Continuation Semantics for Parallel Haskell Dialects.

Mercedes Hidalgo-Herrero[1] and Yolanda Ortega-Mallén[2]

[1] Dept. Didáctica de las Matemáticas
Facultad de Educación, Universidad Complutense de Madrid, Spain
mhidalgo@edu.ucm.es
[2] Dept. Sistemas Informáticos y Programación
Facultad CC.Matemáticas, Universidad Complutense de Madrid, Spain
yolanda@sip.ucm.es

Abstract. The aim of the present work is to compare, from a formal semantic basis, the different approaches to the parallelization of functional programming languages. For this purpose, we define a continuation semantics model which allows us to deal with side-effects and parallelism. To verify the suitability of our model we have applied it to three programming languages that introduce parallelism in very different ways, but whose common functional kernel is the lazy functional language Haskell.

1 Introduction

It is well-known that declarative programming offers good opportunities for parallel evaluation. More precisely, the different ways for exploiting the parallelism inherent in functional programs can be classified in three tendencies:

Implicit Parallelism. The parallelism inherent in the reduction semantics —independent redexes can be reduced in an arbitrary order or even in parallel— is the basis for *automatic* parallelization of functional programs.

Semi-explicit Parallelism. The programmer indicates where a parallel evaluation is desired. Either annotations for the compiler are added to the program, or parallel higher-level constructs, like skeletons or evaluation strategies, are used to express algorithms. Although the programmer controls the parallelism to some extent, the details are still implementation dependent.

Explicit Parallelism. Functional programming languages are extended with constructs for explicit process creation, communication and synchronization, so that general concurrent systems can be modelled.

R.Loogen gives in [Loo99] a similar classification, and provides a complete overview and a detailed discussion on these approaches. It is stated in that work that, when there is no explicit notion of parallel process —that is, in implicit and semi-explicit parallelism—, the denotational semantics remains unchanged. We admit that this is true if a "standard" denotational semantics is considered, where only the functional input-output relationship is considered; but the quality of a denotational semantics resides in the level of abstraction that it discloses, i.e. how many execution details are incorporated in the program denotation, and the utmost abstraction is to keep only the final value. One may argue that other aspects to be observed from a parallel execution, like runtimes, resource consumption, communications, etc. are implementation dependent and, thus, they should be not considered as semantical issues; at most they should be described through abstract machines.

It is our purpose to reach an intermediate level of abstraction, so that we can compare programs in terms of the amount of work, i.e. calculation, that must be done to obtain the same output from the same input. This is particularly interesting in the case of functional languages with demand-driven evaluation, which is inherently sequential and, therefore, parallelization introduces speculative parallelism —some expressions may get evaluated, whose results are never used— that should be detected and avoided. The degree of work duplication and speculation can be then semantically observed, as well as other properties concerning communications between processes. To this extent, we are interested in three different aspects:

– Functionality: the final value computed.
– Parallelism: the system topology (existing processes and connections among them) and its corresponding interactions generated by the computation.
– Distribution: the degree of work duplication and speculation.

Few research has been carried out in these directions for parallel and/or concurrent functional languages. There has been some work addressing the first two aspects, for instance, in [DB97,FH99] two denotational semantics for the strict language Concurrent ML (CML) [Rep92] are presented. Both are based on the Acceptance Trees model [Hen88], originally defined for the analysis of reactive systems. In each paper, the model is extended with value production, in order to express the overall input-output relation of functional programs. The semantic model for strict evaluation is much simpler than one for lazy evaluation, and in a concurrent setting there is no need to bother about the distribution aspects mentioned above.

1.1 Haskell parallel dialects

The *lazy* functional programming language Haskell [Pey03] is a wide-spectrum language widely known and used by the functional programming community. Haskell, like many other functional programming languages, has succumbed to the greediness of the parallel programming community, and different varieties of parallel or concurrent Haskell have emerged in the 90's.

The three parallel dialects of Haskell chosen for the present work, represent excellent examples of each kind of parallelism that we have explained above.

– The language pH (parallel Haskell) [NA01] adopts a parallel evaluation order (implicit parallelism), concretely in the case of data constructors with many arguments, and local definitions in program blocks. Moreover, pH adds special data structures that allow synchronization among parallel tasks, but introduce side-effects and non-determinism.

