Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 258 (2009) 21–39
www.elsevier.com/locate/entcs

To be or not to be... lazy (In a Parallel Context)[1]

Mercedes Hidalgo-Herrero[2]

Dept. Didáctica de las Matemáticas
Universidad Complutense de Madrid
Madrid, Spain

Yolanda Ortega-Mallén[3]

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

Laziness restricts the exploitation of parallelism because expressions are evaluated only on demand. Thus, parallel extensions of lazy functional languages, like Haskell, usually override laziness to some extent. The purpose of the present work is to analyze how and to which extent strictness should be introduced in a lazy language to design a parallel extension of it. Towards this end, we have considered different evaluation strategies mixing laziness and eagerness for the language Eden —a parallel extension of Haskell—, we have given formal definitions for each, and we have implemented them in an interpreter to be able to run examples with alternative evaluation models, so that we can observe the intermediate and final states of the processes in the system, in terms of heaps of closures. Although the study is based on Eden, the concepts involved and the conclusions that we have obtained can be transferred to other parallel and functional languages.

Keywords: Functional programming, parallelism, semantics, distribution.

[1] Work partially supported by the Spanish projects TIN2006-15660-C02-01, S-0505/TIC/0407, and CCG08-UCM/TIC-4124.
[2] Email: mhidalgo@edu.ucm.es
[3] Email: yolanda@sip.ucm.es

doi:10.1016/j.entcs.2009.12.003

1 Introduction

Functional languages provide an excellent basis for reliable parallel programming. The key factor is referential transparency. Thanks to this property alternative orders of evaluation that preserve the meaning of an expression are possible.

Functional parallel approaches can be classified by the level of control of parallelism (see [12]), ranging from automatic parallelizations to explicit process creation. Many of these proposals are parallel extensions of sequential functional languages. For instance, the lazy functional language Haskell [15] has been used as the basis of a large and varied set of parallel and distributed languages (see [18]).

As a lazy language, Haskell adopts normal order evaluation, avoiding repeated computations by sharing reductions. This lazy approach restricts the exploitation of parallelism because expressions are evaluated only on demand. Thus, parallel versions of Haskell override laziness as follows:

Speculative work Some languages allow the evaluation of parts of the code that have not been demanded yet. This does not necessarily change the underlying sequential lazy semantics, because the overall result of the program can be obtained even if some speculative subtask does not finish; this is achieved by guaranteeing that the scheduler prefers to evaluate the computations of the main process. In this case, speculation only influences the efficiency of the system. Examples of this kind of speculative computation are the `par` operator defined in GpH [17], and the eager process creation in Eden [13].

Introducing strictness A more drastic way of overriding laziness is to *force* the evaluation of some portions of the code before the result is really needed. Thus, the underlying lazy semantics is modified. Examples of this are the strict operator (`seq`) introduced in GpH, or forcing the reduction to normal form of the values that are to be transmitted through channels in Eden. Similarly, the transmission of lists in Caliban [10,16] is head-strict, and data-parallel versions of Haskell introduce strictness in the use of some predefined data types (mainly lists) [2].

Mixed (lazy and strict) evaluation has already been introduced in declarative languages (like, for instance, OzFun [8]) and analyzed in a sequential context (see [19,20] for a discussion on advantages and risks of this combination, and [3] for semantic properties), but few work has been done to carefully analyze how and to which extent strictness should be introduced in a lazy language to design a parallel extension of it. Towards this end, in [6,7] we have considered alternative evaluation models for the language Eden, and we

have implemented an interpreter, written in Haskell, capable of dealing with all of them. The interpreter was combined with a set of profiling tools in order to analyze the influence of the evaluation strategies in the performance of some chosen parallel skeletons implemented in Eden. The purpose of the present work is to depart from those experimental evaluations and to achieve a more rigorous and complete comparative analysis. Therefore, the contributions of this paper are an extension of the spectrum of evaluation strategies mixing laziness and strictness, and the formalization of each evaluation model. Although the study is based on Eden, the involved concepts and the conclusions that we have obtained can be transferred to other parallel functional languages.

The paper is organized as follows: We start with a brief introduction to parallelism in Eden, and we describe the calculus used for our analysis. In Section 3 we discuss on the possible evaluation strategies, and we give a classification of these around three concepts. Then in Section 4 we present a distributed operational semantics for the calculus, and we formalize the evaluation strategies defined before. In Section 5 we present a collection of examples that shows how the evaluation strategies may affect issues like termination or deadlock. We conclude with a summary discussion on the lazy-eager combinations and outline future work.

2 Parallelism in Eden

Coordination in Eden is based on two concepts: *explicit definition of processes* and *implicit stream-based communication* [9]. In the same way as there is a distinction between function definition and application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes. For an introduction to the language Eden, its syntax and applications, the reader is referred to [13]. In this section we just explain the basic mechanisms of Eden for parallelism, i.e. process creation and value communication.

Figure 1 shows the restricted[4] (abstract) syntax of an untyped λ-calculus extended with recursive lets and process instantiation. This simple calculus captures the essence of Eden and proves to be sufficient for our purposes.

For simplicity, we have identified process abstractions with one-argument functions, so that new processes are created with only one input channel (from parent to child) and one output channel (from child to parent) channel. Moreover, although Eden deals with streams —possibly infinite sequences of data—, in our calculus just one-value channels are considered. When evaluating an

[4] A restricted syntax is considered to simplify the semantic rules, as in [11].

