Chapter 2

Optimizing Eden by Transformation

Cristóbal Pareja, Ricardo Peña, Fernando Rubio, Clara Segura[1]

Abstract: Eden is a parallel extension of Haskell allowing the programmer to explicitly specify which expressions must be evaluated as parallel processes. Eden is implemented by modifying the *Glasgow Haskell Compiler* (GHC). This decision has saved a lot of work but has also produced some drawbacks: Some optimizing transformations done by GHC are not convenient for Eden, either because they spoil its semantics or because they negatively affect its efficiency. The paper explains how to circumvent these drawbacks and also how to add our own optimizing analysis and transformation steps in order to generate a (correct and) better parallel code.

2.1 INTRODUCTION

The parallel-functional language Eden [BLOP96] extends the lazy functional language Haskell by syntactic constructs to explicitly define and communicate processes. The three main new concepts are *process abstractions*, *process instantiations* and the non-deterministic predefined process abstraction `merge`. They are explained in Section 2.2. Eden has been implemented by modifying the *Glasgow Haskell Compiler* (GHC) [JHH$^+$93] front and back-ends [BKL98]. GHC's *modus operandi* is compiling by transformation. It translates Haskell into a minimal language called Core where a lot of optimizations [San95, JS98] are performed.

In Eden's compiler, process abstractions and instantiations are hidden in Core inside predefined functions, and Core to Core optimizations are disallowed because some of them can spoil Eden's semantics (see [PS00a, PS00b] for more details). This situation is clearly undesirable. In this paper, we propose an extension

[1]Dpto. Sistemas Informáticos y Programación, Facultad CC. Matemáticas, Universidad Complutense de Madrid, Avda. Complutense s/n, 28040 Madrid, Spain; Email: `{cpareja,ricardo,fernando,csegura}@sip.ucm.es`

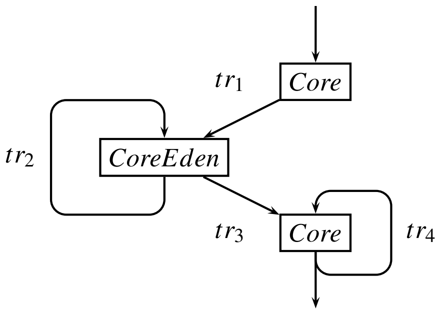

FIGURE 2.1. New transformation scheme

$$
\begin{array}{lll}
\textit{program} & \rightarrow & \textit{binds}_1, \ldots, \textit{binds}_n \\
\textit{binds} & \rightarrow & \ldots \text{ Core bindings } \ldots \\
 & | & \textbf{recpar } \textit{bind}'_1; \ldots; \textit{bind}'_n \\
 & & [\textbf{bypass } \textit{channels}] \\
\textit{bind}' & \rightarrow & v = e \\
 & | & \textit{channels} = v \ \#\# \ \textit{channels} \\
\textit{channels} & \rightarrow & \{v_1, \ldots, v_n\} \\
e & \rightarrow & \ldots \text{ Core expressions } \ldots \\
 & | & \textbf{process } \textit{channels} \rightarrow \textit{body} \\
 & & [\textbf{bypass } \textit{channels}] \\
\textit{body} & \rightarrow & [\textbf{let } \textit{binds } \textbf{in}] \ \textit{channels}
\end{array}
$$

FIGURE 2.2. CoreEden syntax

of the GHC transformation scheme in order to reach the following two objectives:

1. To selectively disallow the potentially dangerous transformations in situations in which they could alter Eden's semantics, but to keep most of them most of the time in order to get an optimized sequential code. They can affect the non-determinism degree of some expressions and the number of instantiated processes (see Section 2.6).

2. To allow useful analyses at Core level to optimize the parallel behaviour of our programs. For this purpose, we extend GHC's Core language into our own *CoreEden* language to make explicitly appear process abstractions, process instantiations, and even their individual channels.

The analyses currently being implemented are *bypassing analysis* and *non determinism analysis*. The first one [KPS00] detects unnecessary threads which simply copy information from an input channel to an output one. This analysis produces annotations attached to process abstractions and process instantiations so that the creation of these threads is avoided. Messages are propagated directly from the producer thread to the consumer process saving much overhead and communication. The second analysis [PS00c] detects those expressions that are sure to be deterministic and those ones which may be non-deterministic, and produces type annotations with this information. Annotations are used to disallow some GHC transformations that are semantically correct in a deterministic environment, but not in presence of non-determinism.