– Glasgow parallel Haskell (GPH) [THM+96] just introduces two special combinators

$$\texttt{seq}, \texttt{par} :: \texttt{a} \to \texttt{b} \to \texttt{b}$$

The former corresponds to sequential composition, while the latter indicates potential parallelism. These combinators are used to define *evaluation strategies* that allow the programmer to overrule laziness in favor of parallelism and to specify the degree of evaluation.

– The language Eden [BLOMP96] extends Haskell by a *coordination* language with explicit parallel process creation and streams, i.e. lazy lists, as communication channels [KM77]. Eden incorporates also a restricted form of non-determinism by means of a predefined non-deterministic process `merge` used to model many-to-one communication.

The following example illustrates the main difference between the three approaches. If we want to obtain the product of the total sum of the elements of two given lists of integers, it can be expressed in Haskell as follows:

```
let s1 = sum l1, s2 = sum l2 in s1 * s2
```

where `sum` is a standard function that sums up all the elements in a list. Supposing that lists `l1` and `l2` have been constructed already, the evaluation of this expression sums up first the elements of `l1` and then those of `l2`. But if we consider this same expression in pH and we have available two processors, then the sum for each list will be done in parallel. Laziness is abandoned in favor of parallelism; `s1` and `s2` are evaluated, eagerly, even if their result would never be demanded by the "main" computation.

The same idea is expressed in GPH with the following expression:

```
let s1 = sum l1, s2 = sum l2 in s2 'par' (s1 * s2)
```

where e_1`'par'`e_2 requires the parallel evaluation of e_1 and e_2, and returns the value obtained for e_2.

Finally, in the case of Eden, we write:

```
let p = process list -> sum list, s1 = p # l1, s2 = p # l2
in s1 * s2
```

where a process abstraction `p` is defined with `sum` as its body, and is instantiated twice —by using the special infix operator `#`— thus creating two processes that will execute `sum` in parallel with the process that is evaluating the let-expression and is considered the *parent* of the other two processes. The parent process has to communicate the corresponding list to each *child*, and each child communicates the result of the sum to the parent.

1.2 A continuation semantics

The difficulties for parallelizing Haskell lie in laziness, a identity-sign of Haskell. As we have mentioned above, parallelization requires some changes in the rules for evaluation. The combination of laziness with eager evaluation is a very interesting point to be studied through a formal semantics. In general, the evaluation of a program of this kind may give rise to different computations, where the amount of speculative parallelism depends on the number of available processors, the scheduler decisions and the speed of basic instructions. Hence, when defining a formal semantics for these languages one can model that speculation ranges from a *minimum*, i.e. only what it is effectively demanded is computed, to a *maximum*, i.e. every speculative computation is carried out.

Formal operational semantics have already been given for each of the three parallel languages that we are considering (or more exactly, for simplified kernels of these languages):

– For pH an operational semantics described in terms of a parallel abstract machine, in the spirit of the G-machine [Pey87], is given in [AAA$^+$95].
– The operational semantics given in [BFKT00] for GpH is small-step for process local reduction, and big-step for the scheduling and parallelization.
– Similarly, a two-level operational semantics is given in [HOM02] for Eden: a local level for process evaluation, and a global level for the system evolution.

We intend to define a general framework where we can express the semantics of each language, in order to be able to compare the differences between the three approaches. For this purpose, the actual operational semantics are unsuitable, because they include too much detail concerning computations and the order of evaluation, and the overall meaning of programs gets lost.

Detecting work duplication and speculation requires to express somehow in the semantics the sharing/copying of closures. In order to achieve this, we depart from a *standard* denotational semantics [Sto77] just expressing the input-output relationship of functional programs, and we extend it with a notion of *process system generation*. Maintaining the process system as part of the denotation of a program means that the evaluation of an expression may produce some side-effects which must be treated with care in a lazy context. For instance, in the case of evaluating an application such as $(\backslash x.3)y$ in a process p, the evaluation of the variable y may imply the corresponding evaluation of other bindings. Whereas this is not relevant in the case of a denotational semantics which is only interested in the final value, in our approach the modifications in the process system should not be carried out, because the λ-abstraction is not strict in its argument and, therefore, the evaluation of y is not going to be demanded. Hence, process creation and value communication are side-effects which must take place only under demand. Moreover, we recall that laziness implies that arguments to function calls are evaluated at most once. Therefore, we must be careful to produce the corresponding side-effects only the first time a value is demanded.