$x \in Var,\ E \in Exp$

$E ::= x$		variable
	$\backslash x.E$	λ-abstraction
	$x_1 x_2$	application
	$x_1 \,\#\, x_2$	process instantiation
	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration

Figure 1. Eden's restricted core syntax

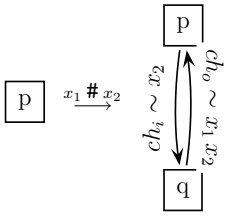

Figure 2. Process creation in Eden

expression $x_1 \,\#\, x_2$ inside a process p, a new child process q is created. Process q evaluates $x_1\ x_2$ and returns to its parent the result via its output channel (ch_o). In order to carry out this evaluation, q receives from p the value of x_2 through its input channel (ch_i). The diagram in Figure 2 illustrates this behavior.

In order to favor parallelization, processes have to be created as soon as possible. Therefore, while functional application is lazy, *parallel application*, i.e. process instantiation, is eager. The next example shows how process instantiation behaves.

Example 2.1 Let us consider the following expression:

$$\begin{array}{rl} \texttt{let} & x1 = x2 \,\#\, x3, \\ & x2 = \backslash x6.x6, \\ & x3 = \backslash x7.x4, \\ & x4 = x5\ x5, \\ & x5 = \backslash x8.x8\ x8 \\ \texttt{in} & x1 \end{array}$$

While evaluating the let-expression, the instantiation reaches the top level:

main (N. Children: 0)

$main \overset{A}{\mapsto} x1$	$x3 \overset{I}{\mapsto} \backslash x7.x4$
$x1 \overset{I}{\mapsto} x2 \,\#\, x3$	$x4 \overset{I}{\mapsto} x5\ x5$
$x2 \overset{I}{\mapsto} \backslash x6.x6$	$x5 \overset{I}{\mapsto} \backslash x8.x8\ x8$

Then a new process is created: [5]

main (N. Children: 1)

$ch_i \overset{A}{\mapsto} x3$	$x3 \overset{I}{\mapsto} (\backslash x7.x4)$
$main \overset{A}{\mapsto} x1$	$x4 \overset{I}{\mapsto} x5\ x5$
$x1 \overset{B}{\mapsto} ch_o$	$x5 \overset{I}{\mapsto} \backslash x8.x8\ x8$
$x2 \overset{I}{\mapsto} (\backslash x6.x6)$	

main.1 (N. Children: 0)

$ch_o \overset{A}{\mapsto} x11\ x9$
$x9 \overset{B}{\mapsto} ch_i$
$x11 \overset{I}{\mapsto} \backslash x10.x10$

[5] The states shown for the examples have been obtained with an interpreter implemented in Haskell. The generation of free variables returns names in the style xn, where n is an increasing integer.

The evaluation of the main variable depends on the result calculated by the child process, which in turn needs the value of channel ch_i to be sent by the main process. The communication of a λ-abstraction implies de copy in the receiver of all the closures related to the free variables of the λ-abstraction ($x4$ in our case). If free variables are to be evaluated to *whnf* before the copy, then the evaluation of the example does not terminate, because the evaluation of $x4$ leads to an endless computation. By contrast, if free variables are allowed to be copied unevaluated, then a value is obtained for the variable *main*, as shown in the Example 5.1.

The following section discusses the different options for evaluating expressions in our calculus.

3 Mixed evaluation strategies

Eden has been designed for distributed environments without shared memory between processes; therefore, bindings have to be copied from one heap to the other when creating new processes or when communicating values. In this context, the following questions can be formulated:

- In the expression $x_1 \,\#\, x_2$, it is clear that x_2 has to be evaluated in the instantiating process. But what about x_1? Should the parent evaluate the expression before copying it into the child's heap?
- How should the free variables in a newly instantiated process be handled?
- What about the values communicated through the channels? To what extent should they be evaluated before being communicated? Is it advisable to send the extra work related to the free variables —with an unknown degree of evaluation— to the receiver?
- Should an instantiation expression be copied from one heap to another? Or is it more advisable to suspend the corresponding communication/process creation?

All these questions are related to the distribution of computation between processes: *How much work should do the parent (resp. producer) of a process (resp. value), and how much work should be left for the child (resp. consumer)?* This is a crucial point in any parallel language, and it is not particular to Eden, although the features of Eden maybe offer more possibilities for discussion.

It turns out that the alternatives can be expressed as different mixtures of *lazy* and *eager* evaluation. In fact, neither pure laziness nor eagerness are optimal, in the sense that, for each proposal, examples can be found showing

that the opposite view would be much more efficient.

3.1 Keystones of the evaluation strategies

We can organize the evaluation strategies around three concepts:

Process Abstraction Evaluation (PAE) In the case of a process instantiation, the evaluation of the process abstraction can be done either by the parent process, or by the child. In the first case, process instantiation could be more costly for the instantiating process, but the programmer has a greater control of the sharing of work between parent and child, that leads to the possibility of designing libraries of process abstractions to create "slaves" to get the "hard work" done. The performance of the processes created from these libraries is guaranteed, because it will not depend on the context were processes are created.

Evaluation Before Copy (EBC) When copying bindings from a heap process to another, it may be required that every needed binding —corresponding to free variables in process/lambda abstractions— is previously evaluated. This corresponds to a strict semantics as can be found for ML [14]. This option applies to two situations: (1) when creating the initial heap of a new process (EBC_p); (2) when communicating a value through a channel (EBC_v).

There is also a choice of how much information should be sent to a child process. For instance, we could decide not to send any information at first, i.e. create processes with empty heaps, so that information is only sent on demand, that fits better to a lazy strategy. This could be an option for shared memory, but we have discarded it in a distributed setting, because it would lead to extra —and out of time— communications; moreover, it would not be easy to determine where to find the needed information, and it would require to look for it through the hierarchy of processes.

