DREAM: the DistRibuted Eden Abstract Machine

S. Breitinger, U. Klusik, R. Loogen[1] and Y. Ortega, R. Peña[2]

[1] Philipps-Universität Marburg
[2] Universidad Complutense de Madrid

Abstract. Eden is being implemented by extending the Glasgow Haskell Compiler (ghc) which is based on the Spineless Tagless G-Machine (STGM). In this paper we present a parallel abstract machine which embodies the STGM for sequential computations and describes a distributed runtime environment for Eden programs on an abstract level.

1 Introduction

The parallel functional programming language Eden is defined by adding a coordination language to the computation language Haskell. In this paper we will show how the implementation of Eden can be developed in the same modular way as the language definition. In contrast to the prototype implementation described in [1], the parallel implementation presented in this paper requires not only the compiler, but also the runtime system to be specifically tailored for Eden.

An important design decision for the implementation of Eden has been to use the Glasgow Haskell implementation as a basis and to reuse of it as much as possible for efficiency purposes. In order to achieve this goal, we define PEARL (parallel Eden abstract reduction language) as a conservative extension of the STG language and DREAM (distributed Eden abstract machine) as a conservative extension of the STG machine. PEARL extends the STG language by constructs for the definition of process abstractions, process instantiations and for message passing.

A parallel computation in Eden is executed by a system of extended STG machine instances, so-called *Dreams*, each of which represents a parallel process. Internally, a Dream maintains multiple flows of control which implement the threads producing independent outputs in Eden processes. Every machine instance uses its own local heap which contains local data structures as well as references to buffers for receiving inputs. Special `queueMe` closures are used to suspend threads that try to read inputs which are not yet available.

As a further step to the implementation in an imperative language, we define a lower level language called *imperative DREAM*. In order to implement Eden on a parallel machine with distributed memory, the Dreams will be handled by a system of schedulers written in C + MPI[1], each of which runs on a different processor and controls the execution of a set of threads.

[1] Message Passing Interface, see [7]

Overview of the paper We start with a short summary of the language Eden in Section 2. In the following section we introduce PEARL, the language on which the machine operates. In Section 4 the DREAM machine is introduced and formally specified by a transition system. Section 5 discusses the compilation of PEARL into an imperative language. Finally we compare our approach with related work and draw conclusions.

2 Eden

In this section, we will present the language Eden very briefly. For a more detailed discussion, the interested reader is referred to [5] and [4].

Process abstractions define abstract schemes for processes. A process that maps inputs $in_1, \ldots, in_m$ to outputs $out_1, \ldots, out_n$ can be specified by:

$$\texttt{process}\ (in_1, \ldots, in_m)\ \texttt{->}\ (out_1, \ldots, out_n)$$
$$\texttt{where}\ equation_1 \ldots equation_r$$

The optional `where` part of this expression is used to define auxiliary functions and common subexpressions which occur within the process definition. A process can have as input (respectively, output) tuples of channels and data structures of channels[2].

Process instantiations create processes. In such an expression, a process abstraction p is applied to a tuple of input expressions, yielding a tuple of output channels. The child process uses n independent threads of control in order to produce these outputs. Likewise, the parent introduces m additional threads that evaluate $input\_exp_1, \ldots, input\_exp_m$:

$$(out_1, \ldots, out_n) = p\ \#\ (input\_exp_1, \ldots, input\_exp_m)$$

Communication channels are unidirectional and connect one writer to exactly one reader. Each channel provides a transmission buffer into which fully evaluated objects are transferred by the sender process. Lists are transmitted in a *stream*-like fashion, i.e piecemeal. The transfer of information via communication channels is done automatically, without explicit send or receive commands. A concurrent thread will be suspended if it needs some value from an inport whose channel buffer is empty.

Nondeterminism is introduced in Eden by a predefined process abstraction `merge` which creates a nondeterministic *fair* merging process for a list of input channels[3].

[2] The latter case is not handled in this paper, see the separate paper [3].

[3] As this process makes use of a channel structure as its input, it is not treated in this paper.

Dynamic reply channels can be used in order to establish new connections at runtime. A process may generate a new input channel and send a message containing the name of this new channel to another process. Such a new input channel *chan* and a reference *ch_name* to it are defined by the following expression:

$$\texttt{new}\ (ch\_name,\ chan)\ exp$$

A process that has received *ch_name* can use it for output[4] by:

$$ch\_name\ \texttt{!*}\ expression_1\ \texttt{par}\ expression_2$$

Before $expression_2$ is evaluated, a new concurrent thread for the evaluation of $expression_1$ is generated, whose result is transmitted via the received reply channel.

The evaluation of Eden processes is driven by the evaluation of the outport expressions, for which there is always demand. This rule overrides normal lazy evaluation in favour of parallelism. There will be a separate concurrent thread of execution for each output channel. Thus, we find in Eden two levels of concurrency: the concurrent or parallel evaluation of processes and the concurrent evaluation of different threads within processes. A user process with all its outputs closed will terminate immediately. On termination its input channels will be eliminated and the corresponding outports in the sender processes will be closed.

Furthermore, each process immediately evaluates all *top level* process instantiations. This may lead to speculative parallelism and again overrules lazy evaluation to some extent, but speeds up the generation of new processes and the distribution of the computation.

No implicit or explicit sharing of information among processes is possible. All kinds of communication and synchronization have to be performed using message passing and all output expressions have to be evaluated to normal form by the sender. These are important prerequisites for an efficient implementation on a system with distributed memory.

Example: The following Eden program specifies a parallel mergesort algorithm:

```
sortNet = process list -> sort list
          where sort []  =  []
                sort [x] =  [x]
                sort  xs =  merger # (sortNet # l1, sortNet # l2)
                            where (l1,l2) = unshuffle xs
                unshuffle []  = ([],[])
                unshuffle [x] = ([x],[])
                unshuffle (x:y:t) = (x:t1,y:t2)
                                    where (t1,t2) = unshuffle t
                merger = process (s1,s2) -> smerge s1 s2
```

[4] For the channels to connect unique writers to unique readers, it will be checked at runtime that no other process has connected to this channel before.

```
where 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
```

3 PEARL

PEARL (parallel Eden abstract reduction language), the language for DREAM (distributed Eden abstract machine), extends the core language of the Glasgow Haskell compiler, the STG language, by the core constructs of Eden. Figure 1 shows the syntax of the STG language as it has been defined in [11]. An STG program is a non-empty sequence of bindings of variables to lambda forms. A lambda form $vars_f \backslash\pi\ vars_a$ `->` $expr$ is a lambda abstraction $\backslash vars_a$ `->` $expr$ which additionally contains information about the free variables $vars_f$ and a flag π which indicates whether the lambda form, or to be precise, a closure with this lambda form, must be updated after its evaluation or not. The body expressions of lambda forms may be literals, applications, local definitions or case expressions.

