Implementing Eden – or: Dreams Become Reality*

Ulrike Klusik[1], Yolanda Ortega[2], and Ricardo Peña[2]

[1] Philipps-Universität Marburg, D-35032 Marburg, Germany
klusik@mathematik.uni-marburg.de
[2] Universidad Complutense de Madrid, E-28040 Madrid, Spain
{yolanda,ricardo}@sip.ucm.es

Abstract. The parallel functional programming language Eden was specially designed to be implemented in a distributed setting. In a previous paper [3] we presented an operational specification of DREAM, the distributed abstract machine for Eden. In this paper we go a step further and present the imperative code generated for Eden expressions and how this code interact with the distributed RunTime System (RTS) for Eden. This translation is done in two steps: first Eden is translated into PEARL (Parallel Eden Abstract Reduction Language), the parallel functional language of DREAM, and then PEARL expressions are translated into imperative code.

1 Introduction

The parallel functional language Eden [6, 5], with its explicit notion of process, represents an alternative to annotated parallel functional languages [14, 16] and to the automatic parallelization of functional programs [9]. The advantages of having explicit processes have been extensively discussed elsewhere [4]. Moreover, Eden processes are autonomous closed entities producing results to be consumed by other processes without any notion of central manager or coordinator, or common shared memory. Thus, the language facilitates the programming of distributed systems. Correspondingly, Eden is specially suited for a distributed implementation.

Actually, Eden extends the lazy functional language Haskell [10] with a coordination language to explicitely express parallelism. There exist already very good and efficient compilers for Haskell. Thus, we decided to use the Glasgow Haskell compiler (GHC) [12] as a basis for the implementation of the computational language, and to concentrate our efforts in the distributed implementation of the coordination features.

Our first step towards an efficient distributed implementation for Eden has been the definition of DREAM (DistRibuted Eden Abstract Machine), consisting of a set of multithreaded abstract machines, each of which executes an Eden

* Work partially supported by German-Spanish Accion Integrada HA1997–0107 and the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

process. In [3] a formal definition of the operational semantics of DREAM in terms of a transition system, in the same spirit as the STG machine description in [11], was given. Our next step has been the construction of a distributed runtime system (RTS), i.e. a compiled version of DREAM. The RTS is explained in [2].

The main contribution of this paper is the translation of each Eden feature into imperative code calling the RTS routines. This translation is done in two phases: first, Eden expressions are *desugared* and translated into a much simpler parallel functional language we call PEARL (Parallel Eden Abstract Reduction Language); then, PEARL expressions are translated into imperative code. We discuss in detail the interaction between the compiled code and the RTS by using scenarios describing the main activities taking place at runtime (i.e. how processes are instantiated, etc).

Outline of the paper: we begin with a short presentation of the coordination features in Eden. Then, in Section 3, we explain the characteristics of PEARL and give translation schemes from Eden expressions into PEARL expressions. Section 4, contains a brief overview of the runtime system. Translation schemes from PEARL expressions into a sequence of RTS routine calls are given in the next section. Then, the most important scenarios are presented. We conclude with some references to related work.

2 An overview of Eden

In this section, the language Eden is presented very briefly. For details, the interested reader is referred to [6] and [5]. Eden extends the lazy functional language Haskell with a set of *coordination* features, aimed to express parallel algorithms.

Coordination features of Eden They are based on two principal concepts: *explicit management of processes* and *implicit communication.*

Functional languages distinguish between function definitions and function applications. Similarly, Eden offers *process abstractions*, i.e. abstract schemes for process behaviour, and *process instantiations* for the actual creation of processes and of the corresponding communication channels. A process mapping input variables $x_1, \ldots, x_n$ to output expressions $exp_1, \ldots, exp_k$ can be specified by the following process abstraction:

$$\texttt{process}\ (x_1, \ldots, x_n)\ \texttt{->}\ (exp_1, \ldots, exp_k)$$
$$\texttt{where}\ equation_1 \ldots equation_r$$

The output expressions can reference the input variables, as well as the auxiliary functions and common subexpressions defined in the optional `where` part. A process can communicate to other processes through tuples of channels and arbitrary data structures of channels. The latter are explained in [1].

A process instantiation expression has the form:

$$(y_1, \ldots, y_k) = p \ \#\ (exp_1, \ldots, exp_n)$$

From now on, we will refer to the new instantiated process as the *child* process, while the process where the instantiation takes place, is called the *parent*. The process abstraction bound to p is applied to a tuple of input expressions, yielding a tuple of output variables. The child process uses k independent threads of control in order to produce these outputs. Correspondingly, the parent process creates n additional threads for evaluating $exp_1, \ldots, exp_n$.

Communication is unidirectional, from one producer to exactly one consumer. In order to provide control over where expressions are evaluated, only fully evaluated data objects are communicated. This also facilitates a distributed implementation of Eden. But retaining the lazy nature of the underlying computational functional language, lists are transmitted in a *stream*-like fashion, i.e. element by element.

