Processes and Functions

Silvio Lemos Meira
Departamento de Informática, Universidade Federal de Pernambuco
PO Box 7851, 50739 Recife - PE - Brazil

ABSTRACT

We discuss the idea of processes in a higher-order, purely functional, modular programming language. Processes are introduced by defining two different and independent language worlds, each of which with a simple semantical basis, one denotational, the other algebraic.

Programming with processes is done by creating static graphs of nondeterministic functions, in a framework separated from the purely functional programming environment defined by a functional language. We consider the characteristics of the approach.

1 Introduction

Several ideas have been recently put forward on how to integrate purely functional languages like Miranda[Tur85] and communication calculi such as CSP[Hoa85].

Most of these have tried to introduce communication (and process) constructs in a functional language[Chr87,Hen84], thus mixing the purely applicative semantics of the functional language considered with the (process) algebra semantics[BeK84] of the process language.

This has the disadvantage of bringing nondeterminism into the functional language, with the likely consequences on its semantics, proof and transformation systems.

Here we show how both functions and processes could be accomodated in a single language, but in two different linguistic levels, as in CONIC[Kra84]. In a purely functional level, Φ, we write purely functional programs. In a processes level, Ξ, we use the function definitions in Φ to create a static network of communicating processes.

By doing so, we (almost completely) isolate the semantics of the two "worlds", making it simpler than the previous approaches to the use of processes in a purely functional setting.

2 A Functional Notation

The functional language discussed herein is a cousin of Miranda[Tur85]. The notation, A[1], is being designed as the functional language of the ETHOS[Tak87] workstation.

A is purely functional, higher-order, polymorphic and non-strict and has abstract and algebraic types. A possible function definition would be

```
DEF _! : Int -> Int;
  0 ! = 1;
  n ! = n * (n-1)!;
END _!;
```

where we define the ubiquitious *factorial* function as a post-fix operator (the _ before the function symbol in the DEF line), of type `Int -> Int` (the type expression after ":" –read *of type*– in the DEF line). The two equations define the possible cases for factorial over the natural numbers. The function is partial over the Integers. When not given, the binding power of operators is equal to that of prefix functions, i.e., maximum. We could have given the binding and associativity of the operator as well, making for the full use of (in, pre, pos, dist)fix operators in expressions.

The principal structured data type is the list. Lists are fully lazy objects, and the domain `List Int`, of *lists* of *integers*, has amongst its values ⊥, `[⊥]`, `[1,⊥,3]`, etc. Functions over lists are defined just the same as over scalar objects, like

```
DEF map : ALL a, b . (a -> b) -> List a -> List b;
  map f []     = [];
  map f (a:x) = f a : map f x;
END map;
```

the higher-order function that applies another (`f`) to all elements of a list. The most general type that can be assigned to such a definition was declared in the DEF line. We assume type declarations are not necessary, given that the language has a Milner-like[Mil78] type system, where inference is possible.

The modules system in A resembles that of Modula-2[Wir82], though there are no Implementation or Local modules, or module Bodies. If we are to be concerned with the purely functional aspects of the language, there exist Definition modules only. Modules will aggregate data type and function definitions, and establish the import/export relationship between component parts of the program. Modules may contain *empty* definitions, i.e., function definitions where only the type declaration has been given, or abstract data types where the implementation is not defined. One example is

[1] Fully described in [Mei88a], in Portuguese and [Mei88b], forthcoming, in English.

```
MOD Sorters;
  EXP ListSorter;
  DEF ListSorter : ALL a . List a -> List a;
  (*  Definition to be given later  *)
  END ListSorter;
END Sorters.
```

The `Sorters` module can be compiled as it stands, making public a definition `ListSorter` of the given type. The `Sorters` module *as is* can be used to define other modules like

```
MOD Use;
  IMP ListSorter FROM Sorters;
  EXP nInOrd;
  DEF nInOrd : ALL a . Int -> List a -> List a;
    nInOrd n = take n . ListSorter;
  END nInOrd;
END Use.
```

with "." as functional composition. `Use` can be compiled (but not run) without the full definition of `ListSorter`. We can see that in its current state, `Sorters` acts like a Modula-2 *definition* module. In fact, that is what it is meant to be. At the programmer's wish, the definitions therein will be completed and/or modified, leading to revisions/recompilations of the user modules.

It is assumed that a *standard* module (`StdMod`) exists, wherefrom definitions like `take`, `map`, `hd`, etc., are imported by default.

