TyCO Gently

Vasco T. Vasconcelos

DI–FCUL TR–01–4

July 2001

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Abstract

TyCO stands for "TYped Concurrent Objects". Not that the language includes any form of primitive objects. Instead, a few basic constructors provide for a form of Object-Based Programming (that is, objects but no inheritance). The language is quite simple. The basic syntax reduces to half-a-dozen constructors. To help in writing common programming patterns, a few derived constructors are available. This report introduces TyCO by example, rather than explaining the language first and giving examples second.

TyCO Gently

This document is dedicated to all those students at the University of Lisbon who have endured programming in the TyCO programming language without such an introduction.

Vasco T. Vasconcelos
Lisbon, May 2001

Chapter 1

Introduction

TyCO [3, 2] stands for "TYped Concurrent Objects". Not that the language includes any form of primitive objects. Instead, a few basic constructors provide for a form of Object-Based Programming (that is, objects but no inheritance). The language is quite simple. The basic syntax reduces to half-a-dozen constructors. To help in writing common programming patterns, a few derived constructors are available.

This report introduces TyCO by example, rather than explaining the language first and giving examples second. The next chapter deals with the basics: channels, process, and how behaviour emerges from processes. It also describes the languages primitive types, and expressions. Chapter 3 carries on: it shows how to encode complex data structures, how to program "functions", how to deal with errors, and how to define synchronous channels. It also describes the basic input/output and program-splitting facilities. Chapter 4 introduces Object-Based Programming via a series of examples. It reveals how TyCO objects may change behaviour. Finally, chapter 5 describes types: how they are assigned to channels and to procedures.

Chapter 2

Processes and Reduction

This chapter introduces the basics of the TyCO programming language: the notion of channel and of process, and the phenomenon of nondeterminism.

2.1 Computation by Communication in TyCO

TyCO supports the idea of computation by communication. Programs in TyCO can be viewed as *processes* that communicate via message passing on shared channels. The behaviour of processes is given by a reduction relation defined on processes.

An example might help to demonstrate these ideas. Suppose you want to ask a messenger to carry a letter to a friend of yours. In TyCO you would write:

```
messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]
```

To witness the process reducing, we proceed as follows:

```
messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]
  → aFriend ! [aLetter]
```

What happened? The *message* messenger ! [aLetter, aFriend] reached the *object* messenger ? (l, f) = f ! [l], the result is the body f ! [l] where the *arguments* aLetter, aFriend replaced the *parameters* l, f. Suppose now that you have two letters to send, but a single messenger at hand.

```
messenger ! [anotherLetter, anotherFriend] |
messenger ! [oneLetter, oneFriend] |
messenger ? (l, f) = f ! [l]
  → messenger ! [anotherLetter, anotherFriend] | oneFriend ! [oneLetter]
```

In turns out that this is not the only possible outcome, as evidenced by this alternative reduction.

```
messenger ! [anotherLetter, anotherFriend] |
messenger ! [oneLetter, oneFriend] |
messenger ? (l, f) = f ! [l]
  ≡ messenger ! [oneLetter, oneFriend] |
      messenger ! [anotherLetter, anotherFriend] |
      messenger ? (l, f) = f ! [l]
  → messenger ! [oneLetter, oneFriend] | anotherFriend ! [anotherLetter]
```

Nondeterminism in message reception is an important property of TyCO programs.

Details. *The* structural equivalence relation*,* ≡*, rearranges processes, bringing messages toward objects. In this case we have used the commutativity and associativity of the parallel composition operator "*| *". Since structural equivalence is embeded in reduction,* →*, we may write:*

```
messenger ! [anotherLetter, anotherFriend] |
messenger ! [oneLetter, oneFriend] |
messenger ? (l, f) = f ! [l]
  → messenger ! [oneLetter, oneFriend] | anotherFriend ! [anotherLetter]
```

To be able to send the two messages, we would have to place two processes of the form messenger ? (l, f) = f ! [l] in parallel. Duplicating code is never a good idea: you end up with two copies to maintain. Instead we may *generalize* the behaviour of the messenger by defining a procedure to perform the message forwarding:

```
def Messenger (self) = self ? (l, f) = f ! [l]
```

Equipped with a messenger procedure, we may create as many messengers as we need. To obtain an *instance* of Messenger *located* at channel messenger we write:

```
Messenger [messenger]
```

It should be clear that Messenger [messenger] reduces to messenger ? (l, f) = f ! [l] (in the presence of the above definition). In fact the proper way to reduce the process is to *unfold* the procedure definition.

messenger ! [aLetter, aFriend] | Messenger [messenger]
  → messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]
  → aFriend ! [aLetter]

We are now in a position of having our two letters forwarded:

messenger ! [oneLetter, oneFriend] |
messenger ! [anotherLetter, anotherFriend] |
Messenger [messenger] |
Messenger [messenger]
  $\rightarrow^4$ oneFriend ! [oneLetter] | anotherFriend ! [anotherLetter]

2.2 Protecting Computations from Interferences

Processes run in concurrent environments; we can never know what these environments may bring. Programs that look correct when executed in isolation, may be "broken" by the environment. As an example, consider the letter-forwarding example of the previous section.

messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]

We have seen that, when run in isolation, it reduces to aFriend ! [aLetter]. We now place the process in an environment that, not only knows the channel messenger, but also uses it to forward a letter:

(messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]) |
messenger ! [envLetter, envFriend]
  ≡ messenger ! [aLetter, aFriend] |
    (messenger ! [envLetter, envFriend] | messenger ? (l, f) = f ! [l])
  → messenger ! [aLetter, aFriend] | envFriend ! [envLetter]

Due to nondeterminism in message reception, the message messenger ! [envLetter, envFriend] may reach the messenger object first, leaving aLetter to be delivered. To eschew the possibility of interference, we must say that channel messenger is *local*. Here is how we do it:

(**new** messenger
messenger ! [aLetter, aFriend] |
messenger ? (l, f) = f ! [l])

We are now safe to place the process in the above environment, to obtain:

(**new** messenger
messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]) |
messenger ! [envLetter, envFriend]
→ (**new** messenger aFriend ! [aLetter]) |
messenger ! [envLetter, envFriend]
≡ aFriend ! [aLetter] | messenger ! [envLetter, envFriend]

Notice that the messenger identifiers on the second and on the third lines denote different channels: they belong to different *scopes*. In fact, the whole processes may be rewritten into:

(**new** m m ! [aLetter, aFriend] | m ? (l, f) = f ! [l]) |
messenger ! [envLetter, envFriend]

where the difference is patent. Channel names introduced with **new** are *bound*, and hence subject to renaming. Also, notice that we have only made channel messenger local; channels aLetter, aFriend are still visible in the environment.