In Figure 2.1, the general view of the new transformation process is shown. There, we call tr_1 to a simple translation from Core to CoreEden that makes explicit process abstractions and process instantiations. Then tr_2 is a set of more complex transformations whose aim is to do a better bypassing analysis. Non-determinism analysis and its annotations are also carried out at this point. Then, tr_3 translates back from CoreEden to Core, and tr_4 consists of the (not disabled) Core to Core transformations currently being done by GHC. After this point, the normal GHC compilation can proceed without alteration. Translation tr_3 is defined in such a way that the (not disabled) transformations and the code generation done by GHC cannot alter Eden's semantics.

The plan of the paper is as follows: In Section 2.2 we give a brief overview of Eden and CoreEden languages. Section 2.3 explains translation tr_1 from Core to CoreEden. Sections 2.4 and 2.5 are the kernel of the paper. They contain the transformations, analyses and annotations produced at CoreEden level (transformation tr_2) and how Eden's semantics is embodied into Core (translation tr_3). Finally, Section 2.6 identifies the dangerous GHC transformations and the situations in which they must be disallowed. The paper ends with a short conclusion.

2.2 EDEN AND COREEDEN OVERVIEW

Eden As it has been said, Eden extends the lazy functional language Haskell by new constructs. There exists a new expression `process x -> e` of a predefined type `Process a b` to define a *process abstraction* having variable `x::a` as input and expression `e::b` as output. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. When the input (resp. output) of a process is a tuple, we will refer to each tuple element as a *channel*. A *process instantiation* is achieved by using the predefined infix operator `(#)::Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, a new process is created. We will refer to the latter as the *child* process, and to the process enclosing the instantiation expression as the *parent* process. Process instantiations are evaluated at runtime so that, in general, the number of processes cannot be determined at compile time. The instantiation protocol deserves some attention in order to understand Eden's semantics:

- Closure `e1` together with all its dependent closures are *copied* to a new processor and the child process is created there to evaluate them. This strategy can lead to some duplication of work.

- Once created, the child process starts producing eagerly its output expression. When this expression is a tuple, a separate concurrent thread is created for the evaluation of each channel. In general, a process is implemented by several threads concurrently running in the same processor.

- Expression `e2` is eagerly evaluated in the parent process. If it is a tuple, an independent concurrent thread is created to evaluate each component.

Once a process is running, only fully evaluated data objects are communicated through channels. The only exception are lists: They are transmitted in a *stream*-like fashion, i.e. element by element. Each element is evaluated to normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way of synchronizing Eden processes.

In Eden, lazy evaluation is changed to eager in two cases: (1) Processes are eagerly instantiated when a binding `o = e1 # e2` is found while evaluating a binding group, and (2) instantiated processes produce their output even if it is not

$$(a)\quad tr_1 \quad (\textbf{let}\ f\ =\ \lambda i.e\ \textbf{in}\ process\ f) \\ =\ \textbf{process}\ \{i_1,\dots,i_n\} \rightarrow \textbf{let}\ i\ =\ (i_1,\dots,i_n)\ \textbf{in let}\ o\ =\ tr_1\ (e)\ \textbf{in}\ \{o\}$$

$$(b)\quad tr_1 \left(\begin{array}{l} \textbf{let} \\ \quad \dots \\ \quad o\ =\ [\textbf{let}\ binds_1 \\ \qquad\quad \textbf{in let}\ binds_2\ \textbf{in}\ \dots]\ \#\ p\ i \\ \quad \dots \\ \textbf{in} \\ \quad e \end{array} \right) = \begin{array}{l} \textbf{let recpar} \\ \quad \dots \\ \quad \{o_1,\dots,o_m\}\ =\ p\ \#\#\ \{i\} \\ \quad o \qquad\qquad = (o_1,\dots,o_m) \\ \quad [tr_1(binds_1{+}{+}binds_2{+}{+}\dots)] \\ \quad \dots \\ \textbf{in} \\ \quad tr_1(e) \end{array}$$

$$(c)\quad tr_1 \quad (\ \#\ p\ i\ )\ =\ \textbf{let recpar}\ \{o_1,\dots,o_m\}\ =\ p\ \#\#\ \{i\}\ \ \textbf{in}\ \ (o_1,\dots,o_m)$$

FIGURE 2.3. Transformation from Core to CoreEden

demanded. These semantic modifications are aimed at increasing the degree of parallelism and at speeding up the distribution of the computation.

Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]`. Each instantiation of `merge` is a process which *fairly* merges a list of input channels, each one consisting of a list, to produce a single non-deterministic list. The implementation of `merge` immediately copies to its output list any value appearing at any of the input lists. So, `merge` can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. It is a genuine *reactive* process and its non-determinism is a consequence of its reactivity.