Instantiation Copy (IC) If the copy —from one heap to another— of process instantiations is not permitted, then the action is blocked until the pending instantiation is resolved. This applies to process creation (IC_p) as well as to value communication (IC_v).

Therefore we have five issues (PAE, EBC_p, EBC_v, IC_p and IC_v), each with two options: parent/child for PAE, yes/no for the rest. This gives a total of $2^5 = 32$ combinations. Some of these can be discarded; for instance, if it is required that every needed binding should be evaluated before its copy (EBC = yes), this should imply the evaluation of pending instantiations too (i.e. IC = no), then the list is reduced to 18 options. Moreover, if it is required that the parent evaluates the process abstraction (PAE = parent) then it is

	PAE	**EBC**$_p$	**IC**$_p$
(1)	parent	yes	no
(2)	child	yes	no
(3)	child	no	yes
(4)	child	no	no

	EBC$_v$	**IC**$_v$
(a)	yes	no
(b)	no	yes
(c)	no	no

Table 1
Evaluation strategies

reasonable that also the parent evaluates the needed bindings before they are copied to the heap of a new child (EBC$_p$ = yes). This then reduces the set to 12 strategies. By separating the discussion relating to process creation from the options corresponding to communication, we can organize the strategies in a table with four combinations. For each combination we have to consider the three options permitted for communication (see table 1).

4 A distributed semantic model

The semantic model that we consider here has already been used to give a formal semantics for Eden [4,5]. It embodies two levels of transitions: a lower level to handle the local behavior of processes, and an upper level to describe global effects on the system, namely process creation and communication.

The evaluation of an expression of the calculus given in Figure 1 will require in general the creation of several parallel processes. Each process will, in turn, encompass a set of independently executing threads, each devoted to the production of one output of the process.

To model communication we need a set of channel identifiers, *Chan*. In the following, we use x, y, z as program variables and ch as a channel, while θ refers to program variables as well as channels. η represents a fresh renaming.

In our model, the state of evaluation of a process is represented by its heap of closures, i.e. the set of bindings of identifiers (variables and channels) to expressions or channels. Following [1], each binding is considered a potential thread and has associated a label indicating its state: $\theta \overset{\alpha}{\mapsto} E$ where $E \in Exp \cup Chan$, and $\alpha ::= I|A|B$ corresponds, respectively, to Inactive (either not yet demanded or already completely evaluated), Active (demanded and in execution), and Blocked (demanded but waiting for the value of another binding).

The set $dom(H)$ contains the left-hand-side identifiers of a heap H. Notation $H + \{\theta \overset{\alpha}{\mapsto} E\}$ (and also $\{\theta \overset{\alpha}{\mapsto} E\} + H$) means that the heap H is extended with the binding $\theta \overset{\alpha}{\mapsto} E$, and it is assumed that $\theta \notin dom(H)$.

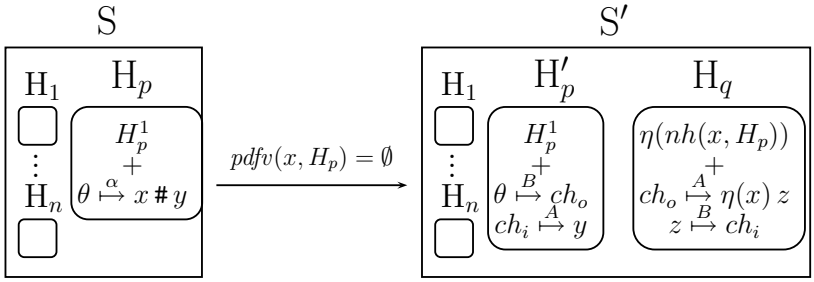

Figure 3. Process creation

To evaluate an expression $E \in Exp$, the initial system consists only of the process *main* with an initial heap $H_0 = \{main \overset{A}{\mapsto} E\}$, where it is assumed that *main* is a fresh variable. The system evolves through global steps. The tasks to be done comprise parallel local evolution of all the processes in the system, process creation, interprocess communication, and thread state management (like for instance, thread unblocking and deactivation).

For the purpose of the present work we only describe here the rules for process creation and communication. The rest of the operational rules are given in the Appendix. For more details and explanations the reader is referred to [4,5].

4.1 Process creation

Processes are created by applying the rule given in Figure 3. As shown in Example 2.1, in our calculus processes are eagerly created [6] when instantiations are found at the top-level, i.e. when a variable in a heap is directly bound to a #-expression, even if that binding is not active (i.e. it has not been demanded yet).

The thread evaluating the instantiation (variable θ at the *parent* side) is blocked on a fresh output channel, ch_o, corresponding to the initial thread in the new *child* process q. Correspondingly, the child gets a thread (z) which is blocked on a new input channel, ch_i, which is served by a new thread in the parent.

The absence of a common shared heap requires that every binding needed for the evaluation of the free variables in the process abstraction is copied from the parent to the child's heap. For this purpose we use the function nh (needed heap), where $nh(E, H)$ collects all the bindings in H that are reachable from E.

As commented in Section 3, another possibility is to make this copy only on demand, that is, whenever a process needs a variable that is not defined

[6] If process creation is lazy, then there is no parallelism.

in its heap, it is demanded to the parent. However, it may occur that the needed variable is not in the parent, but in one of its ancestors, or even in some offspring (due to a communication). Consequently, the cost of searching the owner of the variable may be greater in comparison to the initial copy using function *nh*.

