Functional Skeletons Generate Process Topologies in Eden

Luis A. Galán Cristóbal Pareja Ricardo Peña

Departamento de Informática y Automática
Universidad Complutense de Madrid
e-mail: {lagalan,cpareja,ricardo}@dia.ucm.es
Fax: (34-1) 394 4607

Abstract. We present a collection of skeletons that are appropriate to instantiate process systems in the functional-concurrent language Eden [BLOM96]. Eden is a functional language providing facilities for the explicit definition and instantiation of processes. Skeletons in this language are just higher order functions having process definitions as parameters. We introduce skeletons for both transformational (i.e. deterministic) and reactive (usually non deterministic) process topologies and illustrate their use by applying them to several examples. Some pointers to the skeletons literature are also given.
Keywords: Functional programming, concurrent programming, parallel programming, skeletons, higher order functions.

1 Introduction

Functional languages are often said to be amenable for implicit parallelism because referential transparency allows evaluating expressions in any order, or even in parallel, without changing the denotational meaning of programs. However, to exploit parallelism is not a trivial task. In order to get enough profit from it, decisions have to be taken to see whether to evaluate an expression in parallel is worthwhile or not. Granularity control seems to be one of the reasons why automatic parallelization of programs does not achieve satisfactory enough results. To solve this problem, several solutions have been proposed, ranging between two extremes:

- One of them is to leave the responsibility of deciding when and how to evaluate in parallel to the programmer. This is usually expressed by means of annotations within the functional program [PvE93].
- Another approach is to leave such decisions to the compiler [Bra93]. The difficulty here is for the compiler to extract enough information from the program in order to decide when parallelization is worthwhile.

Functional skeletons [Col89, DFH+93, Ski93a] constitute a compromise between these two extremes: on one hand, a skeleton is a parameterized algorithmic scheme which can be instantiated with particular functions as actual parameters, thus becoming a program. On the other hand, it expresses a concrete process structure that can be recognized as such by the compiler, so that its parallelism can be appropriately exploited.

Higher order functions are the natural way to express skeletons in a functional language [KPS94, Kes95]. It has also been observed that a restricted set of skeletons —just as it happens with higher order functions— are usually enough to express most computation patterns. So, skeletons are receiving much attention in the last few years as a means of exploiting parallelism in functional programs.

M. Cole [Col89] proposed —in an imperative framework— the skeleton principle, and suggested a first collection of them: a fixed degree divide and conquer skeleton, an iterative combination of skeletons, a cluster and a task queue skeletons, etc. For each one, he described how to map them into a two-dimensional grid of processors, and how to predict the resulting program efficiency.

In J. Darlington and his collaborator's work [DFH+93], the emphasis is put on how to transform programs to fit them into known skeletons. The skeletons are related to concrete computer architectures. So, they are proposing a kind of *program derivation* method in which programs are transformed into more efficient parallel versions, having in mind a concrete machine architecture. In a later work [DGTY95], they show how to combine skeletons in different ways to produce complex parallel topologies. The basic skeletons are: pipeline, divide and conquer, process farm, RaMP (reduce and map over pairs) and DMPA (dynamic message passing architecture).

The approach followed by the BMF[1] community [Bir87] is the extensive use of certain higher order functions, such as `map`, `filter` and associative `fold`, having a high degree of implicit parallelism. These functions are related to each other by a rich set of algebraic laws, so that programs containing them can be transformed in many different ways to incorporate higher degrees of parallelism. A cost model is also proposed [Ski93a, Ski93b] to guide the programmer during the transformation process, so that the chosen transformations go in the direction of increasing parallelism.

Other interesting work has been devoted to actual implementations of skeletons, either by including these implementations as predefined ones in the compiler [Bra93], or by using lower level annotations to spawn processes or/and to place them in particular processors [Kes95].

The approach to parallelism followed by Eden is different. Here, we leave explicit granularity control to the programmer, so that programmers are the ones deciding which functions they wish to execute as parallel processes. Eden is aimed not only to the parallel execution of functional programs but also to the definition of general reactive systems, so non-determinism is also allowed. As a consequence, Eden semantics is beyond that of purely functional languages.

Nevertheless, skeletons may play an important role when programming in Eden. Frequently, process topologies are constituted by many instances of a few process schemes. Also frequently, topologies remain the same while component processes can be freely interchanged with other having a similar communication structure. It is a good habit in functional languages to express this redundancy by making an extensive use of higher order functions. This is the approach followed in this paper: skeletons are higher order functions having as parameters what we call *process abstractions*. A process abstraction is just a parameterized process scheme that can be instantiated in different ways. Skeletons receive

[1] Bird-Meertens Formalism.

process abstractions, together with some size/shape parameters, and instantiate process topologies of varying size.