3 Levels of Languages

One of the main appeals of purely functional, higher-order, lazy functional languages is their simple semantics. However, when one "extends" such a language with processes, the previously existing equational theory ceases to work for the "extended" language.

This is what happens in the approach of [Chr87], where CSP is combined with a typed λ-notation. There, processes become first class data objects in an *applicative concurrent language*, thus we can have λ-expressions and processes being manipulated by either, and even processes being sent to other processes. The result is that a full denotational semantics is only thought to be possible via metric spaces [Ame86], making it very complex.

Another approach is the one taken in [Hen84], which is to use a meta-notation which includes both a functional language *and* a deterministic version of CSP, together with a set of rules to transform the meta-programs into recursive definitions. This is not the

same as having *real* processes in the language, the same occurring with Stoye's sorting office approach [Sto86] used by Turner in the functional operating system project [Tur87].

In both cases the result is a language which is neither functional, nor process oriented only, risking the disadvantages of both, which is commonly the case with multi-paradigm languages. The only way to maintain as much properties of a functional language as we can is to separate the functional and process worlds in some sense. The method discussed here is related to, but not derived from, the one used in the language CONIC[Kra84]. There we have two separate languages: one for programming and another for what is called configuration, that is, for describing the ways in which processes are created, connected and communicate. It looks likely that a *natural* property of processes is that their networking characteristics are not necessarily related to the *individual* ones, thus giving rise to this separation of concerns in CONIC and in the approach proposed herein.

4 Streams

Streams are lazy lists, which will be the communication channels between processes. In lazy functional languages, little needs to be added to lists for their use as streams. Lists are computed in a demand-driven way, and infinite ones like

```
DEF ones : List Int;
  ones = 1 : ones;
END ones;
```

can be easily defined and used to compute with. A "channel" in a network is just such a list, with the producers working to supply the consumer demand. Both producer and consumer are functions over lists. Indeed, in general, a process is a function over a number of (list) parameters, producing as output a (partial) tuple of (partial) lists, that is to say that, in general the "type" of processes (in Ξ) is given by

$$p \; : \; \underbrace{I_1 \to I_2 \to \ldots \to I_{m-1} \to I_m}_{m \text{ streams}} \mapsto \underbrace{O_1 \times O_2 \times \ldots \times O_{n-1} \times O_n}_{a \text{ tuple of } n \text{ streams}}$$

The difference between streams (in Ξ) and lists (in Φ) is that

- In Φ, computation of values of members of a list is demanded and waited for;
- In Ξ, *only the strict demand waits for the computation.* If the object to be computed is not involved in a strict operation, and is not ready upon demand, the demanding process will produce the value `N`, whose semantics is *no information.*

All domains in Ξ have N as a possible value, but no function can explicitly produce N as a possible result. The only operation that can use this value is equality. So, the behaviour of

```
DEF Ladd : List Int -> List Int -> List Int;
  Ladd = map (uncurry (_+_)) . pair;
END Ladd;
```

which adds two lists, is the same as a process or as a function. Just as lists, the streams are typed and objects of any type can be passed around.

5 Static Networks

Given that processes are functions, the way to separate both and to define networks of processes is to have a configuration language to create and name the processes and establish the connections. The appropriate syntactic unit that is used in **A** is called a *process module*, or MODP. A MODP can import definitions from other modules and define a number of PROcesses and CONnections. Assuming the existence of a MODule m1

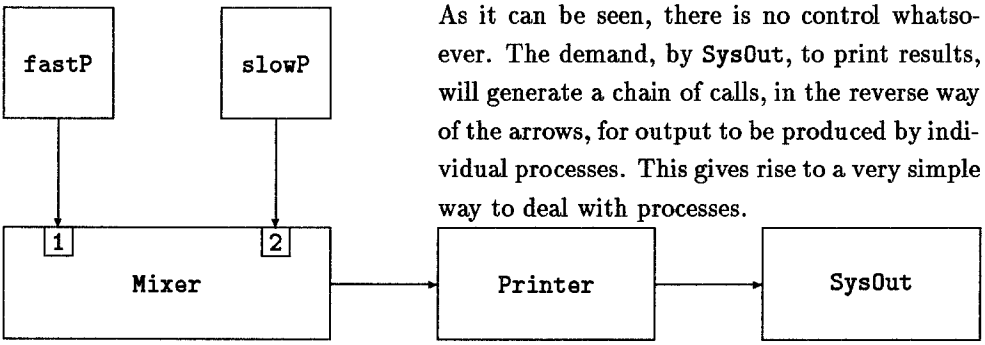
```
MOD m1;
  EXP merge, fast, slow;

  DEF merge;
    merge (a:x) y = a : merge y x;
  END merge;

  DEF fast; (* FAST producer of integers, starting from 1 *) END fast;
  DEF slow; (* SLOW producer of integers, from 1          *) END slow;
END m1.
```