Concurrent threads trying to access not yet available input will be suspended. This is the only way of synchronizing Eden processes.

New communication channels can be established at runtime by means of *dynamic reply channels*, which are specially useful for the description of reactive concurrent systems. A process may generate a new input channel by executing the following expression:

$$\texttt{new}\ (var_{cn},\ var_c)\ exp$$

where the value var_c of the created channel can be used in the expression *exp*, and the channel name var_{cn} can be sent to another process. A process receiving var_{cn} can use it to output a value through the channel[1] referenced by this name by executing:

$$var_{cn}\ \texttt{!*}\ exp_1\ \texttt{par}^2\ exp_2$$

Before exp_2 is evaluated, a new concurrent thread for the evaluation of exp_1 is generated, whose result is transmitted via the received reply channel.

Nondeterminism constitutes a controversial point in language design. On the one hand, it is an essential feature for some program applications but, on the other hand, it spoils the purity of functional languages. In Eden, nondeterminism is gently introduced by means of a predefined process abstraction `merge` which is used to instantiate nondeterministic processes, each one fairly merging a list of input channels into a single list.

The following table summarizes Eden's coordination constructs:

$exp \rightarrow$	`process` *pattern* `->` *exp*	
	`where` *equations*	process abstraction
	$\mid exp_1 \# exp_2$	process instantiation
	$\mid$ `new` $(var_{cn},\ var_c)\ exp$	dynamic channel creation
	$\mid var_{cn}$ `!*` exp_1 `par` exp_2	dynamic channel connection
	$\mid$ `merge`	merge process abstraction

[1] To ensure that channels connect unique writers to unique readers, it is checked at runtime that no other process is already connected (see subsection 5.3).

[2] This construction in Eden is not related to the homonymous function in [14].

The evaluation model for Eden process systems The evaluation of an Eden process is driven by the evaluation of its output expressions, each of which constitutes an independent thread of execution. Although this overrules lazy evaluation to some extent, in order to speed up the generation of new processes and the distribution of the computation all *top level* process instantiations are immediately evaluated. Therefore, if the demand driven evaluation of inner process instantiations hinders the unfolding of the parallel process system, the programmer has the possibility to lift them to the top level where they are immediately evaluated. Nevertheless, the system evolution is still controled by the demand for data, so that a process without output terminates immediately and its input channels are eliminated. This implies the abolition of the corresponding output channels in sender processes, and the propagation of process termination.

Example: The following Eden program specifies a parallel mergesort algorithm, where a parameter `h` controls the granularity by specifying the desired height of the process tree to be generated (`h`=0 for one process, `h`=1 for three processes, etc):

```
parMsort h = process xs -> ys
   where ys | h == 0 = seqMsort xs
            | h > 0  = ordMerge (parMsort (h-1) # x1, parMsort (h-1) # x2)
                       where (x1,x2) = unshuffle xs
seqMsort []  =  []
seqMsort [x] =  [x]
seqMsort xs  =  ordMerge (seqMsort x1, seqMsort x2)
                where (x1,x2) = unshuffle xs
```

Notice the difference between the parameter `h`, which must be known at instantiation time, and the input `xs`, i.e. the list of values to be sorted, which will be gradually provided during the execution of the process. Ignoring the height parameter `h`, the similarity between the sequential and parallel versions is remarkable. While the former makes two function applications —function `seqMsort`—, the latter produces two process instantiations —process abstraction `parMsort`. Sequential functions `unshuffle` (splitting a list into two by putting odd position elements into one list and even position ones into another) and `ordMerge` (merging two ordered lists into a single ordered list) are written in plain Haskell and are not shown here.

3 First translation phase: Eden into PEARL

PEARL is the language of DREAM, the abstract machine for Eden. Before going into the details of the translation from EDEN into PEARL, we give a brief description of the syntax for PEARL.

$prog$	$\rightarrow binds$	
$binds$	$\rightarrow bind_1\,; \ldots ;\, bind_n$	$n \geq 1$
$bind$	$\rightarrow var\ =\ lf$	
lf	$\rightarrow vars_f \backslash\pi\ vars_a$ `->` $expr$	
π	$\rightarrow u \mid n$	
$expr$	$\rightarrow literal$	
	$\mid\ constr\ atoms$	constructor application
	$\mid\ primOp\ atoms$	primitive application
	$\mid\ var\ atoms$	application
	$\mid$ `let` $binds$ `in` $expr$	local definition
	$\mid$ `letrec` $binds$ `in` $expr$	local recursive definition
	$\mid$ `case` $expr$ `of` $alts$	case expression
$vars$	$\rightarrow \{var_1, \ldots, var_n\}$	$n \geq 0$, variable list
$atoms$	$\rightarrow \{atom_1, \ldots, atom_n\}$	$n \geq 0$, atom list
$atom$	$\rightarrow var \mid literal$	