CoreEden The CoreEden language is an extension of Core with constructions for process abstractions and process instantiations. It is defined in Figure 2.2. We follow the convention that v denotes a variable, x an atom (i.e. a variable or a literal), and e an expression. Individual channels are made explicit in both constructions by introducing a variable for each one. Process abstractions are introduced as a new expression while process instantiations are considered as special bindings. Binding groups having process instantiations inside are called `recpar` in order to distinguish them from the usual `rec` binding groups having only sequential bindings. The body of a process abstraction consists of a single `let` with all the auxiliary bindings and process instantiations (if any), and a sequence of output channels. The optional `bypass` construction in both process abstractions and instantiations are the annotations produced by the bypassing analysis (see Section 2.5).

The next sections explain in detail the compilation process of Eden. The introduction of a new intermediate language and the corresponding translation steps intends to minimise the changes to GHC and to reuse most of it.

2.3 FROM CORE TO COREEDEN

The aim of this translation is to make explicit the process abstractions and instantiations. In Core, process abstractions are hidden [BKL98] as an application of the function `process` to a function representing the behaviour of the process. This function has an input parameter representing the tuple of inports. The translation is shown in Figure 2.3a. Each process instantiation is hidden as the application of the function `#` to a couple of arguments, but in CoreEden process instantiations are bindings. If the instantiation already appears inside a `let` (see Figure 2.3b), new fresh variables are created for the output channels, while the original output name is kept in order to maintain the references to it, and to facilitate the subsequent untupling transformations (see Section 2.4.1). When there is no binding associated to the instantiation, a new `let` is created in order to introduce the appropriate binding (see Figure 2.3c).

2.4 INSIDE COREEDEN

Here two things are needed: Collecting as many process instantiation bindings as possible, and making explicit the input and output channels. The intuition behind these transformations is first to instantiate groups of processes in parallel as soon as possible, and second to detect those channels that connect directly one process to another one in the same group. These two things are achieved through two kinds of transformations called *flattening* and *untupling*. Note that these transformations are new with respect to those existing in GHC.

2.4.1 Flattening transformations

There are three different transformations:

$$\frac{\mathbf{let}\ [\mathbf{rec}|\mathbf{recpar}]\ binds_1\ \mathbf{in}\ \mathbf{let}\ \mathbf{recpar}\ binds_2\ \mathbf{in}\ e}{\mathbf{let}\ \mathbf{recpar}\ binds_1; binds_2\ \mathbf{in}\ e}$$

$$\frac{\mathbf{case}\ (\mathbf{let}\ \mathbf{recpar}\ binds\ \mathbf{in}\ e)\ \mathbf{of}\ alts}{\mathbf{let}\ \mathbf{recpar}\ binds\ \mathbf{in}\ (\mathbf{case}\ e\ \mathbf{of}\ alts)}$$

$$\frac{(\mathbf{let}\ \mathbf{recpar}\ binds\ \mathbf{in}\ e)\ x}{\mathbf{let}\ \mathbf{recpar}\ binds\ \mathbf{in}\ (e\ x)}$$

The first is the main one, while the others are useful only because they enable subsequent flattening transformations. In order to flatten as much information as possible, the three transformations are iterated until a fixpoint is reached. Notice that the transformations trivially preserve Haskell's semantics, and that they also preserve Eden's semantics, since the moment in which processes are instantiated is not changed. The reason is that in Core, `case` is strict in its discriminant, and function application is strict in the function to be applied. Notice also that, in the Core to Core transformations, the GHC dependency analysis [JM99] that detects strongly connected components will undo this flattening.

2.4.2 Untupling transformations

These transformations try to make explicit the individual channels, both in process instantiations and in process abstractions. They are are only needed in case the process uses tuples as inputs or outputs for bypassing the individual channels of the tuple to different processes. That is, in case that all the channels of the tuple are to be bypassed to the same process, no untupling is needed. In what follows, we assume that all aliases have been removed.

Process instantiations

After the translation from Core to CoreEden of a process instantiation, an expression like this is obtained:

$$\textbf{let recpar} \ \ \ldots \ \ \{o_1,\ldots,o_m\} \ = \ p \ \#\# \ \{i\} \ \ \ldots \ \ \textbf{in}\ e$$

Untupling the inputs. Only in case there is no computation involving the whole input, it is possible to individually bypass each input channel. Therefore, if the individual input values are used, there must exist a closure like the following:

$$i \ = \ [\textbf{let}\ binds_1\ \textbf{in let}\ binds_2\ \textbf{in}\ \ldots] \ \ (i_1,\ldots,i_n)$$

If such a closure is not found, it is only possible to bypass all the input channels together, but not individually. Thus, untupling is neither possible nor needed.