The PEARL extensions of the STG language (see Figure 1) concern bindings, lambda forms, expressions and primitive operators. In addition to the normal binding of STGL we introduce new bindings which define process instantiations. Process instantiations will only occur in the binding part of `let` and `letrec` expressions.

Lambda forms in PEARL can be either process abstractions for the definition of processes or STGL lambda forms defining functions. Process abstractions are treated as special lambda forms with the tag `process` and a process body that evaluates to an expression $\{expr_1 \,||\, \ldots \,||\, expr_k\}$ defining parallel threads for the process outports. The two lists of variables $vars_f$ and $vars_a$ contain the free variables of the process body and the inports of the process abstraction, respectively.

As process instantiations and abstractions are only handled as bindings and lambda forms and not as special expressions, new expressions are merely introduced for dynamic channel handling. PEARL contains `new` and `par` expressions for the creation and use of dynamic channels. We use three new primitive operations in order to overrule the lazy evaluation strategy for outports and to manage the sending of evaluated values via channels. The function `sendVal` is used for sending a single value, `sendHead` sends the next value of a stream and `closeStrm` closes a stream. These form the basis of the polytypic function `sendChan` with type `a -> ()` shown below, which transmits values on a channel.

```
sendChan  l = case l of {Nil         -> closeStrm;
                         Cons {y,ys} -> force y `seq` sendHead y `seq`
                                        sendChan ys;
                         default     -> force l `seq` sendVal l }
```

$prog$	$\rightarrow binds$	
$binds$	$\rightarrow bind_1 \,;\ldots;\, bind_n$	$n \geq 1$
$bind$	$\rightarrow var\ =\ lf$	
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` $expr$ `in` $alts$	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
$atom$	$\rightarrow var \mid literal$	
$literal$	$\rightarrow$ `0#` $\mid$ `1#` $\mid \ldots$	primitive integers
$primOp$	$\rightarrow$ `+#` $\mid$ `-#` $\mid$ `*#` $\mid \ldots$	primitive operations
$alts$	$\rightarrow calt_1 \,;\ldots;\, calt_r;\ default$	$r \geq 0$, algebraic alternatives
	$\mid\ palt_1;\ldots;palt_n;\ default$	$n \geq 0$, primitive alternatives
$calt$	$\rightarrow constr\ vars$ `->` $expr$	
$palt$	$\rightarrow literal$ `->` $expr$	
$default$	$\rightarrow var$ `->` $expr$	
	$\mid$ `default ->` $expr$	

$bind$	$\rightarrow vars_o = var_p$ `#` $\{var_1$ `||` $\ldots$ `||` $var_n\}$	% **process instantiation**
lf	$\rightarrow vars_f$ **process** $vars_a$ `->` $pBody$	% **process abstraction**
$pBody$	$\rightarrow$ `letrec` $binds$ `in` $parExpr$	% **process body**
	$\mid\ parExpr$	
$parExpr$	$\rightarrow \{expr_1$ `||` $\ldots$ `||` $expr_k\}$	$k \geq 1$
$expr$	$\rightarrow$ `new` $(var_n, var_c)\ expr$	% **dynamic channel handling**
	$\mid\ var_n$ `!*` $expr_1$ `par` $expr_2$	
$primOp$	$\rightarrow$ `sendVal` $\mid$ `sendHead` $\mid$ `closeStrm`	% **primitives for communication**

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

`sendChan` uses another polytypic function `force :: a -> a` to reduce the elements of a list to normal form. The function `seq` evaluates its first argument and then returns its second argument.

```
force x = force' x `seq` x
force' (C a1 ... an) = force a1 `seq` ... `seq` force an `seq` ()

x `seq` y = case x of default -> y
```

The monomorphic instances of `force` will be derived by the compiler for any data type that may be sent. The compiler will also insert `sendChan` in the expressions to be evaluated by a thread, to implement the desired behaviour in the threads.

PEARL has been chosen to be as close as possible to STGL in order to reuse the STG machine for the sequential code of Eden. Note that the only new constructions to be taken care of by DREAM are process abstractions, instantiations, the constructs `new` and `par` as well as the new primitives `sendVal`, `sendHead` and `closeStrm`.

Example: The `sortNet` example shown in Section 2 will be translated into the following PEARL program:

```
sortNet = {sendChan,sortNet} \process {list} ->
    {let merger = {sendChan} \process {s1,s2} ->
                   {letrec smerge =  ... -- normal STG translation
                    in let oute = {smerge,s1,s2} \u {} -> smerge {s1,s2}
                       in  sendChan {oute} }
     in letrec unshuffle =  ...  -- STG-translation
        in letrec sort
             = {sendChan,merger,unshuffle,sort} \n {xs} ->
               case xs of
                {Nil         -> Nil {};
                 Cons {y,ys} ->
                   case ys of ->
                    { Nil     -> let nn = {} \n {} -> Nil {}
                                 in  Cons {y,nn};
                 default  ->
                  case unshuffle {xs} of
                   {(l1,l2) ->
                     let inp1 = {sendChan,l1} \u {} -> sendChan {l1}
                         inp2 = {sendChan,l2} \u {} -> sendChan {l2}
                     in let {l1'} =  sortNet # {inp1}
                            {l2'} =  sortNet # {inp2}
                     in let sl1 = {sendChan,l1'} \u {} -> sendChan {l1'}
                            sl2 = {sendChan,l2'} \u {} -> sendChan {l2'}
                     in let {res} = merger # {sl1,sl2}
                     in res   }     }    }
        in let sorted = {sort,list} \u {} -> sort {list}
        in sendChan {sorted} }
```

The bindings for `seq`, `force` and `sendChan` are provided in the global environment.

In the next section we will present the operational semantics of PEARL, which illustrates on an intermediate level the execution of Eden programs in a distributed machine.