$bind$	$\rightarrow vars_o = var_p$ `#` var_i	**process instantiation**
lf	$\rightarrow vars_f$ `\proc` $vars_a$ `->` $expr$	**process abstraction**
$expr$	$\rightarrow$ `{|`var_1`||` ... `||`var_k`|}`	$k \geq 1$, **parallel expression**
	$\mid$ `new` $\{var_{cn}, var_c\}\ var_e$	**dynamic channel creation**
	$\mid$ `write` $var_{cn}\ var_{e1}\ var_{e2}$	**dynamic channel use**
$primOp$	$\rightarrow$ `sendVal` $\mid$ `sendHead` $\mid$ `closeStrm`	**communication primitives**

Fig. 1. Syntax of STGL (above) and PEARL extensions (below)

3.1 PEARL

It extends the language of GHC, called STGL (Spineless Tagless G-machine Language), by constructs for the coordination features in Eden. A summary of the syntax of STGL, as it has been defined in [11], is presented in the upper part of figure 1, while PEARL extensions are shown in the lower part. An STGL program is a non-empty sequence of bindings of variables to lambda forms: $vars_f \backslash\pi\ vars_a$ `->` $expr$, i.e. lambda abstractions with additional information about the free variables $vars_f$ appearing in the body, and a flag π indicating whether the corresponding closure should be updated or not after its evaluation. PEARL introduces a new binding expression for defining process instantiations: $vars_o = var_p$ `#` var_i, where $vars_o$ denotes the list of variables receiving the output of the process, var_p is bound to a process abstraction, and var_i is bound to the input expression. Process instantiations can only occur in the binding part of `let` and `letrec` expressions.

Lambda forms in PEARL can be either original STGL lambda forms defining functions, or process abstractions defining Eden processes. In this latter case, the arguments $vars_a$ represent the formal input variables of the process, and the process body is an expression which evaluates the tuple of output expressions. As outputs are evaluated in parallel, the whole body eventually evaluates to a parallel expression $\{ | var_1 | | \ldots | | var_k | \}$, where each component expression will give rise to a different concurrent thread. Parallel expressions are used too when instantiating new processes, because the actual inputs for the child process are also evaluated by independent threads in the parent process. Both situations are reflected in the translation schemes of Section 3.2.

Other PEARL expressions are introduced for dynamic channel handling: `new` for the creation and `write` for using it. In the expression for creating a channel, var_{cn} and var_c represent, respectively, the channel name and the channel value, and var_e is bound to a lambda abstraction `ChanName a -> a -> b` expecting the channel name and the channel value to produce a result of type `b`. In the `write` expression, var_{cn} corresponds to the channel name, and var_{e1} and var_{e2} are supposed to be respectively bound to the message expression and to the continuation expression.

At PEARL level, communication is more explicitely handled. The primitive function `sendVal` is used to send a single value of any type, `sendHead` sends the next value of a stream and `closeStrm` closes a stream. However, the actual process intercommunication remains implicit, i.e. the use of these primitives is not required to provide a particular channel or receiver. These primitive functions form the basis of the overloaded function `sendChan`, shown below, which transmits values on a channel. In the translation schemes of the next subsection, the compiler inserts an application of `sendChan` to any expression to be evaluated by a parallel thread. The class `Transmissible` contains all the types whose values can be transmitted from one process to another. The compiler provides *derived* instances of this class for any user defined type.

```
class NFData a => Transmissible a where
  sendChan :: a -> ()
  sendChan x = rnf x ‘seq‘ sendVal x       -- default definition

instance Transmissible [a] where
  sendChan []     =  closeStrm
  sendChan (x:xs) =  rnf x ‘seq‘ sendHead x ‘seq‘ sendChan xs
```

`sendChan` uses the overloaded function `rnf :: a -> ()` to reduce elements to normal form. This is defined in the implementation of *Glasgow Parallel Haskell* [14]. The class definition follows:

```
class Eval a => NFData a where
 rnf :: a -> ()
 rnf a = a ‘seq‘ ()                        -- default definition
```

The function `seq :: a -> b -> b` evaluates its first argument to head normal form and then returns its second argument.

```
x ‘seq‘ y = case x of _ -> y
```

3.2 Translating Eden expressions into PEARL

We present the translation schemes for the new expressions introduced by Eden with respect to Haskell, in the form of a recursive function $tr_1 :: \mathit{Eden} \rightarrow \mathit{PEARL}$. For the rest of the computational language, i.e. normal Haskell expressions, this function provides the original Haskell to STGL translation, and it is not shown here.