a process module that uses its functions can be defined as

```
MODP m2;
  IMP merge, fast, slow FROM m1;
  PRO (* definition of the processes                                  *)
    Mixer   = merge;  (* Mixer is a 2-input, 1-output process       *)
    fastP   = fast;
    slowP   = slow;
    Printer = WriteLnInt;
    (* WriteLnInt is assumed to come from StdMod, formats Integers
```



As it can be seen, there is no control whatsoever. The demand, by SysOut, to print results, will generate a chain of calls, in the reverse way of the arrows, for output to be produced by individual processes. This gives rise to a very simple way to deal with processes.

Figure 1: Processes and Connections of the Module m2.

```
                                        to be displayed in a window  *)
  END;
  CON (* how the processes are connected                              *)
    fastP   -> Mixer.1;
    slowP   -> Mixer.2;
    Mixer   -> Printer;
    Printer -> SysOut;
    (* SysOut is a Standard Process. It will demand computation and force
       the network to produce it.                                     *)
  END;
END m2.
```

As it can be seen, in the PRO...END block, the identifier to the left of the = sign defines the *name* of the process and the equation to its right the *action* it performs. In the CON...END block, we define the connections, with information flowing in the direction given by the arrow. Processes can appear in either side of the arrow, and parameters are named by indexing the process name with integers, with p.1 being the first of p's parameters (or the first element of the output tuple), from left to right.

The network corresponding to the process module m2 is shown in Fig. 1

6 Lazy Functional Processes and Nondeterminism

In a purely functional model, the behaviour of the process described by m2 would be to merge the lists defined by fast and slow. The result of merge fast slow, for

```
fast = [1,2,3,4,5,6,7,8,9,10,...
```

`slow = [1,2,3,4,5,6,7,8,9,10,...`

would be the list

`[1,1,2,2,3,3,4,4,5,5,6,6,7,7,9,9,10,10,...`

with every i-th element, for odd i, coming from `fast` and the even ones from `slow`. In the process network defined by `m2`, however, the *time* at which elements of either `fast` or `slow` are available is important to define the outcome of the `Mixer` process.

The method of computation in a process network is *lazy evaluation*, on demand by the processes that need to produce output, the only ones in the network that are naturally eager. As usual, the process that asks for input is the *consumer* and the one required to produce it the *producer*.

In Fig. 1, `SysOut` will *drive* the network, demanding computation from `Printer`, which in turn drives `Mixer` and in consequence `fastP` and `slowP`.

As it happens, `Mixer` will consume a token at a time, from one of its two inputs. It will do that demanding the head of one stream to be passed to the output –that is required by its functional definition. If the token is available, it is passed to the output and the computation proceeds. Otherwise, the producer hands out `N`, the *not available* value, and starts doing whatever is necessary to produce a token, in the case its head normal form. `N`, on its turn, is passed to the output and from there to where it is being asked.

As the streams are *typed*, there will be an `N` for every type, as there are as many $\perp$ as domains. Just like $\perp$, a process cannot produce `N` explicitly. `N` is the answer to a request when no data is available from the producer. However, we can compare (`=`, `~=`) values against `N`, in order to decide whether a computation should proceed, for example.

Possible outcomes of the experiment described by module `m2`, as viewed in `SysOut` are —as the `N` token is filtered out by the "printing" process—

`[1,[1],2,3,4,5,[2],6,7,8,9,[3],.....`
`[[1],1,2,[2],[3],3,4,5,[4],6,7,8,9,[5],.....`

where the output corresponding to `slowP` is boxed. `merge` is a nondeterministic function, and the above are only two of the possible outcomes. Also, `merge`'s output stream is permeated by `N`'s, such that if we could see (an instance of) it, the two streams above would look like

`[1,[1],2,[N],3,[N],4,[N],5,[2],6,[N],7,[N],8,[N],9,[3],.....`
`[[1],1,[N],2,[2],N,[3],3,[N],4,[N],5,[4],6,[N],7,[N],8,[N],9,[5],.....`

with the boxed output coming from `slowP`.

