]$ itself. Otherwise, the number of potential calls to X is not affected.

There is a call counting function: the constant function that maps any triple into ω. In section 4 we propose a more useful function.

Subtyping

The binary relation $\preceq$ on types is defined as the least equivalence relation closed under the following rule.

$$\frac{\kappa_1 \geq \mu_1 \qquad \kappa_2 \geq \mu_2 \qquad \mu_1 \geq 1 \text{ implies } \vec{\rho}_i \preceq \vec{\sigma}_i \qquad \mu_2 \geq 1 \text{ implies } \vec{\sigma}_i \preceq \vec{\rho}_i}{\{l_i : \vec{\rho}_i\}_{i\in I}^{(\kappa_1,\kappa_2)} \preceq \{l_i : \vec{\sigma}_i\}_{i\in I}^{(\mu_1,\mu_2)}}$$

where $\rho_1 \ldots \rho_n \preceq \sigma_1 \ldots \sigma_n$ means $\rho_i \preceq \sigma_i$, for all $1 \leq i \leq n$. The relation is defined quite conventionally: covariant for input ($\mu_1 \geq 1$), contravariant for output ($\mu_2 \geq 1$), and invariant when both conditions hold.

Type assignment, explicitly typed processes, and reduction with uses

Judgments of the type assignment system are of the form $\Gamma \vdash P$, where Γ, called a *typing*, is a map from names into types (and from procedure identifiers into type sequences), and P is an explicitly typed process (defined below). We do not present the type system here; it can be found in reference [6]. It should be noted that the type system is not syntax-directed because of the presence

$$\text{K-Rcd} \quad K \vdash \{l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n, \ldots\} : \ \langle l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n \rangle$$

$$\text{K-Var} \quad K, t : \ \langle l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n, \ldots \rangle \ \vdash t : \ \langle l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n \rangle$$

Fig. 5. Kind assignment to base types

of the usual subtyping rule,

$$\frac{\Gamma, x : \rho \vdash P \qquad \sigma \preceq \rho}{\Gamma, x : \sigma \vdash P}$$

in addition to the weak rules both for channel names and for procedure identifiers. An arbitrary call counting function is used in the rule for definitions.

We do however present the main property of the system, namely subject-reduction. In order to do so, we need two ingredients: explicitly typed processes, and a reduction relation that records the channel on which communication happened. The *set of explicitly typed processes* is obtained by replacing, in figure 1, $\mathsf{new}\ x\ P$ by $\mathsf{new}\ x : \rho\ P$. We can easily get a (implicitly typed) process from an explicitly typed one. The function erase replaces (sub)process of the form $\mathsf{new}\ x : \rho\ P$ by $\mathsf{new}\ x\ P$.

For the second ingredient, *use-aware reduction*, we label each reduction either with a channel x, or with the special symbol ϵ denoting a communication on a bound channel or a procedure call. We use ℓ to range both over names and over ϵ. The rules for the *reduction relation with uses* are obtained from the rules in figure 2 by a) labeling with l the arrows in rules, Par, Def, and Str, by b) labeling with ϵ the arrow in axiom Call, and by c) replacing rules Com and Res by the rules in figure 4.

The effect of consuming a resource ℓ in a typing Γ is a typing $\Gamma^{-\ell}$, obtained from Γ as follows.

$$\Gamma^{-\ell}(a) = \begin{cases} \Gamma(a) & \text{if } a \neq \ell, \\ \alpha^{(\kappa_1^-, \kappa_2^-)} & \text{if } \Gamma(a) = \alpha^{(\kappa_1, \kappa_2)} \text{ and } \kappa_1^-, \kappa_2^- \text{ defined}, \\ \text{undefined otherwise.} \end{cases}$$

Theorem 3.2 (Subject-reduction) *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\ell} Q$*, then* $\Gamma^{-\ell}$ *is defined and* $\Gamma^{-\ell} \vdash Q$*.*

Notice that the suppression operation (as well as $+$, $\times$, and $\sqcup$ in page 10) only work on the outermost uses in a type. A channel of type $\{\mathsf{val}\colon \mathsf{Integer}\ \{\mathsf{done}\colon\}^{(1,1)}\}^{(0,1)}$ can only be written once. When a message is sent on such a channel, the channel can no longer carry messages. This event is unrelated to the communication capabilities of the channels transmited on the message—the channel $\{done :\}^{(1,1)}$—that are consumed only when actually used.

$$\kappa \ ::= \ 0 \ | \ 1 \ | \ \omega \ | \ u \ | \ \kappa_1 + \kappa_2 \ | \ \kappa_1 \cdot \kappa_2 \ | \ \kappa_1 \sqcup \kappa_2$$

Fig. 6. Syntax of use expressions

4 Linear type inference system

This section describes a linear channel inference system for the TyCO process calculus. We mostly follow Igarashi and Kobayashi [3]; for process definitions we parameterize the type system with a call counting function satisfying definition 3.1. In order to obtain a type system with computable principal record typings' we incorporate the notion of *kinds*, as exploited by Vasconcelos [16]. In what follows we introduce the notions of kinds and constraints needed for the type reconstruction system.

Kinds and kind assignment to types

Intuitively, a *kind* describes a set of record types. A kind of the form $\langle l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n \rangle$ denotes the subset of all record types that contain, at least, the components $l_1 : \vec{\rho}_1, \ldots, l_n : \vec{\rho}_n$.

Judgements of the *kind assignment system* are of the form $K \vdash \alpha : k$, where K, called a *kinding*, is an acyclic map from type variables into kinds [1]. The two axioms composing the kind assignment system are presented in figure 5. Pairs of the form (K, Γ) are called *kinded typings*.