process abstraction

$$
\begin{array}{l}
tr_1(\,\texttt{process}\ (x_1,\ldots,x_n)\ \texttt{->}\ e\ \texttt{where}\ \mathit{eqns}) \stackrel{\text{def}}{=} \\
\quad \texttt{let}\ f = \{\mathit{frees}\}\ \backslash n\ \{x_1,\ldots,x_n\}\ \texttt{->}\ tr_1(\texttt{let}\ \mathit{eqns}\ \texttt{in}\ e) \\
\quad \texttt{in let}\ p = \{f\}\ \backslash\texttt{proc}\ \{x_1,\ldots,x_n\}\ \texttt{->}\ \texttt{case}\ f\ \{x_1,\ldots,x_n\}\ \texttt{of} \\
\qquad\qquad (y_1,\ldots,y_k)\ \texttt{->}\ \texttt{let}\ v_1 = \{y_1\}\ \backslash\ u\ \{\}\ \texttt{->}\ \texttt{sendChan}\ y_1\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad \cdots \\
\qquad\qquad\qquad\qquad\qquad \texttt{let}\ v_k = \{y_k\}\ \backslash\ u\ \{\}\ \texttt{->}\ \texttt{sendChan}\ y_k\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad \{|v_1\,||\ldots||\,v_k|\} \\
\qquad \texttt{in}\ p
\end{array}
$$

We assume the type of the process abstraction to be $\texttt{Process}\ (t_1,\ldots,t_n)\ (t'_1,\ldots,t'_k)$ and *frees* to be the free variables in the body e and in the auxiliary equations *eqns*. The whole expression reduces to a variable p bound to a process abstraction which evaluates its outputs in parallel. The body is just a lambda abstraction applied to the inputs.

process instantiation

$$
\begin{array}{l}
tr_1(\,e_1 \# e_2) \stackrel{\text{def}}{=} \\
\quad \texttt{letrec}\ p = \{\mathit{frees}_1\}\ \backslash u\ \{\}\ \texttt{->}\ tr_1(e_1) \\
\qquad\qquad v = \{\mathit{frees}_2\}\ \backslash u\ \{\}\ \texttt{->}\ \texttt{case}\ tr_1(e_2)\ \texttt{of} \\
\qquad\qquad\qquad (x_1,\ldots,x_n)\ \texttt{->}\ \texttt{let}\ v_1 = \{x_1\}\ \backslash\ u\ \{\}\ \texttt{->}\ \texttt{sendChan}\ x_1\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ v_n = \{x_n\}\ \backslash\ u\ \{\}\ \texttt{->}\ \texttt{sendChan}\ x_n\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \{|v_1\,||\ldots||\,v_n|\} \\
\qquad\qquad (y_1\ldots,y_k) = p \# v \\
\quad \texttt{in}\ (y_1,\ldots,y_k)
\end{array}
$$

We assume the type of e_1 to be $\texttt{Process}\ (t_1,\ldots,t_n)\ (t'_1,\ldots,t'_k)$, the type of e_2 to be $(t_1,\ldots,t_n)$, and frees_i to be the free variables in e_i $(i = 1,2)$. The translation specifies that a process abstraction expression p will be evaluated and that its inputs will be fed in parallel with the result of evaluating the tuple e_2. The result of the whole evaluation is the tuple of outputs produced by the instantiated process.

Actually, process instantiations are not translated one by one. Instead, they are collected in a single `letrec` expression. This feature is crucial for the eager distribution of the computation.

dynamic channel creation and use

$$
\begin{array}{l}
tr_1(\,\texttt{new}\ (cn,c)\ e) \stackrel{\text{def}}{=} \\
\quad \texttt{let}\ v = \{\mathit{frees}\}\backslash n\ \{cn,c\}\ \texttt{->}\ tr_1(e) \\
\quad \texttt{in new}\ \{cn',c'\}\ v
\end{array}
$$

$tr_1($ *cn* `!*` e_1 `par` $e_2) \stackrel{\text{def}}{=}$

$\quad$ `let` $v_1 = \{frees_1\} \setminus u \; \{\}$ `->` $tr_1(e_1)$
$\quad\quad\;\;$ $v_2 = \{frees_2\} \setminus u \; \{\}$ `->` $tr_1(e_2)$
$\quad$ `in write` $cn \; v_1 \; v_2$

The translation in these two last cases is quite straightforward and requires little explanation. As usual, *frees*, $frees_1$ and $frees_2$ are the free variables in their respective expressions. Let us note, in the channel creation case, that the free variables *cn* and *c* in Eden's expression are changed to lambda-bound variables in the translation. This facilitates the further translation of this expression into imperative code, as we only have to apply v to the fresh variables cn' and c'.

4 A distributed runtime system for Eden

The environment to execute an Eden program consists of a set of distributed abstract machines, each machine executing an instance of an Eden process abstraction. DREAM refers to the whole set of machines, while each single abstract machine is referred to as a *Dream*. Recall that an Eden process consists of one or more concurrent threads of control. These evaluate different output expressions which are independent of each other and use a common heap that contains shared information. The destination of the value produced by the thread is called an *outport*. In essence, an outport is a global address consisting of a destination processing element together with a unique local identifier in this processor called an *inport*. The state of a Dream includes information common to all threads, and the individual state of each thread. As input is shared among all threads, the shared part in a Dream includes an *inport table* as well as the heap. The inport table maps inport identifiers to the respective heap addresses, where received messages are to be stored. These heap addresses point to `queueMe` closures which represent not yet received messages. Should a thread enter a `queueMe` closure, it would get blocked into a queue where all threads needing a not received value wait until the value arrives.

