Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 47–68
www.elsevier.com/locate/entcs

Dealing Denotationally With Stream-based Communication

Mercedes Hidalgo-Herrero[1,2]

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

Yolanda Ortega-Mallén[1,3]

Dept. Sistemas Informáticos y Programación
Facultad de CC. Matemáticas, Universidad Complutense de Madrid.
Madrid, Spain

Abstract

We define a denotational semantics for a kernel-calculus of the parallel functional language Eden. We choose continuations to deal with side-effects (process creation and communication) in a lazy context. The calculus includes streams for communication, and their modelization by a denotational semantics is not direct because a stream may be infinite.

Keywords: Denotational semantics, continuation semantics, laziness, parallel programming, functional programming, Eden.

1 Introduction

Assuming that parallelism and distribution are efficiency improvements in programming, the main goal for designing Eden [7] was to profit from both of them in a functional paradigm. Eden extends the functional language Haskell

[1] Work partially supported by the Spanish project TIC2003-01000.
[2] E-mail: mhidalgo@edu.ucm.es
[3] E-mail: yolanda@sip.ucm.es

1571-0661
doi:10.1016/j.entcs.2005.01.039

[10] with constructs for defining explicit processes, so that the Eden programmer controls —from a higher level of abstraction— the process granularity, the data distribution, and the process topology. This circumstance is endorsed by the fact that the programmer has not to worry about synchronization tasks.

The language Eden comprises two layers: the functional level, or computational model, and the processes level, or coordination model [1]. The computational model is the lazy functional language Haskell, while the coordination level includes the following features:

Process abstractions: expressions that define the general behaviour of a process in a purely functional way.

Process creations: the application of some process abstraction to a particular group of expressions produces the creation of a new process to compute the result of that application.

Interprocess communications: these are asynchronous and implicit, since the programmer does not need to specify the message passing. Communications in Eden are not restricted to the transmission of a single value, processes can communicate values in a stream-like manner.

Eden also includes some constructs to model reactive systems:

Dynamic creation of channels: without this possibility communications are only hierarchical, i.e. from parent to child and viceversa. Dynamic channels facilitate the creation of more complex communication topologies [9].

Non-determinism: in order to model communications from many-to-one Eden introduces a predefined process, `merge`, whose inputs are several streams while its output is just one stream; the latter is the non-deterministic merge of the elements of the former.

The introduction of parallelism leads to a certain loss of laziness:

- Processes are eagerly created even if the output of the new process has not still been demanded.
- Communication is achieved even without demand; whenever a process is created, it is initiated the evaluation of the expressions which will yield the values to be communicated through its channels.

In general, the evaluation of an expression comes to an end when a weak head normal form (whnf) is reached. However, when this value has to be communicated and it is not a λ-abstraction, it will be evaluated to normal form. On the one hand, the head of a stream is strict, so that it is evaluated until a communicable value is obtained. On the other hand, the whole stream evaluation is lazy, allowing in this way the existence of potentially infinite

streams.

Our aim in this work is to define a formal semantics to model the main characteristics of Eden. The semantics should consider the functional side of Eden as well as the parallelism and the distribution of the computation. In order to be used by a programmer, it is not necessary to include operational details. Consequently, we consider that it is more suitable to define a denotational semantics. Nevertheless, the chosen denotational model is not a direct denotational semantics, but a continuations denotational model [14] where the semantic value of an expression is a function that transforms a state into another state. This decision is motivated by the wish of modelling together the laziness of the computational kernel of Eden and the *side-effects* resulting from process creations and communications. This kind of semantics has been defined before to express the meaning of laziness in [5]; in that work, the author takes also into account side-effects, like the printing of results. In the case of Eden, the computed value produced by a process creation is the same as if the corresponding functional application was evaluated, but apart from obtaining the corresponding value, a new process is created. This creation together with the underlying communications are considered as side-effects. In the denotational model that we propose here a continuation will take into account these side-effects.

Thanks to the stream-based communication in Eden, processes can be defined whose behaviour is similar to introducing continuously characters with a keyboard. However, the modelization of streams by a denotational semantics is not direct because a stream may be infinite and the denotational value for such a stream could not be computed. In the model of continuations the semantic value for an expression is a function that is well defined although its computational effect may be infinite.

We are interested in observing three different aspects of an Eden program:

Functionality: the final value computed.

Parallelism: the system topology, i.e. the existing processes and connections among them, and the interactions generated by the computation.

Distribution: the degree of work duplication and speculative computation.

The degree of speculative computation depends on the number of available processors, the scheduling decisions and the speed of basic instructions. Therefore, our model embodies two bounds, a minimal, in which only the necessary expressions are evaluated, and a maximal, where every expression is evaluated.

The same semantic model that we develop in this paper was first used in [4] where it was applied to a reduced kernel of Eden that included a simple

form of non-determinism but not streams. Besides, in that work the same model was used for the languages GPH and pH, thus obtaining a common framework to compare three different ways of introducing parallelism in the functional language Haskell.

For the definition of the formal semantics we consider an Eden simplification consisting of a lazy λ-calculus extended with explicit process creation and lists. The calculus is presented in Section 2, while Section 3 is devoted to the definition of the denotational semantics: semantic domains and semantic functions. Finally, we show our conclusions and outline future work in Section 4.

2 The Calculus

The syntax of our calculus is detailed in Figure 1. The simple calculus that we consider here embodies Eden's essentials: a basic λ-calculus with variables, functional abstractions and applications, and local declarations, extended with process creations (parallel application `#`) and lists for modelling streams. [4]

Terms	**Restricted terms**	
$E ::= x$	x	variable
$\mid \backslash x.E$	$\backslash x.E$	λ-abstraction
$\mid E_1 E_2$	$x_1\, x_2$	application
$\mid E_1 \texttt{\#} E_2$	$x_1 \texttt{\#} x_2$	parallel application
$\mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration
$\mid \Pi[x_1 : x_2].E_1 \Vert E_2$	$\Pi[x_1 : x_2].E_1 \Vert E_2$	pattern matching
$\mid L$	L	list
$L ::= \texttt{nil}$	`nil`	empty list
$\mid [E_1 : E_2]$	$[x_1 : x_2]$	non-empty list

Figure 1. Syntax

The effect of evaluating an expression like $E_1 \texttt{\#} E_2$ is the creation of a new process and of two channels communicating parent and child, as it is illustrated in Figure 2. The child will receive from its parent the result of evaluating E_2, and then it will send the value corresponding to the application $E_1\, E_2$. Notice that communication is asynchronous and implicit: there are not communication primitives such as `send` or `receive`, like in CML [11] or in the π-calculus [12], and that are considered harmful by some authors [2]. When the second

[4] Non-determinism is not considered here because its combination with streams would make the semantic definitions more cumbersome.

argument for the parallel application is a list, the corresponding input communication channel is a stream. The same rule applies for the output. The evaluation of a stream is not completely lazy: the constructor and the head must be obtained. In this way, the evaluation is element-wise.