Details. *Structural congruence allows to simplify* (**new** messenger aFriend ! [aLetter]) *into* aFriend ! [aLetter]*, for channel* messenger *does not occur in its scope.*

2.3 Announcing Local Channels

Now suppose that you want to create a messenger, not to use it directly, but else to send it to a friend of yours. Further suppose that you know this friend by the channel myFriend, and that the friend's objective is to send aLetter to someone. Here's how we may proceed:

(**new** m m ? (l, f) = f ! [l] | myFriend ! [m]) |
myFriend ? (x) = x ! [aLetter, someone]
≡ (**new** m m ? (l, f) = f ! [l] | myFriend ! [m] |
myFriend ? (x) = x ! [aLetter, someone])
→ **new** m m ? (l, f) = f ! [l] | m ! [aLetter, someone]
→ someone ! [aLetter]

Details. *Structural congruence allows to extend the scope of channel* m *to encompass process* myFriend ? (x) = x ! [aLetter, someone]*, so that reduction on channel* myFriend *may happen. This is only possible because* m *does not occur free in the new piece of code that felt into its scope.*

In the last example of the previous section such a syntactic rearrangement would not be possible:

```
(new messenger
 messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l]) |
messenger ! [envLetter, envFriend]
  ≢ (new messenger
       messenger ! [aLetter, aFriend] | messenger ? (l, f) = f ! [l] |
      messenger ! [envLetter, envFriend])
```

for channel messenger *occurs in the scope that we are trying to capture. We would not be able to protect our computations from undesired interferences otherwise.*

2.4 Expressions and Primitive Types

In this section we take a look at expressions. Expressions in TyCO include atomic and composite expressions. Among the first we count the *literals* (such as the *integer* 74, the *floating point* number 0.31415e-1, the *truth value* true, and the *string* "hello"), the *variables* (that is parameters, introduced between brackets in methods or procedures), and *channel names* (introduced with a **new**). Composite expressions include, for example, 1 + 2; one step calculation yields the expression 3. Expressions are evaluated before message sending:

```
destiny ! [1 + 2]
  → destiny ! [3]
```

and before procedure unfolding:

```
AProcedure ["Ty" ^ "CO"]
  → AProcedure ["TyCO"]
```

The menu of expressions is quite limited:

1. For *integer* expressions, you can count with the five basic operations: +, -, *, /, %, where / and % represent the quotient and the remainder of integer division, respectively.

2. For (single precision) *floating point* expressions we have the four basic operators +, -, *, /.

3. There is a unary operator to convert an Integer into a Float, named toFloat.

4. *Boolean* expressions are limited to those written with the operators and, or, and not, and the literals false and true.

5. The six comparison operators (==, /=, >, >=, <, <=) are available for all types; comparison of channel types is by reference; that of strings is by contents.

6. Finally, for strings there is a means to construct and deconstruct strings. The concatenation operator is ^; the head and the tail of a string—two unary prefix operators—are hd and tl respectively. The unary operators are defined on non-empty strings only.

Here is a procedure to reverse a string.

```
def Reverse (s, r) =
  if s == ""
  then
    r ! [""]
  else
    let
      reversedTail = Reverse [tl s]
    in
      r ! [reversedTail ^ hd s]
```

2.5 Commenting Out Text

TyCO's comments are those of Haskell. Any text between "- -" and the end of the line is ignored by the compiler. TyCO also permits *nested* comments, which start with "{-" and extend until the next unmatched "-}".

Chapter 3

Data and Value Computing

This chapter shows how to program data structures, how to define procedures that work like functions, and how to deal with errors.

3.1 Value Computing

Communication in TyCO is *asynchronous* and *one-way*. Asynchronous means that the sender of a message does not wait for its reception, very much like when you place a letter in a post-box. One-way means that values are sent in one direction only, from the sender to the recipient. It is sometimes necessary to get the results of a service invocation. Suppose that the procedure call Fib [n, r] replies the n-th Fibonacci number on channel r. We may send the 5-th Fibonacci number to some friend by writing:

```
let x = Fib [5] in aFriend ! [x]
```

There's no magic in here: its all message passing. The above process is in fact an abbreviation for:

```
new r Fib [5, r]  | r ? (x) = aFriend ! [x]
  →* new r r ! [3212]  | r ? (x) = aFriend ! [x]
  → aFriend ! [3212]
```

were we have created a *fresh* channel r used to to convey the result back from the Fib procedure.

Details. *Notice that, although* Fib *is a binary procedure (taking an integer, and a channel to convey the result), it is used with a single argument in the right-hand side of a* **let** *equation. The purpose of* **let***-processes is to hide the second argument.*

How do we program the Fib procedure? Recall its definition.

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

From this definition, a TyCO process can be defined.

```
def Fib (n, r) =
  if n == 1 or n == 2
  then r ! [1]
  else if n > 2
  then
    let v1 = Fib [n - 1]
        v2 = Fib [n - 2]
    in r ! [v1 + v2]
```

This is called a *recursive* procedure definition because Fib "refers to itself" on the right-hand side of the **def**-equation.

