Bypassing of Channels in Eden[⋆]

Ulrike Klusik[1], Ricardo Peña[2], and Clara Segura[2]

[1] Philipps–Universität Marburg, D-35032 Marburg, Germany
Phone# +49-6421-281521 ; Fax# +49-6421-285419
e-mail: `klusik@Mathematik.Uni-Marburg.de`
[2] Universidad Complutense de Madrid, E-28040 Madrid, Spain
Phone# +34-91-3944313; Fax# +34-91-3944602
e-mail: {`ricardop,csegura`}`@eucmax.sim.ucm.es`

Abstract

We describe *automatic bypassing*, an optimization of Eden's implementation to reduce the number of messages and/or threads at runtime. Eden [BLOP97] extends the lazy functional language Haskell with a set of *coordination* features, aimed to express parallel algorithms. These include *process abstractions* (or process schemes) and *process instantiations* (or applications of a process scheme to actual inputs).

When a new process is instantiated, their input and output channels are connected to its parent process. This implies that, in principle, only tree–like process topologies can be created. But the aimed topology may not be tree–like (e.g. pipelines, grids, etc.). It is desirable to be able to connect every producer to its actual consumer, trying to avoid the intermediate processes frequently used only to set up the topology.

The strategy consists of a combination of compile time analysis and runtime support. Both are explained in detail. Also, the savings expected with the proposed strategy are commented.

[⋆]Work partially supported by the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

1 INTRODUCTION

The parallel functional programming language Eden [BLOP97, KOP98] extends the lazy functional language Haskell by syntactic constructs to explicitly de£ne processes. Eden is implemented by modifying the Glasgow Haskell compiler GHC [PHH$^+$93] and its parallel runtime sytem[1] GUM [THMP96]. In [BKL98], more details are given.

Processes in Eden are dynamically instantiated while executing recursive functions and/or process de£nitions. When a new process is created, its channels are connected to its parent process. The parent is responsible for feeding child's input channels with values and for receiving values from child's output channels. This implies that, in principle, only tree-like process topologies can be created, see Figure 1 (left). Frequently, it is the case that the parent process just copies the values received from one child to an input channel of another child that, in turn, forwards these values to another process, and so on. It is a desirable optimization to detect this situation to be able to eliminate intermediate processes in the transmissions, i.e. to directly connect producers to consumers, see Figure 1 (right). This would save a lot of messages and much useless computations at runtime. We call the optimization *automatic bypassing*, which consists of a combination of compile time analysis and runtime support.

The organization of the paper is as follows: in Section 2, we introduce the Eden features relevant to bypassing and explain the bypassing problem in detail. In Section 3, CoreEden abstract syntax, before and after annotations, is presented. Section 4 formally de£nes the compile time analysis and applies it to a simple example. Section 5 explains the bypassing protocol and the support given to it by the RTS. Finally, Section 6 gives an account of the expected savings and summarizes the state of the implementation.

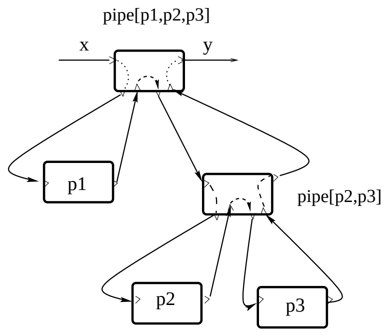
2 EDEN AND BYPASSING

Functional languages distinguish between function de£nitions and function applications. Much in the same spirit, Eden offers *process abstractions*, i.e. abstract schemes for process behaviour, and *process instantiations* for the actual creation of processes. Process abstractions have a polymorphic type `Process a b` and can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. A process abstraction mapping input variables $x_1, \ldots, x_n$ to output expressions $exp_1, \ldots, exp_k$ can be speci£ed by the following expression:

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

The output expressions can reference the input variables, as well as the auxiliary functions and common subexpressions de£ned in the optional `where` part.

[1]In what follows we will write RTS as a shorthand for runtime system.



```
pipe::[Process a a] -> Process a a
pipe [p] = p
pipe (p:ps) =
     process z -> pipe ps # (p # z)
y = pipe [p1,p2,p3] # x
```