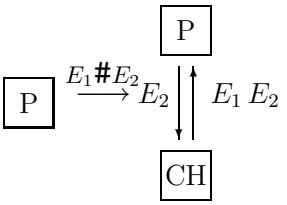


Figure 2. Process creation scheme

From the point of view of the obtained final value, a parallel application is equivalent to a functional application. However, an operational observation of the computation shows that they are qualitatively different:

- The evaluation of any functional application argument is lazy, whereas it is eager in the case of a parallel application, i.e. the input of the new process.
- With respect to the application, the free variables of the abstraction are evaluated only under demand. However, when evaluating a parallel application there are two possibilities: either to evaluate them only if it is strictly necessary, or to generate an initial environment for the new process by previously evaluating the needed variables.

In order to facilitate the implementation in a distributed setting, Eden processes are independent entities, each one with its own memory containing all the information needed to evaluate its main output. Therefore, when a process is created it is associated to some initial environment where all the free variables of the abstraction, i.e. the body of the process, are defined. This initial environment may be built following two different alternatives: (a) Every variable is associated to a final value, or (b) the variables may be bound to expressions which are still unevaluated. The latter gives rise to a potential duplication of work, whereas the former assures that each variable is only evaluated once. In our case we have chosen the first option and, consequently, when a copy of variables takes place all of them must be already evaluated, that is, all the dependencies of free variables must be previously evaluated.

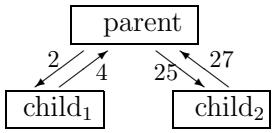
The local declaration of variables makes possible the parallel execution of several processes. The explicit parallelism, which is introduced by the #-expressions, is mostly exploited when some of the local variables are bound to parallel applications (*top-level*), because these processes will be created

without other previous demand, i.e. speculatively.

The next example illustrates the situations where processes are created. [5]

Example 1 *Process creation*

$$\texttt{let}\quad x = y\#25, y = \backslash z.(z+2)\quad \texttt{in}\ y\#2$$



This expression gives rise to the creation of two processes: the first child ($child_1 \equiv y\#2$) is created under demand, whereas the second ($child_2 \equiv y\#25$) is developed speculatively. Depending on the evaluation conditions, the second process finally produces its result value or not.

□

We introduce a special form of abstraction ($\Pi[x_1 : x_2].E_1 \| E_2$) for *lists*. This construction just does a pattern matching when applied to a list:

- Empty list: the evaluation goes on with E_1.
- Non-empty list: E_2 is evaluated after the substitution of the formal parameters by the actual arguments.

Notice that this application is strict in its argument in order to distinguish between an empty list and a non-empty list. Consequently, we consider three kinds of application: lazy (functional), eager (parallel), and strict (pattern matching).

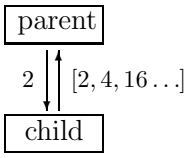
The evaluation of a list —not a stream— is lazy, and we consider sufficient to obtain its constructor (whnf).

The following example shows the creation of an infinite stream.

Example 2 *Infinite streams.*

The following expression gives place to two processes, one generates a stream where each element is the square of its precedent; while the other adds up all the elements of that stream.

$$\texttt{let}\ s = \backslash z.\Pi[y_1 : y_2].z \| (s\,(z + y_1)\,y_2)\quad \texttt{in}\ s\,0\,(\texttt{let}\ p = \backslash x.[x : (p\,(x * x))]\quad \texttt{in}\ (p\,\#\,2))$$



The output of the parent process, i.e. the sum of the stream produced by the child, will never be obtained because of the infiniteness of the stream,

[5] In order to make the examples more understandable, we have considered natural numbers and arithmetic operators as syntactic sugar for the λ-expressions representing them.

although the partial sums are obtained in z, the argument for s.

□

2.1 Normalization

Before defining the meaning of an expression of the given calculus, a process of normalization [6] is carried out. The normalization introduces variables and local declarations in order to transform into variables both subexpressions of an application, as well as the body of a local declaration. Both the head and the tail of a non-empty list become variables too. With these transformations every subexpression gets shared, and, therefore, it is evaluated only once. This is an example of the evaluation based on the *full laziness*. Moreover, the fact that the components of applications are variables simplifies the semantic functions because it is not necessary to introduce fresh variables to model laziness. The restricted syntax was also included in the Figure 1.

Once we have described our calculus informally, we give its formal definition using a continuations-based denotational model.

3 A denotational semantics

The definition of the denotational semantics is done in three stages: definition of the semantic domains, definition of the evaluation function, and definition of the needed auxiliary functions.

3.1 The semantic domains

The semantic domains of our model are given in the Figure 3.

3.1.1 Continuations

A continuation is a function which contains the details of the remainder of the program, that is, if an expression is inside a context, the information about this context is gathered in the continuation. In order to execute all the tasks derived from the context, a continuation is a *state transformer*, i.e. the function is applied to a state that is transformed according to the context information contained in the continuation. We can distinguish two kinds of continuations:

Command continuations: the computation of an imperative program is reflected in the transformations on the state caused by the execution of its instructions, but there is not a returned value. A command continuation behaves similarly, that is, it takes an state and modifies it without any information about returned values.