In what follows, we distinguish two kinds of skeletons: *transformational* ones and *reactive* ones. Transformational skeletons are devoted to deterministic programs, i.e. those (normally terminating) programs calculating a result from a set of arguments. Reactive skeletons abstract reactive systems, i.e. those (normally non-terminating and non-deterministic) programs maintaining some interaction with its environment, and not necessarily yielding a final result. This is a novel aspect of this paper. Reactive process topologies are not usually considered in the skeleton literature.

The plan of the paper is as follows: after this section, we briefly introduce Eden, explaining the syntax and the meaning of the constructions used in this paper. In Section 3, we present a collection of transformational skeletons and illustrate their use with simple examples. In Section 4, reactive skeletons are presented and they are illustrated with typical examples of concurrent systems. The last section draws a brief conclusion.

2 A bird's-eye view of Eden

Eden is a functional language providing constructions for the explicit definition of processes and for establishing communication between them. It is suitable both for parallel and for concurrent programming, including in the later reactive systems and is tailored for execution in distributed memory architectures. A thorough description of the language and of its operational semantics can be found in [BLOM96] and [BLOM95]. In [BLOMP96], a collection of sophisticated examples programmed in Eden is presented.

Eden extends the lazy functional language Haskell [HPJW92] by new syntactic constructs for explicitly defining and instantiating processes. These processes exchange values via communication channels modelled essentially by lazy lists. They are asynchronous, unidirectional, and one-to-one. Communication is implicit, i.e. no special primitives are provided to send or to receive messages. This is achieved by simply connecting an output channel of a process to an input channel of another process. Messages are automatically buffered, so sending a value through a channel never blocks the sending process, while awaiting a value from a channel may block the receiving process. In principle, direct communication is only possible between a parent process and their children processes. But, whenever a parent process simply transmits the values generated by a child to an input channel of another child, it will be "bypassed", and channels will be automatically installed between the corresponding producer and consumer processes. This feature is called *automatic bypassing*.

These constructions are enough to express (deterministic) parallelization of sequential functional programs. In addition, Eden incorporates special concepts for the efficient treatment of general reactive systems:

Dynamic creation of reply channels: This feature simplifies the generation of complex communication topologies and increases the flexibility of the language: at run time, a process may dynamically create new input channels and transmit their "names" to remote processes through previously existent channels. A process receiving these names may either transmit them to another process, or use them to send replies to the originator process.

Predefined nondeterministic processes: In order to model nondeterministic many-to-one and one-to-many communication in process systems, Eden provides predefined nondeterministic `MERGE` and `SPLIT` processes. Eden has a two level semantics: the user-defined process level and the system process level. User-defined processes are seen as deterministic mappings from input channels to output channels. Nondeterminism is handled only at the system level. This distinction, and the maintenance of referential transparency within user-defined processes, supports modularity.

In what follows, we introduce the Eden syntax for the constructions used in this paper, namely the ones for process definition, process instantiation, and nondeterministic processes. Dynamic reply channels are not covered here.

2.1 Process abstractions and process instantiations

Process abstractions specify process behaviour, while process instantiations specify actual creation of processes. The later is done by supplying suitable input values to some process abstraction. A process abstraction maps input values $in_1, \ldots, in_m$ into output values $exp_1, \ldots, exp_n$. It is specified by using the following syntax:

$$\begin{array}{lll} absName\ par_1 \ldots par_k & = \texttt{process} & \texttt{<}in_1, \ldots, in_m\texttt{>} \texttt{ -> } \texttt{<}exp_1, \ldots, exp_n\texttt{>} \\ & \texttt{where} & equation_1 \\ & & \ldots \\ & & equation_r \end{array}$$

The keywords `process` and `where` and the symbol `->` are concrete syntax for a new construction. The symbols `< >` are tuple value constructors, used to group the set of input and that of output channels of the process abstraction. They are mandatory whenever there exists more than one channel. The visibility rules of this construction allow us to use identifiers $in_1, \ldots, in_n$ in the equations following the `where` keyword. When $exp_1, \ldots, exp_n$ are simple enough, the equations and the `where` keyword can be omitted.

A process abstraction is a parameterized process scheme, where $par_1, ..., par_k$ are parameters of any type. The result of the whole construction is a value of the predefined type `Process a b`. The body of a process abstraction consists of the list of expressions $exp_1, ..., exp_n$ defining the outports. The optional `where` part can be used to introduce local variables and some auxiliary functions referenced in the above mentioned expressions.

A process abstraction can be seen as a special kind of λ-abstraction having the following type:

$$absName :: \tau_1 \texttt{ -> } \ldots \texttt{ -> } \tau_k \texttt{ -> Process } \texttt{<} \tau'_1, \ldots, \tau'_m \texttt{>} \texttt{ <} \tilde{\tau}_1, \ldots, \tilde{\tau}_n \texttt{>}$$

where $\tau_1, \ldots, \tau_k$ denote the types of the parameters and $\tau'_1, \ldots, \tau'_m$ and $\tilde{\tau}_1, \ldots, \tilde{\tau}_n$ are respectively the inports and outports types. As it can be seen, symbols `< >` also act as a type constructors. They behave in all aspects as the usual tuple constructor symbols `( )`. This tuple type wrapping up the inports of a process indicates that the corresponding channels must be provided 'in one piece'.