Details. *Note that more than one equation is allowed in a let process. The above* **let** *process is an abbreviation for*

```
new r1 Fib [n - 1, r1] |
new r2 Fib [n - 2, r2] |
r2 ? (v2) = r1 ? (v1) = r ! [v1 + v2]
```

where Fib [n - 1, r1] *and* Fib [n - 2, r2] *run in parallel, and the newly created channels* r1 *and* r2 *are freshly created. The above* **let** *process is different from:*

```
let v1 = Fib [n - 1]
in let v2 = Fib [n - 2]
in r ! [v1 + v2]
```

which constitutes an abbreviation for the process

```
new r1 Fib [n - 1, r1] |
r1 ? (v1) = new r2 Fib [n - 2, r2] |
            r2 ? (v2) = r ! [v1 + v2]
```

where Fib [n - 1, r1] *terminates before* Fib [n - 2, r2] *is launched.*

There is only one problem with this solution: the function is horribly inefficient. It is easy to see that computation blows up exponentially, since many calculations are being repeated. In section 3.7 we devise a more efficient solution.

3.2 Computing with Data

Values in TyCO are either of a primitive type (integer, float, Boolean, string) or are channels. More complex, structured data must be built from the only material available: processes and channels.

Pairs are an example of a *data structure*, and data structures are easily defined in TyCO. To define a pair of values we have to write some code: a pair, located on some channel p is a process that, when invoked, replies its components f and s:

```
def Pair (p, f, s) = p ? (r) = r ! [f, s] | Pair [p, f, s]
```

As an application, we may write a little procedure that concatenates the strings in a pair.

```
def Concat (p, r) =
    let f, s = p ! []
    in r ! [f ^ s]
```

We can easily check that is takes three steps to concatenate the components of a pair of strings (plus two to unfold the definitions).

```
Pair [p, "Hello ", "TyCO"] | Concat [p, r]
  → Pair [p, "Hello ", "TyCO"] |
    let f, s = p ! [] in r ! [f ^ s]
  → (p ? (r) = r ! ["Hello ", "TyCO"] | Pair [p, "Hello ", "TyCO"]) |
    let f, s = p ! [] in r ! [f ^ s]
  → Pair [p, "Hello ", "TyCO"] |
    new r1 r1 ! ["Hello", "TyCO"] | r1 ? (f, s) = r ! [f ^ s]
  → Pair [p, "Hello ", "TyCO"] | r ! ["Hello" ^ "TyCO"]
  → Pair [p, "Hello ", "TyCO"] | r ! ["Hello TyCO"]
```

Details. *Notice the two variables on the left-hand side of the* **let** *equation.*

3.3 A Word on Scope

In TyCO, "scopes extends as far to the right as possible". This means that, in the process

```
new r Fib [5, r] | r ? (x) = aFriend ! [x]
```

the (sub-) process r ? (x) = aFriend ! [x] falls within the scope of the **new** r, requiring no parenthesis to make this explicit. It also means that, in the definition

```
def Pair (p, f, s) = p ? (r) = r ! [f, s] | Pair [p, f, s]
```

the scope of Pair extends to the right, again requiring no parenthesis to the effect. But it also means that, in the process

```
p ? (r) = r ! [f, s] | Pair [p, f, s]
```

the procedure call Pair [p, f, s] falls within the scope of the receptor p ? (r). This is certainly what is intended for the Pair procedure. What if not? What if we want the procedure call to fall outside the scope of the receptor? We have two choices:

1. Use parenthesis (appropriately):

   ```
   (p ? (r) = r ! [f, s]) | Pair [p, f, s]
   ```

2. Take advantage of the commutativity of the parallel composition operator:

   ```
   Pair [p, f, s] | p ? (r) = r ! [f, s]
   ```

3.4 Inductively Defined Data Structures

There are situations when we must deal with collections whose size is not certain: it is useful to represent such a collection as a *list* whose length is arbitrary. Lists are inductively defined: a list is either empty, or a pair composed of an element (the head of the list) and a list (the tail). The former will be represented by a process Nil, the latter by a process Cons. Each of these processes need a location where they may be queried; in addition, Cons needs to know its head and its tail. Nil is a process that, when invoked, answers a message labelled with label nil; Cons in turn answers cons together with its head and its tail. Here is a possible implementation.

```
def Nil (l) = l ? (r) = r ! nil [] | Nil [l]
    Cons (l, h, t) = l ? (r) = r ! cons [h, t] | Cons [l, h, t]
```

Defining list values is cumbersome, for TyCO provides no special syntax for it. The following process describes a list composed of elements 1, 2, and 3 (in this order), and located at channel l.

```
def List123 (l) =
    new n Nil [n] |
    new c1 Cons [c1, 3, n] |
    new c2 Cons [c2, 2, c1] |
    Cons [l, 1, c2]
```

Computation on lists proceeds by taking them apart: given the location of a list, we want to know whether it is a Nil or a Cons, and in the latter case we also want to know (most of the times) the list's head and tail. Here is how we write the Null predicate:

```
def Null (l, r) =
  case l ! [] of {
    nil () = r ! [true]
    cons (_, _) = r ! [false]
  }
```

A **case** process takes two reduction steps (plus two to unfold the definitions):

```
List123 [l] | Null [l, r]
  →⁴ List123 [l] | r ! [false]
```

Details. *The idea of the* **case** *process constructor is similar to that of the* **let**. *The difference is that the latter replies a message always labelled with* val, *whereas in the former we must be ready for different labels. The above* **case** *is an abbreviation for the process:*

```
new s l ! [s] | s ? { nil () = r ! [true] cons (_, _) = r ! [false] }
```

We leave to reader the exercise of expressing (one equation) **let** *in terms of* **case**.