$c \in \mathbf{Cont}$	$= \mathbf{State} \to \mathbf{State}$	continuations
$\kappa \in \mathbf{ECont}$	$= \mathbf{EVal} \to \mathbf{Cont}$	expression continuations
$s \in \mathbf{State}$	$= \mathbf{Env} \times \mathbf{SChan}$	states
$\rho \in \mathbf{Env}$	$= \mathbf{Id} \to (\mathbf{Val} + \{\text{undefined}\})$	environments
$\upsilon \in \mathbf{Val}$	$= \mathbf{EVal} + (\mathbf{IdProc} \times \mathbf{Clo}) + \{\text{not\_ready}\}$	values
$\varepsilon \in \mathbf{EVal}$	$= (\mathbf{Abs} \times \mathbf{Ids}) + (\mathbf{Id} \times \mathbf{Id}) + \{\mathsf{nil}\}$	expressed values
$\alpha \in \mathbf{Abs}$	$= \mathbf{Id} \to \mathbf{Clo}$	abstraction values
$\nu \in \mathbf{Clo}$	$= \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Cont}$	closures
$\mathrm{sch} \in \mathbf{SChan}$	$= \mathcal{P}_f(\mathbf{Chan})$	sets of channels
$ch \in \mathbf{Chan}$	$= \mathbf{IdProc} \times ((\mathbf{Id} + \{\text{closed}\}) \times \mathbf{SCVal}^{<>}) \times \mathbf{IdProc}$	channels
$\mathrm{cv} \in \mathbf{CVal}$	$= \mathbf{Abs} + \mathbf{CList}$	communicable values
$\sigma \in \mathbf{CList}$	$= \{\mathsf{nil}\} + (\mathbf{CVal} \times \mathbf{CList})$	communicable lists
$\mathrm{scv} \in \mathbf{SCVal}$	$= \mathbf{CVal}^+$	sequences
$\mathrm{scw} \in \mathbf{SCVal}^{<>}$	$= \mathbf{SCVal} + \{<>\}$	communicable values
$I \in \mathbf{Ids}$	$= \mathcal{P}_f(\mathbf{Id})$	sets of identifiers
$p, q \in \mathbf{IdProc}$		process identifiers

Figure 3. Semantic domains

Expression continuations: the evaluation of a functional program yields a value. An expression continuation transforms the state taking into account this returned value; that is, it takes the value as its argument and then behaves like a command continuation.

In our denotational model we use both types of continuations, so that a *continuation*, $c \in \mathbf{Cont}$, transforms a state into another state, while an *expression continuation*, $\kappa \in \mathbf{ECont}$, takes a value, obtained from the evaluation of an expression, and yields a continuation.

3.1.2 States

Usually, a *state* is a dynamic entity representing the place where the program evaluation produces irreversible changes; they are considered irreversible because it would be very expensive to keep copies of the whole state in order to be able to return to previous versions after leaving a block. On the other hand, an *environment* is defined locally for each program block, so that its local

references disappear after leaving each block. Environments are much smaller than states and its copy is therefore feasible. This separation between the environment and the state is usual in the imperative languages, or in those languages that include imperative instructions —such as variable assignment—. However, in a functional language, where each variable is bound only once to a value and this association is never broken, it is more suitable to rename the variables and manage a combination state/environment, considering only one environment for the whole program instead of partial environments for each block.

Besides an environment, a state, $s \in \mathbf{State}$, includes a set of channels, $\mathrm{sch} \in \mathbf{SChan}$, representing the actual process topology by defining a graph whose nodes are the processes, and the edges are labelled by the values that have been communicated through those channels.

As usual, an *environment*, $\rho \in \mathbf{Env}$, bounds identifiers in **Id** to values; the value undefined indicates that the identifier has not been used yet.

3.1.3 Values

There are three possible evaluation states of a variable:

(i) It has been completely evaluated: the variable is associated to the value returned after evaluating the previously associated expression.

(ii) It has not been demanded yet: the identifier is associated to a closure.

(iii) Its evaluation has been demanded but not finished yet: the evaluation will go wrong if this variable is demanded again (*self-reference*).

In the first case, the identifier is bound to an *expressed value*, $\varepsilon \in \mathbf{EVal}$, or final value that includes *abstraction values*, $\alpha \in \mathbf{Abs}$. Each denotational abstraction has associated a set of identifiers corresponding to the free variables of the syntactic abstraction.[6] Why do we need these free variables? Because everything that is *copied* must have been evaluated previously.

The domain **EVal** also includes the denotational value nil corresponding to the empty list and the domain of non-empty lazy lists. In normalized expressions (see Section 2.1) every subexpression is shared, and we do not want to lose this property in this denotational framework. Therefore, non-empty lists are represented by a pair of identifiers, $\mathbf{Id} \times \mathbf{Id}$: the former is bound to the value of the head, while the latter is associated to the value of the tail.

[6] Although this set could be computed from the denotational value of the abstraction, that task would be much more complex.

A *closure*, $\nu \in \mathbf{Clo}$, is associated to a process identifier corresponding to the process which introduced the variable in the environment. This process will be the parent in the case that the evaluation of that variable generates a new process. However, a closure does not need this information for generating its own descendants; in this case it is necessary to know in which process the closure is being evaluated. This information will be passed to the closure via an argument in the semantic function. Notice that this process identifier is not needed to partition the environment into different processes, and that the copy of variables is only virtual, that is, they are shared instead of being copied explicitly.

Example 3 *Process identifiers and closures.*

Let us observe the environment

$$\{x_1 \mapsto \lambda y.\nu_1, x_2 \mapsto \langle p, \nu_2 \rangle, x_3 \mapsto \langle p, \mathcal{E}\,[\![x_1 \# x_2]\!]\,\rangle\}.$$

The 'owner' of x_3 is p. However, the evaluation of the closure associated to x_3 implies the creation of a new process, let us say q; so that the application $x_1\,x_2$ will be evaluated inside q, instead of p. We keep the identifier p because x_2 has to be evaluated inside p, and the process creations caused by the evaluation of x_2 will be descendant of p; but any process generated by $x_1\,x_2$ a will be a child of q.

□

A closure is only meaningful once the corresponding process identifier has been provided. Nevertheless, it is still necessary to establish its context; for this purpose the closure has to receive an expression continuation. In this way, the closure becomes a state transformer.

When a variable is bound to the special value not_ready that means that the variable is being evaluated. If after applying an initial continuation to a state, the final state still contains a variable bound to not_ready, then a self-reference —direct or indirect— has been detected.

