Eden — The Paradise of Functional Concurrent Programming*

S. Breitinger[1], R. Loogen[1], Y. Ortega-Mallén[2], R. Peña-Marí[2]

[1] Philipps-Universität Marburg, Fachbereich Mathematik, Fachgebiet Informatik, Hans Meerwein Straße, Lahnberge, D-35032 Marburg, Germany, {breiting,loogen}@informatik.uni-marburg.de

[2] Universidad Complutense de Madrid, Sec. Dept. de Informática y Automática, Facultad de C.C. Matemáticas, E-28040 Madrid, Spain, {yolanda,ricardo}@dia.ucm.es

The functional concurrent language *Eden* [1] is an extension of the lazy functional language Haskell [4] by constructs for the explicit specification of dynamic process systems. It employs stream-based communication and is tailored for distributed memory systems. Eden supports and facilitates the task of parallel and concurrent programming.

Eden incorporates special concepts for the efficient treatment of general reactive systems, i.e. systems which maintain some interaction with the environment and which may be time-dependent. The *dynamic creation of reply channels* simplifies the generation of complex communication topologies and increases the flexibility of the language. *Predefined nondeterministic processes* MERGE and SPLIT are used to model many-to-one and one-to-many communication in process systems.

Eden incorporates a two level structure: the level of user-defined *processes* and the level of *process systems*. User-defined processes can be seen as deterministic mappings from input channels to output channels. Nondeterminism is only handled at the system level which consists of all (predefined and user-defined) processes interacting via communication on channels.

1 Eden in a nutshell

Haskell forms the *computation language* of Eden. This is extended by a *coordination model* that introduces processes in a functional style, embodying constructs which allow for the definition and creation of processes, communication and synchronization, and the specification of interconnections between processes.

Eden distinguishes between *process abstractions*, which specify process behaviour in a purely functional way, and *process instantiations* in which process abstractions are supplied with input values in order to create new processes.

A process abstraction defines a general parameterized process scheme. It specifies a process which maps (streams of) input values $\mathtt{in}_1, \ldots, \mathtt{in}_m$ to (streams of) output values $\mathtt{out}_1, \ldots, \mathtt{out}_n$:

* Supported by the DAAD (Deutscher Akademischer Austauschdienst) and the Spanish Ministry of Education and Science in the context of the German-Spanish Acción Integrada n.142B.

process abstraction:

$$\begin{array}{lll} p :: \tau_1 \to \ldots \to \tau_k & \to & \mathit{Process}\ (\tau'_1, \ldots, \tau'_m)\ (\tau''_1, \ldots, \tau''_n) \\ p \quad par_1 \ \ldots \quad par_k & = & \text{process}\ (in_1, \ldots, in_m) \to (out_1, \ldots, out_n) \\ & & \text{where} \quad equation_1 \quad \ldots \quad equation_r \end{array}$$

Process creation takes place when a process abstraction without unspecified parameters[3], e.g. $(p\ e_1 \ldots e_k)$, is applied to a tuple of inport expressions. This is called *process instantiation* and defines the tuple of outports of the newly created process:

process instantiation: $(p\ e_1 \ldots e_k) \sharp (in\_exp_1, \ldots, in\_exp_m)$

Often, process instantiations occur in equations of the form

$$(out_1, \ldots, out_n) = (p\ e_1 \ldots e_k) \sharp (in\_exp_1, \ldots, in\_exp_m)$$

The left hand side of such an equation is the tuple of output channels of the created process.

It is important that a process abstraction must be closed, i.e. the expressions in the body may depend only on parameters, input values, local auxiliary definitions or functions contained in the standard Haskell prelude. This guarantees that a process is an independent unit of computation which communicates only via its ports and can be executed without any implicit need to access global information. This property is essential as Eden is a language for distributed memory systems.

We do not allow the duplication of running processes because they are dynamic entities with an internal state. Therefore it is essential to presuppose a lazy semantics, which guarantees that process instantiations passed as argument to functions, to other processes, or to abstractions will be shared.

Communication channels transmit completely evaluated values of arbitrary type. In order to model the transmission of a stream of values we introduce a new algebraic data type *Strm* a with data constructor $< . >:: [a] \to \mathit{Strm}\ a$. A channel of type *Strm* τ transmits values of type τ one by one. Streams correspond to lazy lists, but note that a channel of type *list* τ is assumed to transfer exactly one list of type τ, which is finite because its evaluation must be completed before the transmission, while a channel of type *Strm* τ transmits a potentially infinite list component-wise.