The state of a thread comprises the state of a sequential abstract machine and a specification of the associated outport referencing the connected inport. Therefore, the interface of a process is visible in the following way: The inports are enumerated in the inport table and the outports are associated to threads (either runnable or blocked). Communication channels are only represented by inports and outports which are connected to each other. There are no explicit channel buffers. Instead, the data is transferred from the producer's heap to the consumer's heap using the inport table to find the location where the data should be stored.

Let us now obtain a compilable version of our DREAM abstract machine in terms of a distributed runtime system consisting of a set of *processor elements* (PE), each containing a pool of runnable threads, each represented by a heap-allocated Thread State Object (TSO) containing the thread's registers and pointers to the heap-allocated argument and return stacks.

Mapping of several Dreams to one PE: As we allow more Eden processes than we have PEs available, several Eden processes need to share a PE. We can easily map several Dreams onto one abstract machine, by joining their interface representations, threads, heaps, and code. This enables a cheap dynamic creation of Eden processes, whereas the process system of the underlying message passing system is static: one abstract machine process per available PE.

Runtime tables: In order to be able to reference inports and outports of other PEs, we need immutable global references of the form *(PEId,Id)*, where *Id* is a locally unique identifier. We cannot simply use the heap addresses of the `queueMe` closures and TSOs —for inports and outports, respectively—, because they may change under garbage collection. Therefore, each PE keeps one *inport* and one *outport* runtime table, where the current heap addresses are stored, and indexes to these tables are used as local *Id*s. For termination purposes, the global reference to the sender thread is saved, together with the `queueMe` closure address, in the inport table; while outport identifiers are mapped to TSOs. As several Eden processes share the same inport and outport tables, each PE also needs a separate *process* table, with the lists of the current inports and outports of each process.

Communication unit and protocol: The communication unit is responsible for communicating with other PEs, providing sending and receiving routines for the following messages:

Data messages concerning the communication of values through channels:

- *VAL* for sending a value over a one value channel.
- *HEAD* for sending an element of a stream.
- *CLOSE-STRM* for closing a stream.

System messages implying modifications on the runtime tables of a remote PE. Messages of this kind always include the identification of the sending PE, to be used appropriately at the receiver PE.

- *CREATE-PROCESS*: Is sent by the parent to a remote PE[3] to instantiate a new process there. The process abstraction and the new local inports and outports of the parent are included in the message.
- *ACK*: Is the response to a *CREATE-PROCESS* message, acknowledging the creation of the child process and including information to properly connect the inports and outports of the child in the parent process.
- *DCH*: When a process uses a dynamic channel, it must communicate to the creator of the channel the new connection.
- *TERM-THREAD*: For terminating a remote thread, represented by its associated outport.

[3] The specific remote PE will be decided by the RTS of the parent PE, depending on the global scheduling and load balancing policies.

Termination When a thread has completed its computation and has sent its output value, it terminates closing the corresponding outport. If the last outport in a process is closed —this can be looked up in the process table—, the whole process is terminated by closing all its inports. When an inport is closed, the sending thread is informed that its results are not longer needed. This is done via a *TERM-THREAD* message. Inports may also be closed at garbage collection, when it is detected that the input is not referenced anymore.

Only a brief overview of the runtime system for Eden has been given here. It suffices to understand the second translation phase described in the next section and its interaction with the RST. For a more detailed presentation and a discussion on topics not adressed here, e.g scheduling, process distribution, garbage collection, etc., the interested reader is referred to [2].

5 Second translation phase and runtime scenarios

In this section we present the translation of PEARL expressions into imperative code, i.e. a sequence of calls to the procedures of the RTS. As we did it in section 3.2 for the first translation phase from Eden to Pearl, we present the translation schemes in the form of a recursive function $tr_2 :: PEARL \to RTS$. Function tr_2 is here only described for the new expressions introduced by PEARL with respect to STGL.

In order to better understand the runtime behaviour of Eden programs, we present the translation schemes grouped into two "scenarios": *process creation* and *dynamic channel handling*. In each scenario, we explain the interaction between the code generated by PEARL expressions and the code executed by the RTS upon the reception of system messages. In this way, the reader can get a whole picture of what happens at runtime.

In the following, we will assume that process abstractions always have type `Process` $(t_1, \ldots, t_n)$ $(t'_1, \ldots, t'_k)$, i.e. they have n inports and k outports. The following notation and conventions will also be used:

inport (outport) identifiers Local inport (outport) identifiers will be denoted by $i_1, \ldots, i_n$ $(o_1, \ldots, o_k)$ in the child process and by $i'_1, \ldots, i'_k$ $(o'_1, \ldots, o'_n)$ in the parent process. The *destination* of a thread (*origin* of an inport) is its corresponding remote inport (outport), and it is represented by a global inport (outport) identifier, i.e. a tuple (idp, i) $((idp, o))$ where idp is a remote PE and i (o) is a local inport (outport) in that PE. Destinations (origins) may also be denoted by D (O).

closure addresses $a, b, p, q, e, \ldots$

thread state objects are distinguished from other closure addresses and denoted by tso.

procedure calls are of the form $o_1, \ldots, o_r \leftarrow name(i_1, \ldots, i_s)$, where $o_1, \ldots, o_r$ are the output parameters, $name$ is the name of the called procedure and $i_1, \ldots, i_s$ are the input parameters.