4 DREAM

DREAM consists of a sequence of extended STG machines, called *Dreams*, one for each instantiated process. We use a set *DreamState* for the description of the current state of a Dream. A DREAM state is represented by a finite mapping of the set *Process_Id* of process numbers to the set *DreamState*. The transition rules define a binary relation on the set of DREAM states:

$$\text{DREAM} = \langle \mathit{Process\_Id} \rightarrow_{\text{finite}} \mathit{DreamState}, \Rightarrow \rangle,$$

where $\mathit{Process\_Id} := \mathbb{N}$. Before we define the state of the components as well as the transitions, we give a short description of the state and behaviour of the STG machine. The sequential transitions of the STG machine as they have been defined in [11] remain unchanged in the parallel context. They only have to be adapted to the extended state.

4.1 The STG machine

Let us denote the syntactic domains of the STG language as follows:

Expr	STG expressions
Var	variables, identifiers
Alts	alternatives of `case` expressions
Lfs	lambda forms
Γ	typed data constructors.

Let furthermore *Int* be a set of tagged integer values
Addr be a set of tagged heap addresses
$\mathit{Val} := \mathit{Int} \cup \mathit{Addr}$
$\mathit{Env} := [\mathit{Var} \rightarrow_{\text{finite}} \mathit{Val}]$

The state of the (sequential) STG machine consists of the following six components:

code	c	$\in \mathit{Instr}$, where *Instr* is defined below,
argument stack	as	$\in \mathit{Val}^*$,
return stack	rs	$\in \mathit{Cont}^*$, where $\mathit{Cont} = \mathit{Alts} \times \mathit{Env}$
update stack	us	$\in \mathit{Frame}$, where $\mathit{Frame} = \mathit{Val}^* \times \mathit{Cont}^* \times \mathit{Addr}$
heap	h	$: \mathit{Addr} \rightarrow \mathit{Closures}$, where $\mathit{Closures} = \mathit{Lfs} \times \mathit{Val}^*$, and
global environment	σ	$: \mathit{Var} \rightarrow \mathit{Addr}$.

It implements a single thread of control. The machine code instructions are the following:

$$\begin{array}{rlll} Instr = & \{Eval & e\ \rho & \mid e \in Expr, \rho \in Env\} \\ \cup & \{Enter & a & \mid a \in Addr\} \\ \cup & \{ReturnCon & (C\ w_1 \ldots w_n) & \mid C \in \Gamma, w_i \in Val\} \\ \cup & \{ReturnInt & k & \mid k \in Int\} \end{array}$$

where e is an STG expression, ρ is the local environment, a is a closure address, C is a constructor, w_i are either basic values or closure addresses, and k is a basic value. They have the following intuitive meaning:

Eval $e\ \rho$	evaluate expression e in the environment ρ and apply its value to the arguments on the argument stack
Enter a	apply the closure at address a to the arguments on the argument stack
ReturnCon $c\ ws$	return the constructor c applied to values ws to the continuation on the return stack
ReturnInt k	return the primitive integer k to the continuation on the return stack

A detailed specification and explanation of the STG machine can be found in [11]. In the following, we will show how this machine has to be extended in order to express the full range of parallel computations that can be programmed in Eden.

4.2 Extension of the STG machine

The parallel DREAM machine contains a Dream for each process instantiation. Within a Dream, multithreading is used to concurrently compute the different outputs of the corresponding process. There is a one to one correspondence between machine instances and processes and between threads and outports.

The state of a Dream contains for each thread the following components:

code	c
argument stack	as
return stack	rs
update stack	us
outport (list)	$out \in (Process\_Id \times Channel\_Id)^*$

and, in common for all threads:

heap	h
global environment	σ
inport table	$ipt : Channel\_Id \rightarrow Addr \times Process\_Id$

The first four components of the thread state are the same as those of the STG machine except that we extend the underlying sets of expressions (PEARL expressions), lambda forms and closures (see next section). The outport component contains information about the connection to inports of another instance which will be filled by the thread.

At this abstract level, an outport specification is a pair (m, c) where $m \in$ *Process_Id* is the identifier of a remote Dream and $c \in$ *Channel_Id* is a channel identifier in this remote machine. It denotes the destination of the output produced by the thread. In general each thread contains exactly one outport specification. There are two exceptions. The very first thread of a process which evaluates the process body contains the list of all outports communicating with the parent process. The thread evaluating the main expressions has no outport.

The inport table *ipt* maps inport identifiers to the respective heap addresses, where received messages are stored. Additionally, the sender is logged here to be able to check the 1:1 correspondance of inports to outports. The heap addresses point to `queueMe` closures, until channels are closed.

The set of heap closures is extended by a closure `queueMe` of the form:

$$\{\}\ \backslash n\ \{\}\texttt{->}\ \texttt{queueMe}\ q$$

which causes the suspension of a thread when entered, appending its state (code, argument stack, return stack, update stack, outport) to the associated queue q. In the following, we will represent this closure as a pair $\langle \texttt{queueMe}, [t_1, \ldots, t_n] \rangle$ where the t_i are thread states.

The suspension of threads is necessary to wait for not yet received messages. Thus, for a stream, a `queueMe` closure marks the write end where new incoming values must be appended and for a single valued channel the position for storing the value.

This special closure is also used to prevent the simultaneous evaluation of updatable closures by multiple threads. The thread which enters the updatable closure first overwrites it with a `queueMe`-closure. Any thread trying to enter the closure in the time interval between the start and the end of this evaluation reads this `queueMe` closure instead of the original closure, and thus will be suspended. In addition, we introduce `ChanName` closures

$$\{\}\ \backslash n\ \{\}\texttt{->}\ \texttt{ChanName}\ \ (pid, cid).$$

to represent the names of dynamic channels, where (pid, cid) is a regular outport specification.

The initial state of the system consists of a unique Dream embodying a thread for evaluating the main expression. Its local heap h_0 contains a closure for each globally defined variable, and the global environment σ_0 binds each global variable to the heap address of its closure. The input channel table is empty. The initial thread has empty stacks and no outport specification.

$$\boxed{\langle 0 \mapsto ((Eval\ \ \mathbf{main}\ \{\ \}\ \ \rho_{\emptyset})\ \ \varepsilon\ \ \varepsilon\ \ \varepsilon\ \ \varepsilon)\ \ h_0\ \ \sigma_0\ \ [\,]\rangle}$$

4.3 DREAM transitions

The execution of the code for a thread may cause changes not only to the thread itself, but also to the global state of the corresponding Dream. Moreover, there are evaluations which cause interactions with other Dreams, thus affecting the state of the whole system; for instance, process instantiation or communication. In the specifications we adopt the notation used in [11]. We only show the states of the machine instances involved in the transitions and use the symbol $\|$ to separate concurrent threads and $|||$ to separate parallel Dream instances.