One operation on kinds is useful to describe the type inference system. The *sum* of two kinds $\langle l_i : \vec{\alpha}_i \rangle_{i \in I}$ and $\langle l_j : \vec{\alpha}_j \rangle_{j \in J}$ is the kind $\langle l_k : \vec{\alpha}_k \rangle_{k \in I \cup J}$.

Constraints

We extend the syntax of uses to incorporate variables and expressions. Let u range over a countable set of *use variables*. The syntax of *use expressions* is given by the grammar in figure 6. We call the uses that may appear in types—0, 1, ω— *constants*.

A *subtype constraint set* (constraint set, for short) C is a set of subtype expressions $\rho_1 \preceq \rho_2$, called *constraints*. We extend $\preceq$ to typings, and let $\Gamma \preceq \Delta$ denote the constraint set $\{\Gamma(x) \preceq \Delta(x) \mid x \in \mathrm{dom}(\Delta)\}$, when $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma)$.

A substitution S is a map from type variables into types (ρ/t), and from use variables into use expressions (κ/u). For any type ρ define $S\rho$ to be the type obtained by simultaneously applying the substitutions in S. We similarly apply a substitution S to kinds k, to finite sequences of types $\vec{\rho}$, to typings Γ, to kindings K (images only), to constraint sets C, and also to processes P. Moreover, if SC contains no variables (type or use), we call S a *ground substitution* on C.

[1] A cycle in a set of kind assignments is a sequence of elements $t_1 : k_1, \ldots, t_n : k_n$, such that t_{i+1} occurs in k_i and t_1 occurs in k_n.

$$\textsc{Par}\,\frac{K_1; C_1; \Gamma_1 \vdash P_1 \quad K_2; C_2; \Gamma_2 \vdash P_2 \quad C \models \Gamma \preceq \Gamma_1 + \Gamma_2 \quad C \models C_1 \cup C_2}{K_1 + K_2; C; \Gamma \vdash P_1 \mid P_2}$$

$$\textsc{Msg}\,\frac{K \vdash \alpha : \langle l : \vec{\rho} \rangle \quad C \models \Gamma \preceq (a : \alpha^{(0,1)}, \vec{v} : \vec{\sigma}) \quad C \models \{\vec{\sigma} \preceq \vec{\rho}\}}{K; C; \Gamma \vdash a\,!\,l[\vec{v}]}$$

$$\textsc{Obj}\,\frac{\begin{array}{c}(\forall i \in I)\ K_i; C_i; \Gamma_i, \vec{x}_i : \vec{\sigma}_i \vdash P_i, \\ C \models \bigcup_{i \in I}(\{\Gamma \preceq \Gamma_i\} \cup C_i \cup \{\vec{\rho}_i \preceq \vec{\sigma}_i\}) \\ C \models \Gamma \preceq (a : \{l_i : \vec{\rho}_i\}^{(1,0)}_{i \in I} + \bigsqcup_{i \in I} \Gamma_i)\end{array}}{\sum_{i \in I} K_i; C; \Gamma \vdash a\,?\,\{l_i(\vec{x}_i) = P_i\}_{i \in I}} \qquad \textsc{Nil}\ \ \emptyset; \emptyset; \Gamma \vdash \mathsf{0}$$

$$\textsc{Res}\,\frac{K; C; \Gamma, x : \rho \vdash P}{K; C; \Gamma \vdash \mathsf{new}\ x\ P} \qquad \textsc{Call}\,\frac{C \models \vec{\sigma} \preceq \vec{\rho}}{\emptyset; C; \Gamma, X : \vec{\rho}, \vec{v} : \vec{\sigma} \vdash X[\vec{v}]}$$

$$\textsc{Def}\,\frac{\begin{array}{c}(\forall i \in I)\ K_i; C_i; \bigcup_{j \in I} X_j : \vec{\rho}_j, \Gamma_i, \vec{x}_i : \vec{\sigma}_i \vdash P_i, \quad K; C'; \bigcup_{j \in I} X_j : \vec{\rho}_j, \Delta \vdash Q \\ C \models \Gamma \preceq (\Delta + \sum_{j \in I} \mathcal{U}(X_j, (X_i(\vec{x}_i) = P_i)_{i \in I}, Q) \times \Gamma_j) \\ C \models C' \qquad\qquad C \models \bigcup_{j \in I}(C_j \cup \{\vec{\sigma}_j \preceq \vec{\rho}_j\})\end{array}}{\sum_{i \in I} K_i + K; C; \Gamma \vdash \mathsf{def}_{i \in I}\ X_i(\vec{x}_i) = P_i\ \mathsf{in}\ Q}$$

Fig. 7. Type reconstruction

The ground substitutions of interest are those that turn true all the constraints in a constraint set. A ground substitution S is a *solution* of C, if $S\rho_1 \preceq S\rho_2$ holds for every constraint expression $\rho_1 \preceq \rho_2$ in C. A constraint set C_1 *satisfies* another constraint set C_2, denoted by $C_1 \models C_2$, if every solution of C_1 is also a solution of C_2.

A kinded type system for reconstruction

We introduce a syntax-directed typing system that identifies linear channels. Judgments are now of the form $K; C; \Gamma \vdash P$, for P an (implicitly typed, figure 1) process. Rules can be found in figure 7. The notation is explained along with the rules.