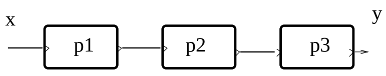


FIGURE 1: Created and Desired Topology for a Pipeline

A *process instantiation* is achieved by using the prede£ned in£x operator `(#)` [2] in the following way: $(y_1, \ldots, y_k) = p \# (exp_1, \ldots, exp_n)$. We will refer to the new instantiated process as the *child* process, while the process where the instantiation takes place will be 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 $exp_1, \ldots, exp_n$. In both cases, value production is done eagerly. Communication is unidirectional, from one producer to exactly one consumer. Only fully evaluated data objects are communicated. Lists are transmitted in a *stream*-like fashion, i.e. element by element. To illustrate the bypassing problem, in Figure 1 the de£nition and the instantiation of a pipeline appear.
Eden's current implementation [KOP98] unfolds at runtime the recursive de£nition and generates the process topology of Figure 1 (left). It can be seen that this structure is far from desirable (see Figure 1 (right)) as processes are created not only for `p1`, `p2` and `p3`, but also for `pipe[p1,p2,p3]` and `pipe[p2,p3]`, which only forward data from their input to their children and from them to their output. Channels are connected by following the unfolding of the recursive de£nition.

Automatic bypassing will deal with such situations in order to connect producer to consumer directly, as in Figure 1 (right). Nevertheless, with our approach the additional processes will still be instantiated, but they will terminate after creating their children processes as they neither produce nor consume data.

This example illustrates two of the three possible kinds of bypassing: *bypassing between siblings* (e.g. `p2` and `p3`) and *bypassing between generations* (e.g. `pipe[p1,p2,p3]` and `p2`). There is a third kind, called *bypassing between ancestors*, in which an input channel value is directly copied to exactly one output channel. The analysis will detect the three cases.

[2]`(#)::Process a b -> a -> b`

3 COREEDEN AND ANNOTATED COREEDEN

Currently in Core[3], process abstractions and instantiations are hidden inside pre-de£ned functions. In order to do bypassing analysis we need to make input and output channels explicit. So, we introduce a new intermediate language called CoreEden, which is an extension of Core. The basic extensions are the following:

$$
\begin{array}{l}
binds \rightarrow \mathbf{recpar}\ bind'_1; \ldots ; bind'_n\ [\mathbf{bypass}\ channels] \\
bind' \rightarrow var = exp \\
\qquad\qquad \mid\ channels = var\ \texttt{\#}\ channels \\
channels \rightarrow \{var_1, \ldots, var_n\} \\
exp \rightarrow \mathbf{process}\ channels \rightarrow body\ [\mathbf{bypass}\ channels] \\
body\ \rightarrow\ [\mathbf{let}\ binds\ \mathbf{in}]\ channels
\end{array}
$$

The idea is £rst to translate from Core to CoreEden; then to do the bypassing analysis, producing an annotated CoreEden program[4] and £nally to translate back from CoreEden to Core to go on with the rest of the compilation.

4 BYPASSING ANALYSIS

Through the bypassing analysis we want to detect those situations in which a variable is used exactly once as input channel and once as output channel (from the parent's point of view) and is not used in any other expression. This problem is similar to a usage analysis [Ses91], [LGH$^+$92].

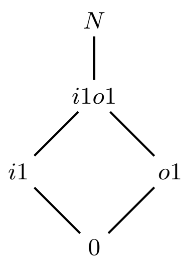
We use an abstract bypassing domain $B^{\#}$, shown in Figure 2, where $i1o1$ represents that the channel is used only once as input and only once as output. In such a case the channel will be bypassed. The value N represents the fact that the channel's value is used too many times, for example twice as input or as a free variable. We de£ne a commutative $+$ operation over bypassing values, shown in Figure 2, which extends easily to environments. We also need a $\bigplus$ operation over sets of environments due to the fact that a variable may appear more than once as input channel and then bypassing is not possible.

The analysis uses three main functions to analyse expressions (A_{exp}), bindings (A_{binds}) and process bodies ($A_{bindsbody}$) shown in Figure 3[5]. The function A_{exp} receives a CoreEden expression e and decorates it with bypassing information, which in the end is attached to process abstraction expressions and $\mathbf{recpar}$ bindings in e. Such information is local to the process abstraction, so we analyse its body with a bypassing environment carrying all the information about input and output channels. In a $\mathbf{let}\ binds\ \mathbf{in}\ e$ expression we analyse e and then the bindings, taking into account that the free variables in e cannot be bypassed (so

[3]Core is a minimal functional language to which Haskell is translated in GHC's £rst steps.

[4]The analysis decorates process abstractions (bypassing between generations and between ancestors) and $\mathbf{recpar}$ bindings (bypassing between siblings) with **bypass** clauses.