In case the closure is found, the `let`–bindings must be floated, so that i is just defined as a tuple $(i_1,\ldots,i_n)$. After that, the following transformation is used to make explicit the input channels:

$$\frac{\{o_1,\ldots,o_m\} \ = \ p \ \#\# \ i}{\{o_1,\ldots,o_m\} \ = \ p \ \#\# \ \{i_1,\ldots,i_n\}}$$

Then all references to i must be removed. Hence, the following transformation is applied wherever possible:

$$\frac{\textbf{case}\ i\ \textbf{of}\ (\ldots,v_j,\ldots) \ \rightarrow \ e}{e[i_j/v_j]}$$

At this moment, if i was not used as a whole, all references to it should have disappeared. Therefore, the dead-code removal transformation of GHC is reused to remove i's closure.

The steps can be summarized as follows: (1) Search the abstract tree for a closure $i \ = \ (i_1,\ldots,i_n)$; (2) if such a closure is found, *maybe_remove(i)* is applied, defined as:

$$maybe\_remove(i) \stackrel{\text{def}}{=} \begin{cases} \text{a) Replace uses of } i \text{ by the corresponding } i_j \\ \text{b) Remove the definition of } i \text{ (if possible)} \end{cases}$$

Untupling the outputs. In the Core to CoreEden translation new output channels have already been introduced:

$$
\begin{array}{l}
\textbf{let recpar} \\
\qquad \ldots \\
\qquad \{o_1,\ldots,o_m\} \quad = \; p \;\#\#\; \{i_1,\ldots,i_n\} \\
\qquad o \qquad\qquad\quad\; = (o_1,\ldots,o_m) \\
\qquad \ldots \\
\textbf{in } e
\end{array}
$$

In order to be able to bypass individual output channels, there should be no applied occurrences of o. Thus, the same two last steps as in the inputs case must be performed, i.e. *maybe_remove(o)* is applied.

Process abstractions

After Core to CoreEden translation, the abstraction is an expression like:

$$\textbf{process}\ \{i_1,\ldots,i_n\} \rightarrow \textbf{let}\ i\ =\ (i_1,\ldots,i_n)\ \textbf{in let}\ o\ =\ e\ \textbf{in}\ \{o\}$$

Untupling the inputs. It is possible to bypass individual input channels only in case there are no applied occurrences of i. Therefore, the solution is to apply *maybe_remove(i)*.
Untupling the outputs. If the expression e produces a tuple $(o_1,\ldots,o_m)$ as final result, and the individual outputs can be independently bypassed, it must be possible to directly access the individual output channels. Thus, the definition of the output must be of the form

$$\textbf{let}\ o = [\textbf{let}\ \ldots\ \textbf{in}]\ (o_1,\ldots,o_m)$$

In that case, if o is not used, the process abstraction will be translated to:

$$\textbf{process}\ \{x1\ldots x_n\} \rightarrow \textbf{let}\ \ldots\ \textbf{in}\ \{o_1,\ldots,o_m\}$$

2.4.3 Bypassing and non-determinism analysis

Automatic **bypassing** is an optimization of Eden's implementation to reduce the number of messages and/or threads at runtime. The strategy is a combination of compile time analysis and runtime support. Both are explained in [KPS00]. The analysis decorates process abstractions and `recpar` bindings with `bypass cs` clauses, where `cs` are variables representing channels or tuples of channels. In the last case bypassing of all the channels in the tuple will be done. The annotations in a process abstraction correspond to two different types of bypassing:

Bypassing between generations An input – resp. output – channel of a process connected through a variable to an input – resp. output – channel of a descendant process.

(a)

```
q::[Process a (a,a)] →
   Process (a,a) (a,a)
q [ ]    = pid
q (p:pp) =
 process (v1,v2)→(v2,o2)
 where
  (o2,o3) = (q pp) # (p # v1)
```

(b)

```
q= λ ps. case ps of
   [ ] → pid
   p:pp→ process {v1,v2} →
          let recpar
           {o1} = p ## {v1}
           recq = q pp
           {o2,o3} = recq ## {o1}
           bypass o1
          in {v2,o2}
         bypass v1,v2,o2
```

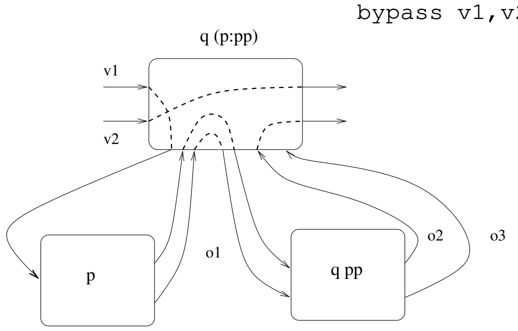

FIGURE 2.4. A bypassing example

Bypassing between ancestors An input channel of a process connected to an output channel of the same process.