3.1.4 Streams

The reason for introducing lists in our calculus is to model communication stream-channels. A non-stream channel is closed once it has sent its unique value; by contrast, a stream is kept open until the value nil is communicated. Thus, our denotational model needs some mechanism to express the channel state, that is either open or closed:

Open: the capability of going on communicating values through the channel is modelled by introducing in the denotational value an identifier, or variable, which corresponds, in the environment, to the list-stream obtained (or to

the closure which, after being evaluated, will give place to the list-stream).

Closed: the identifier is replaced by the special value closed.

It follows that the representation of a *channel*, $ch \in \mathbf{Chan}$, is a 4-tuple, containing the information mentioned above. Besides, as streams may be infinite, the value sent through a channel is represented as a sequence of values, $scw \in \mathbf{SCVal}^{<>}$, which is built with communicable values, $cv \in \mathbf{CVal}$; initially, this sequence is empty, and represented by $<>$.

Communicable values also include abstractions (for non-stream channels). In the communication channel this single value is represented by a stream with only one element. In case of a stream-channel, the value will be the whole sequence of sent values. These values belong to the domain **CList** of communicable lists: each of these may be either the empty list or a list whose head is a communicable value.

3.2 The evaluation function

We have already mentioned the necessity of indicating the process where an expression is going to be evaluated, that is, the parent of the potential new processes. Consequently, the signature of the semantic function for the evaluation of expressions is as follows:

$$\mathcal{E} :: \mathbf{Exp} \to \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Cont}.$$

The expression continuation contains the information needed to use the value returned by the evaluation of the expression, and thus accumulating the effects of evaluating the given expression to those of the expression continuation.

The meaning of a 'program' E corresponds to the evaluation of the special variable *main* in an initial environment with just the binding $main \mapsto \langle p_{main}, \mathcal{E}\,[\![E]\!]\,\rangle$, where p_{main} stands for the main process in the state.

The definition of the evaluation function is shown in the Figures 4, 5, 6 and 7, where we use the operator $\oplus_e$ to extend/update environments, such as in $s \oplus_e \{x \mapsto \langle p, \nu \rangle\}$.[7]

3.2.1 Basic λ-calculus

The evaluation of an *identifier* 'forces' the evaluation of the associated value in the given environment. The definition of the function force appears in the Figure 8 and it will be explained later on in the Section 3.3.1.

For the evaluation of a *λ-abstraction* the corresponding expressed value has to be built: the denotational abstraction —a function that takes an identifier

[7] $\oplus_{ch}$ in the case it is an extension/update of the set of channels of a state, and $\oplus$ when both components of the state are extended/updated.

$$
\begin{array}{l}
\mathcal{E}\,[\![x]\!]\,p\,\kappa = \mathsf{force}\,x\,\kappa \\
\mathcal{E}\,[\![\backslash x.E]\!]\,p\,\kappa = \kappa\langle \lambda x.\mathcal{E}\,[\![E]\!]\,, \mathsf{fv}(\backslash x.E)\rangle \\
\mathcal{E}\,[\![x_1\,x_2]\!]\,p\,\kappa = \mathcal{E}\,[\![x_1]\!]\,p\,\kappa' \\
\quad \texttt{where}\ \kappa' = \lambda\varepsilon.\lambda s.\texttt{case}\ \varepsilon\ \texttt{of} \\
\qquad\qquad \langle \alpha, I\rangle \in \mathbf{Abs} \times \mathbf{Ids} \longrightarrow (\alpha\,x_2)\,p\,\kappa\,s \\
\qquad\qquad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad \texttt{endcase}
\end{array}
$$

Figure 4. Evaluation function $\mathcal{E}$: basic λ-calculus

and returns a closure— is put together with the set of free variables (fv) of the syntactic abstraction. Afterwards, the given expression continuation is applied to this semantic value.

Since *functional application* in the calculus is lazy, the evaluation of its argument is delayed, and it will take place only if it is demanded. In order to model this behaviour, the given expression continuation κ is substituted by κ', i.e. the expression continuation for evaluating the variable corresponding to the abstraction, x_1. In this way, once the value of the abstraction has been obtained, α, it is applied to the argument variable x_2, and the resulting closure is evaluated with the expression continuation κ.

3.2.2 Parallel application

The evaluation of a *parallel application* implies the creation of a process with two communication channels: one 'input' from parent to child and one 'output' from child to parent. Each channel has a variable associated whose evaluation will give place to the (first) value to be communicated. In the following we designate by i the variable corresponding to the input channel, while o is for the output.

The evaluation of a parallel application goes through the following 'stages':

(i) To force the evaluation of the variable x_1 corresponding to the abstraction as well as all its free dependencies the function $\mathsf{forceFV}$ (see Figure 9 and Section 3.3.1) is used.

(ii) The context of x_1 is gathered in the expression continuation κ', which creates two new channels communicating the parent p and the new child q, one suspended in i and the second waiting in o. Both of them are created with the initial value $<>$.

(iii) To evaluate the application which is bound to the variable o. This evaluation yields a value that is communicated from the new process to its

$\mathcal{E}\,[\![x_1 \# x_2]\!]\,p\,\kappa = \mathsf{forceFV}\,x_1\,\kappa'$ (i)

where $\kappa' = \lambda\varepsilon.\lambda s.\mathtt{case}\ \varepsilon\ \mathtt{of}$ (ii)

$\langle \alpha, I\rangle \in \mathbf{Abs} \times \mathbf{Ids} \longrightarrow \mathsf{forceFV}\,o\,\kappa''\,s'$ (iii)

where $q = \mathsf{newIdProc}\,s$

$\{i, o\} = \mathsf{newId}\,2\,s$

$s' = s \oplus \langle \{o \mapsto \langle q, (\alpha\, i)\rangle, i \mapsto \langle p, \mathcal{E}\,[\![x_2]\!]\,\rangle\}, \{\langle p, i, <>, q\rangle, \langle q, o, <>, p\rangle\}\rangle$

$\kappa''_{min} = \lambda\varepsilon'.\lambda s''.\kappa\,\varepsilon'\,s''\ (= \kappa)$
(iv) and (v)
$\kappa''_{max} = \lambda\varepsilon'.\lambda s''.\kappa\,\varepsilon'\,s_o$
where $s_i = \mathsf{forceFV}\,i\,(\mathsf{kstr}\ i)\,s''$
$s_o = \mathsf{forceFV}\,o\,(\mathsf{kstr}\ o)\,s_i$

otherwise $\longrightarrow$ wrong

endcase

Figure 5. Evaluation function $\mathcal{E}$: parallel application

parent. These tasks are carried out by the function forceFV.