As an example of a recursive definition, let us try to compute the number of elements present in a list.

```
def Length (l, r) =
  case l ! [] of {
    nil () =
      r ! [0]
    cons (_, t) =
      let n = Length [t]
      in r ! [n + 1]
  }
```

Here is an example of Length in action:

```
List123 [l] | Length [l, r]
  →² new n Nil [n] | ... | Cons [l, 1, c2] | case l ! [] of { ... }
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     let n = Length [c2] in r ! [n + 1]
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1 case c2 ! [] of {...} | r1 ? (n) = r ! [n + 1]
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1 let n1 = Length [c1] in r1 ! [n1 +1] | r1 ? (n) = ...
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1, r2 case c1 ! [] of {...} | r2 ? (n1) = r2 ! [n1 + 1] | ...
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1, r2 let n2 = Length [n] in r2 ! [n2 +1] | r2 ? (n1) = ...
  →² new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1, r2, r3 r3 ! [0] | r3 ? (n2) = r2 ! [n2 + 1] | ...
  → new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1, r2 r2 ! [1] | r2 ? (n1) = r1 ! [n1 + 1] | ...
  → new n Nil [n] | ... | Cons [l, 1, c2] |
     new r1 r1 ! [2] | r1 ? (n) = r ! [n + 1]
  → List123 [l] | r ! [3]
```

Suppose that l1 and l2 are the locations of two lists, and that we want to compute (more precisely, to write on channel r) the sum of their lengths.

```
let n1 = Length [l1]
    n2 = Length [l2]
in r ! [n1 + n2]
  →* r ! [n1 + n2]
```

There is an alternative definition for the procedure Length. The idea is to setup a persistent, stateless object (a server) that accepts requests for computing the length of lists.

```
def LengthServer (self) =
  self ? (l, r) =
    case l ! [] of {
      nil () =
        r ! [0]
      cons (_, t) =
        let n = self ! [t]
        in r ! [n+1]
    } |
  LengthServer [self]
```

Using this second version we need a single instance of LengthServer for the whole program: requests for computing the length of lists are all directed to the same server. Here is how we compute the sum of the lengths of lists l1 and l2.

```
new length
LengthServer [length] |
let n1 = length ! [l1]
    n2 = length ! [l2]
in r ! [n1 + n2]
  →* (new length LengthServer [length]) | r ! [n1 + n2]
```

Which one to choose? I feel this is largely a matter of taste. Nevertheless, notice that, in this case, LengthServer remains, whereas in the previous case no trace of Length appears in the resulting process. We have placed the parenthesis in the resulting process to emphasize that the LengthServer is no longer inaccessible and hence may be garbage collected.

3.5 Splitting Your Program

TyCO provides for a rather crude means of splitting a program amongst several files. To sort the code in the previous section, we could place the list constructors:

```
def
Nil (l) = l ? (r) = r ! nil [] | Nil [l]
Cons (l, h, t) = l ? (r) = r ! cons [h, t] | Cons [l, h, t]
in
```

in a file and call it listConstructors.tyc. Notice that the above code does not

constitute a TyCO program: the process after the **in** is missing. I suggest using the extension ".tyc" to distinguish files that are to be included, from those that constitute programs. We may then prepare another file for the list operations, and call it listOperations.tyc:

```
include "listConstructors.tyc"
def
Null (l, r) =
  case l ! [] of {
    nil () = r ! [true]
    cons (_, _) = r ! [false]
  }
...
in
```

An application would then start with **include** "listOperations.tyc", and go on with the process that constitutes the "main" program.

Details. *No harm arises from including both* listConstructors.tyc *and* listOperations.tyc*:*

```
include "listConstructors.tyc"
include "listOperations.tyc"
```

File listConstructors.tyc *gets included twice: the definitions of* Nil *and* Cons *in the first include are immediately hidden by that in the second include. No harm; just waste.*

3.6 Errors

Before closing this chapter, let's us try one more operation on lists: the function Last that returns the last element of a non empty list. The first problem we must face is what to do when the list, argument to the function, is non-empty. Our choices are as follows:

1. We can ignore the problem. Indeed, if all we are concerned with is the proper behaviour of Last on non-empty lists, then we may ignore calls conveying empty lists, by just not replying to these sort of calls. The case for the empty list would be just the terminated process **inaction**. The problem with this solution is that clients that inadvertently invoke Last with an empty list will wait forever, without even realising that calls to Last "may not be answered".

2. We could fix a channel, say error, where to signal erroneous operations on last. This channel could be global to the function, or passed as an argument. The case for the empty list would be handled by issuing a message error ! ["last: empty list"].

3. Alternatively, we could make the client aware that calls may go wrong, by replying two sorts of messages: messages labelled with nothing for invalid calls, and labelled with just for the last element of the list.

The code for the three solutions is quite similar; we develop that of the last.

```
def Last (l, r) =
  case l ! [] of {
    nil () =
      r ! nothing ["Last: empty list"],
    cons (h, t) =
      case t ! [] of {
      nil () =
        r ! just [h]
      cons (_, _) =
        Last [t, r]
    }
  }
```

These sort of functions may not be invoked with a **let**-process as before. Instead, the client must explicitly wait for the two possible replies.

```
case Last [l] of {
  nothing (s) = -- Handle the error somehow
  just (e) = -- Do something with e
}
```

Details. *The* **case** *within the* **case** *is a nuisance. Functional languages allow to consider three cases:* nil (), cons (_, nil ()), *and* cons (_, t). *In TyCO, one can only deconstruct data structures one step at a time, hence the nested pattern* cons (_, nil ()) *needs two nested* **case***s.*

3.7 Synchronous Message Passing and Streams

Channels are used to transmit information. We have seen on section 2.1 that TyCO makes no assumption on the order of reception of two messages sent

on the same channel: communication is *asynchronous*. There are cases when we would like to proceed only when a message gets accepted by its recipient, that is, we would like to communicate *synchronously*. Asynchronous channels can be converted into synchronous channels, by following a simple protocol.