We will present the transitions in the following order: We start with communication, which is performed between threads of different instances, afterwards explain the interaction of threads in the same instance and finally explain process creation.

Communication. In PEARL, lists which have to be communicated between processes, are transmitted in the form of streams. The predefined function `sendChan` forces the reduction of every list element to normal form and uses the primitives `sendHead` and `closeStrm` for sending the head of the stream and closing the channel respectively. Non-list values are evaluated to normal form and sent as a single value by `sendVal`.

Sending and receiving a single value: The outport in a thread's state contains a remote inport address consisting of a process instance number and a channel identifier. The remote inport table maps the channel identifier to a `queueMe`-closure address and a sender id, which must correspond to the id of the sender or be unknown ($\perp$) in the case of a dynamically created channel. The suspended threads collected in the `queueMe`-closure are reactivated and the closure is updated with the transmitted closure using the auxiliary function *graph_copy*. In fact, the whole subheap which can be referenced by the sent closure must be transmitted and embedded into the receiver's heap.

$$
\begin{array}{l}
\langle i \mapsto ((\mathit{Eval}\ (\mathbf{sendVal}\ v)\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ (j,c)) \parallel \mathit{threads}_i\ h_i\ \sigma_i\ \mathit{ipt}_i\rangle \\
|||\quad \langle j \mapsto \mathit{threads}_j\ h_j[a_{in} \longmapsto \langle \mathtt{queueMe}, ts\rangle]\ \sigma_j\ \ \mathit{ipt}_j[c \longmapsto (a_{in}, s)]\rangle \\
\qquad\qquad\qquad \text{with } s \in \{\perp, i\} \\
\Rightarrow \\
\langle i \mapsto \mathit{threads}_i\ \ h_i\ \ \sigma_i\ \ \mathit{ipt}_i\rangle \\
|||\quad \langle j \mapsto (ts \parallel \mathit{threads}_j)\ \ h'_j\ \ \sigma_j\ \mathit{ipt}_j[c \longmapsto \perp]\rangle \\
\qquad\qquad \text{where } h'_j = \mathit{graph\_copy}(h_j, a_{in}, \mathit{subHeap}(h_i, a), a) \\
\qquad\qquad\qquad\quad \mathit{Addr}\ a = \mathit{val}\ \sigma\ \rho\ v
\end{array}
$$

$\mathit{graph\_copy}(h_1, a_1, h_2, a_2)$ creates a copy of heap h_2 with new unique addresses within another heap h_1 starting at address a_1 where the node $h_2(a_2)$ is inserted. We omit details. To determine the subheap of a heap h which contains all nodes reachable (*reach*) from a list of node addresses ws we use the function *subHeap*:

$$\mathit{subHeap}(h, ws) = h_{|\mathit{reach}(h,ws)}.$$

The sending thread terminates after completing its task, namely evaluating an expression to normal form and sending its value via its output channel. The receiving process adds the value to its heap and deletes the inport from the inport table.

Sending and receiving a stream element: Sending a stream of values is similar to sending a single value except that the sending of values and the termination of the thread are done by separate primitive functions. Streams are stored in the receiving process heap as normal lists, the only difference being that a `queueMe`-closure marks the end of the sequence of values received up to now. This closure will be rewritten when a new message for this stream arrives. Thus, sending a stream element updates the inport's closure with a new closure representing a list, the head of which points to the transmitted subheap and the tail of which is a new `queueMe`-closure with an empty list of suspended threads. The address of the new `queueMe`is written into the inport table.

The sending thread continues its computation with the continuation given on top of the return stack. In fact, the continuation is the sending of the remaining list via the channel.

$$
\begin{array}{l}
\langle i \mapsto ((\mathit{Eval}\ (\mathbf{sendHead}\ v)\ \rho)\ \varepsilon\ (\mathit{default} \to \mathit{cont}, \tilde{\rho}) : rs)\ \varepsilon\ (j,c)) \parallel \mathit{threads}_i \\
\hfill h_i\ \sigma_i\ \mathit{ipt}_i\rangle \\
||| \\
\langle j \mapsto \mathit{threads}_j\ \ h_j[a_{in} \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ \mathit{ipt}_j[c \longmapsto (a_{in}, s)]\rangle \\
\hfill \text{with } s \in \{\perp, i\} \\
\Rightarrow \\
\langle i \mapsto ((\mathit{Eval}\ \ \mathit{cont}\ \ \tilde{\rho})\ \ \varepsilon\ \ rs\ \ \varepsilon\ \ (j,c)) \parallel \mathit{threads}_i\ \ h_i\ \ \sigma_i\ \ \mathit{ipt}_i\rangle \\
||| \\
\langle j \mapsto (ts \parallel \mathit{threads}_j)\ \ h'_j\ \ \sigma_j\ \mathit{ipt}_j[c \longmapsto (a_t, i)]\rangle \\
\qquad \text{where } h'_j = \mathit{graph\_copy}(h_j \begin{bmatrix} a_{in} \longmapsto \langle \texttt{Cons}\ \{a_v, a_t\}\rangle \\ a_t\ \ \longmapsto \langle \texttt{queueMe}, [\,]\rangle \end{bmatrix}, a_v, \mathit{subHeap}(h_i, a), a) \\
\qquad\qquad \mathit{Addr}\ a = \mathit{val}\ \sigma\ \rho\ v \\
\qquad\qquad a_v \text{ and } a_t \text{ are new heap addresses}
\end{array}
$$

Closing a channel: The closing of a channel implies the termination of the corresponding thread. On the receiver's site the threads depending on the inport are reactivated, and the `queueMe`-closure is overwritten with a `Nil` closure marking the end of the list associated to the stream.