(iv) In the case of a *maximal semantics*, after evaluating the application, if the channel is a stream, i.e. the identifier is bound to a list, then it must be evaluated completely. Again, the function forceFV is invoked, and using the expression continuation kstr x (see Section 3.3.2), each component of the stream is forced, including the free dependencies as well.

(v) Moreover, in the special case of a *maximal semantics*, both channels have to be evaluated completely. Therefore the expression continuation forces i as well as o.

The functions newId and newIdProc return fresh variables and fresh process identifiers, respectively.

3.2.3 Local declaration

Before evaluating the body of a *local declaration of variables* it is compulsory to introduce the declared variables into the environment. A renaming is necessary to avoid name clashes. For each x_i $(1 \leq i \leq n)$, a fresh variable y_i is introduced, which is associated to the closure $\mathcal{E}\,[\![E_i[y_j/x_j]]\!]$. Each of these closures is labelled by p, the identifier corresponding to the process where the evaluation takes place. The different levels of speculation are achieved by the new expression continuation κ'. To model a minimal semantics, κ' must create the top-level processes encountered in the local declaration, but the values of

$$\mathcal{E}\,[\![\texttt{let}\ \{x_i = E_i\}_n\ \texttt{in}\ x]\!]\,p\,\kappa = \lambda s.\mathcal{E}\,[\![x]\!]\,p\,\kappa'\,s'$$
$$\texttt{where}\ \{y_1,\ldots,y_n\} = \mathsf{newId}\,n\,s$$
$$s' = s \oplus_e \{y_i \mapsto \langle p, \mathcal{E}\,[\![E_i[y_1/x_1,\ldots,y_n/x_n]]\!]\,\rangle \mid 1 \le i \le n\}$$

$$\kappa'_{min} = \lambda\varepsilon.\lambda\langle\rho,\mathrm{sdh}\rangle.\kappa\,\varepsilon\,s''$$
$$\texttt{where}\ I = \{y_i \mid E_i \equiv x_1^i \#x_2^i \wedge (\rho\,y_i) \in \mathbf{IdProc}\times\mathbf{Clo} \wedge 1 \le i \le n\}$$
$$m = \mathsf{card}\,I$$
$$\{q_1,\ldots,q_m\} = \mathsf{newIdProc}\,m\,\langle\rho,\mathrm{sdh}\rangle$$
$$s'' = s \oplus_{ch} \{\langle p,\mathrm{closed},<>,q_j\rangle, \langle q_j,\mathrm{closed},<>,p\rangle \mid 1 \le j \le m\}$$

$$\kappa'_{max} = \lambda\varepsilon.\lambda s.\kappa\,\varepsilon\,s_f$$
$$\texttt{where}\ I = \{y_i \mid E_i \equiv x_1^i \#x_2^i \wedge 1 \le i \le n\}$$
$$s_f = \mathsf{mforce}\,I\,s$$

Figure 6. Evaluation function $\mathcal{E}$: local declaration

$$\mathcal{E}\,[\![\Pi[y_1 : y_2].E_1 \| E_2]\!]\,p\,\kappa = \kappa\langle\alpha, I\rangle$$
$$\texttt{where}\ \alpha = \lambda x.\lambda p'.\lambda\kappa'.\lambda s.\texttt{case}\ (\rho_a\,x)\ \texttt{of}$$
$$\mathsf{nil} \longrightarrow \mathcal{E}\,[\![E_1]\!]\,p\,\kappa'\,\langle\rho_a,\mathrm{sdh}_a\rangle$$
$$\langle z_1, z_2\rangle \longrightarrow \mathcal{E}\,[\![E_2[z_1/y_1, z_2/y_2]]\!]\,p\,\kappa'\,\langle\rho_a,\mathrm{sdh}_a\rangle$$
$$\texttt{otherwise} \longrightarrow \mathsf{wrong}$$
$$\texttt{where}\ \langle\rho_a,\mathrm{sdh}_a\rangle = \mathsf{force}\,x\,id_\kappa\,s$$
$$\texttt{endcase},$$
$$I = \mathsf{fv}(\Pi[y_1 : y_2].E_1 \| E_2)$$
$$\mathcal{E}\,[\![\texttt{nil}]\!]\,p\,\kappa = \kappa\,\mathsf{nil}$$
$$\mathcal{E}\,[\![[x_1 : x_2]]\!]\,p\,\kappa = \kappa\,\langle x_1, x_2\rangle$$

Figure 7. Evaluation function $\mathcal{E}$: lists

their channels are not demanded. In the case of a maximal semantics, every local variable which was bound originally to a parallel application is forced; and this causes the creation of new processes and channels. The function mforce (*multiple forcing*) forces a set of identifiers; therefore, its definition, not detailed here, invokes repeatedly the function force (see Section 3.3.1). The function card calculates the cardinal of a set.

3.2.4 List evaluation

The evaluation function $\mathcal{E}$ for *pattern matching* is similar to that of the functional abstraction. The only difference, due to the strictness of a pattern-

$$\mathsf{force} :: \mathbf{Id} \to \mathbf{ECont} \to \mathbf{Cont}$$

$$\mathsf{force}\, x\, \kappa = \lambda\langle \rho, \mathrm{sdh}\rangle.\mathtt{case}\ \langle p, x, \mathrm{cv}, q\rangle \in \mathrm{sdh}\ \mathtt{of}$$

$$\text{true} \longrightarrow \mathsf{forceFV}\, x\, \kappa$$

$$\text{false} \longrightarrow \mathsf{forceS}\, x\, \kappa$$

$$\mathtt{endcase}$$

$$\mathsf{forceS} :: \mathbf{Id} \to \mathbf{ECont} \to \mathbf{Cont}$$

$$\mathsf{forceS}\, x\, \kappa = \lambda\langle \rho, \mathrm{sdh}\rangle.\mathtt{case}\ (\rho\, x)\ \mathtt{of}$$

$$\varepsilon \in \mathbf{EVal} \longrightarrow \kappa\, \varepsilon\, \langle \rho, \mathrm{sdh}\rangle$$