Let us start with a simple producer-consumer system communicating via a *stream*. A process Ints produces consecutive integers some output stream:

```
Ints (next, outStream) =
  outStream ! [next] ;
  Ints [next + 1, outStream]
```

Here is how the protocol works on the sender side: together with whatever is sent on the stream (the integer next, in this case), we sent a newly created channel; before proceeding we wait for a done message at this channel. The process above is an abbreviation of the more verbose process below, where the protocol is made explicit.

```
Ints (next, outStream) =
  new r
  outStream ! [next, r]  |
  r ? { done () = Ints [next + 1, outStream] }
```

Another process, Odds, forwards the odd integers on some other stream, ignoring even numbers.

```
Odds (inStream, outStream) =
  inStream ? (x) :
  if x % 2 == 1
  then outStream ! [x] ; Odds [inStream, outStream]
  else Odds [inStream, outStream]
```

To match the protocol of the client, the server side of the stream receives, not only whatever is transmitted on the stream, but also a synchronization channel. Then, it replies a done message on this channel. The process below, equivalent to the above, makes the protocol explicit.

```
Odds (inStream, outStream)=
  inStream ? (x, r) =
    r ! done [] |
    if x % 2 == 1
    then
      new s
      outStream ! [x, s] |
      s ? { done () = Odds [inStream, outStream] }
    else
      Odds [inStream, outStream]
```

Details. *One might be tempted to simplify the* Odds *procedure by replacing the two recursive calls with a single call, placed right after the* **if-then** *process.*

```
Odds (inStream, outStream)=
  inStream ? (x) :
  ( if x % 2 == 1 then outStream ! [x] ) ;
  Odds [inStream, outStream]
```

Such a piece of code is not syntactically correct: on left of a semicolon one can only place a message or a procedure call. We thus see that the semicolon operator in TyCO is quite distinct from that of conventional imperative programming languages.

3.8 Efficient Fibonacci

We are now ready to devise an efficient solution for the Fibonacci problem discussed in section 3.1. Rather than explicitly computing Fib [n - 1] and Fib [n - 2], we put the already computed Fibonacci numbers on a stream.

The following solution, taken from [1], uses a *stream duplicator* and a *stream adder*. A stream duplicator re-sends whatever comes in a given input stream into two output streams. Here is our first try.

```
Duplicator (inStream, outStream1, outStream2) =
  inStream ? (x) :
  outStream1 ! [x] ;
  outStream2 ! [x] ;
  Duplicator [inStream, outStream1, outStream2]
```

The above solution is purely sequential. We wait for the acknowledgment from the first stream before sending on the second. To send on two streams

in parallel, we must hand-code the protocol.

```
Duplicator (inStream, outStream1, outStream2) =
  inStream ? (x) :
  new r1, r2
  outStream1 ! [x, r1] |
  outStream2 ! [x, r2] |
  r1 ? { done () =
    r2 ? { done () =
      Duplicator [inStream, outStream1, outStream2]
    }
  }
```

A stream adder gets two integers on two given streams; outputs its sum on another given stream.

```
Adder (inStream1, inStream2, outStream) =
  inStream1 ? (x) :
  inStream2 ? (y) :
  outStream ! [x + y] ;
  Adder [inStream1, inStream2, outStream]
```

To prepare a procedure that feeds a stream with the Fibonacci numbers, we only have to be careful about the "plumbing".

```
Fibs (f1) =
  new f2, tf1
  Duplicator [tf1, f1, f2] | tf1 ! [1] ; inaction |
  new tf2, ttf
  Duplicator [ttf, tf1, tf2] | ttf ! [1] ; inaction |
  Adder [f2, tf2, ttf]
```

If we are interested on the n-th Fibonacci number, rather than the whole lot, we may write a function that takes the the n-th element of a (generic) stream.

```
Take (n, inStream, r) =
  inStream ? (x) :
  if n == 0
  then r ! [x]
  else Take [n - 1, inStream, r]
```

We are finally in a position to write the function that computes the n-th Fibonacci.

```
Fib (n, r) =
  new stream
  Fibs [stream]  |
  Take [n, stream, r]
```

Details. *Notice the* tail-call optimization*: the* Fib*'s reply-to channel* r *is passed to the* Take *procedure. This, in turn, will reply directly to the* Fib*'s client. The alternative is to replace the last line with* **let** x = Take [n, stream] **in** r ! [x].

3.9 Basic Input/Output

TyCO quietly runs any program that successfully compiles: it needs no input to start execution; it yields no value. How do we then read what the program is supposed to produce? and how may we influence the behaviour of programs by providing input at adequate places? TyCO input/output system is quite crude: I/O operations are from the standard input (usually, the keyboard) and to the standard input (usually, the screen), and are managed by an object named io.

For each primitive type—Integer, Float, Boolean, String—there is a method—geti, getf, getb, gets—in the io object that reads a value from the standard input.

Again, for each primitive type, there is a method—puti, putf, putb, puts—in the io object that prints the value of expressions in the standard output. In order for a client to be able to print two values in a sequence, these method return an acknowledge (adone-labelled message) on a channel supplied by the client.

When the acknowledgment is not important—either because it is the last expression in a series of expressions to print, or because the order is not important—the io objects provides a third series of methods: printi, printf, printb, and prints.

Here is how we read two Boolean values, and print its conjunction and its disjunction.

```
let b1 = io ! getb []
    b2 = io ! getb []
in io ! puts ["\nThe conjunction is "] ; io ! prints [b1 and b2]  |
   io ! puts ["\nThe disjunction is "] ; io ! prints [b1 or b2]
```

We can almost specify the behaviour of the io object. We concentrate on the methods for integer values; the methods for the remaining primitive types are similar.

```
def IO () =
    io ? {
      geti (r) =
        let x = -- Read an integer
        in r ! [x] | IO []
      puti (x, r) =
        -- Write the value of x and then
        r ! done [] | IO []
      printi (x) =
        io ! puti [x] ; inaction
      ...
    }
in IO []
```

Chapter 4

Object-based Programming

This chapter introduces object-based programming in TyCO. We say "based" (and not "oriented") because the language offers no support for code reuse. As mentioned in the introduction, TyCO incorporates no primitive notion of objects, at least not in the form that we are used to in object-oriented languages: persistent (ie., that survive method invocations), allocating local variables (there is no primitive notion of imperative variables), offering a collection of distinct services, at most one object per reference, and created via some distinguished expression (new in Java).

TyCO support for objects comes into two forms: *recursive procedures* to account for persistence and for a form of local variables, and branching structures to account for service offering.

4.1 A Bag of Things

Let us start with a bag that keeps items of some kind. We allow to put and to get elements in and from the bag, and provide a means to ask the number of elements in the bag, the size of the bag.