The definition of *nh* follows the same pattern as other recursive auxiliary functions of our semantics. Hence, we show the *common cases* by means of *cc* (that replaces the corresponding function name):

$$
\begin{array}{lll}
cc(E, \emptyset) & = \emptyset & \\
cc(x, H) & = \emptyset & \text{if } x \notin dom(H) \\
cc(\backslash x.E, H) & = cc(E, H) & \\
cc(x_1\, x_2, H) & = cc(x_1, H) \cup cc(x_2, H) & \\
cc(x_1 \,\#\, x_2, H) & = cc(x_1, H) \cup cc(x_2, H) & \\
cc(\texttt{let } \{x_i = E_i\}_{i=1}^{n} \texttt{ in } x, H) & = cc(x, H) \cup (\bigcup_{i=1}^{n} cc(E_i, H)) &
\end{array}
$$

Notice that the case $cc(x, H)$ where $x \in dom(H)$ is missing. This is precisely the *significant* case in our auxiliary functions, and for *nh* —that is independent of the semantic option chosen— is defined as follows:

$$nh(x, \{x \overset{\alpha}{\mapsto} E\} + H) = \{x \overset{I}{\mapsto} E\} + nh(E, H).$$

A process creation takes place only if it is *feasible*. This is detected by functions *dfv* (demand of free variables) and *pdfv* (previous to *dfv*), that will be formalized in Section 4.3. The feasibility of a process creation depends on the evaluation strategy, but at any case it is required that the process body does not depend on a value to be communicated from some other process. Alternative design options could be considered where process creations (and communications) are allowed even if there is a dependency on a channel. However, this would lead to transform communications 1-1 between processes to communications 1-n, that are much more costly and difficult to implement.

4.2 Communication

The rule for communication is sketched in Figure 4. A communication takes place if there is a process with a channel (*ch*) bound to a value —a λ-abstraction in our calculus— and another process contains a variable (θ) blocked on this channel. The bindings needed for the evaluation of the free variables in the communicated value are copied from the producer's to the consumer's heap (*nh*). Similarly to the case of process creation, this copy is done only if there is no dependency on pending communications. Notice that the channel disappears after the communication

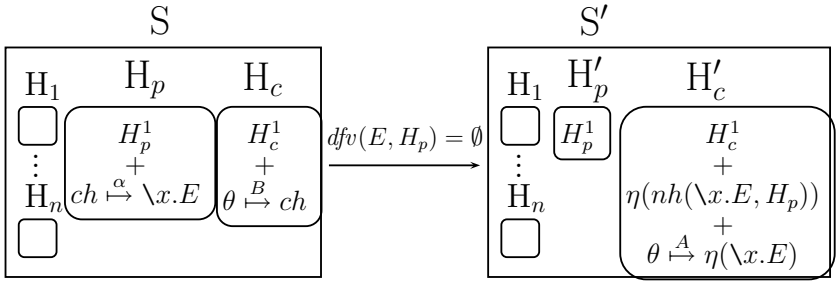

Figure 4. Value communication

4.3 Formalization of semantic options

The semantics is parameterized by functions *dfv* and *pdfv*. Function *dfv* (demand of free variables) checks the circumstances that cause a process creation (or a communication) to be suspended:

- A pending communication.
- A pending process creation.
- A free variable not bound to a λ-abstraction.

Whereas all the evaluation strategies suspend when pending communications are found, pending process creations are only considered when IC=no, and the last condition only when EBC=yes. Consequently, three different versions of *dfv* are needed to express the evaluation strategies considered in table 1.

In order to take into account the option PAE, in the process creation rule the function *pdfv* (previous to demand of free variables) is applied just before *dfv*:

$$
\begin{array}{lll}
pdfv(x, H) & = \emptyset & \text{if } x \notin dom(H) \\
pdfv(x, \{x \overset{\alpha}{\mapsto} \backslash x.E\} + H) & = dfv(E, H) & \text{if PAE=parent} \\
pdfv(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = \{x \overset{\alpha}{\mapsto} E\} & \text{if PAE=parent} \wedge E \neq \backslash x.E' \\
pdfv(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = dfv(E, H) & \text{if PAE=child}
\end{array}
$$

If PAE=parent then the process abstraction x_1 —for an instantiation expression $x_1 \,\#\, x_2$— must be evaluated before proceeding with the creation. In this case, and only if the corresponding expression is still unevaluated, i.e. it is not a λ-abstraction, *pdfv* returns a heap with a unique binding for x_1. Otherwise, *pdfv* just calls *dfv*.

Although *dfv* and *pdfv* test the feasibility of process creations and communications, they are not defined just as boolean functions because the sets of dependencies are needed for generating demand in the scheduling rules (see [4,5] and the Appendix).

Next, the significant cases for *dfv* (i.e. those not considered by the pattern *cc*) are defined for each evaluation strategy.