Operations `fst`, `snd`, etc. are overloaded for this new tuple type. The reason to introduce it is to avoid ambiguities. A process abstraction of type `Process (a,b) -> (c,d)` would denote a process with a single input channel and a single output channel, both carrying as messages a tuple of values. For the same reason we use in the examples below a type *channel list*, denoted [<a>], and even tuples of channel lists, denoted <[<a>],[<b>]>. Channel lists overload the list constructors `[]` and `(:)`, and the usual operations for lists. In this paper we provide explicit declaration for all types and do not claim that type inference could be done in an automatic way. This is a subject still under study in Eden.

Process creation takes place when a process abstraction with no unbound parameters is applied to a tuple of inport expressions. This is called *process instantiation* and defines the tuple of outports of the newly created process:

$$\mathit{absName}\ e_1 \ldots e_k\ \texttt{<}\mathit{input\_exp}_1, \ldots, \mathit{input\_exp}_m\texttt{>}$$

Often, process instantiations occur in equations of the form:

$$\texttt{<}\mathit{out}_1, \ldots, \mathit{out}_n\texttt{>} = \mathit{absName}\ e_1 \ldots e_k\ \texttt{<}\mathit{input\_exp}_1, \ldots, \mathit{input\_exp}_m\texttt{>}$$

The left hand side of such an equation is a tuple of terms which must match the output channels of the created process. We do not duplicate running processes because they are dynamic entities with an internal state. If a process instantiation expression is passed as an actual parameter to a function duplicating the corresponding formal one, it is essential to presuppose a lazy semantics. This guarantees that results produced by the process instantiation will be shared.

While in lazy functional programs there is always demand for the evaluation of the topmost expression driving the computation, in Eden there is always demand for the evaluation of the process outports. If a process has several outports, it is divided into several *concurrent threads*, one for each outport definition. If outports are assembled in a tuple and defined by a single expression, there will be a unique thread for their evaluation. This leads to a natural distinction of two levels of concurrency: processes and threads within processes.

2.2 Communication and synchronization

Communication channels transmit completely evaluated values of arbitrary type. In order to model the transmission of a stream of values we introduce a new unary type constructor *Strm*. A channel of type *Strm* τ transmits values of type τ one by one. Streams correspond essentially to lazy lists and, accordingly, stream types are fully compatible with the corresponding list types. Streams are lazy with respect to the `(:)` constructor, but eager with respect to the head of the list, in correspondence with the decision of communicating completely evaluated expressions. Note then, that a channel of type $[\tau]$ is assumed to transfer exactly one list of type τ. This one must be finite because its evaluation has to be completed before the transmission. Instead, a channel of type *Strm* τ transmits —component-wise— a potentially infinite list.

Communication between processes only takes place via communication channels. They are asynchronous and connect normally one sender to one receiver process. Multicasting can be easily achieved by simply connecting the output channel of one process to input channels of several other processes. The semantics of this construction is message duplication. Each channel provides a transmission buffer into which fully evaluated objects are transferred by the sender process. As soon as a value of the corresponding type has been transmitted in full, the channel is closed and the communication port is abolished. Stream typed channels are closed when the `[]` constructor is transmitted. Information transfer via communication channels is done automatically. No explicit send or receive commands are needed by the programmer who simply specifies *what* has to be output.

If a concurrent thread within a process needs some value from an inport whose channel buffer is empty, this thread will be suspended until the corresponding sender process writes some value into the channel buffer. However, the evaluation of outports is independent of the consumption of the produced values.

The input and output ports specified in the process abstraction are established on process creation, but additional ones can be introduced later during the process "life" when either child processes are instantiated or dynamic reply channels are used. How is this reflected in the process interface? To answer this question we distinguish two different points of view: from the denotational one, where processes are seen as functions mapping input parameters into output ones, the process interface does never change. This coincides with the (process abstraction) users view. But from the operational point of view process interfaces in fact change because, when children processes are created, new connections with parent processes are established. However, this interface can be seen as an "internal" one, only known to the process abstraction implementer . Therefore, Eden supports modular design by making explicit the external interface in the process header, but hiding the internal one inside the implementation.

2.3 Nondeterministic Processes

Nondeterminism is introduced in Eden by special predefined process abstractions. They encapsulate nondeterminism and thus do not destroy referential transparency within user-defined processes. Furthermore, the introduction of separate processes to model nondeterministic behaviour improves modularity and simplifies the incremental analysis of process systems.