$$
\begin{array}{l}
\langle i \mapsto ((\mathit{Eval}\ \ \mathbf{closeStrm}\ \ \rho)\ \ \ \varepsilon\ \ \varepsilon\ \ \varepsilon\ \ (j,c)) \parallel \mathit{threads}_i\ \ h_i\ \ \sigma_i\ \ \mathit{ipt}_i\rangle \\
||| \\
\langle j \mapsto \mathit{threads}_j\ \ h_j[a_{in} \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ \mathit{ipt}_j[c \longmapsto (a_{in}, s)]\rangle \\
\Rightarrow \\
\langle i \mapsto \mathit{threads}_i\ \ h_i\ \ \sigma_i\ \ \mathit{ipt}_i\rangle \\
||| \\
\langle j \mapsto (ts \parallel \mathit{threads}_j)\ \ h_j[a_{in} \longmapsto \langle \texttt{Nil}\rangle]\ \sigma_j\ \ \mathit{ipt}_j\rangle
\end{array}
$$

Dynamic reply channels: The creation of a dynamic channel means the creation of a new inport, i.e. the inport table is extended by an entry which contains the address of a newly allocated `queueMe`-closure. As the sender to this inport is not yet known, $\perp$ is noted as sender id. When the first value of a stream is sent, $\perp$ will be overwritten by the actual sender's id. In addition a `ChanName`-closure is included in the heap which contains the machine instance number and the new channel identifier.

$$\begin{array}{l}
\langle i \mapsto ((Eval\ \ \mathbf{new}\ (chn, ch)\ e\ \ \rho)\ \ \ as\ \ rs\ \ us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ \ e\ \ \rho[chn \mapsto Addr\ achn, ch \mapsto Addr\ ach])\ as\ rs\ us\ outp) \parallel \\
\hfill threads h'\ \ \sigma\ \ ipt'\rangle \\
\qquad\qquad \text{where } ipt' = ipt[c \longmapsto (ach, \perp)] \\
\qquad\qquad\qquad h' = h \begin{bmatrix} achn \longmapsto \langle \texttt{ChanName}\ \ (i, c)\rangle \\ ach \ \ \longmapsto \langle \texttt{queueMe}, [\,]\rangle \end{bmatrix} \\
\qquad\qquad\qquad achn \text{ and } ach \text{ are new heap addresses,} \\
\qquad\qquad\qquad c \text{ is a new channel identifier}
\end{array}$$

When using a dynamic channel the current thread is split into two: one will continue with the evaluation of the main expression, and the other will be responsible for the new output.

$$\begin{array}{l}
\langle i \mapsto ((Eval\ \ v\ \texttt{!*}\ \ e_1\ \mathbf{par}\ e_2\ \ \rho)\ \ \ as\ \ rs\ \ us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{with } val\ \ \rho\ \ \sigma\ \ v = Addr\ \ a \text{ and } \ h(a) = \langle \texttt{ChanName}\ \ out_{\text{dyn}}\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ e_1\ \ \rho)\ \ \ \varepsilon\ \ \varepsilon\ \ \varepsilon\ \ out_{\text{dyn}}) \parallel \\
\qquad ((Eval\ e_2\ \ \rho)\ \ \ as\ \ rs\ \ us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle
\end{array}$$

Organizing multithreading

Entering an updatable closure: As explained in Subsection 4.2, we have to prevent multiple threads from simultaneously evaluating the same updatable closure. Therefore, when an updatable closure is encountered, it is overwritten with a `queueMe`-closure.

$$\begin{array}{l}
\langle i \mapsto ((Enter\ a)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\rangle \\
\qquad\qquad \text{with } h(a) = \langle (vs\ \backslash u\ \{\}\ \ \rightarrow\ \ e)\ ws_f\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ \ e\ \ [vs/ws_f])\ \ \ \varepsilon\ \ \varepsilon\ \ \ (as, rs, a) : us\ \ \ outp) \parallel threads\ \ h'\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{where } h' = h[a \longmapsto \langle \texttt{queueMe}, [\,]\rangle]
\end{array}$$

Updating a closure: When the `queueMe`-closure is finally updated, the suspended threads are reactivated. This is the only difference between the following rules and the corresponding sequential ones. Updates are triggered in two cases. Firstly, when a *ReturnCon* instruction is executed with an empty return stack.

$$
\begin{array}{l}
\langle i \mapsto ((ReturnCon\ \ c\ \ ws)\ \ \ \varepsilon\ \ \varepsilon\ \ (as_u, rs_u, a_u) : us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{with } h(a_u) = \langle \texttt{queueMe}, ts\rangle \\
\Rightarrow \\
\langle i \mapsto ((ReturnCon\ c\ ws)\ as_u\ rs_u\ us\ outp) \parallel ts \parallel threads\ h'\ \sigma\ ipt\rangle \\
\qquad\qquad \text{where } h' = h[a_u \longmapsto \langle vs \setminus n\ \{\} \rightarrow\ \ c\ vs)\ \ ws\rangle] \\
\qquad\qquad\qquad\quad vs \text{ is a list of distinct variables with } length(vs) = length(ws)
\end{array}
$$

Secondly, when a closure is entered for which there are not enough addresses on the argument stack (partial application).

$$
\begin{array}{l}
\langle i \mapsto ((Enter\ a)\ \ \varepsilon\ \ \varepsilon\ \ (as_u, rs_u, a_u) : us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{with } h(a_u) = \langle \texttt{queueMe}, ts\rangle \\
\qquad\qquad\qquad h(a) = \langle (vs \setminus n\ xs \rightarrow e)\ \ ws\rangle \\
\qquad\qquad\qquad length(as) < length(xs) \\
\Rightarrow \\
\langle i \mapsto ((Enter\ \ a)\ \ as \mathbin{+\!\!+} as_u\ \ rs_u\ \ us\ \ outp) \parallel ts \parallel threads\ \ h'\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{where } xs_1 \mathbin{+\!\!+} xs_2 = xs \\
\qquad\qquad\qquad\quad length(xs_1) = length(as) \\
\qquad\qquad\qquad\quad f \text{ is an arbitrary variable} \\
\qquad\qquad\qquad\quad h' = h[a_u \longmapsto \langle (f : xs_1) \setminus n\{\ \} \rightarrow f\ xs_1)\ \ a : as\rangle]
\end{array}
$$

Suspending a thread: As has been mentioned earlier, there are the following two cases when a thread has to suspend itself on entering a closure: The closure is already under evaluation by another thread in the same process, or it represents an empty inport. In both cases the closure to be entered is a `queueMe`-closure and the following transition will be performed:

$$
\begin{array}{l}
\langle i \mapsto ((Enter\ \ a)\ \ \ as\ \ rs\ \ us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{with } h(a) = \langle \texttt{queueMe}, ts\rangle \\
\Rightarrow \\
\langle i \mapsto threads\ \ h'\ \ \sigma\ \ ipt\rangle \\
\qquad\qquad \text{where } h' = h[a \longmapsto \langle \texttt{queueMe}, ((Enter\ \ a)\ \ \ as\ \ rs\ \ us\ \ outp) \parallel ts\rangle]
\end{array}
$$