Before we proceed, we must agree on a representation for the elements in the bag. We could use a list as defined in section 3.4, but a better solution is to use *messages in transit*. The state of the bag is given by a series of messages in transit, all targeted at the same (local) channel, say contents, one for each element in the bag. To get an item, we read from contents; to put an item, we write on contents. Here's a possible solution.

```
def Bag (self) =
  new contents
  def Go (size) =
    self ? {
      get (r) =
        Go [size - 1] |
        contents ? (x) = r ! [x]
      put (e) =
        Go [size + 1] |
        contents ! [e]
      size (r) =
        Go [size] |
        if size > 0 then r ! [size] then r ! [0]
    }
  in Go [0]
```

We shall use the identifier self for the location of objects (if you dislike the name, try this, or else your favourite identifier). A few points are worth noticing:

- Procedures that generate objects are of a special form: **def** X (...) = self ? { *methods* }.
- Each method re-constructs the object if and *when* needed.
- There are no (imperative) variables: object's attributes are all updated at the same time, when the procedure recurs.
- We use a definition inside another definition. Each serves its purpose: the outermost hides channel contents and initializes size, the innermost provides for recursion and for the update of size.
- The value of size changes throughout the life of the object: we pass it as an argument to Go. The channel attached to variable contents is kept constant: no need to pass it as an argument to Go. The same applies to self.

Notice that the body of method get cannot be rewritten as

```
contents ? (x) = r ! [x] | Go [size - 1]
```

for a get on an empty bag would hang the bag (recall that the scope "=" encompasses Go [size - 1]). Notice also that size reflects the number of ele-

ments in the bag, if positive, and the number of clients waiting on a get, if negative.

In a concurrent setting, one must be conscious about the usefulness of testing the size of bag before getting something. For example:

```
let n = aBag ! size []
in if n /= 0 then let x = aBag! get []
```

may hung forever when placed in a suitable environment. On the other hand,

```
let n = aBag ! size []
in if n == 0 then let x = aBag! get []
```

may succeed pretty fast. We leave as exercise the description of the necessary environments for both cases.

4.2 A Simple Bank Account

We proceed with a bank account procedure that keeps a balance, and serves requests for deposit deposit, withdraw and balance. The procedure needs two parameters: the current balance (balance) and its location (self). Here is a possible solution:

```
def Account (self, balance) =
  self ? {
    deposit (amount) =
      Account [self, balance + amount]
    balance (replyto) =
      replyto ! [balance] |
      Account [self, balance]
    withdraw (amount, replyto) =
      if amount >= balance
      then
        replyto ! overdraft [] |
        Account [self, balance]
      else
        replyto ! dispense [] |
        Account [self, balance - amount]
  }
```

In the above definition, there is no difference between the location of

an instance (self) and an attribute (balance): they are all parameters to the procedure. Even though, the channel attached to variable self is kept constant (that is often the case), in this case there is no advantage in using the technique of section 4.1 with nested definitions.

We may now create an instance of Account and perform a few operations. Object creation, in this case, is no different from a regular procedure call: we provide an argument for each of the parameters in the procedure.

```
new a
Account [a, 50] |
a ! deposit [20] |
case a ! withdraw [60] of {
  overdraft () =
    io ! prints ["Did not make it!"]
  dispense () =
    io ! prints ["Got it!"]
}
```

4.3 The Statement of an Account

Suppose that we would like to extend our account in order to provide for a new service: a list of all the (successful withdraw or deposit) operations. The first thing we have to realise is that we cannot reuse any code from the above section, the most we can do is cut-and-paste.

In order to manage the history of the account we rely on a list of operations: DepositOperation and WithdrawOperation nodes hold the amount, and a reference to the next element in the list; EmptyOperation terminates the list. Each element in the list provides for a single operation: toString; other operations can be added (counting the number of deposit/withdraw operations, or the total amount deposited/withdrawn), if needed.

```
def DepositOperation (self, amount, next) =
  self ? {
    toString (replyto) =
      DepositOperation [self, amount, next] |
      let a = IntToString [amount]
          n = next ! toString []
      in replyto ! ["\ndeposit " ^ a ^ n]
  }
```

```
WithdrawOperation (self, amount, next) =
  self ? {
    toString (replyto) =
      WithdrawOperation [self, amount, next] |
      let a = IntToString [amount]
          n = next ! toString []
      in replyto ! ["withdraw " ^ a ^ n]
  }
```

```
EmptyOperation (self) =
  self ? {
    toString (replyto) =
      replyto ! [""]
  }
```

DepositOperation and WithdrawOperation's toString methods are quite similar: each converts its amount into a string using function IntToString and converts the rest of the list into a string by invoking the toString method on next. The difference is that the former replies "deposit" together with the gathered information, whereas the second replies "withdraw". EmptyOperation terminates the list. To toString requests, it answers the empty String.

Notice that we may in-distinctively place in the list DepositOperation and WithdrawOperation objects: the only restriction is that they have the same *interface*. In this case, each accepts a single method named toString, requiring for argument a channel capable of (at least) accepting a val-labelled message containing a string. More on types in chapter 5.

For the bank account itself, we start from the Account in the previous section, and add a new attribute: operations, a new method: statement, and change methods deposit and withdraw to register the operations. Here is a possible solution:

```
def Account (self, balance) =
  new operations
  EmptyOperation [operations] |
  def Go (balance, ops) =
    self ? {
      deposit (amount) =
        new op
        DepositOperation [op, amount, ops] |
        Go [balance + amount, op]
      balance (replyto) =
        replyto ! [balance] |
        Go [balance, ops]
      withdraw (amount, replyto) =
        if amount >= balance
        then
          replyto ! overdraft [] |
          Go [balance, ops]
        else
          replyto ! dispense [] |
          new op
          WithdrawOperation [op, amount, ops] |
          Go [balance - amount, op]
      statement (replyto) =
        ops ! toString [replyto] |
        Go [balance, ops]
    }
  in Go [balance, operations]
```

Let us analyse the deposit method: we place at the head of list ops a new DepositOperation cell located at a new channel and with the given amount. Then we recur with the new list and the updated amount. Method withdraw only performs this operation if successful; this time we use a new WithdrawOperation list node. In method statement, the call toString replies directly to the client. If it important that the account remains unavailable while the statement is being generated, we may rewrite the method as follows:

```
statement (replyto) =
  let str = ops ! toString []
  in replyto ! [str] |
    Go [balance, ops]
```

4.4 Changing the Behaviour of Objects

The last section of this chapter deals with a feature unusual on most object-oriented programming languages: the ability to change the behaviour of objects half-way through computation.