		Evaluation Strategies											
		(1)(a)	(1)(b)	(1)(c)	(2)(a)	(2)(b)	(2)(c)	(3)(a)	(3)(b)	(3)(c)	(4)(a)	(4)(b)	(4)(c)
Global Rules	Process Creation	$pdfv^I$	$pdfv^I$	$pdfv^I$	$pdfv^I$	$pdfv^I$	$pdfv^I$	$pdfv^{II}$	$pdfv^{II}$	$pdfv^{II}$	$pdfv^{II}$	$pdfv^{II}$	$pdfv^{II}$
	Communication	dfv^I	dfv^{II}	dfv^{III}	dfv^I	dfv^{II}	dfv^{III}	dfv^I	dfv^{II}	dfv^{III}	dfv^I	dfv^{II}	dfv^{III}

Table 2
Definition of *pdfv* and *dfv* for each evaluation strategy

I. EBC=yes (⇒ IC=no)

$$
\begin{array}{lll}
(1) & dfv^I(x, \{x \overset{\alpha}{\mapsto} \backslash x.E'\} + H) & = dfv^I(E', H) \\
(2) & dfv^I(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = \{x \overset{\alpha}{\mapsto} E\} \qquad \text{if } E \neq \backslash x.E' \wedge \alpha \neq B \\
(3) & dfv^I(x, \{x \overset{B}{\mapsto} y\} + H) & = \{x \overset{B}{\mapsto} y\} \cup dfv^I(y, H) \\
(4) & dfv^I(x, \{x \overset{B}{\mapsto} x_1\, x_2\} + H) & = \{x \overset{B}{\mapsto} x_1\, x_2\} \cup dfv^I(x_1, H) \\
(5) & dfv^I(x, \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} + H) & = \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} \cup dfv^I(x_1, H) \\
(6) & dfv^I(x, \{x \overset{B}{\mapsto} ch\} + H) & = \{x \overset{B}{\mapsto} ch\}
\end{array}
$$

If x is already bound to an abstraction (1), then dfv^I must gather the free variables corresponding to this value. Otherwise, the binding for x is collected (2), and if this binding is blocked, then the following cases must be considered:

(i) If it is blocked on another variable (3), then dfv^I continues with the binding for this second variable.

(ii) If it is blocked either on an application (4) or an instantiation (5), then dfv^I continues checking the corresponding abstraction (function/process).

When x is blocked on a channel (6), it is unnecessary to go further, because a channel appears at most once in a heap.

II. EBC=no, IC=yes

With this combination, the only reason to suspend a communication or a process creation is a dependency on a channel (i.e. pending communication):

$$
\begin{array}{l}
dfv^{II}(x, \{x \overset{B}{\mapsto} ch\} + H) = \{x \overset{B}{\mapsto} ch\} \\
dfv^{II}(x, \{x \overset{\alpha}{\mapsto} E\} + H) \; = dfv^{II}(E, H) \;\; \text{if } E \notin Chan
\end{array}
$$

III. EBC=no, IC=no

In this case dfv^{III} detects dependencies on instantiation expressions and channels:

$$
\begin{array}{ll}
dfv^{III}(x, \{x \overset{\alpha}{\mapsto} x_1 \,\#\, x_2\} + H) & = \{x \overset{\alpha}{\mapsto} x_1 \,\#\, x_2\} \\
dfv^{III}(x, \{x \overset{B}{\mapsto} ch\} + H) & = \{x \overset{B}{\mapsto} ch\} \\
dfv^{III}(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = dfv^{III}(E, H) \qquad \text{if } E \neq x_1 \,\#\, x_2 \wedge E \notin Chan
\end{array}
$$

The versions of *dfv* and *pdfv* corresponding to the evaluation strategies given in Table 1 are summarized in Table 2.

5 Applications

In this section we include some examples that show the behavior of a program under different evaluation strategies.

5.1 Termination

The termination of a program may depend on the semantic option chosen. It is the case of the following example:

Example 5.1 Let us consider again the expression given in Example 2.1. As explained before, if free variables are to be evaluated to whnf before being copied (EBC=yes), then the evaluation never terminates, because the communication from the main process to its child is impossible:

main (N. Children: 1)		**main.1** (N. Children: 0)
$ch_i \overset{I}{\mapsto} \backslash x7.x4$	$x3 \overset{I}{\mapsto} \backslash x7.x4$	$ch_o \overset{B}{\mapsto} x11\ x9$
$main \overset{B}{\mapsto} x1$	$x4 \overset{A}{\mapsto} x5\ x5$	$x9 \overset{B}{\mapsto} ch_i$
$x1 \overset{B}{\mapsto} ch_o$	$x5 \overset{I}{\mapsto} \backslash x8.x8\ x8$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x2 \overset{I}{\mapsto} \backslash x6.x6$		

By contrast, if free variables are allowed to be copied unevaluated (EBC=no), then the evaluation comes to an end, and the final system obtained is:

main (N. Children: 1)		**main.1** (N. Children: 0)
$main \overset{I}{\mapsto} \backslash x20.x23$	$x4 \overset{I}{\mapsto} x5\ x5$	$x9 \overset{I}{\mapsto} \backslash x12.x15$
$x1 \overset{I}{\mapsto} \backslash x20.x23$	$x5 \overset{I}{\mapsto} \backslash x8.\ x8\ x8$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x2 \overset{I}{\mapsto} \backslash x6.x6$	$x23 \overset{I}{\mapsto} x24\ x24$	$x15 \overset{I}{\mapsto} x16\ x16$
$x3 \overset{I}{\mapsto} \backslash x7.x4$	$x24 \overset{I}{\mapsto} \backslash x22.x22\ x22$	$x16 \overset{I}{\mapsto} \backslash x14.x14\ x14$

The example shows that strictness on free variables may lead to non-termination.

5.2 Deadlock

In some contexts, and depending on IC, a deadlock state is reached.