The `MERGE` process abstraction creates a nondeterministic process *fairly* merging a list of stream channels of type `Strm a` and producing a single channel of the same type. We also introduce an analogous process abstraction `SPLIT` nondeterministically distributing the contents of its single input stream into a set of output streams:

```
MERGE :: Process [<Strm a>] (Strm a)
SPLIT :: Process (Strm a) [<Strm a>]
```

A typical application of the `MERGE` process is the modelling of many-to-one communications. For instance, in a client-server system several clients may send requests to a single server. This situation can be modelled by using a `MERGE` process merging the requests in a fair way into a single input channel going to the server, so without fixing a deterministic order in which process requests will arrive. Our fair merge generalizes easily to a *priorized* merge which would have a list with the priorities associated to the input channels as parameter.

3 Transformational skeleton

In this section, we make a review of most of the skeletons proposed in the literature and express them as Eden process networks. We apply them to some examples and comment on how Eden features may extract the parallelism implicit in these topologies. Frequently, to appropriately exploit this parallelism several skeletons must be combined, or/and some pre/post-processing is needed in order to adapt the input data to the skeleton or to extract the final results.

3.1 Pipelines

A pipeline is just a list of processes composed in such a way that the output channel of every process is connected to the input channel of the next one in the pipe. Parallelism is achieved if these channels transmit *streams* of data. Pipes in Eden can be easily described by a `foldr` application to a binary composition of processes:

```
(>>):: Process a b -> Process b c -> Process a c
p >> q = process left -> q (p left)
pipe:: [Process (Strm a) (Strm a)] -> Process (Strm a) (Strm a)
pipe = foldr (>>) pipeId
       where pipeId = process left -> left
```

Sometimes, it is useful to provide for a bi-directional data flow. This can be done with a simple modification of the above definition:

```
(<<>>):: Process <a,b> <a,b> -> Process <a,b> <a,b> -> Process <a,b> <a,b>
p1 <<>> p2 = process <iright,ileft> -> <oright,oleft>
             where <midr,oleft>  = p1 <iright,midl>
                   <oright,midl> = p2 <midr,ileft>
pipeB:: [ Process <Strm a,Strm b> <Strm a,Strm b> ]
         -> Process <Strm a,Strm b> <Strm a,Strm b>
pipeB = foldr (<<>>) pipeBId
        where pipeBId = process <iright,ileft> -> <iright,ileft>
```

The binary operators themselves can be useful to define pipes in which processes have different types. For instance, if we have `p1::Process (Strm a) (Strm b)`, `p2::Process (Strm b) (Strm c)` and `p3::Process (Strm c) (Strm d)`, we can define:

```
mypipe :: Process (Strm a) (Strm d)
mypipe = p1 >> p2 >> p3
```

Example 1. We present here an application of the one-directional pipe to the generation of Hamming numbers. A previous Eden solution to this problem can be found in [BLOM96]. A list of different prime numbers, is given and the pipeline must return the list of all natural numbers having (some of) these primes as the only prime factors. The pipe has as many processes as the initial list, each one devoted to a prime factor:

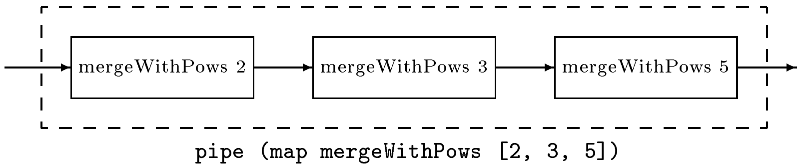

Fig. 1. One-directional pipeline for the Hamming numbers problem

```
hamming :: [Int] -> [Int]
hamming primes  =  pipe (map mergeWithPows primes) [1]
mergeWithPows p  = process left -> right
                   where right = 1 : merge (tail left) (map (*p) right)
```

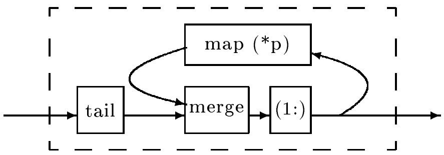

Fig. 2. A pipeline component

The `merge` function is just the standard one merging two ordered lists. The resulting pipeline and a detail of the component processes can be respectively seen in Figure 1 and in Figure 2.