Process instantiation Now that we have presented the mechanisms used for message passing and for the interaction of the threads in the same process, we can explain the creation of processes, which relies on both. Process instantiations only appear as special bindings in local declarations, i.e. let and letrec expressions. Thus, the rule for evaluating these expressions must be extended:

$$\langle i \mapsto \; ((Eval\ \mathbf{let(rec)}\ binds\ \mathbf{in}\ e\ \rho)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ \rangle$$
$$\Rightarrow$$
$$\langle i \mapsto ((Eval\ e\ \rho')\ as\ rs\ us\ outp) \parallel threads \parallel threads'\ h'\ \sigma\ ipt'\rangle \;|||\; newInstances$$
where
$$\rho' = extend\_env(\rho, binds)$$
$$(threads', h', ipt', newInstances) = handle\_bindings(i, binds, \rho', h, \sigma, ipt)$$

The function *extend_env* provides heap addresses for the variables on the left hand side of the bindings:

$$extend\_env(\rho, \{\}) = \rho$$
$$extend\_env(\rho, \{x = lf; bind_2; \ldots; bind_n\})$$
$$= extend\_env(\rho[x \mapsto Addr\ a], \{bind_2; \ldots; bind_n\})$$
where a is a new heap address
$$extend\_env(\rho, \{\{o_1, \ldots, o_m\} = p\#\{i_1 \parallel \ldots \parallel i_k\}; bind_2; \ldots; bind_n\})$$
$$= extend\_env(\rho[o_1 \mapsto Addr\ a_1, \ldots, o_m \mapsto Addr\ a_m], \{bind_2; \ldots; bind_n\})$$
where $a_1, \ldots, a_m$ are new heap addresses

The corresponding heap closures are created within *handle_bindings*. If the bindings contain process instantiations, *handle_bindings* additionally produces new threads, further entries in the inport table and new machine instances.

$$handle\_bindings(i, \{bind_1; \ldots; bind_n\}, \rho, h_0, \sigma, ipt_0)$$
$$= (threads_1 \parallel \ldots \parallel threads_n, h_n, ipt_n, newInst_1 \;|||\; \ldots \;|||\; newInst_n)$$
where for $1 \leq j \leq n$:
$$(threads_j, h_j, ipt_j, newInst_j) = handle\_binding(i, bind_j, \rho, h_{j-1}, \sigma, ipt_{j-1})$$

The transition rule specifies that the executed thread continues with the evaluation of the body expression e in the extended environment. The new threads are run concurrently. The heap and inport table are adjusted accordingly. The new machine instances are executed in parallel.

The auxiliary function *handle_binding* distinguishes two cases. The simple case of binding is the heap allocation of a closure as in the sequential STG machine. Only the heap is modified. In this case the effect of the transition exactly corresponds to the sequential rule. Let $val\ \rho\ \sigma\ x = Addr\ a$. Then:

$$handle\_binding(i, x = vars_f \setminus\pi\ vars_a \rightarrow e_{\text{rhs}}, \rho, h, \sigma, ipt) = (\varepsilon, h', ipt, \varepsilon)$$
where $h' = h[a \longmapsto \langle(vars_f \setminus\pi\ vars_a \rightarrow e_{\text{rhs}})\ (\rho\ vars_f)\rangle]$

A process instantiation leads to the creation of new input channels, new threads for the evaluation of outports and a new DREAM instance for the newly created process.

Let $val\ \rho\ \sigma\ p = Addr\ a$ and $h(a) = \langle (xs\ \backslash\mathbf{process}\ ins \rightarrow e_{\text{body}})\ ws \rangle$. Then:

$$
\begin{aligned}
& handle\_binding(i, os = p\#vs, \rho, h, \sigma, ipt) = (newThs, h', ipt', newInst) \\
& \quad \text{where } (ins, h', ipt') = create\_inports(i, new_i, os, \rho, h, ipt) \\
& \qquad\qquad (new_i, newInst, outs) \\
& \qquad\qquad\qquad = create\_process(\langle (xs\ \backslash\mathbf{process}\ is \rightarrow e_{\text{body}})\ ws \rangle, h_{ws}, i, ins, \sigma) \\
& \qquad\qquad h_{ws} = h_0 \cup subHeap(h, ws) \\
& \qquad\qquad newThs = spawn\_threads(vs, \rho, outs)
\end{aligned}
$$

In the state of the current machine instance, input channels are created by *create_inports* to receive messages from the outports of the new process. The identifiers of these channels together with the parent id are passed as arguments to the function *create_process* that yields a new machine instance for the process to be created. For each inport of the newly created process a new thread is created via *spawn_threads* to evaluate the associated expression.

The heap h_{ws} contains the static closures of the global environment in h_0 and all the relevant closures needed to correctly bind the free variables $x_1, \ldots, x_r$ of the process abstraction as well as all closures reachable from these closures via variable bindings computed via *subHeap*[5]. It is used as the initial heap for the new machine instance.
The further auxiliary functions are defined below.

- *create_inports* creates new input channels. It allocates `queueMe`-closures in the heap and extends the environment by mapping the input variables in_i to the addresses of the new closures. Finally, it extends the inport table by mappings of new channel identifiers to the `queueMe`-closures. The updated state components and the list of the new channel names are returned as result. Let $val\ \rho\ \sigma\ in_j = Addr\ ai_j$ for $1 \leq j \leq n$. Then:

$$
\begin{aligned}
& create\_inports(i_{\text{recv}}, i_{\text{send}}, [in_1, \ldots, in_n], \rho, h, ipt) \\
& \qquad\qquad\qquad\qquad = ([(i_{recv}, c_1), \ldots, (i_{recv}, c_n)], h', ipt') \\
& \quad \text{where } h' = h[ai_1 \longmapsto \langle \texttt{queueMe}, [\,] \rangle, \ldots, ai_n \longmapsto \langle \texttt{queueMe}, [\,] \rangle] \\
& \qquad\qquad ipt' = ipt[c_1 \rightarrow (ai_1, i_{\text{send}}), \ldots, c_n \rightarrow (ai_n, i_{\text{send}})] \\
& \qquad\qquad c_1, \ldots, c_n \text{ are new channel identifiers}
\end{aligned}
$$