Example 5.2 Let us consider the following expression:

$$\begin{array}{ll} \texttt{let} & x1 = x1\,\#\,x1, \\ & x2 = x3\,\#\,x4, \\ & x3 = \backslash x5.x5, \\ & x4 = \backslash x6.\backslash x7.x1 \\ \texttt{in} & x2 \end{array}$$

With the option IC=no, the communication from the parent to the child cannot take place because $x1 \mapsto\ x1 \,\#\, x1$ cannot be copied. Consequently, the system gets deadlocked:

main (N. Children: 1)	**main.1** (N. Children: 0)
$ch_i \overset{I}{\mapsto} \backslash x6.(\backslash x7.\ x1)$ $main \overset{B}{\mapsto} x2$ $x1 \overset{B}{\mapsto} x1 \,\#\, x1$ $x2 \overset{B}{\mapsto} ch_o$ $x3 \overset{I}{\mapsto} \backslash x5.\ x5$ $x4 \overset{I}{\mapsto} \backslash x6.(\backslash x7.\ x1)$	$ch_o \overset{B}{\mapsto} x8$ $x8 \overset{B}{\mapsto} ch_i$ $x16 \overset{I}{\mapsto} \backslash x9.\ x9$

Nevertheless, in a lazier context with IC=yes, the communication from the parent to the child is achieved successfully, and the evaluation ends with a whnf value bound to the *main* variable:

main (N. Children: 1)	**main.1** (N. Children: 0)
$main \overset{I}{\mapsto} \backslash x13.(\backslash x14.\ x12)$ $x1 \overset{B}{\mapsto} x1 \,\#\, x1$ $x2 \overset{I}{\mapsto} \backslash x13.(\backslash x14.\ x12)$ $x3 \overset{I}{\mapsto} \backslash x5.\ x5$ $x4 \overset{I}{\mapsto} \backslash x6.(\backslash x7.\ x1)$ $x12 \overset{B}{\mapsto} x12 \,\#\, x12$	$x8 \overset{I}{\mapsto} \backslash x11.(\backslash x19.\ x10)$ $x16 \overset{I}{\mapsto} \backslash x9.\ x9$ $x10 \overset{B}{\mapsto} x10 \,\#\, x10$

Although three process instantiations remain (self)blocked, these are unimportant because the results are not needed (speculative work).

Therefore, in order to avoid this deadlock, IC should be *yes*. However, sharing is reduced in this option, and if a process creation is copied to several children, then a duplication of work may occur.

5.3 Too costly children

Creating a child process may not be profitable for the parent. The following example illustrates this situation when EBC_v=yes.

Example 5.3 Let us consider the following expression:

$$\begin{array}{rl} \texttt{let} & x1 = x3\, x2, \\ & x2 = \backslash x6.x6, \\ & x3 = x2\, x2, \\ & x4 = \backslash x7.\backslash x8.x1\, x2, \\ & x5 = x2 \,\#\, x4 \\ \texttt{in} & x5 \end{array}$$

The new process is created during the first (global) step:

main (N. Children: 1)		**main.1** (N. Children: 0)
$ch_i \overset{A}{\mapsto} x4$	$x3 \overset{I}{\mapsto} x2\,x2$	$ch_o \overset{A}{\mapsto} x11\,x9$
$main \overset{A}{\mapsto} x5$	$x4 \overset{I}{\mapsto} \backslash x7.\backslash x8.x1\,x2$	$x9 \overset{B}{\mapsto} ch_i$
$x1 \overset{I}{\mapsto} x3\,x2$	$x5 \overset{B}{\mapsto} ch_o$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x2 \overset{I}{\mapsto} \backslash x6.x6$		

Variables $x1$ and $x3$ must be evaluated before the communication from the parent to the child takes place:

main (N. Children: 1)	**main.1** (N. Children: 0)
$main \overset{B}{\mapsto} x5$	$ch_o \overset{B}{\mapsto} x9$
$x1 \overset{A}{\mapsto} \backslash x6.x6$	$x9 \overset{A}{\mapsto} \backslash x12.\backslash x13.x16\,x17$
$x2 \overset{I}{\mapsto} \backslash x6.x6$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x3 \overset{I}{\mapsto} \backslash x6.x6$	$x16 \overset{I}{\mapsto} \backslash x14.x14$
$x4 \overset{I}{\mapsto} \backslash x7.\backslash x8.x1\,x2$	$x17 \overset{I}{\mapsto} \backslash x15.x15$
$x5 \overset{B}{\mapsto} ch_o$	

Therefore, the parent has to do all the work in order to send to the child everything already evaluated, while the activity of the child is reduced to return back (to the parent) the same value that it has just received from it!

The last example shows a situation where a process creation is not profitable because PAE=parent.

Example 5.4 Let us consider the following expression:

$$\begin{aligned} \texttt{let}\ & x1 = x2\,\texttt{\#}\,x3, \\ & x2 = x3\,x4, \\ & x3 = \backslash x7.x7, \\ & x4 = x3\,x5, \\ & x5 = x3\,x6, \\ & x6 = x3\,x3 \\ \texttt{in}\ & x1 \end{aligned}$$

After evaluating the let-expression we obtain:

main (N. Children: 0)	
$main \overset{A}{\mapsto} x1$	$x4 \overset{I}{\mapsto} x3\,x5$
$x1 \overset{I}{\mapsto} x2\,\texttt{\#}\,x3$	$x5 \overset{I}{\mapsto} x3\,x6$
$x2 \overset{I}{\mapsto} x3\,x4$	$x6 \overset{I}{\mapsto} x3\,x3$
$x3 \overset{I}{\mapsto} \backslash x7.\ x7$	

The process creation is delayed until the process abstraction is evaluated. This work is carried out by the parent in eleven global steps.

main (N. Children: 1)		**main.1** (N. Children: 0)
$ch_i \overset{A}{\mapsto} x3$	$x3 \overset{I}{\mapsto} \backslash x7.x7$	$ch_o \overset{A}{\mapsto} x18\, x8$
$main \overset{B}{\mapsto} x1$	$x4 \overset{I}{\mapsto} \backslash x7.x7$	$x8 \overset{B}{\mapsto} ch_i$
$x1 \overset{B}{\mapsto} ch_o$	$x5 \overset{I}{\mapsto} \backslash x7.x7$	$x18 \overset{I}{\mapsto} \backslash x9.x9$
$x2 \overset{A}{\mapsto} \backslash x7.x7$	$x6 \overset{I}{\mapsto} \backslash x7.x7$	

Afterwards, the calculation performed by the child process only takes two further steps. Once again, the parent has done most of the computation. Hence, if the purpose of creating children is to delegate work, the suitable option is PAE=child.