In some circumstances, it is appropriate to have pipelines in which all the component processes are identical. Its definition is just a particular case of the previously defined pipes:

```
repeatN :: Int -> a -> [a]
repeatN n x = take n (repeat x)
pipeN :: Int -> Process (Strm a) (Strm a) -> Process (Strm a) (Strm a)
pipeN n p = pipe (repeatN n p)
pipeBN :: Int -> Process <Strm a,Strm b> <Strm a,Strm b>
              -> Process <Strm a,Strm b> <Strm a,Strm b>
pipeBN n p = pipeB (repeatN n p)
```

Finally, it could also be useful to have pipes in which the number of components is dynamically decided depending on the input data. We propose the following generic definition for them:

```
dynPipe :: ([a] -> Bool) -> Process (Strm a) (Strm a)
                         -> Process (Strm a) (Strm a)
dynPipe q p = process xs -> if q xs then xs else dynPipe q p (p xs)
```

3.2 Grids

A grid is a two-dimensional structure in which component processes are connected to four neighbours. We convene that input channels come in from left and top sides of each component, and output channels go out from right and bottom sides of it. A natural way of defining such a topology is by using arrays. The following function receives a list of m process abstractions lists, each one of length n, and instantiate them in a grid fashion:

```
gridMN :: [[ Process <a,b> <a,b> ]] -> Process <[<a>],[<b>]> <[<a>],[<b>]>
gridMN pss = process <lefts,tops> -> <rights,bots> where
  grid =  array ((0,0),(m-1,n-1))
    ([((0,0),pss!!0!!0 <lefts!!0,tops!!0>)] ++
     [((0,j),pss!!0!!j <fst (grid!(0,j-1)),tops!!j>) | j<-[1..n-1]] ++
     [((i,0),pss!!i!!0 <lefts!!(i),snd (grid!(i-1,0))>) | i<-[1..m-1]] ++
     [((i,j),pss!!i!!j <fst (grid!(i,j-1)),snd (grid!(i-1,j))>) |
                  i<-[1..m-1], j<-[1..n-1]])
  m = length pss
  n = length (head pss)
  rights  = [fst (grid!(i,n-1)) | i <- [0..m-1]]
  bots    = [snd (grid!(m-1,i)) | i <- [0..n-1]]
```

Here, the auxiliary array `grid :: Array (Int,Int) (a,b)` is used to provide channel connections between all the processes.

An alternative way of defining the same topology is by making an extensive use of higher order functions as we did with pipelines. Now, we define binary operators to add a component to a row and to add a row to a grid:

```
(>:) :: Process <a,b> <a,b> -> Process <a,[<b>]> <a,[<b>]>
        -> Process <a,[<b>]> <a,[<b>]>
p >: ps = process <left,tops> -> <right,bots>
          where (x:xs)     = tops
                <mid,y>    = p  <left,x>
                <right,ys> = ps <mid,xs>
                bots       = y:ys
(>>:) :: Process <a,[<b>]> <a,[<b>]> -> Process <[<a>],[<b>]>
         -> <[<a>],[<b>]>
ps >>: pss = process <lefts,tops> -> <rights,bots>
             where (l:ls)    = lefts
                   <r,mids>  = ps  <l,tops>
                   <rs,bots> = pss <ls,mids>
                   rights    = r:rs
```

We need identity processes to start with when constructing each row or the complete grid:

```
idRow  :: Process <a,[<b>] >  <a,[<b>] >
idRow  =  process <left,[] > -> <left,[] >
idGrid :: Process < [<a>],[<b>] >  < [<a>],[<b>] >
idGrid =  process <[],xs> -> <[],xs>
```

Then, the grid definition is just a repeated application of the `foldr` higher order function:

```
gridMN :: [[Process <a,b> <a,b> ]] -> Process < [<a>],[<b>] > < [<a>],[<b>] >
gridMN pss = foldr (>>:) idGrid rows
             where rows = [foldr (>:) idRow ps | ps <- pss]
```

Example 2. A well known example of grid topology application is matrix multiplication. To multiply a $m \times n$ matrix by an $n \times l$ one, we will instantiate a $m \times n$ grid in which each processing element will contain the corresponding value of the first matrix. Consecutive length-n columns of the second matrix will be feed through the n top channels of the grid. The resulting m rows of the product matrix will go out from the m right channels of the grid.

```
multElem :: Num a => a -> Process <Strm a,Strm a>  <Strm a,Strm a>
multElem x = process <ileft, itop> -> <zipWith f itop ileft, itop>
             where f y z = x*y + z
matMult :: Num a => [[a]] -> [[a]] -> [[a]]
matMult ass bss = fst outProcess
                  where outProcess = gridMN pss <zeroesRows,bss>
                        zeroesRows = repeatN (length ass) (repeat 0)
                        pss = map (map multElem) ass
```

In Figure 3 we show the following multiplication example, in which the first matrix is 2×2 and the second one is $2 \times z$, being z any natural:

$$\begin{pmatrix} 3 & 2 \\ 5 & 7 \end{pmatrix} \times \begin{pmatrix} 8 & 2 & 5 & \ldots \\ 7 & 3 & 4 & \ldots \end{pmatrix}$$