Those in the `recpar` bindings correspond to a third kind of bypassing:

Bypassing between siblings An output channel of a process connected through a variable in the parent to an input channel of a sibling process.

Figure 2.4 shows an example: (a) is an Eden program and (b) is the corresponding annotated CoreEden program.This is an *ad hoc* example which illustrates the three types of bypassing. It also contains the case where all the channels in a tuple are to be bypassed. In that case only a variable of tuple type, `o1` in the example, appears in the annotation.

A **non-determinism** analysis, fully explained in [PS00c], is also performed at this point. It detects whether an expression is sure to be deterministic or whether it may be non-deterministic, and annotates the variables with this information.

2.5 FROM COREEDEN TO CORE

It is time to go back to Core, to go on with the normal GHC compilation. The main goals of this translation phase are:

- To eagerly instantiate the processes.
- To embody the bypassing information.
- To make possible the reuse of later GHC's optimizations as far as possible.

$$
\begin{array}{l}
tr_3 \begin{pmatrix} \textbf{process}\ \{i_1,\ldots,i_n\} \rightarrow body \\ \textbf{bypass}\ bcs \end{pmatrix} = \\
\\
\quad \textbf{case}\ createBypass\ (\ )\ \textbf{of}\ h_1 \rightarrow \\
\quad\quad \textbf{case}\ createBypass\ (\ )\ \textbf{of} \ldots h_m \rightarrow \\
\quad\quad\quad \textbf{let}\ f = \lambda\, i_1.\ \ldots \lambda\, i_n.\ body' \\
\quad\quad\quad \textbf{in}\ processBy\ f\ (byp_1, byp_2) \\
\quad \text{where} \\
\quad\quad\quad (body', byp_2) = trbd\ body\ cs \\
\quad\quad\quad cs = zip'\ bcs\ [h_1,\ldots,h_m] \\
\quad\quad\quad byp_1 = \{i_1,\ldots,i_n\} \sqcap cs
\end{array}
$$

$$
\begin{array}{l}
tr_3\ (\textbf{let recpar}\ bs\ \textbf{bypass}\ bcs\ \textbf{in}\ e) = \\
\quad \textbf{case}\ createBypass\ (\ )\ \textbf{of}\ h_1 \rightarrow \\
\quad\quad \textbf{case}\ createBypass\ (\ )\ \textbf{of} \ldots h_m \rightarrow \\
\quad\quad\quad \textbf{let rec}\ bs' \\
\quad\quad\quad \textbf{in case}\ v_1\ \textbf{of}\ \_ \rightarrow \ldots \\
\quad\quad\quad\quad\quad \textbf{case}\ v_n\ \textbf{of}\ \_ \rightarrow tr_3(e) \\
\quad \text{where} \\
\quad\quad\quad (bs', vs) = trbs\ bs\ cs \\
\quad\quad\quad v_i = vs!!i \\
\quad\quad\quad cs = zip'\ bcs\ [h_1,\ldots,h_m]
\end{array}
$$

$$
\begin{array}{l}
trbd \begin{pmatrix} \textbf{let recpar}\ bs \\ \quad\quad \textbf{bypass}\ bcs \\ \textbf{in}\ os \end{pmatrix}\ cs = \\
\\
\quad (\textbf{case}\ createBypass\ (\ )\ \textbf{of}\ h_1 \rightarrow \\
\quad\quad \textbf{case}\ createBypass\ (\ )\ \textbf{of} \ldots h_m \rightarrow \\
\quad\quad\quad \textbf{let rec}\ bs' \\
\quad\quad\quad \textbf{in case}\ v_1\ \textbf{of}\ \_ \rightarrow \ldots \\
\quad\quad\quad\quad\quad \textbf{case}\ v_n\ \textbf{of}\ \_ \rightarrow os, os \sqcap cs) \\
\quad \text{where} \\
\quad\quad\quad (bs', vs) = trbs\ bs\ (cs{+}{+}ds) \\
\quad\quad\quad v_i = vs!!i \\
\quad\quad\quad ds = zip'\ bcs\ [h_1,\ldots,h_m]
\end{array}
$$