$$\langle p, \nu\rangle \in (\mathbf{IdProc} \times \mathbf{Clo}) \longrightarrow \nu\, p\, \kappa'\, s''$$

$$\mathtt{where}\ s = \langle \rho, \mathrm{sdh}\rangle$$

$$\kappa' = \lambda\varepsilon'.\lambda s'.$$

$$\kappa\, \varepsilon'\, (s' \oplus_e \{x \mapsto \varepsilon'\})$$

$$s'' = s \oplus_e \{x \mapsto \mathrm{not\_ready}\}$$

$$\mathtt{otherwise} \longrightarrow \mathsf{wrong}$$

$$\mathtt{endcase}$$

Figure 8. Auxiliary semantic functions for forcing

matching, is the way of building the abstraction: first of all, the argument must be forced, and then the evaluation proceeds according to the form obtained. If the value is an empty list, the returned closure will be determined by the expression E_1; whereas in the case of a non-empty list, the corresponding closure is built with E_2. Any other case is considered erroneous, and the continuation wrong is returned. The strictness is achieved by forcing the evaluation of the argument, where id_κ is the identity expression continuation.

The evaluation of a *list* (empty or non-empty) applies the expression continuation to the corresponding denotational value.

In the next section we define and explain the auxiliary semantic functions that have been used for the evaluation function $\mathcal{E}$.

3.3 Auxiliary semantic functions

In this section we give the definition of those semantic functions that have occurred in the definition of $\mathcal{E}$. Some of them are used to force the demanded variables, while others deal with streams and compose the sequences of the communication values.

3.3.1 Forcing the evaluation

The function force (see Figure 8) just decides how the demanded variable must be forced, since in the case of a communication variable —which appears in a channel definition— the function forceFV (see Figure 9) is used to propagate the forcing to the free dependencies. The function forceS (*simple force*, see Figure 8) compels the evaluation of an identifier. The context where this evaluation takes places is included in the expression continuation, and the

result of this forcing is a continuation. There are three possibilities:

(i) The identifier is bound to an expressed value: just apply the remainder of the program —expression continuation κ— to that value.

(ii) The identifier is bound to a closure which must be evaluated in the appropriate process. While the identifier is being evaluated it is bound to the value not_ready, once the value is obtained, it is bound to the variable —this association is carried out by the expression continuation κ'—, and the initial expression continuation, κ, is applied to that value.

(iii) Otherwise, the variable is undefined (it has never been declared) or not_ready (it is being evaluated, i.e. it is a self-reference). Both erroneous cases are interpreted by the continuation wrong .

The main details derived from the evaluation of streams appear in the function forceFV (see Figure 9). After having forced a variable three cases are possible. At each case one must distinguish whether the variable is for communication or not. First of all the variable is forced. Thereafter, the free variables are evaluated.

(i) If the obtained value is an abstraction: its free dependencies are also demanded, and thus evaluated recursively. If it is the case of a communication variable then the abstraction is communicated and the channel is closed. Finally, the expression continuation is applied to the value that has been obtained.

(ii) If the obtained value is nil: the expression continuation is applied to nil and to the state passed as argument. In the case of a communication the channel is closed.

(iii) Otherwise, the obtained value is a non-empty list: if the variable does not correspond to a communication then the whole list must be evaluated to normal form; thus, its head and its tail are forced, propagating the demand to their free dependencies. However, in the case of a communication, the list is a stream and the head is forced —together with all its free dependencies— in order to be communicated, and the communication identifier in the channel is substituted by a fresh one which is bound to the tail of the stream.

The function ++ adds a new item in the sequence of values that have been communicated through a stream-channel.

$$
\begin{array}{l}
\mathsf{forceFV} :: \mathbf{Id} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{forceFV}\ x\ \kappa = \mathsf{forceS}\ x\ \kappa' \\
\quad \texttt{where}\ \kappa' = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sdh}\rangle.\texttt{case}\ \varepsilon\ \texttt{of} \\
\quad (\mathrm{i}) \quad \langle\alpha, I\rangle \in \mathbf{Abs} \times \mathbf{Ids} \longrightarrow \kappa\,\varepsilon\,s'_f \\
\qquad\quad \texttt{where}\ s_f = \langle\rho_f, \mathrm{sdh}_f\rangle = \mathsf{mforceFV}\ I\ \langle\rho, \mathrm{sdh}\rangle \\
\qquad\qquad\quad s'_f = \texttt{case}\ \langle p, x, \mathrm{scw}, q\rangle \in \mathrm{sdh}_f\ \texttt{of} \\
\qquad\qquad\qquad \mathrm{false} \longrightarrow s_f \\
\qquad\qquad\qquad \mathrm{true} \longrightarrow \texttt{case}\ \mathrm{scw}\ \texttt{of} \\
\qquad\qquad\qquad\quad <> \longrightarrow s \oplus_{ch} \{\langle p, \mathrm{closed}, <\alpha>, q\rangle\} \\
\qquad\qquad\qquad\quad \mathrm{scv} \longrightarrow s_f \oplus_{ch} \{\langle p, \mathrm{closed}, \mathrm{scv} +\!\!+ \alpha, q\rangle\} \\
\qquad\qquad\qquad \texttt{endcase} \\
\qquad\qquad\quad \texttt{endcase} \\
\quad (\mathrm{ii}) \quad \mathsf{nil} \longrightarrow \texttt{case}\ \langle p, x, \mathrm{scw}, q\rangle \in \mathrm{sdh}\ \texttt{of} \\
\qquad\quad \mathrm{false} \longrightarrow \kappa\ \mathsf{nil}\ \langle\rho, \mathrm{sdh}\rangle \\
\qquad\quad \mathrm{true} \longrightarrow \kappa\,\varepsilon\,s' \\
\qquad\qquad \texttt{where}\ s' = \texttt{case}\ \mathrm{scw}\ \texttt{of} \\
\qquad\qquad\qquad\quad <> \longrightarrow \langle\rho, \mathrm{sdh}\rangle \oplus_{ch} \{\langle p, \mathrm{closed}, <\mathsf{nil}>, q\rangle\} \\
\qquad\qquad\qquad\quad \mathrm{scv} \longrightarrow \langle\rho, \mathrm{sdh}\rangle \oplus_{ch} \{\langle p, \mathrm{closed}, \mathrm{scv} +\!\!+ \mathsf{nil}, q\rangle\} \\
\qquad\qquad\quad \texttt{endcase} \\
\qquad \texttt{endcase} \\
\quad (\mathrm{iii}) \ \langle x_h, x_t\rangle \in \mathbf{Id} \times \mathbf{Id} \longrightarrow \texttt{case}\ \langle p, x, \mathrm{scw}, q\rangle \in \mathrm{sdh}\ \texttt{of} \\
\qquad\quad \mathrm{false} \longrightarrow \kappa\,\varepsilon\,s_f\ \texttt{where}\ s_f = \mathsf{mforceFV}\ \{x_h, x_t\}\ \langle\rho, \mathrm{sdh}\rangle \\
\qquad\quad \mathrm{true} \longrightarrow \kappa\,\langle x_h, y\rangle\,s'_h \\
\qquad\qquad \texttt{where}\ s_h = \langle\rho_h, \mathrm{sdh}_h\rangle = \mathsf{forceFV}\ x_h\ id_\kappa\ \langle\rho, \mathrm{sdh}\rangle \\
\qquad\qquad\quad \mathrm{cve}_h = \mathsf{comval}\ x_h\ s_h \\
\qquad\qquad\quad s'_h = \texttt{case}\ \mathrm{cve}_h\ \texttt{of} \\
\qquad\qquad\qquad \mathrm{cv}_h \in \mathbf{CVal} \longrightarrow \texttt{case}\ \mathrm{scw}\ \texttt{of} \\
\qquad\qquad\qquad\qquad <> \longrightarrow s_h \oplus \langle\{y \mapsto \langle p, \mathcal{E}\,[\![x_t]\!]\,\rangle, x \mapsto \langle x_h, y\rangle\}, \{\langle p, y, \mathrm{cv}_h, q\rangle\}\rangle \\
\qquad\qquad\qquad\qquad \mathrm{scv} \longrightarrow s_h \oplus \langle\{y \mapsto \langle p, \mathcal{E}\,[\![x_t]\!]\,\rangle, x \mapsto \langle x_h, y\rangle\}, \{\langle p, y, \mathrm{scv} +\!\!+ \mathrm{cv}_h, q\rangle\}\rangle \\
\qquad\qquad\qquad\quad \texttt{endcase} \\
\qquad\qquad\qquad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad\qquad\quad \texttt{endcase} \\
\qquad\qquad\quad y = \mathsf{newId}\ s_h \\
\qquad \texttt{endcase} \\
\texttt{endcase}
\end{array}
$$