Lazy Streams, Strict Messages

Individual processes as in Fig. 1 behave the same as a lazy functional language, driven by the need to produce output. Whether or not a particular process will be asked to produce output depends on the overall behaviour of the process network. If such output is on demand, it will be delivered in *normal form*, which means either the *value* of the item or `N`. If the consumer accepts `N` as input (i.e., it is non-strict) computation can proceed. Otherwise, it will wait until the producer delivers a *value*. This method of computation avoids widespread deadlocking in the network [Mei88c], which would happen if partially evaluated objects could be sent through the streams.

Many definitions can be carried over from Φ to Ξ which keep their functional behaviour. One such example is `Ladd` (cf. Sec.4), whose other definition is

```
DEF Ladd;
  Ladd []     []     = [];
  Ladd (a:x) (b:y) = a+b : Ladd x y;
END Ladd;
```

As the "+" operator is strict on its arguments, the *messages* sent by a process defined by `Ladd` need both `a` and `b` to be defined before the sum is computed and output. Thus, the network defined by[2]

```
MODP m3;
  IMP merge, fast, slow, Ladd FROM m1;
  PRO
    Mixer   = merge;      fastP   = fast;         slowP   = slow;
    Adder   = Ladd;       Printer = WriteLnInt;
  END;
  CON
    fastP   -> Mixer.1;    slowP   -> Mixer.2;
    Mixer   -> Adder.1;    fastP   -> Adder.2;
    Adder   -> Printer;    Printer -> SysOut;
  END;
END m3.
```

will have its throughput bound by the speed of `fastP` and one of its possible outputs would be

```
[2,4,4,7,.....
```

assuming `fastP` would always have a token to output.

[2] Assuming `Ladd` in `m1`.

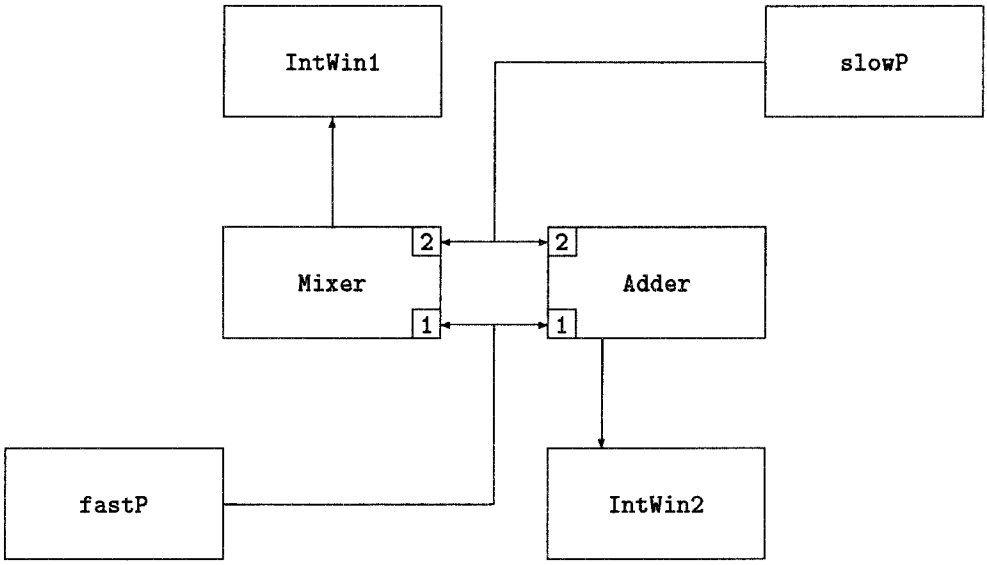


Figure 2: The newtork for module m4.

7 One Further Example

Now we show one short example on synchronization. First define

```
MODP m4;
  IMP merge, slow, fast, Ladd FROM m1;
  PRO
    Mixer   = merge;      fastP   = fast;         slowP   = slow;
    Adder   = Ladd;       slowP   = slow;
  END;
  CON
    fastP   -> Mixer.1;    slowP   -> Mixer.2;
    fastP   -> Adder.1;    slowP   -> Adder.2;
    Mixer   -> IntWin1;    Adder   -> IntWin2;
  END;
END m4.
```

assuming $\texttt{IntWin}_n$ to be an "integer" window. The network is shown in Fig. 2. The messages from slowP and fastP are sent to Mixer and Adder by duplicating the output stream of the first two. As Mixer is always demanding a token and it will not wait for slowP, the output generated by fastP will accumulate as the computation proceeds.