$$
\begin{array}{l}
trb\ (\{o_1,\ldots,o_m\} = p\ \#\#\ \{v_1,\ldots,v_n\})\ cs = \\
\quad (bs, [o']) \\
\quad \text{where} \\
\quad\quad bs = [o = \textbf{case}\ o'\ \textbf{of}\ Lift\ o'' \rightarrow o'', \\
\quad\quad\quad\quad o' = instantiateBy\ p\ (v_1,\ldots,v_n)\ bo, \\
\quad\quad\quad\quad o_1 = \textbf{case}\ o\ \textbf{of}\ (o'_1,\ldots,o'_m) \rightarrow o'_1 \\
\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad o_m = \textbf{case}\ o\ \textbf{of}\ (o'_1,\ldots,o'_m) \rightarrow o'_m] \\
\quad\quad extout = \{o_1,\ldots,o_m\} \sqcap cs \\
\quad\quad extin = \{v_1,\ldots,v_n\} \sqcap cs \\
\quad\quad bo = (extout, extin)
\end{array}
$$

FIGURE 2.5. Translation from CoreEden to Core

Figure 2.5 shows function tr_3 performing the translation. Only the more relevant cases (process abstraction and `recpar` bindings) are shown. Two new primitive functions, `processBy` and `instantiateBy`, are introduced in order to respectively hide process abstractions and process instantiations. The first one has two arguments, the process abstraction represented as a function and the bypassing information. The second one has three arguments: the variable bound to the process abstraction, the input and the bypassing information.

The bypassing information It is now represented in a way closer to the implementation. In the bypassing protocol local forwards are represented by unique names called *handles*, whose creation is specified in this phase using a primitive function `createBypass`. If the CoreEden annotation is `bypass cs`, one handle must be created for each channel variable in `cs`. If there is a variable representing an *n*-tuple of channels, *n* handles must be created. They are created eagerly using `case` expressions. Should `let` bindings be used, some Core to Core transformations could collapse all the bindings together to increase sharing.

Bypassing information is represented as a pair of lists, corresponding to output and input channels in process instantiations. In process abstractions the convention is the opposite. Each list specifies the index of the channel and the handle it

```
q = λ ps. case ps of
    [ ]  → pid
    p:pp → case createBypass () of hv1 →  … hv2 →  … ho2 →
     let
      f = λ v1.λ v2.case createBypass () of ho11 →  … ho12 →
           let rec
            o1   = case o1′ of Lift o1′′ → o1′′
            o1′  = instantiateBy p v1
                          ([(1,ho11)(2,ho12)],[(1,hv1)])
            recq = q pp
            o    = case o′ of Lift o′′ → o′′
            o′   = instantiateBy recq o1
                          ([(1,ho2)],[(1,ho11),(2,ho12)])
            o2   = case o of (v1,v2) → v1
            o3   = case o of (v1,v2) → v2
           in case o1′ of _ → case o′ of _ → (v2,o2)
     in processBy f ([(1,hv1),(2,hv2)],[(1,hv2),(2,ho2)])
```

FIGURE 2.6. Translation to Core of the example in Figure 2.4

will be connected to. As a handle represents a forward from an input channel to an output channel, each handle will appear in an output list and in an input list. Figure 2.6 shows a translation.

In Figure 2.5, zip' pairs each variable with its corresponding list of handles. If the variable represents just a channel, it will be a singleton list, but if it represents an n-tuple of channels it will be a list with n handles. In the example of Figure 2.6 the pair for `v1` is `(v1,[hv1])` and that one for `o1` is `(o1,[ho11,ho12])`. Operator $\sqcap$ obtains the list of pairs index-handle from the list of variables *bcs* and from the corresponding list of pairs channel-list of handles *cs*. For example, in Figure 2.6, the pairs obtained for `o1` are `(1,ho11)` and `(2,ho12)`.

Process abstractions Each process abstraction is hidden as a function (f in Figure 2.5) and used as argument for the primitive function *processBy* together with the bypassing information inferred by the analysis ((byp_1, byp_2) in Figure 2.5). Non-determinism information is attached to the types of the binders so it is not necessary to care about them. As we have some knowledge about the names of each channel ($\{i_1, \ldots, i_n\}$ in Figure 2.5), they can be provided to *processBy*. As part of the information corresponds to bypassing between generations, the process abstraction's *body* must be translated, so that such information (*cs* in Figure 2.5) is included in the corresponding places. In Figure 2.5, *trbd* translates a process abstraction's body. It is very similar to the translation of a `let recpar` expression.