5.1 Interface between the imperative code and the RTS

The self-explanatory procedure names and the accompanying text will, in general, suffice to understand the translation. The less obvious procedures offered by the RTS are explained in advance.

$tso \leftarrow createThread(e, D)$ Creates a thread tso, where e is the heap address of the expression to be executed, and D is the destination of the thread. If this is not known yet, or if the thread has no destination, the value *undefD* is used.

$idp \leftarrow sendCreateProcess(p, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$ The RTS will compose and send a *CREATE-PROCESS* message to some remote PE, and return the identifier *idp* of the chosen remote machine. The parameter p represents a closure that will eventually evaluate to a process abstraction, and o_{tso} is the outport associated with the thread (in parent's PE) responsible for spawning the threads feeding the inports of the child process. It is passed around in order to have a handle to this thread and to be able to activate it after the corresponding acknowledgement message arrives. The rest of the parameters are the inports —$(i'_1, \ldots, i'_k)$— and the outports —$(o'_1, \ldots, o'_n)$— at the parent's side, which correspond to outports, respect. inports, in child's PE.

$sendAck(idp, o, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$ Sends an *ACK* message to the matching thread for outport o in PE *idp*, acknowledging that the process instantiated from that thread has been created with inports $i_1, \ldots, i_n$ and outports $o_1, \ldots, o_k$. The rest of the parameters are just a repetition of the remote inports and outports received in the corresponding *CREATE-PROCESS* message.

5.2 Scenario 1: process creation

A graphical description of this scenario is shown in Figure 2. The parent and the child side, and the messages exchanged between both are represented there. As we will see in the generated code, newly created inports in both sides are initially bound to `queueMe` closures so that threads trying to consume values not yet produced by the remote process will be suspended. Outports should be matched to their corresponding threads, but in many cases these are not already known at the moment of creating the outport. Thus, outports are usually initially matched to *undefT* threads.

Translation of the process instantiation expression
The environment at the parent's side has to be created in order to be able to feed the inports of the remote process and to receive the values delivered by it through its outports. Therefore, k inports and n outports are created. However, at instantiation time of a new process not all the information is at hand at the parent's side to create the new outports. In fact, the output expressions which will feed each outport and the destinations of the new threads are not yet

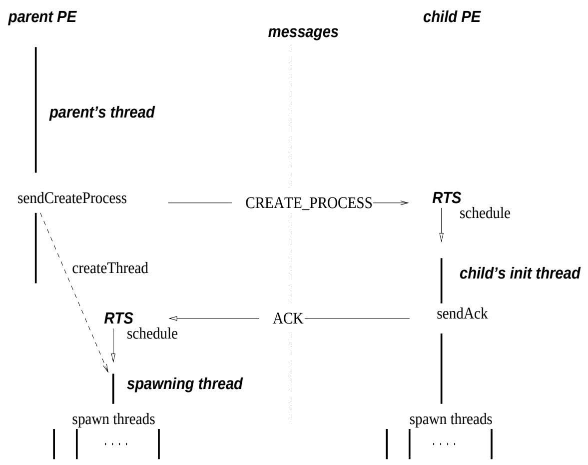

Fig. 2. Process creation scenario

known. Therefore, instead of directly matching the outports to their corresponding threads, a *spawning thread* is created that will eventually reduce to a parallel expression. When this happens, threads are spawned and matched to the previously created outports (see translation of the parallel expression below). Notice that, at this moment, this spawning thread is created but not yet activated. Then, the remote process creation is initiated —by sending the corresponding message—, and the tuple of new input channels is delivered to the continuation stack as the result of the whole instantiation. The addresses of the newly created `queueMe` closures form this tuple.

$$
\begin{aligned}
&tr_2\ (p\#e) \stackrel{\text{def}}{=} \\
&\quad a_1 \leftarrow createQM(); \ldots; a_k \leftarrow createQM(); \\
&\quad i'_1 \leftarrow createInport(a_1); \ldots; i'_k \leftarrow createInport(a_k); \\
&\quad o'_1 \leftarrow createOutport(undefT); \ldots; o'_n \leftarrow createOutport(undefT); \\
&\quad tso \leftarrow createThread(e, undefD); \\
&\quad o_{tso} \leftarrow createOutport(tso); \\
&\quad idp_{child} \leftarrow sendCreateProcess(p, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n); \\
&\quad returnTuple(a_1, \ldots, a_k)
\end{aligned}
$$

Arrival of the process creation message

When the RTS of the PE where the child process must be instantiated, receives the message $CREATE\text{-}PROCESS(idp_{parent}, p, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$, it first unpacks the closure p and copies it to the local heap — let p' denote the address of the copied closure— then, it creates the initial thread:

$$
\begin{aligned}
&tso \leftarrow createThread(p', undefD); \\
&initArg(tso, idp_{parent}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n); \\
&scheduleThread(tso)
\end{aligned}
$$

Before scheduling the initial thread, all parameters in the message, except p, are saved in the argument stack, so that the code generated for the process abstraction (see below) can use them.