Adder, the other user of the information, needs a token from both input processes, and that will lead to a long queue of **fastP** tokens waiting to be consumed.

If that sort of behaviour is not wanted, we could build a buffer to avoid a process leading another by more than a number of tokens. That can be defined as a function of two input to two output streams, and a state consisting of two integers: the maximum and the current difference between the two streams. The process to the left adds to later and the right one subtracts:

```
DEF buffer : ALL a . Int -> Int -> List a -> List a -> (List a, List a);
  buffer Lim Count (a:x) y = lcons a p, a ~= N & abs Count < Lim;
  buffer Lim Count x (b:y) = rcons b q, b ~= N & abs Count < Lim;
  LOC lcons, rcons, p, q;
    p = buffer Lim (Count+1) x y;
    q = buffer Lim (Count-1) x y;
    lcons a (x,y) = (a:x, y);
    rcons a (x,y) = (x, a:y);
  END lcons, rcons, p, q;
END buffer;
```

where **LOC** introduces local definitions, **abs** is the absolute value and **&** a non-strict "and". **buffer** will allow one of the processes to be at most **Lim** tokens ahead, and it is a fairly generical definition, although not being a real *buffer* as we know, given that the tokens are stored in the network, and not in a space local to **buffer**.

8 Conclusions and Further Work

Static networks of processes were introduced in a purely functional language to cater for distributed and concurrent programming. No restrictions about the actual distribution of the processes need to exist, they could run on a single machine.

The approach keeps the functional language separated from the process language and helps keeping down the complexity of both. Some experimentation has been done to justify this claim, and a complete formal semantics of the language **A** is being written.

In our initial approach, **MODPs** could not export process networks to other modules. That restriction is being raised now, allowing for a rather general use of the concepts of process and network.

9 Acknowledgements

The work reported here was partly financed by CNPq and FINEP. Much of the results have strong connections with the author's discussions with Rafael Lins (UFPE) and Simon Thompson (UniKent), during the Summer 87/88 in Recife.

References

[Ame86] America, P. *et al.*: *A Denotational Semantics of a Parallel Object Oriented Language*. Report CS-R8626, Computer Science/DST, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1986.

[BeK84] Bergstra, J. A. and J. W. Klop: "Process Algebra for Synchronous Communication". *Information and Control* (**60**) 1/3, pp. 109-137, 1984.

[Chr87] Christensen, P.: *Combining CSP with an Applicative Language*. Internal Report, Dept. of Comp. Sci., Tech. Univ. of Denmark, Lingby, DN, 1987.

[Hen84] Henderson, P.: *Communicating Functional Programs*. Tech. Report FPN-8, Comp. Sci. Dept., University of Stirling, UK, 1984.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall Intl., 1985.

[Kra84] Kramer, J. *et al.*: *The Conic Programming Language: Version 2.4*. Res. Report 84/19, Dept. of Computing, Imp. College London, UK, 1984.

[Mei88a] Meira, S. R. de L.: *Introduction to Functional Programming*. Escola de Computação, Campinas-SP, 1988. (In Portuguese)

[Mei88b] Meira, S. R. de L.: *A Modular Functional Language with Processes*. Working Report, Departamento de Informática, UFPE, Recife. To be submitted to SIGPLAN Notices.

[Mei88c] Meira, S. R. de L.: *Functional Processes and Their Semantics*. Working Report, Departamento de Informática, UFPE, Recife.

[Mil78] Milner, R.: "A Theory of Type Polymorphism in Programming". *Jour. of Comp. and Sys. Sci.* (**17**) 3, pp. 348-375, 1978.

[Mil80] Milner, R.: *A Calculus of communicating Systems*. Springer-Verlag, LNCS 92, Berlin, 1980.

[Sto86] Stoye, W. R.: *A New Scheme for Writing Functional Operating Systems.* Cambridge Univ. Computer Lab., Tech. Report 56, Cambridge, 1986.

[Tak87] Takahashi, T. (Ed.): *Anais do IV Encontro de Trabalho do Projeto Ethos.* Petrópolis, Abr. 1987.

[Tur85] Turner, D.: *Miranda: A non-Strict Functional Language with Polymorphic Types.* LNCS 201, Springer-Verlag, Sep. 1985.

[Tur87] Turner, D.: "Functional Programming and Communicating Processes". *Proc. of PARLE Conference*, Eidhoven, 1987.

[Wir82] Wirth, N.: *Programming in Modula-2.* 2nd. Ed., Springer-Verlag, New York, 1982.