Process instantiations The `let recpar` expression may contain several instantiations. In this moment the processes have to be eagerly instantiated, so that they are immediately created. The idea is to use the definition of `#` instead of using `#` directly: `# p v = case (instantiateBy p v) of Lift a → a`.

transformation	before	after
(a) Full laziness	$\textbf{let } g = \lambda y.\textbf{let } x = e$ $\textbf{in } e'$ $\textbf{in } \ldots$	$\textbf{let } x = e$ $\textbf{in let } g = \lambda y.e'$ $\textbf{in } \ldots$
(b) Static arguments	$foldr\ f\ z\ l =$ $\textbf{case } l \textbf{ of}$ $[\,] \quad \to z$ $(a : as) \to$ $\textbf{let } v = foldr\ f\ z\ as$ $\textbf{in } f\ a\ v$	$foldr\ f\ z\ l =$ $\textbf{let } foldr'\ l =$ $\textbf{case } l \textbf{ of}$ $[\,] \quad \to z$ $(a : as) \to$ $\textbf{let } v = foldr\ f\ z\ as$ $\textbf{in } f\ a\ v$ $\textbf{in } foldr'\ l$
(c) Specialization	$g = \Lambda ty.\lambda dict.\lambda y.$ $\textbf{let } f = \Lambda ty.\lambda dict.e$ $\textbf{in } f\ ty\ dict\ (f\ ty\ dict\ y)$	$g = \Lambda ty.\lambda dict.\lambda y.$ $\textbf{let } f = \Lambda ty.\lambda dict.e$ $\textbf{in let } f' = f\ ty\ dict$ $\textbf{in } f'\ (f'\ y)$
(d) *let* floating from *let* rhs	$\textbf{let } x = \textbf{let } bind$ $\textbf{in } e$ $\textbf{in } b$	$\textbf{let } bind$ $\textbf{in let } x = e$ $\textbf{in } b$
(e) *case* floating from *let* rhs	$\textbf{let } v = \textbf{case } e_v \textbf{ of}$ $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \to e_i$ $\ldots$ $\textbf{in } e$	$\textbf{case } e_v \textbf{ of}$ $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \to \textbf{let } v = e_i$ $\textbf{in } e$ $\ldots$
(f) *let* to *case*	$\textbf{let } v = e_v \textbf{ in } e$	$\textbf{case } e_v \textbf{ of } v \to e$
(g) Unboxing *let* to *case*	$\textbf{let } v = e_v \textbf{ in } e$	$\textbf{case } e_v \textbf{ of } C\ v_1 \ldots v_n \to$ $\textbf{let } v = C\ v_1 \ldots v_n \textbf{ in } e$

FIGURE 2.7. Dangerous rules in GHC

In Figure 2.5, the function *trb* carries out the translation of each process instantiation. It generates new bindings: one to maintain all the references to the original outport variable o, another one for a fresh variable o' corresponding to the `instantiateBy` application and the m (number of outports) projections necessary to use the outports independently. The eager instantiation of the process takes place by using a `case` expression scrutinising the variable o'. The function *trbs* applies *trb* to each of the bindings in *bs*, collects the new bindings, bs', and all the variables *vs* and returns them to the function (*trbd* or tr_3) responsible for building the `case` expressions. When forcing the evaluation of `instantiateBy`, our primitive functions take control of the situation, and demand the output of the processes without needing to force them with another `case`.

In [PS00a] it is shown that the eager instantiation of processes is not affected by the subsequent transformations done by the GHC.

2.6 INSIDE CORE

At the last stage of the compilation process, we have non-optimized Core code. In [JM99, JS98] it is shown that non-optimized code is in average 2.7 times slower than the (by GHC) optimized one, so it is desirable to preserve as many sequential transformations as possible. The aim of this section is to briefly address the risks of some GHC Core to Core optimizations. First, a transformation rule cannot be accepted if it can change the semantics of programs. Specifically, three possible changes are unacceptable:

- Triggering a process too early or too late.
- Changing the number of instantiated processes.
- Reducing or increasing the non-determinism of an Eden program.

Additionally, program efficiency can be affected in several ways:

- Changing work from a child process to its parent, or vice-versa.
- Increasing the communication overhead due to the copy of free variable closures.
- Increasing memory allocation.