Translation of the process abstraction expression

The code generated for a process abstraction assumes that the executing thread's stack contains the arguments mentioned above. This is safe as we call it only in this setting.

Now, the environment of the new instantiated process must be established: n inports and k outports are created. As the origins of the inports are already known, the connection is completed. Thereafter, the destinations and the outports of the future parallel threads are saved —in a table associated to the spawning thread state—, an ACK message is sent, and the body of the process abstraction is executed. The arguments $x_1, \ldots, x_n$ are free in the body of the process abstraction, and the values bound to them are supposed to be in the argument stack. That is the reason for pushing there the addresses of the `queueMe` closures.

$$
\begin{array}{l}
tr_2\ (\{frees\}\ \texttt{\textbackslash proc}\ \ \{x_1,\ldots,x_n\} \rightarrow body) \stackrel{\mathrm{def}}{=} \\
\quad idp_{parent} \leftarrow popArg(); \\
\quad o_{tso} \leftarrow popArg(); \\
\quad i'_1 \leftarrow popArg(); \ldots ; i'_k \leftarrow popArg(); \\
\quad o'_1 \leftarrow popArg(); \ldots ; o'_n \leftarrow popArg(); \\
\quad b_1 \leftarrow createQM(); \ldots ; b_n \leftarrow createQM(); \\
\quad i_1 \leftarrow createInport(b_1); \ldots ; i_n \leftarrow createInport(b_n); \\
\quad o_1 \leftarrow createOutport(undefT); \ldots ; o_k \leftarrow createOutport(undefT); \\
\quad connectInport(i_1, (idp_{parent}, o'_1)); \ldots ; connectInport(i_n, (idp_{parent}, o'_n)); \\
\quad tso \leftarrow ownTSO() : \\
\quad saveDestinations(tso, (idp_{parent}, i'_1), \ldots, (idp_{parent}, i'_k)); \\
\quad saveOutports(tso, o_1, \ldots, o_k); \\
\quad sendAck(idp_{parent}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k); \\
\quad pushArg(b_n); \ldots ; pushArg(b_1); \\
\quad tr_2(body)
\end{array}
$$

Arrival of the acknowledgement message

When a message $ACK(idp_{child}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$ arrives to the parent's PE, the outports and destinations of the threads, which will be later created by the parallel expression, are stored in the state of the spawning thread. Inports are connected, and the spawning thread is scheduled:

$$
\begin{array}{l}
tso \leftarrow getThread(o_{tso}); \\
saveDestinations(tso, (idp_{child}, i_1), \ldots, (idp_{child}, i_n)); \\
saveOutports(tso, o'_1, \ldots, o'_n); \\
connectInport(i'_1, (idp_{child}, o_1)); \ldots ; connectInport(i'_k, (idp_{child}, o_k)); \\
scheduleThread(tso)
\end{array}
$$

Translation of the parallel expression

The executing thread will spawn a set of m parallel threads and thereafter die. The translation below is valid both for parallel expressions in process instantiations (parent's side) and for parallel expressions in process abstractions (child's side). It is assumed that the destinations and the outports associated to the spawned threads have been previously saved in the state of the executing thread.

$$
\begin{aligned}
& tr_2\ (\{\,|\,e_1\,|\,|\ldots|\,|\,e_m\,|\,\}) \stackrel{\text{def}}{=} \\
& \quad D_1, \ldots, D_m \leftarrow getDestinations(); \\
& \quad o_1, \ldots, o_m \leftarrow getOutports(); \\
& \quad tso_1 \leftarrow createThread(e_1, D_1); \ldots; tso_m \leftarrow createThread(e_m, D_m); \\
& \quad matchOutportThread(o_1, tso_1); \ldots; matchOutportThread(o_m, tso_m); \\
& \quad scheduleThread(tso_1); \ldots; scheduleThread(tso_n); \\
& \quad exitThread()
\end{aligned}
$$

5.3 Scenario 2: dynamic channel handling

Translation of the channel creation expression

A new inport is created in the current machine, bound to a `queueMe` closure and the global address of this inport is saved in a `ChanName` closure which, in essence, is a tuple. Then, the main expression is entered, having previously pushed in the stack the arguments of this lambda abstraction.

$$
\begin{aligned}
& tr_2\ (\texttt{new}\ \{cn, c\}\ e) \stackrel{\text{def}}{=} \\
& \quad idp_{creator} \leftarrow ownPE(); \\
& \quad q \leftarrow createQM(); \\
& \quad i \leftarrow createInport(q); \\
& \quad a \leftarrow createCHN(idp_{creator}, i); \\
& \quad pushArg(q); \\
& \quad pushArg(a); \\
& \quad enter(e)
\end{aligned}
$$

Let us note that the binding variables c and cn are ignored and replaced by the fresh heap addresses q and a. The first one, i.e. q, is bound to a `queueMe` closure, representing the new channel, whereas the second one, a, is bound to a `ChanName` closure containing the global address of the channel.