[5]We also use the functions: $cs \cap\!\!\!\cdot\, ds = \{x \mid x \in cs \wedge \exists! y \in ds. y = x\}$ and $(f \wedge\!\!\cdot\, g)\ x = (f\ x) \wedge (g\ x)$. The £rst one is used to detect bypassing between ancestors.



$$
\begin{aligned}
&N + x = N\\
&0 + x = x\\
&i1o1 + x = N \ (x \neq 0)\\
&i1 + o1 = i1o1\\
&i1 + i1 = N\\
&o1 + o1 = N
\end{aligned}
$$

FIGURE 2: Bypassing Domain

we set all the free variables to N in the environment used to analyse the bindings). The rest of the cases are trivial recursive calls to A_{exp}.

The functions A_{binds} and $A_{bindsbody}$ combine the information coming from each individual binding (see Figure 3). In a binding of the form $var = e$ and in a **let** $binds$ **in** e expression, those variables free in e cannot be bypassed, so they must be set to a bypassing value of N. The de£nition of *freevars* is the usual one extended as expected for the new constructions.

Applying this algorithm to the (translated to CoreEden) example of Section 2, we would obtain the following annotations for the binding of $pipe$:

```
$pipe = \lambda\, ps \rightarrow$ case $ps$ of
          $[p] \rightarrow p$
          $p : pp \rightarrow$ process $z \rightarrow$ let recpar $inter = p \# z$
                                                $p' = pipe\ pp$
                                                $y = p' \# inter$
                                                bypass $inter$
                                         in $y$ bypass $z, y$
```

As there are no £xpoint computations, the ef£ciency of this analysis is linear with respect to code's size.

5 THE BYPASSING PROTOCOL

In what follows, we will call a connection from a real ouport[6] (also called the *producer*) to a real inport (also called the *consumer*) through a sequence of intermediate local forwards a *forward chain*. At process creation time, we must introduce these £nal ports to each other.

5.1 The protocol without bypassing

The original communication protocol has been described in [KOP98]. Here, we only present the process creation part, which uses the following messages:

[6]An inport (respectively an outport) is the implementation view of an input (respectively an output) of a process instantiation.

```
$A_{exp} :: Exp \rightarrow DExp$
$A_{exp}\ (\mathbf{let}\ binds\ \mathbf{in}\ e) = \mathbf{let}\ (A_{binds}\ binds\ (freevars\ e))\ \mathbf{in}\ (A_{exp}\ e)$
$A_{exp}\ (\mathbf{process}\ cs_1 \rightarrow cs_2) =$
    $\mathbf{process}\ cs_1 \rightarrow cs_2\ (\texttt{if}\ cs_1 \cap cs_2 = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ (cs_1 \cap cs_2))$
$A_{exp}\ (\mathbf{process}\ cs_1 \rightarrow \mathbf{let}\ binds\ \mathbf{in}\ cs_2) =$
    $\mathbf{process}\ cs_1 \rightarrow \mathbf{let}\ binds'\ \mathbf{in}\ cs_2\ byp$
     $\texttt{where}\ \rho_0 = \{c \mapsto o1 \mid c \leftarrow cs_1\} + {\Large +}\{\{d \mapsto i1\} \mid d \leftarrow cs_2\}$
           $(binds', \rho') = A_{bindsbody}\ binds\ \rho_0\ (cs_1 \cup cs_2)$
           $b = filter\ (== i1o1)\ (map\ \rho'\ (cs_1 \cup cs_2))$
           $byp = \texttt{if}\ b = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ b$
$A_{binds} :: Binds \rightarrow Free \rightarrow DBinds$
$A_{binds}\ binds\ free = fst\ (A_{propagate}\ binds\ \{x \mapsto N \mid x \leftarrow free\}\ \emptyset)$

$A_{bindsbody} :: Binds \rightarrow Env \rightarrow (DBinds, Env)$
$A_{bindsbody}\ binds\ \rho\ cs = A_{propagate}\ binds\ \rho\ cs$

$A_{propagate} :: Binds \rightarrow Env \rightarrow Channels \rightarrow (DBinds, Env)$
$A_{propagate}\ (v = e)\ \rho\ cs = (bind', \rho + \rho')$
    $\texttt{where}\ (bind', \rho') = A_{onebind}\ (v = e)$
$A_{propagate}\ (\mathbf{rec}\ binds)\ \rho\ cs = (\mathbf{rec}\ binds', \rho + \rho')$
    $\texttt{where}\ (binds', \rho') = A_{listbinds}\ binds$
$A_{propagate}\ (\mathbf{recpar}\ binds)\ \rho\ cs =$
   $(\mathbf{recpar}\ binds'\ byp, \rho'')$
    $\texttt{where}\ (binds', \rho') = A_{listbinds}\ binds$
          $\rho'' = \rho + \rho'$
          $b = filter\ ((== i1o1 \cdot get) \wedge (not \cdot in\ cs))\ (dom\ \rho'')$
          $byp = \texttt{if}\ b = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ b$

$A_{listbinds} :: [Bind] \rightarrow ([DBind], Env)$
$A_{listbinds}\ (bind : binds) = (bind' : binds', \rho' + \rho'')$
    $\texttt{where}\ (bind', \rho') = A_{onebind}\ bind$
          $(binds', \rho'') = A_{listbinds}\ binds$
$A_{listbinds}\ [] = ([], \emptyset)$

$A_{onebind} :: Bind \rightarrow (DBind, Env)$
$A_{onebind}\ (v = e) =$
   $(v = e', \{v \mapsto N\} + \{x \mapsto N \mid x \leftarrow freevars\ e\})$
    $\texttt{where}\ e' = A_{exp}\ e$
$A_{onebind}\ (cs_1 = p\ \#\ cs_2) = (cs_1 = p\ \#\ cs_2, \rho')$
  $\texttt{where}\ \rho' = \{c \mapsto o1 \mid c \leftarrow cs_1\} + {\Large +}\{\{d \mapsto i1\} \mid d \leftarrow cs_2\} + \{p \mapsto N\}$
```