It is worthwhile to remark that, thanks to the default Eden feature of providing

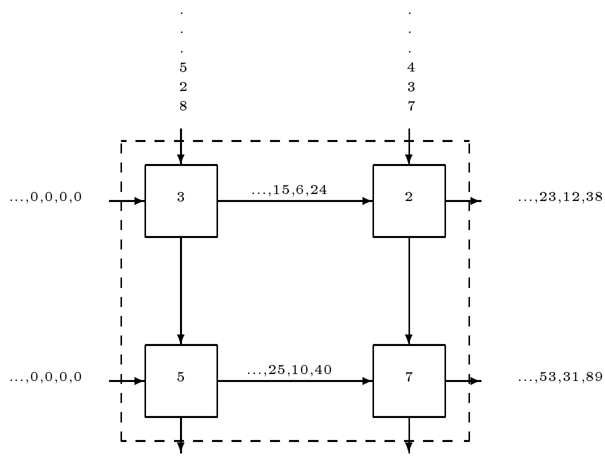

Fig. 3. Matrix multiplication as an application of the grid topology

a concurrent thread to compute each output channel of a process, the vertical data flow, where each element copies the value received from the top channel to the bottom channel, proceeds in parallel with the horizontal computation of scalar products.

3.3 Farms

This skeleton exploits parallelism by applying the same process abstraction to a set of independent input channels. Conceptually, it corresponds to the higher order function map:

```
farm :: Process a b -> Process [<a>] [<b>]
farm p = process xs -> f xs
         where f []     = []
               f (x:xx) = p x : f xx
```

The purpose of function f here is to repeatedly instantiate copies of process p. It could also have been done by a recursive invocation to function farm but, in this case, we would have instantiated a number of not very useful processes. We present an application of farm in the next section, where farm is used in the context of another well known skeleton.

3.4 Divide and conquer

This is a parameterized skeleton instantiating a dynamic tree of processes, each one receiving an input channel from his parent process and returning an output channel to it. The farm skeleton is used to generate the set of children processes of a given one. Its description in Eden is as follows:

```
divCon :: (a -> Bool) -> (a -> b) -> (a -> [<a>]) -> (a -> [<b>] -> b)
          -> Process a b
divCon trivial solve split combine = process x -> y where
 y = if trivial x then solve x
     else combine x (farm (divCon trivial solve split combine) (split x))
```

There are many applications of this skeleton. In particular, R. Bird's homomorphisms over lists $B(\oplus, f) = \oplus/ \circ f*$ can be expressed as a divide and conquer scheme[2].

Example 3. We present here an Eden version of the well known *quicksort* algorithm:

```
quickSort :: Ord a => Process (Strm a)  (Strm a)
quickSort = divCon trivial id splitHd combine
            where trivial []     = True
                  trivial [_]    = True
                  trivial _      = False
                  splitHd (x:l) = splitAux x l
                  splitAux x [] = [[],[]]
                  splitAux x (y:l) | y<=x = [y:u,v]
                                   | y>x  = [u,y:v]
                                   where [u,v] = splitAux x l
                  combine (x:_) [l1, l2] = l1 ++ [x] ++ l2
```

It is interesting to remark that Eden's eager creation of processes and eager evaluation of output channels increase the parallelism of this solution: the downwards data flow proceeds as soon as each node process receives the first element of its input list. So, splitting the original list is done in parallel by an increasing number of processes. The upwards "pipeline" is also done in parallel due to eager evaluation of output channels. In particular, processes in right subtrees start producing their output lists without waiting for their parent processes to ask for it. This may have the cost of having big buffers in the channels, but from the time efficiency point of view, the solution is optimal.

---

[2] It corresponds to the particular case in which the combine function ignores x: $a \oplus b =$ combine _ [a, b]

4 Reactive skeletons

In this section, we show how to instantiate in Eden some process topologies which are common in reactive systems. Usually, reactive systems constitute "closed" process graphs in the sense that each output channel of every process is connected to some input channel of another process. From the functional point of view, a skeleton instantiating such a process net would produce no value as a result. To cope with these no value returning functions, we introduce a new predefined type `System` and a new kind of expression:

`(system where` *declarations*`) :: System`

The type `System` is similar to the type `()` except for the fact that values of the new type represent running concurrent systems. The `system` expression evaluates, in an eager way, all process instantiations appearing in the declaration part and returns a running system.