Translation of the channel use expression

The message expression v_1 and the main expression v_2 are evaluated by independent threads. Therefore, a new thread is created and scheduled for the message evaluation, whereas the current thread continues with the evaluation of the main expression. A new local outport o matching the new thread is created. Then a DCH message is sent to communicate to the creator of the dynamic channel the new connection.

tr_2 (`write` $cn\ v_1\ v_2$) $\stackrel{\text{def}}{=}$

$$
\begin{array}{l}
idp_{creator}, i \leftarrow accessCHN(cn); \\
tso \leftarrow createThread(v_1, (idp_{creator}, i)); \\
o \leftarrow createOutport(tso); \\
sendDch(idp_{creator}, i, o); \\
scheduleThread(tso); \\
enter(v_2)
\end{array}
$$

Arrival of the dynamic channel connection message
When the message $DCH\,(idp_{user}, i, o)$ arrives to the dynamic channel creator's PE, the RTS will just connect the local inport i to its new established origin (idp_{user}, o), by executing:

$connectInport(i, (idp_{user}, o));$

The fact that no other process has connected to this channel before can be checked by looking up in the inport table.

6 Related and future work

The specialized code generation defined here instantiates Eden processes, each one consisting of a number of threads and a complex communication structure, whereas languages like Concurrent ML [13] or Facile [8] use explicit message passing primitives on the language level, and they treat these primitives as ordinary functions during code generation. Other systems like GpH [14] and *Concurrent Clean* [16] handle communication implicitely by relying on a virtually shared memory. Their parallel annotations are also implemented in a straightforward manner by primitive operations. Closer to our approach is Caliban [7], which extracts topology information from the program text and needs a special code generation, but it is restricted to static process networks.

The starting point for the implementation of Eden has been the Glasgow Haskell Compiler (GHC) [12]. Only small modifications to the parser were needed in order to accept the new expressions Eden has with respect to Haskell. As a consequence, our abstract machine DREAM is an extension of the STG machine [11] on which GHC is based. The extensions concern multithreading and communication features. The current runtime system for Eden has much profited from the runtime system GUM for Glasgow Parallel Haskell [15]. The implementation status is as follows: a compiler for the basic features of Eden is running and producing C code; PEARL constructions have been implemented by a set of predefined C functions called from this code; the RTS has been coded and the whole system is being integrated. In brief, we will present efficiency results with this compiler. Preliminary figures were obtained with a first distributed prototype directly calling MPI from Haskell, and they were published elsewere [2]. They promise good speedups for the actual implementation. Eden features not covered up to now are channel structures and implementation optimizations such as channel bypassing.

Acknowledgements We thank Luis Antonio Galán, Rita Loogen, Cristóbal Pareja, Steffen Priebe, Fernando Rubio and Clara Segura for useful discussions, and comments provided while preparing this paper.

References

1. Silvia Breitinger, Ulrike Klusik, and Rita Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Functional Programming 1997*, 1998. revised version.
2. Silvia Breitinger, Ulrike Klusik, and Rita Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98*, Springer LNCS 1490, pages 318–334, 1998.
3. Silvia Breitinger, Ulrike Klusik, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. DREAM - the DistRibuted Eden Abstract Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Symposium on the Implementation of Funct. Lang. 1997, St. Andrews, selected papers*, LNCS 1467. Springer, 1998.
4. Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Concurrency in Functional and Logic Languages. In *Proceedings of the Fuji Int. Workshop on Functional and Logic Programming, Japan*. World Scientific Publishing Company, 1995. ISBN 981-02-2437-0.
5. Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.
6. Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Progr. Models and Supportive Env. (HIPS)*. IEEE Press, 1997.
7. Stuart Cox, Shell-Ying Huang, Paul Kelly, Junxian Liu, and Frank Taylor. An implementation of static functional process networks. In *Parallel Architectures and Languages Europe*, pages 497–512. Springer Verlag, 92.
8. A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Prog.*, 18(2), 1989.
9. Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic parallelization of lazy functional programs. *ESOP*, 1992.
10. John Peterson and Kevin Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical Report YALEU/DCS/RR-1106, Yale University, 1997.
11. S. Peyton-Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, version 2.5. *Journal of Functional Programming*, 2(2):127–202, April 1992.
12. Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *JFIT*, 93.
13. J. Reppy. CML: A higher-order concurrent language. In *ACM PLDI*, 1991.
14. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.
15. P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *ACM PLDI*, 1996.
16. Marco van Eekelen and Rinus Plasmeijer. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.