In general, those transformations increasing sharing have to be considered. These include moving bindings out of lambdas or floating them out of `let` rhs (right hand side). In the first case the number of instantiated processes may change if the floated binding embodies any process instantiation: once the binding has been floated, the instantiation takes place only the first time the function is applied. In the second case, recall that a process instantiation is represented in Core as `instantiateBy p v by`, where the binding for `p` must be evaluated in the child process. So, moving bindings out of `p`'s rhs changes the allocation of such bindings, and consequently increases closure traffic. Also, transformations aimed at performing an earlier evaluation of needed expressions have to be studied, as they can lead to changing work between processes.

One would expect many GHC transformations to be dangerous. However, only a few rules have negative effects on CoreEden programs because the previous transformation stages accomplished the precise goal of minimizing this danger. A fully detailed study is done in [PS00a].

2.6.1 Transformations affecting non-determinism

A few rules may change the non-deterministic behaviour expressed by the programmer. In this case, they will be deactivated. The general reason for all of them is the increasing of closure sharing: before the transformation, several evaluation of a non-deterministic expression can produce several different values; after the transformation, a shared non-deterministic expression is once evaluated, yielding a unique value. These rules are the following:

Full Laziness In Figure 2.7a, let us assume *e* to be non-deterministic. Before applying this rule, `x` may get a different value in each application of `g`, and after the rule is applied, `x` will get a unique value.

Static Argument Transformation The danger is present (see Figure 2.7b) when the partial application of the function to its static arguments is not a whnf and it is non-deterministic.

Specialization In Figure 2.7c, let us assume that *e* is a non-deterministic function. Then, the two partial applications `f ty dict` appearing in `g` denote two possibly different functions. But after applying this rule, the two occurrences of `f'` denote the same function.

2.6.2 Transformations affecting process instantiations

The three previous rules reduce the number of times a process is instantiated. For example, let us consider *full laziness* rule in Figure 2.7a assuming that *e* contains a process instantiation. Before applying this rule, the process would be instantiated several times, once in each application of function `g`; after applying the rule, the process would be instantiated only the first time the function is applied.

2.6.3 Rules with other dangerous effects

There are some other rules correct with respect to the (denotational) program semantics, but that can affect in some way its operational semantics, and hence some aspect of its cost behaviour. The possible modifications are changing work between a parent and a child process (e, f and g in Figure 2.7), changing the communication overhead (d, e, f and g in Figure 2.7), or changing the space cost (d, e, f and g in Figure 2.7). These changes can be sometimes positive, and sometimes negative. Also, a positive effect increases opportunities for the application of other rules. Therefore, we have decided to keep these rules active.

2.7 CONCLUSIONS

Process creation in Eden can be seen as a 'side effect' of evaluating functional expressions. However, Eden semantics insists in that the number of instantiated processes has to be decided by the programmer and must be preserved by the compiler. On the other hand, nondeterminism in a functional language spoils in some cases equational reasoning. So it can be expected that some of the transformations done by an optimizing compiler, which are semantically correct in a sequential, deterministic environment, are not correct anymore in a parallel nondeterministic one. The paper has explored this problem and provided solutions to it: the selective disallowing of some transformations when non-determinism is present or when the number of instantiated processes may change. A second problem addressed in the paper has been the addition of an intermediate language CoreEden and of a bypassing analysis, together with the propagation of its annotations, in order to generate a better parallel code. A final contribution has been the embodiment of Eden eager semantics (in process instantiations) into the Core language in such a way that subsequent transformations done by the compiler cannot destroy it. The current state of the implementation includes the eager process instantiation, a prototype of the non-determinism analysis and the runtime system support for bypassing.

REFERENCES

[BKL98] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In PLILP'98. Springer Verlag LNCS 1490, pp. 318–334, 1998.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1998, Philipps-Universität Marburg, Germany, 1996.

[JHH+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 *Joint Framework for Information Technology, Keele, DTI/SERC*, pp. 249–257, 1993.

[JM99] S. L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, September 1999.

[JS98] S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, Sept. 1998.

[KPS00] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In *Trends in Functional Programming. Proceedings of the 1st Scottish Functional Programming Workshop, SFP'99*, pp. 2–10. Intellect, 2000.

[PS00a] C. Pareja and C. Segura. Efecto de las Transformaciones de GHC sobre Edén. Tech. Report 101-00. Dpto. Sistemas Informáticos y Programación (Universidad Complutense de Madrid), 2000.

[PS00b] R. Peña and C. Segura. Two Non-determinism Analyses in Eden. Technical Report 108-00, 46 pages. Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2000.

[PS00c] R. Peña and C. Segura. Non-Determinism Analysis in a Parallel Functional Language. 2000. *Implementation of Functional Languages, IFL'00*.

[San95] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science. University of Glasgow, 1995.

Contents