FIGURE 3: The Analysis Functions

- *CREATE-PROCESS(pabs,ins$_p$,outs$_p$)*: Initiates the creation of a child process in a PE by using the process abstraction *pabs* and the parent's inports and outports, respectively *ins$_p$* and *outs$_p$*, which will be connected to the child.
- *ACK(outs$_c$ $\rightarrow$ ins$_p$,outs$_p$ $\rightarrow$ ins$_c$)*: Acknowledges the creation of the child process to the parent, including the connections between parent and child.

While the child immediately starts sending values to the parent, the parent must wait for the *ACK* message in order to know to which child ports the values should be sent. These are sent by using a *SENDVAL(in,value)* message.

5.2 New RTS components: Forward handles and forward table

To represent each local forward we use a unique identi£er within a processing element (PE), called *forward handle*: `type HForward = (PE, Int, Level)`. The third component is the level of the process in the process creation tree to which the forward handle belongs. It is an additional information whose utility will be explained later. Forward handles can take the place of inports and outports in the messages of the bypassing protocol. To store intermediate connection information we add to the RTS a new runtime table called the *forward* table.

5.3 The revised protocol

5.3.1 *The messages*

In contrast to what happened in the old *CREATE-PROCESS* message, the ports now do not necessarily belong to the parent itself but forward handles are also allowed. Apart from the *CREATE-PROCESS* message, there are three new bypassing messages:

- *CONSUMER(out $\rightarrow$ in)*: Tells the producer which is the real consumer of the chain. Always, *in* is a real inport, whereas *out* may be an outport or a handle.
- *PRODUCER(out $\rightarrow$ in)*: Tells the consumer which is the real producer of the chain. Always, *out* is a real outport, whereas *in* may be an inport or a handle.
- *LOOP(out $\rightarrow$ in)*: In this case, both *out* and *in* are descendant forwards. It is used in special cases in which an ancestor forward is involved in the chain. This is explained below.

The old *ACK* message can be described by the new bypassing messages, so it is logically not needed anymore.

5.3.2 *Protocol Scheme*

Phase 1: modi£ed process creation In the presence of a descendant forward, some inports and/or outports will not be created. In the *CREATE-PROCESS* message, forward handles replace such absent ports. This happens, for example, in the second and third *CREATE-PROCESS* messages of Figure 4, where we

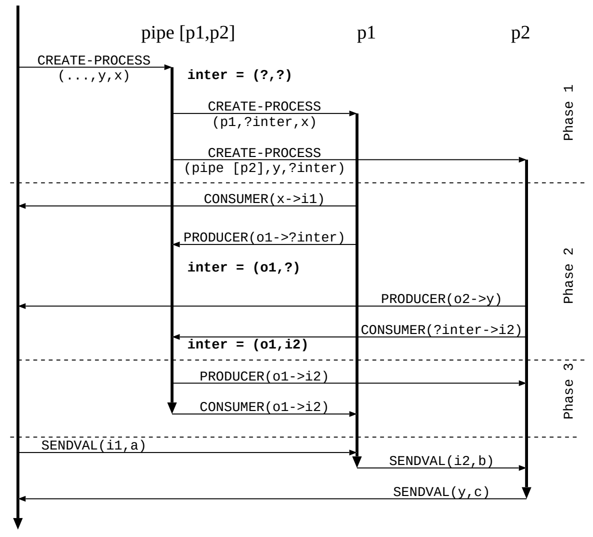


FIGURE 4: Protocol with Bypassing for a two Stages Pipeline

denote forward handles by pre£xing their names with a ? symbol. If some connection information is already known, it is sent instead of the local forward handle. This is the case with inter-generational forwards. In Figure 4, the real outport `x` is propagated downwards through the descending forward of `pipe[p1,p2]`. Ancestor forwards are not involved in the process instantiation on the parent side. After all processes have been created, we have the following situation: all inter-generational forwards have been removed. Each process at the end of a chain has received as opposite end either a descendant forward or a real port of an ancestor process.