To illustrate this feature, we develop a program to produce the prime numbers using the algorithm of Eratosthenes. We start with a procedure that produces integer values on some output stream: for example the Ints procedure in section 3.7.

The stream produced by Ints is 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 then in 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] ;
    Sieve [inStream, grain, outStream]
  else
    Sieve [inStream, grain, outStream]
```

An invariant of the program is 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 (call it n) ever arrives the last chain, it must be a prime. The Sink then outputs the number on a given stream, 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, outStream) =
  inStream ? (n) :
  outStream [n] ;
  new newsieve
  Sink [newsieve, outStream]  |
  Sieve [inStream, n, newsieve]
```

Notice how the change of behaviour is accomplished:

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

The only restriction is that channels Sink's inStream and Sieve's inStream share the same type: a stream of integers, in this case (more on types in chapter 5).

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 I decided to call stream. The function that computes the n-th prime is then:

```
Primes (n, r) =
  new inStream, primeStream
  Ints [2, inStream] |
  Sink [inStream, primeStream] |
  Take [primeStream, n, r]
```

Chapter 5

Types?

Up to this point we have not mentioned types. We have not written a single piece of type information in our programs. This does not mean that TyCO is untyped: you do not write types; instead the compiler infers them for you. TyCO is *implicitly typed*.

Types are useful in early detection of programming errors, as well as in guiding the compiler's code generation process. Types prevent you from writing programs that may call methods for which a target object may not be prepared to deal with.

Types stay in the background of a typed program except when a type error occurs, in which case the error message may refer to the type conflict. Programmers that write well-typed programs at the very first try would never notice types. Since the class of such programmers is possibly empty, this section introduces types, and how they look like in error messages.

5.1 Monomorphic Types

The TyCO compiler infers a type for each expression. The primitive types are called Integer, Float, Boolean, and String. Types for channels describe the sort of messages the channel may carry. Take for example the following piece of code.

```
mod3 ? { do (n) = replyto ! val [n % 3] }
```

What do we know about identifier mod3? That it is a channel that carries a do-labelled message, whose contents is an integer. We write this information in the form below.

```
{do: Integer}
```

What do we know of identifier replyto? That it is a channel that carries a val-labelled message composed of an integer. A type for replyto is:

{val: Integer}

This is not the only type for the channel. "val: Integer" is one kind of message that channel replyto may transmit; there may be others. In fact, the above program does not allow us to infer much about the messages that channel replyto may carry: all we know is that it carries *at least* val-labelled messages carrying an integer. The *principal type* for replyto is:

⟨val: Integer⟩

Consider now the following variant of the mod3 function, where channel replyto has been made a parameter:

```
mod3bis ? { do (n, replyto) = replyto ! val [n % 3] }
```

The principal type of mod3bis is, as before, a record with a single component, labelled with do. The difference is that the do method has now two parameters: the Integer (as before), and the type of the replyto channel:

{do: Integer ⟨val: Integer⟩}

For another example, the principal type of the location of a simple bank account (described on page 24), channel self, is:

```
{balance: ⟨val: Integer⟩,
 deposit: Integer,
 withdraw: Integer ⟨dispense: , overdraft: ⟩}
```

5.2 Polymorphic Types

Computation in TyCO proceeds by message exchanging; we have seen that in Chapter 2. There are times however when one needs to store values for later retrieval. An imperative variable is exactly that: it allows writing now and reading later. A simple variable cell can be written as follows:

```
Cell (self, value) =
  self ? {
    read (replyto) =
      replyto ! [value]  |
      Cell [self, value]
    write (newValue) =
      Cell [self, newValue]
  }
```

The variable cell stores a *value*. What should its type be? We need not be particular about the precise type of its value; just call it a. What is the type of channel self? It carries two kinds of messages: read and write. The former carries a value of the same type of value, that is, a; the latter carries a replyto channel: we have seen in Section 5.1 that its type is ⟨val: a⟩. Putting it all together, the type of self is:

{read: ⟨val: a⟩, write: a}

Details. *Generic names for types, such as* a *above, are called* type variables, *and are uncapitalized to distinguish from specific types such as* Integer.

The type for procedure Cell comprises the type for its arguments, self and value, plus an indication that "a is any type":

forall a. {read: ⟨val: a⟩, write: a} a

Given the variable Cell procedure, we may now construct different cells. The example below creates an integer cell, and a bank account (page 24) cell.

```
new myAccount Account [myAccount, 500]  |
new anAccountCell Cell [anAccountCell, myAccount]  |
new anIntegerCell Cell [anIntegerCell, 25]
```

What are the types for channels anAccountCell and anIntegerCell? That of the Cell's first parameter—{read: ⟨val: a⟩, write: a}—with type variable a replaced by the appropriate type: that of the first parameter of procedure Account (page 31), and Integer, respectively. For the integer cell we get the type:

{read: ⟨val: Integer⟩, write: Integer}

I'm sure you can figure out the type for channel anAccountCell.

Details. *You may wonder why we have to be explicit about the genericity of type*

variable a. *Consider the following process:*

```
def Y (y) = x ! do [y]
```

The type of y *is any, say* a*; then, that of* x *is* ⟨do: a⟩. *We cannot generalise the type of* Y *to* **forall** a. a*, for the type variable* a *appears in the type of* x. *Now consider the following convoluted way of writing* u ! do [5] | v ! do [false]*:*

```
def X (x, i) =
  def Y (y) = x ! do [y]
  in Y [i]
in
X [u, 5] |
X [v, false]
```

We have seen that the type of Y *is* a*, and that of* x *is* ⟨do: a⟩*; it should be easy to see that the type of* i *is that of* y*, that is,* a. *Then, the monomorphic type for* X *is the sequence comprising the type for* x *and for* i*, that is,* ⟨do: a⟩ a. *In this type sequence, type variable* a *may be generalised, yielding* **forall** a. ⟨do: a⟩ a*, allowing to type the process* X [u, 5] | X [v, false].