The $+$ and $\times$ operations on uses (defined in page 6) are extended, first to types, and then to typings. The sum of two types is defined only when the base types are identical: $\alpha^{(\kappa_1,\kappa_2)} + \alpha^{(\mu_1,\mu_2)} = \alpha^{(\kappa_1+\mu_1,\kappa_2+\mu_2)}$. For typings, $+$ is extended as follows,

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) + \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \setminus \mathrm{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_2) \setminus \mathrm{dom}(\Gamma_1) \end{cases}$$

and similarly for the sum of kindings and the product of typings. When $x \notin \mathrm{dom}(\Gamma)$, we use $\Gamma, x : \rho$, instead of $\Gamma + x : \rho$.

Rule Par says that, in order to type $P_1 \mid P_2$ one has to type each P_i, find a constraint set C that satisfies each C_i (we can easily show that $C \models C_1 \cup C_2$

iff $C \models C_1$ and $C \models C_2$), and a typing Γ (whose domain contains those of each Γ_i) such that C satisfies each constraint in the set $\Gamma \preceq \Gamma_1 + \Gamma_2$.

Rule Msg expresses the fact that a must be a channel with, at least, a component $l : \vec{\rho}$ (notice the kind $\langle l : \vec{\rho} \rangle$ assigned to α) and output capabilities (usage $(0,1)$). The typing $\vec{v} : \vec{\sigma}$ (meaning the n-fold sum $v_1 : \sigma_1 + \cdots + v_n : \sigma_n$ when $\vec{v} = v_1 \cdots v_n$, $\vec{\sigma} = \sigma_i \cdots \sigma_n$) take into account the use of $\vec{v}$ by the receiver, keeping in mind that the v_i are not necessarily disjoint.

The $(1,0)$ in rule Obj expresses the fact that a must be a channel with, at least, input capabilities. We take the least upper bound of the typings for the methods, since only one of them will ever be activated. Also, we throw away type information on $\vec{x}_i$ from the resulting type, but keep the subtype information $\vec{\rho}_j \preceq \vec{\sigma}_j$ in the resulting constraint set.

For rule Res we throw away type information on x since x is bound in the conclusion. The constraint $\vec{\sigma} \preceq \vec{\rho}$ in rule Call accounts for the fact that the types of the arguments must be subtypes of the parameters; Call is essentially an output operation. For rule Def, one might expect that the sum of the parts, that is $\Delta + \sum_{j \in I} \Gamma_j$, would be enough for typify the whole def-process. This is not the case, since every time a procedure P_j is called we must supply a set Γ_j of resources. Thus, Γ must hold enough resources to cover every call to X_j, hence, at least $\mathcal{U}(X_j, (X_i(\vec{x}_i) = P_i)_{i \in I}, Q)$—the number of times that X_j is called from Q—copies of Γ_j must exist in Γ.

The equivalence between the system in figure 7 and the one mentioned in section 3 is made precise by the following theorem.

Theorem 4.1 *Let P be an explicitly typed process.*

(i) *If $K; C; \Gamma \vdash \mathrm{erase}(P)$, C' (resp. Γ') is obtained from C (resp. Γ) by recursively replace type variables t for records $\{l_i : \vec{\rho}_i\}_{i \in I}$ whenever $t : \langle l_i : \vec{\rho}_i \rangle_{i \in I}$ occurs in K, S is a solution of C whose domain includes all type/use variables in Γ and in P, then $S\Gamma' \vdash SP$.*

(ii) *If $\Gamma \vdash P$, then $\emptyset; \emptyset; \Gamma \vdash \mathrm{erase}(P)$.*

Proof. Assertion one and two are proved by induction on the structure of the derivation of the typing of P. □

A type reconstruction algorithm

Typings are not uniquely determined. The principal kinded typing—a triple (K, C, Γ)—for processes allows to recover all such typings.

Definition 4.2 (i) A pair (K', S) *respects* K, if $K' \vdash St : SKt$, for all $t \in \mathrm{dom}(K)$.

(ii) A triple (K', C', Γ'), called a *kinded constraint typing*, is an *instance of* (K, C, Γ), if $\mathrm{dom}(K) \subseteq \mathrm{dom}(K')$, $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$, and there is a substitution S such that (K', S) respects K, $S\Gamma \subseteq \Gamma'$, and $C' \models SC$.

(iii) A *kinded set of equations* is a pair (K, E) composed of a kinding K, and a set of equations E of the form $\alpha = \beta$, for α and β types.

(iv) The pair (K', S) is a *unifier* of (K, E) if (K', S) respects K, and $S\alpha = S\beta$ for all $\alpha = \beta \in E$.

(v) (K, S) is *more general than* (K', S') if there is a substitution S'' such that $S' = S''S$ and (K', S'') respects K.

(vi) A solution S of C is *minimal* if, for any solution S' of C, $S'j \geq Sj$ for each use variable j appearing in C.

Definition 4.3 The triple (K, C, Γ) is *principal* for P, if

(i) $K; C; \Gamma \vdash P$, and

(ii) If $K'; C'; \Gamma' \vdash P$, then (K', C', Γ') is an *instance* of (K, C, Γ).

There is an algorithm, call it LTR for *linear type reconstruction*, that computes a quadruple (K, C, Γ, E), where K is a kinding, Γ is a typing, C is a constraint set, and E is a set of type equations. From (K, C, Γ, E) we can compute the principal typing of a process if it exists, or announce failure otherwise. The correctness of the algorithm is given by the following result.