4.1 Client-server

In this topology, several *client* processes ask a single *server* process for some service. There is an output channel for each client to send requests, and an input channel for each one to receive acknowledgements. We include a `MERGE` process to model the non determinism in the arrival of requests to the server, and a *distributor* process in order to split the output from the server into a list of channels, each one dedicated to one client. The resulting topology is shown in figure 4, where a client-server skeleton is instatiated with a semaphore. The distributor process is deterministic and can be made generic provided a "distribution" function `f::a -> Int` is given. The relevant declarations follow:

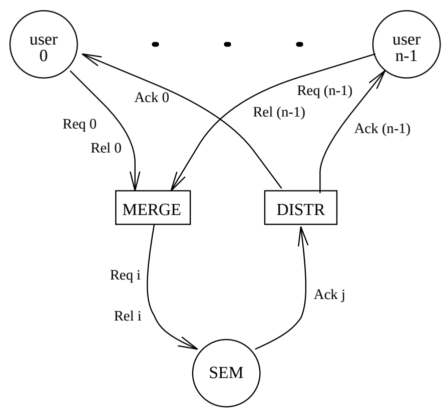

Fig. 4. A client-server skeleton instantiation

```
client :: Process (Strm b)  (Strm a)
server :: Process (Strm a)  (Strm b)
distr  :: (b -> Int) -> Process (Strm b) [<Strm b>]
distr f = process input -> cycle input
          where  cycle (m:ms) = insert m (f m) (cycle ms)
                 insert x 0 (ys:yss) = (x:ys):yss
                 insert x (k+1) (ys:yss) = ys:(insert x k yss)
clientServer :: [client] -> server -> (b -> Int) -> System
```

```
clientServer cs serv f = system where
                            outClients = zipWith apply cs outDist
                            inServer   = MERGE outClients
                            outServer  = serv inServer
                            outDist    = distr f outServer
                            apply p x  = p x
```

Example 4. We show a simple FIFO semaphore to achieve mutual exclusion between the critical regions of a set of users. The protocol between a user and the semaphore is as follows: when the user wishes to enter the critical region, first it sends a `Req i` message to the semaphore, `i` being the user identity. Then it waits for an `Ack i` message. When this one arrives, it enters the critical region. Finally, it sends a `Rel i` message to the semaphore to indicate the end of the critical actions. For the semaphore implementation we assume the existence of a type `Queue a` implementing polymorphic FIFO queues with the typical operations: `emptyQ, empty, add, first` and `del`.

```
data Signal = Req Int | Ack Int | Rel Int
distFun (Req i) = i
distFun (Ack i) = i
distFun (Rel i) = i
user :: Int -> Process (Strm Signal)  (Strm Signal)
user i    = process input -> thinking input
            where thinking xs         = Req i : hungry xs
                  hungry (Ack i : xs) = eating xs
                  eating xs           = Rel i : thinking xs
semaphore = process input -> cycle True emptyQ input where
 cycle True  q (Req i : ss) = Ack i : cycle False q ss
 cycle False q (Req i : ss) = cycle False (add q i) ss
 cycle False q (Rel i : ss) = if empty q then cycle True q ss
                             else Ack (first q) : cycle False (del q) ss
semMutex  = clientServer (map user [0..n-1]) semaphore distFun
```

4.2 Ring

A ring topology having a single channel connecting every two neighbours in the ring can be easily defined in terms of the `pipe` function presented in Section 3.1:

```
ring :: [ Process a a ] -> System
ring (p:ps) = system where link = p next
                           next = pipe ps link
```

Similarly, we can define bi-directional rings by using the `pipeB` function defined in the same section:

```
ringB :: [ Process <a,b> <a,b> ] -> System
ringB (p:ps) = system where <in2,out2> = p <in1,out1>
                            <in1,out1> = pipeB ps <in2,out2>
```

Example 5. A typical example of two-ways ring is a token-passing ring, i.e. a mutual exclusion system where a ring-shaped set of processes share a `token`. A process must hold the token in order to enter the critical region. If it does

not hold it, it asks to its right neighbour for
it. Requests for the token travel counterclockwise through the ring, while token acknowledgements travel clockwise, as it is shown in Figure 5, a ring skeleton instatiation implements mutual exclusion. The token is released whenever there is a request for it and it is not currently being used by the (only) process holding it. We present only the external view of this ring:

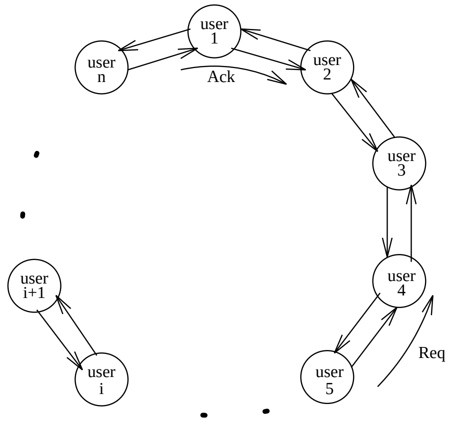

Fig. 5. A ring skeleton instantiation

```
data Req     = Req
data Ack     = Ack
type RingProc = Process <Strm Ack, Strm Req> <Strm Ack, Strm Req>
mutexRing    = ringB ps
```

where `ps :: [RingProc]` is the list of the ring component processes.