6 Discussion and future work

The combination PAE=parent, EBC_p=yes and EBC_v=yes (IC_p=no and IC_v=no), i.e. entry (1)(a) in Table 1, could be considered the most eager approach. This evaluation strategy tends to be more efficient, because in many cases work duplication is avoided, and the size of the data transmitted —either through communication channels or by copying from heap to heap— is much smaller. It also benefits from a greater control of load balance and of communications, as the size of the transmissions depends exclusively on the type of the value to be communicated; while in a context with EBC_v=no there is no way to determine the expected size of a transmission, as the "current state" of the free variables of the communicated value must be packed and sent to the consumer, and the evaluation of these may depend on very large objects.

The main argument against this eager strategy is that, as we have seen in the first example in Section 5, the evaluation of free variables in advance to create a child may lead to a *loss of the normal order*, and this is a critical matter. As a consequence, we cannot replace equals by equals, as any functional programmer would expect.

As eagerness may lead to spend a lot of energy on useless work or even to endless loops (Example 5.1), we can look for a way to provide the programmer with some means to pass, when desired, unevaluated definitions as subexpressions of the process abstraction, in order to be (or not to be) evaluated by the child process. A natural way to do this is to encapsulate the expressions within λ-abstractions. For example, if the programmer desires that some subexpression e_y (bound to y, a free variable of the process abstraction) is to be evaluated by the child process —instead of the parent process—, then it can be encapsulated as $\backslash dummy.e_y$, and bound to y'; the variable y must be substituted by $y'(\backslash x.x)$ in the abstraction. Thus, though the option is EBC_p=yes, the parent will not evaluate e_y.

Eager	**Lazy**
$\backslash x.e$	$\backslash x.e'$
$y \in Free(e)$	$y' \mapsto \backslash dummy.e_y$
$y \mapsto e_y$	$e' = e[y'(\backslash x.x)/y]$

At the opposite extreme, the combination PAE=child, EBC_p=no, EBC_v=no, IC_p=yes, and IC_v=yes ((3)(b) in Table 1) can be considered as the laziest one.

The load balance may be better under this strategy in the cases where a parent process does not share variables with its children and the children themselves do not share variables between them, because the parent has not to do all the work, but can divide it among its children. In Example 5.3 we showed that load balance is not well achieved when EBC_p=yes. Moreover, as the parent does not evaluate the free variables necessarily, less time is needed to create each child, although the real gain depends on factors such as the work necessary to evaluate the free variables, the amount of graph to be packed, etc. We wonder how often this kind of situation occurs. We think that this problem can be solved methodologically if the programmer tries not to use free variables or, at least, to use free variables that do not require a big amount of work. In such cases, the performance is nearly the same for both options of EBC_p.

To gain efficiency in this approach, we can provide the programmer with some means for evaluating the free variables eagerly at the parent side. This would allow to share values, and also to send less work and/or data when packing the closures for the child's heap.

There are two ways of introducing eagerness:

- Sending the expressions (evaluated) bound to free variables through channels.
- Using the functions `nf` (evaluation of an expression to normal form) and `seq` (strict sequential composition).

The problem with the first approach is that currying is lost because in Eden parameters can be curried while channels cannot. The second alternative is not as elegant as the first one, but it preserves currying. The programmer only needs to force the evaluation of each free variable that it is desired to be evaluated before the creation of the process.

For these reasons, we think that the evaluation strategy (3)(b) is the best option and it favors parallelism, as processes are not burdened with much work before creating a child. In fact, Eden is implemented according to this strategy.

In the case of a sharing memory setting, an option could be considered

where the evaluation of free variables in process abstractions (and communications) is carried out by the parent, but only on demand. This would avoid work duplication.

As a future task, we will investigate how to apply similar techniques to other parallel functional languages like GpH. Besides, it is our intention to widen the calculus in order to include other Eden features such as streams. In this way, we will be able to extend the work in [7] to consider all the semantic options explained in this paper, and thus to analyze the influence of these evaluation strategies on Eden skeletons.

Acknowledgement

This work is much indebted to former members of the Eden Group at Madrid and a lively discussion on eagerness vs. laziness. We are also grateful to Fernando Rubio for contributing with the title, some ideas, and invaluable support.

References

[1] C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173, Montreal, Canada, September 2000.

[2] M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP*, pages 10–18, 2007.

[3] M. J. Gabbay, S. H. Haeri, Y. Ortega-Mallén, and P. W. Trinder. Reasoning about selective strictness: operational equivalence, heaps and call-by-need evaluation, new inductive principles. (Work in progress available from authors), 2009.

[4] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo.* PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[5] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[6] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Analyzing the influence of mixed evaluation on the performance of Eden skeletons. *Parallel Computing*, 32(7-8):523–538, 2006.