Theorem 4.4 (Correctness of LTR) *Let (K, C, Γ, E) be the output of the LTR(P) algorithm.*

(i) *If (K', S') is the most general unifier of (K, E) and S is a minimal solution of C, then $(K', S'SC, S'S\Gamma)$ is principal for P.*

(ii) *If (K, E) is not unifiable, then P is not typable.*

We omit the algorithm (see reference [6]), but describe its main features. The construction of the principal kinded constraint typing triple proceeds in four phases: (1) compute a quadruple (K, C, Γ, E) using the LTR algorithm; (2) compute the substitution pair (K', S') from the set of kinded equations (K, E) using Ohori's algorithm [8]; (3) generate a set of use constraints from C; (4) resolve these constraints using [3] to obtain S. Then, the triple $(K', S'SC, S'S\Gamma)$ is principal for P. If the kinded set of equations, (K, E), has no solution, then P is not typable.

The algorithm for the first phase is obtained by reading the rules in figure 7 bottom-up. Consider rule Par. We recursively call the algorithm on P_1 and P_2, thus obtaining $(K_1, C_1, \Gamma_1, E_1)$ and $(K_2, C_2, \Gamma_2, E_2)$. To combine these we use a function $\oplus$ that computes the most general pair (Γ, C) such that $C \models \Gamma \preceq \Gamma_1 + \Gamma_2$, and $C \models C_1 \cup C_2$. The result of the call on $P_1 \mid P_2$ is the quadruple $(K_1 + K_2, C, \Gamma, E_1 \cup E_2)$. The remaining rules are handled similarly, with new additional functions developed as necessary.

Notice that in the forth phase, we solve the subtype constraints in the constraint set obtained during the first phase. We are however interested on a *optimal* type annotation for the new-channels in the input process, in the sense that the uses of the channels are estimated as small as possible. Igarashi

$$\begin{aligned}
\mathcal{W}(Y, D, \mathsf{0}, V) &= \mathcal{W}(Y, D, a\,!\,l[\vec{v}], V) \;=\; 0\\
\mathcal{W}(Y, D, P \mid Q, V) &= \mathcal{W}(Y, D, P, V) + \mathcal{W}(Y, D, Q, V)\\
\mathcal{W}(Y, D, a\,?\,\{l_i(\vec{x}_i) = P_i\}_{i\in I}, V) &= \textstyle\bigsqcup_{i\in I} \mathcal{W}(Y, D, P_i, V)\\
\mathcal{W}(Y, D, \mathsf{new}\ x\ P, V) &= \mathcal{W}(Y, D, P, V)\\
\mathcal{W}(Y, D, Y[\vec{v}], V) &= 1, \quad \text{if } Y \notin \{X_i\}_{i\in I}\\
\mathcal{W}(Y, D, Z[\vec{v}], V) &= 0, \quad \text{if } Z \notin \{X_i\}_{i\in I} \text{ and } Y \neq Z\\
\mathcal{W}(Y, D, X_i[\vec{v}], V) &= 0, \quad X_i \in V, \text{ and } X_i \not\rightsquigarrow Y\\
\mathcal{W}(Y, D, X_i[\vec{v}], V) &= \omega, \quad X_i \in V, \text{ and } X_i \rightsquigarrow Y\\
\mathcal{W}(X_i, D, X_i[\vec{v}], V) &= 1 + \mathcal{W}(X_i, D, P_i, V \cup \{X_i\}), \quad \text{if } X_i \notin V\\
\mathcal{W}(Y, D, X_i[\vec{v}], V) &= \mathcal{W}(Y, D, P_i, V \cup \{X_i\}),\\
&\qquad \text{if } X_i \notin V, \text{ and } Y \neq X_i\\
\mathcal{W}(Y, D, \mathsf{def}\ D' \ \mathsf{in}\ Q, V) &= \mathcal{W}(Y, D \cup D', Q, V)
\end{aligned}$$

where D is $(X_i(\vec{x}_i) = P_i)_{i\in I}$.

Fig. 8. The number of times a procedure is called.

and Kobayashi show how to optimally solve a constraint set [3].

Computing the use of a procedure identifier

The type systems mentioned in section 3 and presented in figure 7, as well as the algorithm LTR described above are parametric on a call counting function (definition 3.1). We now present an algorithm that computes the number of times that a procedure is called within a process. Notice that the algorithm has to deal with recursive calls to procedures (possibly defined using mutually recursive equations) and, more importantly, with *free names in each definition*.

Our approach is to interpret procedure calls as a graph that models the dependencies between each procedure. The number of times $(0, 1, \text{or } \omega)$ that a certain procedure X is called within a process P is given by the number of paths starting on every Y free in P and ending in X.

Definition 4.5 Consider the procedure definitions $(X_i(\vec{x}_i) = P_i)_{i\in I}$ and a procedure identifier Y. We say that X_i *calls* Y *directly*, denoted by $X_i \rightsquigarrow_1 Y$, if $P_i \equiv \mathsf{new}\ \vec{x}\ \mathsf{def}\ D\ \mathsf{in}\ Y[\vec{v}] \mid R$. The relation $\rightsquigarrow$ is the reflexive-transitive closure of $\rightsquigarrow_1$. When $X \rightsquigarrow Y$ we say that X *calls* Y, or that Y *is reachable from* X.

Finding whether a procedure X calls another procedure Y amounts to determine if two nodes are connected in a direct-graph; algorithms can be easily found in the literature ([1], for example).