4.3 Graph

A connected graph of processes is the most general topology we can define. We present in this section a skeleton receiving as parameters the graph description and a list of *node* processes. Then, it instantiates a node process for each vertex of the graph, and connects the nodes according to the graph edges. We are assuming that the graph is an undirected one[3]. For each edge joining vertices A and B, it provides a channel going from $node_A$ to $node_B$, and another one in the opposite direction. The graph description is just an array of adjacency lists, and nodes in the node list are given in correspondence with the array indices, i.e. the first node in the list is the node we wish to instantiate for vertex zero, and so on. Undirected edges are represented in the graph description by two directed edges (i.e. if vertex u is adjacent to vertex v, then u is in the adjacency list of v and v is in the adjacency list of u). We adopt the convention that, if v is the vertex in position p_v in u's adjacency list, and u is vertex in position p_u in v's adjacency list, then the p_v-th output channel of u is connected to the p_u-th input channel of v:

```
type Graph  = Array Int [Int]
type Node   = Process [<a>]  [<a>]
net        :: Graph -> [Node] -> System
```

[3] It will be straightforward to provide a skeleton for a directed one.

```
net g nodes = system where
   outchans = [node (inchans i)|(node,i) <- zip nodes [0..]]
   inchans i = [outchans!!k!!p|k <- g!i,(j,p) <- zip (g!k) [0..],j==i]
```

Example 6. We show an application of the graph topology to the generalized dining philosopher problem presented by Chandy and Misra in [CM84]. A detailed description of the Eden solution to this problem can be found in [BLOMP96]. There, we only instantiated a ring of five nodes. Every node contained four processes: a philosopher, a secretary, a `MERGE` and a distributor process. We are assuming here that the nodes corresponding to each graph vertex have already been defined and our problem now is to instantiate the complete graph of nodes. Let us call `node0`, `node1`, `node2`, `node3`, `node4` and `node5` to the `Node` processes corresponding to the graph vertices of Figure 6. Then, the instantiation of the process graph would be as follows:

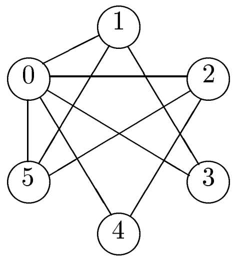

Fig. 6. A concrete graph topology

```
philGraph = array (0,5) [(0,[1,2,3,4,5]),(1,[0,3,5]),(2,[0,4,5]),
                         (3,[0,1]),(4,[0,2]),(5,[0,1,2])]
philNet   = net philGraph [node0, node1, node2, node3, node4, node5]
```

5 Conclusions

We have presented two skeleton families that are appropriate to instantiate both transformational parallel algorithms and reactive concurrent systems. They have been expressed as higher order functions written in the functional-concurrent language Eden.

Having a skeletons catalogue very much simplifies concurrent programming in Eden because the programmer can concentrate on writing process abstractions, and this essentially amounts to defining functions. So, writing concurrent programs in this framework presents a difficulty rather similar to that of writing ordinary functional programs.

Eden special characteristics, in particular eager evaluation of output channels and the possibility of providing a concurrent thread for each output channel, seems to extract a high degree of parallelism from the presented skeletons.

Further work is currently devoted to the denotational semantics of Eden (at present we only have defined an operational one), to the refinement of the type system and to develop formal techniques to analyze safety and liveness properties in Eden programs.

Acknoledgements

We want to thank Rita Loogen, Herbert Kuchen, Yolanda Ortega, Manuel Núñez and Pedro Palao their comments on a previous version of this paper.

References

[Bir87] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer Verlag. NATO ASI Series, vol. F36, 1987.

[BLOM95] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Eden: Language definition and operational semantics. Technical Report, Philipps-Universität Marburg, Germany, 1995.

[BLOM96] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In *Glasgow Workshop on Functional Programming 1995*. Springer-Verlag, 1996.

[BLOMP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden, the paradise of functional-concurrent programming. Technical Report DIA-UCM-20/96, Universidad Complutense de Madrid, Spain. Submitted to EUROPAR'96, 1996.

[Bra93] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages*, pages 23–33, September 1993.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, 1984.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research monographs in parallel and distributed computing. Pitman, 1989.

[DFH+93] J. Darlington, A. J. Field, P. J. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming using Skeleton Functions. In *PARLE'93: Parallel Architectures and Languages Europe*. Springer-Verlag, Jun. 1993.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel Skeletons For Structured Composition. In *Proceedings of the 15th ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 1995.

[HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell. version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Kes95] M. Kesseler. Constructing skeletons in Clean: The bare bones. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 182–192, April 1995.

[KPS94] H. Kuchen, R. Plasmeijer, and H. Stoltze. Distributed Implementation of a Data Parallel Functional Language. In *Parallel Architectures & Languages Europe, PARLE'94. LNCS 817*, pages 464–477. Springer-Verlag, 1994.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Ski93a] D.B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[Ski93b] D.B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 20, No.3:205–213, June 1993.