Figure 9. Propagating the forcing

$$
\begin{array}{l}
\mathsf{comval} \;::\; \mathbf{Id} \to \mathbf{State} \to \mathbf{CValE} \\
\mathsf{comval}\; x\; s = \mathtt{case}\; (\rho\, x)\; \mathtt{of} \\
\qquad \langle \alpha, I \rangle \longrightarrow \alpha \\
\qquad\quad \mathsf{nil} \longrightarrow \mathsf{nil} \\
\qquad \langle x_1, x_2 \rangle \longrightarrow \langle (\mathsf{comval}\; x_1\; s), (\mathsf{comval}\; x_2\; s) \rangle \\
\quad \mathtt{endcase} \\
\quad \mathtt{where}\; s = \langle \rho, \mathrm{sdh} \rangle
\end{array}
\qquad
\begin{array}{l}
\mathsf{kstr} \;::\; \mathbf{Id} \to \mathbf{ECont} \\
\mathsf{kstr}\; x = \lambda\varepsilon.\lambda s.\mathtt{case}\; (\rho\, x)\; \mathtt{of} \\
\qquad \langle \alpha, I \rangle \longrightarrow \{s\} \\
\qquad\quad \mathsf{nil} \longrightarrow \{s\} \\
\qquad \langle x_h, x_t \rangle \longrightarrow \mathsf{forceFV}\; x_h\; (\mathsf{kstr}\; x_t)\; s \\
\quad \mathtt{endcase} \\
\quad \mathtt{where}\; s = \langle \rho, \mathrm{sdh} \rangle
\end{array}
$$

Figure 10. Auxiliary semantic functions for streams

3.3.2 Auxiliary functions for streams

The value communicated through a stream may be a list —which is then evaluated to normal form— and to add this list to the sequence of the channel it has first to be 'composed', that is, it has to be transformed into a list of communicable values. This task is carried out by the function comval (composition of values) whose definition is given in Figure 10. Initially, the list to be composed belongs to the domain of generic lists

$$\mathbf{CValE} = \mathbf{Abs} + \mathbf{CList} + \{\mathsf{nil}\} + (\mathbf{CValE} \times \mathbf{CValE})$$

which, in particular, contains the domain **CList**. However, this list may be 'amorphous' —it may not finish with the empty list—, and we do not allow to communicate a list which is not well-formed.

As we explained in Section 3.2.2, during process creation a special expression continuation kstr (expression continuation of streams) is used. Its definition is given in Figure 10: if the variable is bound to a non-empty list, the head and the tail are forced; otherwise, the function is like the identity expression continuation.

3.4 Example

The next example compares the denotational values for a stream following the minimal and maximal guidelines.

Example 4 Let us consider the following expression:

$$\mathtt{let}\; x_0 = [x_1 : x_2], x_1 = x_3\, x_3, x_2 = \mathtt{nil}, x_3 = \backslash x.x, x_4 = x_3 \# x_0\; \mathtt{in}\; x_4$$

where the evaluation of x_4 demands the creation of a process for evaluating $x_3\, x_0$. The input for this process is a stream because x_0 evaluates to a list.