The recursive function $\mathcal{U}$ computes the number of times that a procedure X_i is called in a process of the form $\mathsf{def}_{i \in I}\ X_i(\vec{x}_i) = P_i\ \mathsf{in}\ Q$. It uses an auxiliary function $\mathcal{W}$ that maintains a set V of visited procedures to avoid infinite recursion.

$$\mathcal{U}(X_i, D, Q) \stackrel{\text{def}}{=} \mathcal{W}(X_i, D, Q, \emptyset).$$

Figure 8 describes function $\mathcal{W}$, assuming that all bound procedure identifiers are pairwise distinct. If Q is inaction or a message, the number of calls is obviously 0. If Q is an object, we compute the least upper bound of the uses of Y, since only at most one of the methods is selected in reduction.

The first and the second clauses for a call $Y[\vec{v}]$ do not descend the body of the corresponding procedure, since it is not defined in the def-process we are analyzing. The reachability tests performed at the third and forth clauses are necessary when a procedure was already visited ($X_i \in V$). When $X_i \rightsquigarrow Y$ there is a cycle starting in X_i, since X_i is the first procedure that belongs to V. Thus, if Y is part of that cycle its use is obviously ω, otherwise is 0. The fifth and sixth clauses for $Y[\vec{v}]$ describe a call to a procedure defined in D. In this case we must analyze P_i (the process bound to X_i) as well. When Y is the same as X_i, we add 1 to the result yield by the analysis of P_i.

Theorem 4.6 *Function $\mathcal{U}$ is a call-counting function.*

Theorem 4.7 *Function $\mathcal{U}$ is invariant under process congruence.*

As an example, consider the process $\mathsf{def}_{i \in I}\ X_i(\vec{x}_i) = P_i\ \mathsf{in}\ P$ corresponding to our running example; we analyze its channel and procedure use. From the Sieve process definition we find that channel inStream is used for input once and channel outStream is used for output once in the then branch of the **if** process. But Sieve is recursive, and is reachable from Sink, that is reachable from P; then $\mathcal{U}(\mathsf{Sieve}, D, P) = \omega$ and the use of both inStream and outStream is $(\omega, 0)$ and $(0, \omega)$, respectively.

Analyzing the Sink procedure, we find that channel inStream is used for input once and io is used for output (also once). Then Sink passes inStream to Sieve that inputs from it ω times. So inStream has $(\omega, 0)$ use. We also find that *Sink* is recursive and is reachable from P, then $\mathcal{U}(\mathsf{Sink}, D, P)$ is ω, which makes io to be used ω times for output. The newly created channel newSieve has the same usage as inStream, that is, $(\omega, 0)$.

The channel aStream created in P is used for output by Ints an infinite number of times (recall that Ints produce integer numbers ad eternum; $\mathcal{U}(\mathsf{Ints}, D, P) = \omega$) and, as discuss above, ω times for input by Sink. Then the usage of aStream is (ω, ω).

Finally, synchronization channels (ack for example) are always linear, despite the fact that they may belong to recursive procedures, because they are newly created for each synchronization.

5 Comparing with Igarashi's type system

The def construct used in Igarashi and Kobayashi [3] is syntax sugar for the replicated input construct. Thus,

$$\mathsf{def}\ x[\vec{y}] = P\ \mathsf{in}\ Q\ \mathsf{end} \quad \text{stands for} \quad \mathsf{new}\ x(x\,?^*[\vec{y}].P \mid Q)$$

TyCO uses (mutually) recursive definitions instead of replication. It is well-known how to translate replication into recursive definitions and vice-versa (see, for instance, [12, pages 132–138]). This section compares our approach (using the U function defined in page 13) with that of Igarashi and Kobayashi.

Translation into TyCO

The translation function $[\![\cdot]\!]$ maps the Igarashi and Kobayashi process

$$\mathsf{def}\ x[\vec{y}] = P\ \mathsf{in}\ Q\ \mathsf{end}$$

into

$$\begin{cases} [\![Q]\!], & \text{if } x \notin \text{fn}(Q) \\ \mathsf{new}\ x\ [\![Q]\!] \mid x\,?\,(y) = [\![P]\!], & \text{if } x \in \text{fn}(Q) \text{ and } x \notin \text{fn}(P) \\ \mathsf{new}\ x\ \mathsf{def}\ A() = x\,?\,(y) = [\![P]\!] \mid A[] & \\ \qquad \mathsf{in}\ A[] \mid [\![Q]\!], & \text{otherwise} \end{cases}$$

and is an homomorphism in all other cases.

Theorem 5.1 *Let P be a process in [3]. if $\Gamma \vdash_{[3]} P$, then $\Gamma \vdash [\![P]\!]$.*

Proof. The proof is by straightforward induction on the structure of the derivation of the typing of P considering the translation function defined above.□

Encoding (mutually) recursive definitions into Igarashi's calculus

We consider a function $[\![\cdot]\!]^V_D$ that translates a TyCO process into Igarashi's calculus extended with objects and messages *à lá* TyCO. The sets D and V represent the procedure definitions and the variables translated (so far), respectively.