- The function *create_process* is used to define a new machine instance with a new identifier and a list of outport specifications. The state of this new machine contains the threads for its outports, the heap passed as a parameter extended by the inports of the process, the global environment and the new inport table. The threads for the outports are spawned using *spawn_threads*. The addresses of the corresponding inports (of the parent process) are passed as a parameter. Inport channels are created using *create_inports* to which the new outports of the parent process will be connected.

$$
\begin{aligned}
& create\_process(\langle (xs\ \backslash\ \mathbf{process}\ is\ \rightarrow\ e_{\text{body}})\ ws \rangle, h_{ws}, i, outs, \sigma) \\
& \quad = (new_i, [new_i \mapsto state_i], ins)
\end{aligned}
$$

[5] Note that this heap must not contain any `queueMe`-closure.

where $\rho_0 = [xs \mapsto ws, i_1 \mapsto Addr\ a_1, \ldots, i_n \mapsto Addr\ a_n]$
$state_i = ((Eval\ e_{\text{body}}\ \rho_0)\ \epsilon\ \epsilon\ \epsilon\ outs)\ h_{in}\ \sigma\ ipt_{in})$
$(ins, h_{in}, ipt_{in}) = create\_inports(new_i, i, is, \rho_0, h_{ws}, [\,])$
new_i is a new process identifier,
$a_1, \ldots, a_n$ are new heap addresses.

– *spawn_threads* takes a parallel expression of the form $\{e_1 \parallel \ldots \parallel e_n\}$, an environment ρ and a list of outport specifications and generates threads which evaluate these expressions in the given environment and send the results via the respective outports.

$spawn\_threads\ (\{e_1 \parallel \ldots \parallel e_k\}, \rho, [out_1, \ldots, out_k]) = t_1 \parallel \ldots \parallel t_k$
where $t_i = ((Eval\ e_i\ \rho)\ \epsilon\ \epsilon\ \epsilon\ out_i),\ \forall i \in \{1, \ldots, k\}$

The evaluation of a process body leads to a parallel expression which defines the threads for the outports to the generator process. The single control thread for the creation of the heap and top level subprocesses terminates. k threads for the evaluation of the outports are started.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ \{e_1 \parallel \ldots \parallel e_k\}\ \rho)\ \ \epsilon\ \epsilon\ \epsilon\ outs) \parallel threads\ h\ \sigma\ ipt\rangle \\
\Rightarrow \\
\langle i \mapsto threads' \parallel threads\ h\ \sigma\ ipt\rangle \\
\qquad \text{where } threads' = spawn\_threads(\{e_1 \parallel \ldots \parallel e_k\}, \rho, outs)
\end{array}
$$

Termination: A machine instance without threads terminates immediately. The threads in other instances which fill its inports are also removed.

$$
\begin{array}{l}
\langle i \mapsto \emptyset\ h\ \sigma\ [c_1 \mapsto (j_1, a_1), \ldots, c_k \mapsto (j_k, a_k)]\rangle \\
|||\,_{j=1}^{k}\ \langle j \mapsto (instr_j\ as_j\ rs_j\ us_j\ (i, c_j)) \parallel threads_j\ h_j\ \sigma_j\ ipt_j\rangle \\
\Rightarrow \\
|||\,_{j=1}^{k}\ \langle j \mapsto threads_j\ h_j\ \sigma_j\ ipt_j\rangle
\end{array}
$$

This completes the specification of the DREAM machine.

5 Imperative DREAM

In the current section we will discuss the compilation of PEARL, which is a step further to the parallel implementation.

The target code is a set of low level machine instructions for DREAM. The operational semantics of the instructions is given in terms of transitions over the DREAM's state. The lambda forms in the code component of closures are replaced by code pointers. The instructions can be easily defined in an imperative language and we distinguish three kinds of instructions:

Thread level instructions only change the local state of the executing thread without affecting the rest of the Dream or the global DREAM. The instructions here are derived from the compilation of the sequential STGM, except the operations on the heap, which belong to the *process level*, because the threads of one process share a single heap.

Process level instructions modify the thread set, the heap or the channel table of a Dream. They may also have a local effect on the state of the executing thread.

$a \leftarrow$ `allocateClosure` n	Allocates a record of length n + length(code ptr) from the heap and returns its address a
`WriteClosure` a cd ws	Overwrites allocated closure a with code pointer cd and bindings ws
$c \leftarrow$ `CreateChannel` a	Associates a new input channel to closure a and returns its name c
`CreateThread` cd out	New thread executing code cd with outport out
`SuspendThread`	Suspends current thread

System level instructions affect the state of the whole DREAM by creating new Dreams or sending messages from one Dream to another.

$$id_{new} \leftarrow \texttt{CreateProcess}\; id_{fath}\; a_p\; [c_1, \ldots, c_k]$$

It is given a process abstraction binding a_p (i.e. the heap address of the process abstraction closure), the identity of the executing Dream (id_{fath}), and the identity of the receiving channels in this machine ($[c_1, \ldots, c_k]$). Then it creates a remote Dream returning the identity id_{new} of the new Dream. This instruction just creates a most simple initial state for the new Dream, which will immediately enter the code associated to the process abstraction being instantiated. This code is responsible for creating the corresponding inports, spawning the output threads and executing the process.

We do not present the complete set of instructions here, but only a subset that is used in the following compilation example. Likewise, we do not give the whole set of compilation schemes in this section. The interested reader is referred to [2]. As an illustrative example, we will discuss here the compilation of process instantiations. We make use of two auxiliary functions *val* which is analogous to *val* ρ σ in Section 4 and *acode* which returns the code pointer for a given closure.

Compilation of process instantiation As in PEARL process instantiations may only appear in local bindings, we show their translation in this context.