The initial environment is:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus_e$$
$$\{main \mapsto \langle p_{main}, \mathcal{E}\,[\![\texttt{let } x_0 = [x_1 : x_2],$$
$$x_1 = x_3\, x_3,$$
$$x_2 = \texttt{nil},$$
$$x_3 = \backslash x.x,$$
$$x_4 = x_3 \texttt{\#} x_0 \texttt{ in } x_4]\!]\,\rangle\}$$

And the final state obtained from $s_0 = \langle \rho_0, \emptyset \rangle$ and $\kappa_0 = id_\kappa$:

(i) $\mathcal{E}_{min}\,[\![main]\!]\,p_{main}\,\kappa_0\,s_0 =$

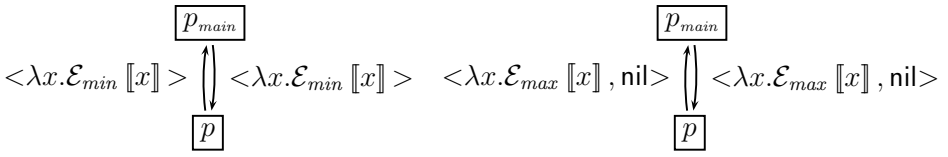
$$\langle\{x_0 \mapsto \langle x_1, x_2\rangle, x_1 \mapsto \langle \lambda x.\mathcal{E}_{min}\,[\![x]\!], \emptyset\rangle, x_2 \mapsto \langle p_{main}, \mathcal{E}_{min}\,[\![\texttt{nil}]\!]\,\rangle,$$
$$x_3 \mapsto \langle \lambda x.\mathcal{E}_{min}\,[\![x]\!], \emptyset\rangle, x_4 \mapsto \langle x_1, c_1\rangle, main \mapsto \langle x_1, c_1\rangle,$$
$$c_0 \mapsto \langle p_{main}, \mathcal{E}_{min}\,[\![x_2]\!]\,\rangle, c_1 \mapsto \langle p, \mathcal{E}_{min}\,[\![c_0]\!]\,\rangle, o \mapsto \langle x_1, c_1\rangle, i \mapsto \langle x_1, c_0\rangle, \},$$
$$\{\langle p_{main}, c_0, <\lambda x.\mathcal{E}_{min}\,[\![x]\!]>, p\rangle, \langle p, c_1, <\lambda x.\mathcal{E}_{min}\,[\![x]\!]>, p_{main}\rangle\}\rangle$$

(ii) $\mathcal{E}_{max}\,[\![main]\!]\,p_{main}\,\kappa_0\,s_0 =$

$$\langle\{x_0 \mapsto \langle x_1, x_2\rangle, x_1 \mapsto \langle \lambda x.\mathcal{E}_{max}\,[\![x]\!], \emptyset\rangle, x_2 \mapsto \mathsf{nil},$$
$$x_3 \mapsto \langle \lambda x.\mathcal{E}_{max}\,[\![x]\!], \emptyset\rangle, x_4 \mapsto \langle x_1, c_1\rangle, main \mapsto \langle x_1, c_1\rangle,$$
$$c_0 \mapsto \mathsf{nil}, c_1 \mapsto \mathsf{nil}, o \mapsto \langle x_1, c_1\rangle, i \mapsto \langle x_1, c_0\rangle, \},$$
$$\{\langle p_{main}, \text{closed}, <\lambda x.\mathcal{E}_{max}\,[\![x]\!], \mathsf{nil}>, p\rangle, \langle p, \text{closed}, <\lambda x.\mathcal{E}_{max}\,[\![x]\!], \mathsf{nil}>, p_{main}\rangle\}\rangle$$

From these final states we conclude that the values bound to *main* are the same in both cases, but in the case of the minimal semantics, the channels have been left open because only the heads have been demanded, while in the maximal semantics they have been wholly evaluated and closed. Moreover, in the maximal semantics all the variables, for instance x_2, have been evaluated.

Graphically, the process topologies are the following:



Minimal Maximal

□

4 Conclusions and future work

Some experts may argue that the denotational semantics presented here has not a very high level of abstraction because continuations are too 'operational'. And they are right. Denotational semantics with continuations were introduced in the seventies [13,14] to express the breaking in the sequence of instructions of a program. In our case, the choice of a denotational model based on continuations —instead of a direct denotacional semantics— has allowed us to express the combination of a lazy computational kernel with the *side-effects* produced by the coordination layer, i.e. process creations and underlying communications. In fact, a continuation takes into account these side-effects produced during the evaluation of some expression. In short, we have defined a formal model for a lazy λ-calculus that is suitable to describe parallel processes.

With the newly defined semantics, we can associate each program with two denotations: a *minimal* one, which represents an almost completely lazy evaluation —processes are created eagerly but their bodies are evaluated only if they are demanded— and a *maximal* one, where laziness is restricted to functional applications. Consequently, the set of all the possible resulting computational states ranges between the minimal and the maximal denotations.

Our denotational model also allows to extract the degree of parallelism and the amount of speculative computation. In the final state, the nodes of the graph corresponding to the set of channels are the processes which have been created in the system during the evaluation of the main expression. Other degrees of parallelism can be obtained by modifying the expression continuation for `#`-expressions and local declarations; these degrees would be greater than the minimal one and smaller than the maximal one. One might, for instance, demand the evaluation of the output but not of the input of a process, or to evaluate only some subset of the parallel applications in a local declaration depending on the number of available processors. In order to get information of the speculative computation, we must analyze the edges of the system: if the edge from a child to its parent is labelled with <> then the child (together with all its descendants) is a speculative process because its output has not been used for the main result.

Although this denotational model is suitable for studying the equivalence between process systems, it has also some limitations because its abstraction level does not allow, for instance, to observe work duplication.

Regarding the future, the defined semantics can be used to prove the correctness of the transformations defined for Eden in [8]. Besides, we want to

extend the calculus with dynamic channels.

We are working on the formal relationship between the operational semantics in [3] and the denotational one of this paper. This relationship will be based on the process topology which can be obtained from both semantics. Although in the operational model the topology cannot be built from the final system, this information can be obtained from the whole computation as we have just mentioned. In the denotational semantics this structure is contained in the channels set of the state. We want to prove that the structure obtained from the operational computation is equivalent to the one obtained from the channels set of the final state in the denotational semantics.

Acknowledgement

The authors would like to thank David de Frutos for his valuable comments on the denotational model.

References

[1] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[2] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, January 2004.

[3] 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.

[4] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems*, pages 303–321. LNCS 2895, Springer, 2003.

[5] M. B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

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

[7] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 2004. (To appear).

[8] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 13–26. Intellect, 2000.

[9] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Trends in Functional Programming (Selected papers of the 3rd Scottish Functional Programming Workshop)*, volume 3, pages 51–62. Intellect, 2002.

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

[11] J. H. Reppy. Concurrent ML: Design, application and semantics. In *Proceedings of Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. LNCS 693, Springer, 1993.

[12] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[13] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[14] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.