For the rest of this section let D be the procedure definition $(X_i(\vec{x}_i) = P_i)_{i\in I}$. We define $[\![\cdot]\!]^V_D$ for def and call processes and stipulate that $[\![\cdot]\!]^V_D$ is a homomorphism for the remaining process contructors.

$$\begin{aligned} &[\![Y[\vec{v}]]\!]^V_D \stackrel{\text{def}}{=} y\,!\,[\vec{v}], && \text{if } Y \notin \{X_i\}_{i\in I} \text{ or } Y \in V, \\ &[\![X_i[\vec{v}]]\!]^V_D \stackrel{\text{def}}{=} \mathsf{def}\ x_i[\vec{v}] = [\![P_i]\!]^{V\cup\{X_i\}}_D\ \mathsf{in}\ x_i\,!\,[\vec{v}]\ \mathsf{end}, && \text{if } X_i \notin V, \\ &[\![\mathsf{def}\ D'\ \mathsf{in}\ Q']\!]^V_D \stackrel{\text{def}}{=} [\![Q']\!]^V_{D\cup D'} \end{aligned}$$

The intuitive idea is that we substitute each procedure call $X_i[\vec{v}]$ by Igarashi's process $\mathsf{def}\ x_i[\vec{v}] = [\![P_i]\!]^{V\cup\{X_i\}}_D\ \mathsf{in}\ x_i\,!\,[\vec{v}]\ \mathsf{end}$ and proceed with the translation inside P_i, the process bound to X_i in D. If P_i is recursive we substitute $X_i[\vec{v}]$ within P_i by $x_i\,!\,[\vec{v}]$, since we have already introduced the definition of x_i. The set V tracks the expanded procedures at each point during translation.

Definition 5.2 The *out use* of a channel x in a typing Γ is

$$\operatorname{out}(x,\Gamma) = \begin{cases} \kappa_2, & \text{if } \Gamma(x) = \rho^{(\kappa_1,\kappa_2)} \\ 0, & \text{if } x \notin \operatorname{dom}(\Gamma) \end{cases}$$

Lemma 5.3 *Let* $\Gamma \vdash [\![X_i[\vec{v}]]\!]_D^V$*, and* $X_i \notin V$*, then*

$$\operatorname{out}(y,\Gamma) \begin{cases} \geq 1, & \textit{if } X_i \rightsquigarrow Y \\ = 0, & \textit{otherwise} \end{cases}$$

Proof. (outline) Since $X_i \notin V$, then $[\![X_i[\vec{v}]]\!]_D^V \overset{\text{def}}{=} \mathsf{def}\ x_i(\vec{y}) = [\![P_i]\!]_D^{V\cup\{X_i\}}\ \mathsf{in}\ x_i\,!\,[\vec{v}]\ \mathsf{end}$. By definition of $X_i \rightsquigarrow Y$, P_i calls (eventually indirectly) Y, meaning that $[\![P_i]\!]_D^{V\cup\{X_i\}}$ includes at least an output to y. Thus, $\operatorname{out}(y, [\![P_i]\!]_D^{V\cup\{X_i\}}) \geq 1$. The equality $\operatorname{out}(y,\Gamma) = 0$ is proved using similar arguments. □

Lemma 5.4 *Let* $\Gamma \vdash [\![Q]\!]_D^{\emptyset}$*. Then* $\mathcal{U}(X_i, D, Q) = \operatorname{out}(x_i,\Gamma)$*.*

Proof. Notice that the function $\mathcal{U}$ has a structure similar to the translation function $[\![\cdot]\!]$—the substitution of each call process by its definition. In what follows we proceed by induction on the structure of the translation and present only the more interesting cases—the call and the def processes.

For call, we prove that if $\Gamma \vdash [\![Z[\vec{v}]]\!]_D^V$, then $\mathcal{W}(Y, D, Z[\vec{v}], V) = \operatorname{out}(y,\Gamma)$. The proof is divided in 6 cases that match the definition of $\mathcal{U}$. We present the last one. Let $Z = X_i$ for some i. if $X_i \notin V$ and $Y \neq X_i$, then by translation $\Gamma \vdash \mathsf{def}\ x_i(\vec{x}) = [\![P_i]\!]_D^{V\cup\{X_i\}}\ \mathsf{in}\ x_i\,!\,[\vec{v}]\ \mathsf{end}$. Let $\Delta \vdash [\![P_i]\!]_D^{V\cup\{X_i\}}$, then $\operatorname{out}(y,\Gamma) = 1 \cdot (1 + \operatorname{out}(x_i,\Delta)) \cdot \operatorname{out}(y,\Delta)$. We have to consider two cases: (1) when $\operatorname{out}(x_i,\Delta) = 0$, then $1 \cdot (1 + \operatorname{out}(x_i,\Delta)) \cdot \operatorname{out}(y,\Delta) = \operatorname{out}(y,\Delta)$, that, by induction hypothesis, is $\mathcal{W}(Y, D, P_i, V \cup \{X_i\})$; and (2) when $\operatorname{out}(x_i,\Delta) \neq 0$. We need to analyse two subcases: (2.a) when $X_i \rightsquigarrow Y$, then at least one output in y is performed in $[\![P_i]\!]_D^{V\cup\{X_i\}}$, therefore $\operatorname{out}(y,\Gamma) = \omega$, which is the same as $\mathcal{W}(Y, D, P_i, V \cup \{X_i\})$, since X_i is recursive; (2.b) when $X_i \not\rightsquigarrow Y$, then $\operatorname{out}(y,\Delta) = 0$ and hence $\operatorname{out}(y,\Gamma) = 0$. The value of $\mathcal{W}(Y, D, P_i, V \cup \{X_i\})$ is also 0 when X_i is recursive and $X_i \not\rightsquigarrow Y$.