A **let/letrec** compilation scheme (see Figure 2) defines the processing of its bindings in sequence and the evaluation of the body expression e. An abstraction binding — either a lambda abstraction binding or a process abstraction one — gives rise to a new closure in the heap. As in the interpretative version, we allocate the closures and bind all the variables before processing the right hand sides to be able to deal with recursive bindings.

$$
tr \left(\begin{array}{l} \textbf{letrec} \\ \quad \{o_1, \ldots, o_k\} = \\ \qquad\quad p \,\#\, \{v_1 \parallel \ldots \parallel v_m\} \\ \textbf{in} \;\; e \end{array} \right) \stackrel{\text{def}}{=} \begin{array}[t]{l} ao_1 \leftarrow \texttt{allocateClosure}\ 1; \\ \ldots \\ ao_k \leftarrow \texttt{allocateClosure}\ 1; \\ \rho \leftarrow \rho[\ldots o_1/ao_1 \ldots o_k/ao_k \ldots]; \\ \texttt{WriteClosure}\ ao_1\ pcd_{qm}\ q_\epsilon; \\ c_1 \leftarrow \texttt{CreateChannel}\ ao_1; \\ \ldots \\ \texttt{WriteClosure}\ ao_k\ pcd_{qm}\ q_\epsilon; \\ c_k \leftarrow \texttt{CreateChannel}\ ao_k; \\ id \leftarrow \texttt{CreateProcess}\ id_{own}\ (val\ p)\ [c_1, \ldots, c_k]; \\ \texttt{CreateThread}\ (acode\ (val\ v_1))\ (id, 1); \\ \ldots \\ \texttt{CreateThread}\ (acode\ (val\ v_m))\ (id, m); \\ tr\ (e) \end{array}
$$

$$
pcd_{qm} \stackrel{\text{def}}{=} \begin{array}[t]{l} \texttt{SuspendThread}; \\ \texttt{Call}\ cc\ [\,] \end{array}
$$

Fig. 2. Compiling PEARL

A process instantiation gives rise to as many closures as outports are bound. First of all input channels are created (`CreateChannel`) as well as `queueMe`-closures to receive the outports of the instantiated process (created by `Create-Process`, where id_{own} is the identity of the executing Dream). Finally, it creates new threads (`CreateThread`) to evaluate the inport expressions. It is assumed that the receiving channels of these inport expressions will be the first m input channels of the newly created process.

Each `queueMe` closure is created with code pointer pcd_{qm} and an empty queue q_ϵ for suspended threads. The precompiled code pcd_{qm} just suspends the thread entering it (`SuspendThread`). The thread is only reactivated when the closure has been updated. Thus, the current closure is reentered (`Call` cc [ ], where [ ] is the empty argument list.).

6 Related Work

In the last decade parallel functional programming has been an active field of research. Hammond [8] provides a historical overview of the languages developed and discusses the key ideas that underly their implementations.

Most of the parallel abstract machines for general purpose distributed memory architectures are based on a low level "fork and wait" mechanism, which supports coarse grained tasks. These are for example GUM ([12]), the STGM_{MT} ([6]), another multi threaded STGM by Horn ([9]) and the PABC ([10]). All of them are extensions of sequential abstract machines and the first three of them use the STG machine as their basis, like DREAM.

The common idea of these approaches is to annotate (mostly by the programmer) expressions that are worthwhile to be executed remotely. These expressions will be "sparked" (i.e. their heap addresses are put into a work pool, as in PABC, STGM_{MT} and GUM), or new threads are created to evaluate these expressions and added to the thread or task list as done by Horn. Sparked expressions and threads can be either evaluated locally or remotely, depending on the load balancing algorithm and the current work load.

In order not to duplicate work, the graph of the expression to be evaluated (or only the top node, or only a part of it) is moved to the remote processor element and the local heap nodes are overwritten with indirections to their new remote locations. When there is demand for such an indirection, the remote data is either requested, or the evaluation is only suspended until the data is delivered automatically. The latter can only be performed for the expressions that are evaluated in a thread and sent back to their original location. While waiting for the data, the evaluation is continued with a reactivated thread or a new spark.

In all approaches, a thread does not know where an expression it is evaluating will be needed, hence the result is only sent on request. In Eden, the situation is different, because due to one-to-one channels it is known which data is needed on which processor and this information can be used to automatically send the data to the respective processor. We expect this to reduce communication costs considerably.

Another advantage of DREAM is that it always sends normal forms[6], whereas the other machines evaluate the expressions at most to head normal form before they are sent, possibly resulting in closures being sent back and forth. In a Dream there are no heap nodes that are only referenced externally: remotely demanded expressions are under evaluation and hence referenced from a local thread. So we do not need an expensive distributed garbage collection, as the other approaches do.

The approaches also differ in the scheduling behaviour. Only DREAM and the PABC have a fair scheduler, while the "multi threaded" approaches and GUM only have threads that are only suspended when waiting for data.

7 Conclusions and Future Work

In this paper we have developed an extension of the STG language called PEARL which handles processes explicitly. PEARL's operational semantics is defined as an conservative extension of the STGM and its transition system. Our parallel abstract machine is a sequence of Dreams, which are multithreaded STGM's, communicating via message passing. Each Eden process is evaluated on a separate Dream. Each thread has its own stacks and current instruction, but it shares the heap and global environment with the other threads belonging to the same process.

[6] except for the free variables of process abstractions, which are copied at process creation time

The communication channels are implemented by an inport table in each process and an outport for each thread. For simplicity the communication is modelled synchronously here, but in the low level implementation it will be asynchronous and we will have to deal with bounded message buffers, which will force us to control the evaluation and sending of outport values.

We have given an overview of the next step towards an implementation of Eden, namely the compilation of PEARL into code which is imperative and more low-level. What remains for future work on this machine is the treatment of channel structures, non-deterministic merge and automatic bypassing of channels.

References

1. S. Breitinger, U. Klusik, and R. Loogen. An Implementation of Eden on Top of Concurrent Haskell. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.
2. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. Imperative DREAM. Internal Report, 1997.
3. S. Breitinger and R. Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Funct. Prg.*, 1997.
4. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.
5. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS).* IEEE Press, 1997.
6. M. M. Chakravarty. Integrating Multithreading into the Spineless Tagless G-machine. In D. N. Turner, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer Verlag, 1995.
7. M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, 1994.
8. K. Hammond. Parallel Functional Programming: an Introduction. In *PASCO, Linz, Austria*. World Scientific, 1994.
9. M. Horn. A Different Integration of Multithreading into the STGM. In W. Kluge, editor, *Impl. of Funct. Lang.* Christian-Albrechts-University Kiel, 1996.
10. M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Katholieke Universiteit Nijmegen, 1996.
11. S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. of Funct. Prog.*, 2(2), 1992.
12. P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. P. Jones. GUM: A portable parallel implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.