Phase 2: sending bypassing messages upwards When a child process is the real producer/consumer of a chain, it must inform to the other side of the chain. This is done by a *CONSUMER*/*PRODUCER* message. If an outport already knows its consumer, it can directly start sending data. In Figure 4 these situations are re¤ected by the four upwards messages. In this phase we also take care of the ancestor forwards. When at least one of the ends is a real port, we pass the information to the other end by using a *CONSUMER*/*PRODUCER* message. A special case arises when both ends are connected to descendant forwards. This is separately explained below.

Phase 3: £nal connection When a descendant forward has received the messages from the real consumer and the real producer of the chain, it introduces them to each other by also sending a *CONSUMER* and a *PRODUCER* message. In Figure 4 this situation is re¤ected by the next two downwards messages.

5.3.3 The special case: Loop

A *loop* consists of an ancestor forward connecting two descendant forwards as illustrated in Figure 5. For this special case a *LOOP* message is sent to the descendant forward in the lower process. The upper forward handle will eventually be informed by the lower one. This is the reason why we need the level information in forward handles. After receiving the *LOOP* there may be a fusion of forward handles according to the class of the receiver's forward.

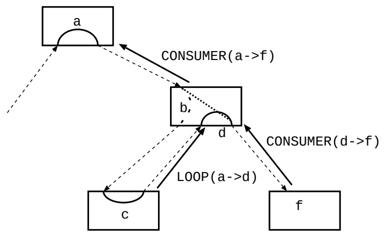


FIGURE 5: Special Case: Loop

6 COMMUNICATION COSTS AND CONCLUSIONS

We said before that the old *ACK* message was not needed anymore, but in the real implementation it is still used to collect several bypassing messages into one message to the parent. Then, if the program contains no forwards, the messages in the old and in the revised protocol are identical, i.e. the creation of tree topologies costs exactly the same. For each descendant forward in the chain, two messages are needed in the second phase and two additional messages in the £nal connection phase. In chains not containing descendant forwards, only one message is needed from the lower to the upper end. The expected savings of bypassing come from the fact that the number of data messages is usually much greater than the number of protocol messages needed to create the direct connection. This is because, when programming in Eden, many of the channels are lists whose transmission implies as many data messages as list elements. For a forward chain of length $n, n \geq 2$, a total of $n-1$ messages are saved for every data message sent through the direct connection. On the other hand, the number of additional messages of the new protocol, with respect to the old one, increases linearly with the length of the chain. Even in the case of one data value the savings are clear when the length of the forward chain is greater than 4. For example, in a pipeline with n processes and m data values passing through it, the saved messages are $3m(n-1) - (3n-2)$ (where $3m(n-1)$ are data messages and $3n-2$ are the additional protocol messages). So, if $n = 2$ then the optimization is worth if at least 2 data values are passed. The savings must also take into account the overheads of the threads not created in all the intermediate processors. In the previous example, the number of saved threads is $3(n-1)$.

The current state of the implementation is as follows: the RTS has already been modi£ed for the new protocol and it is under debugging. The compile time analysis and the many transformations involved to produce a 'good' CoreEden are still being implemented.

REFERENCES

[BKL98] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *Programming Languages: Implementations, Logics, and Programs, PLILP'98*, pages 318–334. Springer LNCS 1490, Pisa, September 1998.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, Geneva, April 1997.

[KOP98] U. Klusik, Y. Ortega Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98*, pages 1–16, London, September 1998. Springer-Verlag, LNCS 1595.

[LGH$^+$92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In *Glasgow Functional Programming Workshop*. Springer-Verlag, 1992.

[PHH$^+$93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Proceedings of Joint Framework for Information Technology, Keele, DTI/SERC*, pages 249–257, March 1993.

[Ses91] P. Sestoft. *Analysis and Ef£cient Implementation of Functional Programs*. PhD thesis, DIKU, October 1991.

[THMP96] P. W. Trinder, K. Hammond, J. S. Mattson Jr., and A. S. Partridge. GUM: A Portable Parallel Implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, USA*, pages 259–280. ACM Press, May 1996.