For def, we prove that if $\Gamma \vdash [\![\mathsf{def}\ D'\ \mathsf{in}\ Q']\!]_D^V$, then $\mathcal{W}(Y, D, \mathsf{def}\ D'\ \mathsf{in}\ Q', V) = \operatorname{out}(y,\Gamma)$. By definition of translation, $\Gamma \vdash [\![Q']\!]_{D\cup D'}^V$, and by induction hypothesis $\operatorname{out}(y,\Gamma) = \mathcal{W}(Y, D \cup D', Q', V)$ holds. The definition of $\mathcal{W}$ supports $\mathcal{W}(Y, D \cup D', Q', V) = \mathcal{W}(Y, D, \mathsf{def}\ D'\ \mathsf{in}\ Q', V)$. □

Theorem 5.5 *Let P be a process. If* $\Gamma \vdash P$*, then* $\Gamma \vdash_{[3]} [\![P]\!]_{\emptyset}^{\emptyset}$*.*

Proof. The proof is a straightforward induction on the structure of the derivation of the typing of P using lemmas 5.3 and 5.4. □

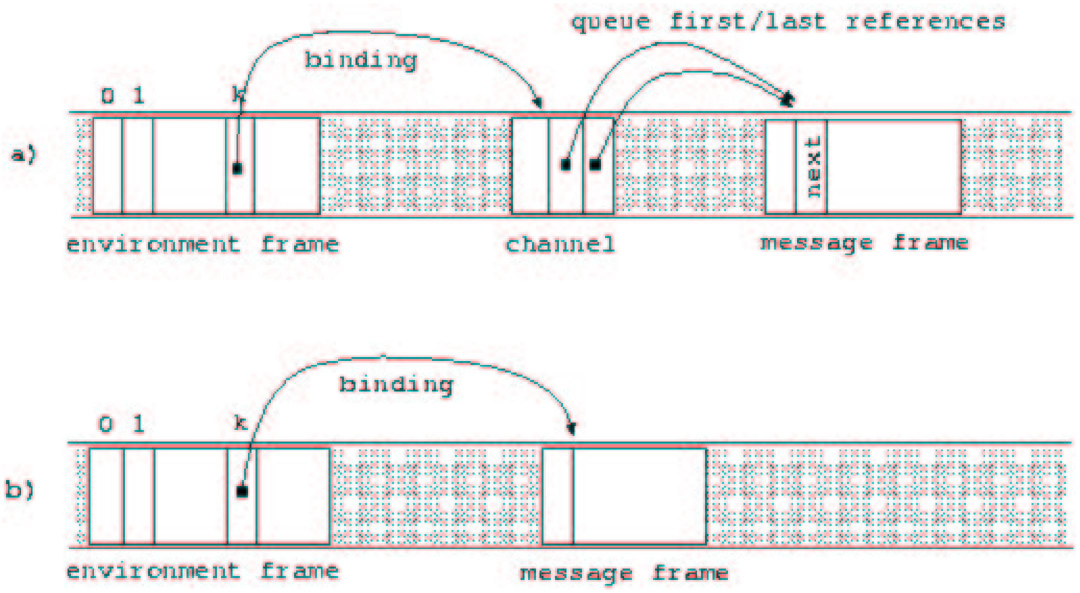

Fig. 9. Message reduction: a) general case; b) linear channel.

6 Optimizing linear channels

The run-time system of the TyCO programming language is implemented as a virtual machine [5] that emulates byte-code format program files generated by TyCO compiler [10]. Linear usage of channels enables optimizations that may substantially increase the performance of the virtual machine.

Optimization

The optimization described in the sequel can be applied to any channel for which we can ensure that it receives exactly one message and one object through its life time. Reduction, of course, also occurs exactly once. The main contribution to performance lies in the fact that, in this case, we never allocate an intermediate channel queue in the heap to hold the frames[2] for the object and for the message. Instead, we create a single frame for the first component of the redex that arrives and keep the frame reference directly in the environment frame. In other words, we let some *environment entries be directly bound to message or object frames*, as opposed to channel queues. Reduction is performed using this single frame.

In the non-optimized case, trying to reduce a message in a given channel requires testing the state of the queue (empty, no messages, no objects) and accordingly either enqueuing the message or creating a new thread in the run-queue if an object was found in the queue. The case for object reduction is symmetric. The queue is required for we have no information on the number and on the arrival order of objects and messages. Figure 9a shows the heap

[2] Also called activation records: a block of words, allocated from the heap, containing relevant information for the execution of a thread.

configuration for the general case of message reduction.

The compile-time recognition of linear channels allows the following simplifications to be performed:

- avoid the allocation of a queue in the heap to hold messages and objects (diminishes heap usage and garbage collection overheads);
- use references to messages or objects directly as bindings, minimizing indirections (increases speed);
- simplify the instruction for reduction (increases speed).

We introduce two new instructions to handle linear reduction: `forkLinearObj (k, n, t)` and `forkLinearMsg (k, n, l, a)`. Initially our binding at offset `k` in the current frame has a null reference. The first component of the redex to appear creates a frame of size `n` to hold data such as the method table `t` for the object, or the method name `l` and the arguments `a` for the message, plus some extra space for local variables. The reference for this frame is kept at offset `k`. The second component to arrive reduces using data from the instruction arguments and from the frame held at `k`. The code for these instructions may be coarsely defined as follows, where `FP` is a reference for the environment frame.

```
forkLinearObj (k, n, t) {
 if FP[k]
   newThread (FP[k], t);
 else
   FP[k] = newFrame (n, t);
}
```

```
forkLinearMsg (k, n, l, a) {
 if FP[k]
   newThread (FP[k], l, a);
 else
   FP[k] = newFrame (n, l, a);
}
```

Figure 9b shows the heap configuration in this optimized case. There is still some room for improvement. If, for example, we find that, at run-time, the object always gets executed first, we may further optimize the code by removing the test. The instruction `ForkLinearObj` simply keeps the binding for the frame created, whereas the `ForkLinearMsg` instruction produces a thread immediately.