[7] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Comparing alternative evaluation strategies for stream-based parallel functional languages. In *Proceedings of the 18th International Workshop on Implementation of Functional Languages, (IFL'06 selected papers)*, pages 55–72. LNCS 4449, Springer, 2007.

[8] K. Ibach. Ozfun: A functional language for mixed eager and lazy programming. In Jean-Luc Cochard, editor, *International Workshop on Oz Programming*, pages 87–92. IDIAP, 1995.

[9] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP'77*, pages 993–998. Eds. B. Gilchrist. North-Holland, 1977.

[10] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, 1989.

[11] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.

[12] R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs, pages 63–92. Eds. K. Hammond and G. Michaelson. Springer, 1999.

[13] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[14] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[15] S. L. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[16] F. S. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Imperial College, 1997.

[17] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Funcional Programming*, 8(1):23–60, 1998.

[18] P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4+5):469–510, 2003.

[19] M. van Eekelen and M. de Mol. Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics. In *Draft Proceedings of the 14th International Workshop on Implementation of Functional Languages, IFL'02*, pages 357–373. Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2002.

[20] M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, 2007.

Appendix

Local rules are included in Figure 5 and global rules in Figure 6, but for process creation and communication that were explained in Section 4.

$$H + \{x \overset{I}{\mapsto} \backslash x.E\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{I}{\mapsto} \backslash x.E, \theta \overset{A}{\mapsto} \backslash x.E\} \quad \textbf{(value)}$$

$$\text{si } E \not\equiv \backslash x.E' \qquad H + \{x \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\} \quad \textbf{(demand)}$$

$$H : x \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{B}{\mapsto} x\} \quad \textbf{(blackhole)}$$

$$\text{si } E \not\equiv \backslash x.E' \qquad H + \{x \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\, y\} \quad \textbf{(app-demand)}$$

$$H + \{x \overset{I}{\mapsto} \backslash z.E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{I}{\mapsto} \backslash z.E, \theta \overset{A}{\mapsto} E[y/z]\} \quad (\beta\textbf{-reduction})$$

$$1 \leq i \leq n, fresh(y_i)$$
$$H : \theta \overset{A}{\mapsto} \texttt{let } \{x_i = E_i\} \texttt{ in } x \longrightarrow \quad \textbf{(let)}$$
$$\longrightarrow H + \{y_i \overset{I}{\mapsto} E_i[y_1/x_1, \ldots, y_n/x_n]\}_{i=1}^{n} + \{\theta \overset{A}{\mapsto} x[y_1/x_1, \ldots, y_n/x_n]\}$$

Figure 5. Local rules

The parallel evolution of all the processes in the system is achieved by rules **(local parallel)** and **(parallel)**:

$$\textbf{(local parallel)}$$
$$\frac{\{H_i^1 + H_i^2 : \theta_i \overset{A}{\mapsto} E_i \longrightarrow H_i^1 + K_i^2 | H = H_i^1 + H_i^2 + \{\theta_i \overset{A}{\mapsto} E_i\} \wedge \theta_i \overset{A}{\mapsto} E_i \in \mathcal{EB}(H)\}_{i=1}^{n}}{H \overset{lpar}{\longrightarrow} (\cap_{i=1}^{n} H_i^1) \cup (\cup_{i=1}^{n} K_i^2)}$$

(WHNF unblocking)

$$(S, \langle p, H + \{x \overset{A}{\mapsto} \backslash x.E', \theta \overset{B}{\mapsto} E_B^x\}\rangle) \overset{wUnbl}{\longrightarrow} (S, \langle p, H + \{x \overset{A}{\mapsto} \backslash x.E', \theta \overset{A}{\mapsto} E_B^x\}\rangle)$$

(WHNF deactivation)

$$(S, \langle p, H + \{\theta \overset{A}{\mapsto} \backslash xz.E\}\rangle) \overset{deact}{\longrightarrow} (S, \langle p, H + \{\theta \overset{I}{\mapsto} \backslash z.E\}\rangle)$$

(blocking process creation)

$$(S, \langle p, H + \{\theta \overset{IA}{\mapsto} x \,\#\, y\}\rangle) \overset{bpc}{\longrightarrow} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x \,\#\, y\}\rangle)$$

(process creation demand)

$$\text{si } y \overset{I}{\mapsto} E \in pdfv(x_1, H)$$
$$(S, \langle p, H + \{\theta \overset{B}{\mapsto} x_1 \,\#\, x_2\}\rangle) \overset{pcd}{\longrightarrow} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x_1 \,\#\, x_2, y \overset{A}{\mapsto} E\}\rangle)$$

(value communication demand)

$$\text{si } x \overset{I}{\mapsto} E \in dfv(\backslash x.E, H)$$
$$(S, \langle p, H + \{ch_o \overset{I}{\mapsto} \backslash x.E\}\rangle) \overset{vComd}{\longrightarrow} (S, \langle p, H + \{ch_o \overset{I}{\mapsto} \backslash x.E, x \overset{A}{\mapsto} E\}\rangle)$$

Figure 6. Scheduling rules

where $n = |\mathcal{EB}(H)|$, i.e. the number of evolvable bindings.

(parallel)

$$\frac{\{H_p \overset{lpar}{\longrightarrow} H'_p\rangle\}_{\langle p, H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}$$

The evolution of the system is defined by the following sequence, where $\overset{\dagger}{\Longrightarrow}$ stands for the reflexive transitive closure of $\overset{\dagger}{\longrightarrow}$:

$$\Longrightarrow = \overset{par}{\Longrightarrow} ; \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow}$$