5.3 Polymorphism and Multiple Definitions

One must understand the nature of (parametric) polymorphism. Recall the variable cell of the previous section, and suppose we want to write two constructors: one to create a floating point cell with initial value 0.0; another to create a Boolean cell, initial value false. We could prepare the following code

```
def
  Cell (self, value) = ...
  FloatCell (self) = Cell [self, 0.0]
  BooleanCell (self) = Cell [self, false]
in ...
```

But such a program is not typable. My favourite compiler issues the following message:

> For the Cell procedure call in line 4, I was expecting the type Float but found the type Boolean.

The reason is that all definitions in a **def**-**in** are typed together; generalisation (the **forall** thing) is done only after. To fix the problem, a crucial **in def** comes to rescue:

```
def
  Cell (self, value) = ...
in def
  FloatCell (self) = Cell [self, 0.0]
  BooleanCell (self) = Cell [self, false]
in ...
```

Now, the Cell procedure is typed in isolation, and the types of its parameters generalised. Then we can construct two concrete (float and Boolean) cases, as we have seen in the previous section.

5.4 Types for Inductively Defined Data Types

Recall the list constructors of page 11.

```
def Nil (l) = l ? { val (r) = r ! nil [] | Nil [l] }
    Cons (l, h, t) = l ? { val (r) = r ! cons [h, t] | Cons [l, h, t] }
```

What are the types of the location l of each of these constructors? No magic: the principal types of Nil and Cons are respectively:

```
{val: ⟨nil: ⟩}
forall a b. {val: ⟨cons: a b⟩}
```

There is nothing "list-ish" about these types. The former is the type of an empty tuple, the latter that of a pair. They are not even related. It is the way we use these constructors that make the "list nature" we have thought for them arise. Recall from page 13 the function that computes the length of a list. This time we use no abbreviations.

```
Length (l, r) =
  new r1
  l ! val [r1] |
  r1 ? {
    nil () =
      r ! val [0]
    cons (_, t) =
      new r2
      Length [t, r2] |
      r2 ? {val (n) = r ! val [n + 1]}
  }
```

What does Length expects from its parameter l? That it answers (at least) val-requests. And what do these answers look like? There are two kinds: nil and cons. The first component of cons is any (of type, say, a); the second is an argument to Length (on line 9), thus of the same type of l. The type of a list of elements of type a is then

```
x where x = ⟨val: {nil: , cons: a x}⟩
```

The type of Length is now easy to derive:

```
forall a. x ⟨val: Integer⟩
  where x = ⟨val: {nil: , cons: a x}⟩
```

To make it clear that there is nothing "list-ish" about the Nil and Cons constructors, we use them to write trees, a particular form of trees: nodes are Cons cells; leafs are Nil cells. The only information the tree carries is its structure. We may then write a function to count the number of leafs in such a tree.

```
Count (tree, replyto) =
  case tree ! [] of {
    nil () =
      replyto ! [1]
    cons (left, right) =
      let
        n1 = Count [left]
        n2 = Count [right]
      in
        replyto ! [n1 + n2]
  }
```

The type for variable tree reveals the tree structure induced by the Count function:

```
x where x = ⟨val: {nil: , cons: x x: }⟩
```

The type of Count is the sequence:

```
x ⟨val: Integer}⟩
  where x = ⟨val: {nil: , cons: x x: }⟩
```

5.5 Record Polymorphism

There is a second form of polymorphism that is not introduced with keyword **forall**. We have seen in section 3.4 how to define list constructors and how to define the Null predicate. From section 5.4, recall that Cons has the following type

forall a b. {val: ⟨cons: a b⟩}

Then, it should not be difficult to see that the replyto name, implicit in the Null predicate (page 12), has the type

{cons: c d, nil: }

where the *open record* ⟨cons: a b⟩ has become a *closed record* {cons: c d, nil: }, with a second component nil.

We now define binary trees, and the LeafTree predicate. For the internal nodes we use the Cons cell; for leaves we define a Leaf cell.

```
include "listConstructors.tyc"
def Leaf (l, x) = l ? (r) = r ! leaf [x] | Leaf [l, x]
in def LeafTree (t, r) =
  case t ! [] of {
    leaf () = r ! [true]
    cons (_, _) = r ! [false]
}
```

In this case, the replyto, name implicit in the LeafTree predicate, has the type

{cons: e f, leaf: g}

where the open record ⟨cons: a b⟩ was closed with a *different* component leaf to become {cons: e f, leaf: g}.

5.6 Expressions and Polymorphism

The equality operator is defined on all types: one can compare integer expressions, float expressions, Boolean expressions, string expressions, and channels. However we cannot write a polymorphic comparator: a procedure that outputs on its third parameter the result of comparing the first two arguments, such as:

```
X (a, b, r) = r ! [a == b]
```

Details. *The reason is of a pragmatic nature. One the one hand we do not want to keep type information at runtime. One the other hand, TyCO 0.2 provides no means to annotate programs with type information.*

Still, such procedures are sometimes useful. For each primitive type, there is get-around solution. The idea is to take advantage of the neutral element of some operator on the type (say, `0` for integer +, `0.0` for float +, `true` for `and`, `""` for ^), to implicitly say which type we are comparing. For example, a comparator for Boolean values can be written as follows.

```
X (a, b, r) = r ! [a == (b and true)]
```

Similarly, we cannot write an adder that is good both for integer and for float values. The trick of the neutral element works here as well. An adder of float values can be written as follows.

```
X (a, b, r) = r ! [a + b + 0.0]
```

Bibliography

[1] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

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

[3] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Informatics, Faculty of Sciences, University of Lisbon, March 1998.