Preliminary performance results

We wish to measure the performance increment in the virtual machine implementation that results from optimizing linear channels in programs. For this reason we chose not to perform any other optimization on the program source for our runs. To evaluate the effect of the optimization we use three metrics:

- *execution time*, measured in seconds (time) without garbage collections;
- *heap usage*, measured in machine number of words (space); and

program	time(s)			heap(kw)			♯gc			
	¬opt	opt	%	¬opt	opt	%	heap	¬opt	opt	%
`tak 22,16,8`	3.07	2.70	88	19925	17208	86	5500	67	32	47
`fib 30`	5.26	4.72	89	31617	26625	84	21000	46	22	47
`hanoi 15`	4.53	4.00	88	45023	38633	85	7560	40	33	82
`sieve 10000`	4.01	3.62	90	20385	18040	82	250	376	338	90
`mirror 5626`	0.29	0.27	93	1181	1071	90	224	140	122	87

Fig. 10. Performance results.

- number of *garbage collections* (♯gc) for a specific amount of heap memory.

The programs we use for this set of runs range from pure functional such as: `tak` (Takeuchi numbers), `fib` (Fibonacci numbers), and `hanoi` (the Towers of Hanoi); to object-based such as `sieve` (Eratosthenes' sieve) and `mirror` (mirroring a huge random tree). The results of our experiments are presented in figure 10. The arguments used for each benchmarks are also shown.

The TyCO compiler performs linearity analysis quite fast, being at most 16% slower than when using the default type inference algorithm. Note that this is only critical for very large benchmarks, as in the other cases the individual compile times are rather small. The benchmarks were run over Linux on a laptop equipped with a Pentium III at 600MHz, 256L2 cache and 256Mbytes of RAM.

As can be observed in figure 10 the preliminary results indicate an average decrease in the execution time to values around 89% those of non-optimized code. The effect of the optimization on the heap usage is also significant, with values around 85% of the non-optimized case.

These performance results may be further improved by eliminating or simplifying the code for reduction of linear channels. In terms of heap usage it is also possible to improve. In fact, the frames allocated for messages or objects at linear channels do not require some fields that are otherwise crucial in the non-optimized case (e.g., a `next` field to queue the object or message in a channel).

7 Related and future work

The framework supporting the sections 3 and 4 on type systems is adapted from the work of Igarashi and Kobayashi [3]. Our main contribution is the handling of mutually recursive process definitions, and free channel names within procedures.

The language Igarashi and Kobayashi study allows only for a simple form of definitions, namely $\mathsf{def}\ x[\vec{y}] = P\ \mathsf{in}\ Q$, where the names of definitions are conventional channels. But the x above is not a conventional channel: its input and output usage is exactly the same. On this kind of channel we are only interested on how many times the process definition can be expanded, hence the usage assigned to such a channel is (ω, κ), where the ω is there merely for technical convenience. On the other hand, TyCO features process definitions using identifiers from a syntactic category different from that of channels. As a result, we may assign a single use (the $\mathcal{U}$ in sections 3 and 4) to such identifiers.

The rules for definitions in both works follow the same pattern. In reference [3], a formula is found for the particular case of definitions $(\kappa_2 \cdot (\kappa_1 + 1))$; we have decided to parameterize the type system with a function $\mathcal{U}$ that tells how many times a procedure is called within a process. Notice that mutual recursion can only be transformed into simple recursion at the expense of code duplication; a really undesirable feature in a compiler. Nevertheless, using appropriate encodings from one calculus into the other, the type environments computed with [3] and with our type system (parametrized with the call-counting function defined in figure 8) coincide.

Further work includes the extension of the type inference system to handle recursive types and predicative polymorphism, and the study of the complexity of the call count function $\mathcal{U}$. Benchmarking with realist programs is under way.

Acknowledgements.

The authors would like to thank anonymous referees for important comments. This work was supported by projects Mikado (IST–2001–32222) and MIMO (POSI/CHS/39789/2001).

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Series in Computer Science and Information Processing. Addison-Wesley, 1974.

[2] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[3] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Journal of Information and Computation*, 161(1):1–44, 2000. An extended abstract appeared in the *Proceedings of SAS '97*, LNCS 1302.

[4] N. Kobayashi. Quasi-linear types. In *POPL'99*, pages 29–42, 1999.

[5] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, September 1999.

[6] Francisco Martins and Vasco T. Vasconcelos. TyCO + linear channels. DI/FCUL TR 01–11, Department of Informatics, Faculty of Sciences, University of Lisbon, December 2001.

[7] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[8] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *19th Annual Symposium on Principles of Programming Languages (POPL)*, pages 154–165. ACM Press, 1992.

[9] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000.

[10] Luís Lopes, Vasco T. Vasconcelos, Francisco Martins, and Rui Bastos. The TyCO programming language—compiler and virtual machine. URL: http://www.ncc.up.pt/˜lblopes/tyco, 1988–2002.

[11] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256, pages 303–313.

[12] Davide Sangiorgi and David Walker. *The π-calculus, A Theory of Mobile Processes*. Cambridge University Press, 2001.

[13] Vasco T. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.

[14] Vasco T. Vasconcelos. Core-TyCO, appendix to the language definition, yielding version 0.2. DI/FCUL TR 01–5, Department of Informatics, Faculty of Sciences, University of Lisbon, July 2001.

[15] Vasco T. Vasconcelos. Tyco gently. DI/FCUL TR 01–4, Department of Informatics, Faculty of Sciences, University of Lisbon, July 2001.

[16] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 472 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.