Example. A sorting network which transforms an input stream into a sorted output stream by subsequently merging sorted sublists with increasing length is specified by the following process abstraction:

```
mergesort :: Process (Strm a) (Strm a)
mergesort  = process  <s> -> <sort s>
             where
```

[3] Actual parameter expressions are copied into the body of process abstractions before the process is created.

```
sort []  =  []
sort [x] =  [x]
sort  xs =  smerge (mergesort # <l1>) (mergesort # <l2>)
            where (l1,l2) = unshuffle xs
smerge [] l = l
smerge l [] = l
smerge (x:l) (y:t) = if x<=y then x:smerge l (y:t)
                             else y:smerge (x:l) t
unshuffle []  = ([],[])
unshuffle [x] = ([x],[])
unshuffle (x:y:t) = (x:t1,y:t2)
                    where (t1,t2) = unshuffle t
```

Streams with at least two elements are split into two substreams for which recursive instantiations of the mergesort process are generated. As a result a tree of mergesort processes is created.

2 Programming in Eden

Due to space limitations we present only the specification of a parallel matrix multiplication algorithm. Matrix multiplication is an important operation in many scientific and engineering problems. It has a good potential for parallelization and has become a standard example for parallel programming. We consider a parallel algorithm which uses a torus topology. The matrices are partitioned into submatrices which are distributed on the torus nodes. For simplicity we assume the partition size to be 1 and the matrix dimension to be $n \times n$.

Each element of the torus first gets the corresponding elements of the input matrices. The torus node with the position (i, j) then has to compute the element (i, j) of the result matrix, i.e. the scalar product of the ith row of the first input matrix and the jth column of the second input matrix. In order to place the elements to be multiplied on the same node, the rows of the first matrix and the columns of the second matrix are rotated by $(i-1)$ and $(j-1)$ positions respectively before the proper computation starts. Then all torus nodes perform an iteration of $n-1$ steps in which they multiply corresponding elements of the matrices which they read subsequently from their input channels.

We presuppose the definition of a process abstraction torus which creates a torus topology according to the dimension of the given matrix (see [2] or [3]):

```
type Matrix a  = [[a]]
type NodeProc a b c d = (Int -> Int -> a ->
                         Process (Strm b, Strm c) (d, Strm b, Strm c)
torus :: Matrix a -> NodeProc a b c d -> Process () (Matrix d)
```

The node processes are connected row- and column-wise by stream channels from right to left and bottom to top. Their behaviour is specified by an abstraction which is passed as a parameter to the torus process abstraction.

```
matmult :: Int -> (Matrix Int) -> (Matrix Int) -> (Matrix Int)
```

```
matmult n ass bss
  = torus (map uncurry.zip (zip ass bss)) (scalarprod n)
scalarprod :: Int -> Int -> Int -> (Int,Int)
              -> Process (Strm Int, Strm Int) (Int, Strm Int, Strm Int)
scalarprod n i j (a,b) -- process abstraction for each torus node
  = process  (inrow, incol) -> (result, outrow, outcol)
    where
     outrow = a:inrow  -- rotate row of first matrix
     outcol = b:incol  -- rotate col of second matrix
     result = iterate n arow bcol (a0 * b0)
     iterate 0        row     col  val = val
     iterate (n+1) (r:row) (c:col) val = iterate n row col (val+r*c)
     arow = (drop (i-1) inrow)
     bcol = (drop (j-1) incol)
     a0 = (a:inrow)!!(i-1)
     b0 = (b:incol)!!(j-1)
```

3 Where to find more about Eden

The interested reader can find a more detailed discussion of Eden features in [2]. There, more programming examples are provided. A collection of skeletons for the instantiation of various process topologies can be found in [3].

The semantics of Eden is an extension of the standard operational semantics of Haskell. It reflects the two-layered nature of Eden: systems and processes. On the upper level global effects on process systems are described. The lower level handles local effects within processes. The interface between the two levels consists of so-called 'actions' by which the need for global events is communicated to the upper level.

In a prototype implementation of Eden the low-level primitives of Concurrent Haskell have been used to model communication and synchronization between Eden processes explicitly. In the future there will be a direct implementation of Eden on an IBM SP-2 using the MPI (Message Passing Interface) standard.

References

1. S. Breitinger, R. Loogen, Y. Ortega-Mallén: Towards a Declarative Language for Concurrent and Parallel Programming, Glasgow Workshop on Functional Programming, Springer Workshops in Computing 1995.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, R. Peña-Marí: Eden — The Paradise of Functional Concurrent Programming, TR DIA-UCM-20/96, Universidad Complutense de Madrid 1996, also available as `http://www.mathematik.uni-marburg.de/~loogen/paper/paradise.ps`.
3. L. A. Galán, C. Pareja, R. Peña: Functional Skeletons Generate Process Topologies in Eden, APPIA-GULP-PRODE'96 Joint Conference on Declarative Programming, San-Sebastian, Spain, July 1996.
4. P. Hudak, Ph. Wadler (eds.): Report on the Programming Language Haskell: a non-strict, purely functional language, SIGPLAN Notices, 27(5):1–162,1992.