Continuations resolve elegantly the problem of dealing with command sequencing in denotational semantics. The meaning of each command is a function returning the final result of the entire program. The command meaning receives as an extra argument a function from states to final results. This extra argument —the *continuation*— describes the "rest of the computation", that will take place if the command ever ends its execution [Rey98].

In the case of functional languages, a continuation is a function from values to final results. The semantic function for the evaluation of an expression depends on an environment and a continuation, so that the result of the entire program is obtained by applying the continuation to the value obtained for the expression.

In order to deal with side-effects in functional programming, Josephs [Jos86] combines the two views of continuations to obtain a *expression continuation*, which is a function that receives a value and produces a "command continuation" or state transformer. In [Jos89] it is described how to use these expression continuations to model accurately the order of evaluation in a denotational semantics for a lazy λ-calculus. We use these ideas to define a continuation-based denotational semantics for parallel lazy functional languages, where process creation and interprocess communication and synchronization are considered as side-effects produced during the evaluation of expressions.

In [EM02] a mixed lazy/strict semantics is presented. Apart from the fact that it does not consider any form of parallelism, the strict semantics just tests whether an expression would evaluate to a normal form, but the actual evaluation will only take place when the value is needed. This is not adequate for our case, where evaluation of an expression may produce the side-effects explained above.

As far as we know, ours is the first denotational semantics which considers this problem and expresses not only the final result, but also the interaction between processes in a lazy context.

It is not our objective to give a complete denotational semantics for each language. For our purposes, it is sufficient to concentrate on a very simple functional kernel —a lazy λ-calculus— extended with the parallel features particular to each approach. To facilitate the comparison, a uniform syntax is adopted.

The paper is structured as follows: We devote the next three sections to the parallel Haskell dialects Eden, GPH and pH. For each language we explain its main characteristics, we give its kernel syntax, and define its denotational semantics, i.e. semantic domains and semantic functions. In the last section we draw some conclusions and comment on future work.

2 Eden

Eden extends the lazy functional language Haskell with a set of *coordination* features that are based on two principal concepts: *explicit management of processes* and *implicit communication*. Functional languages distinguish between function definitions and function applications. Similarly, Eden offers *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes and of the corresponding communication channels.

Communication in Eden is unidirectional, from one producer to exactly one consumer. Arbitrary communication topologies can be created by means of *dynamic reply channels*. In order to provide control over where expressions are evaluated, only fully evaluated data objects are communicated; this also facilitates a distributed implementation of Eden. However, Eden retains the lazy nature of the underlying computational functional language, and lists are transmitted in a *stream*-like fashion, i.e. element by element. Concurrent threads trying to access not yet available input will be suspended. This is the only way of synchronizing Eden processes.

In Eden, *nondeterminism* is gently introduced by means of a predefined process abstraction which is used to instantiate nondeterministic processes, that fairly merge a list of input channels into a single list.

For details on the Eden language design, the interested reader is referred to [BLOMP96]. Here we just concentrate on Eden's essentials which are captured by the untyped λ-calculus whose (abstract) syntax, with two syntactic categories: identifiers ($x \in \mathbf{Ide}$) and expressions ($E \in \mathbf{Exp}$), is given in Figure 1.

$E ::= x$	identifier
$\mid$ $\backslash x.E$	λ-abstraction
$\mid$ $x_1 \,\$\, x_2$	lazy application
$\mid$ $x_1 \,\$\#\, x_2$	parallel application
$\mid$ $\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration
$\mid$ $x_1 [\!] x_2$	choice

Fig. 1. Eden-core syntax

The calculus mixes laziness and eagerness, using two kinds of application: lazy (\$), and parallel (\$#). The evaluation of $x_1 \,\$\, x_2$ generates demand on the value for x_1, but the expression corresponding to x_2 is only evaluated if needed. By contrast, the evaluation of $x_1 \,\$\#\, x_2$ implies the creation of a new parallel process for evaluating $x_1\ x_2$. Two channels are established between the new process and its creator: one for communicating the value of x_2 from the parent to the child, and a second one for communicating from the child to the parent the result value of $x_1 x_2$. Eden-core includes also an operator that carries out a non-deterministic choice between its two arguments.

Following [Lau93], the calculi presented in this paper have been normalized to a restricted syntax where all the subexpressions, but for the body of λ-abstractions, have been associated to variables. We have several reasons for doing this: (1) In this way all the subexpressions of any expression are shared. This maximizes the degree of sharing within a computation and guarantees that every subexpression of a program is evaluated at most once. (2) The semantic definition of lazy application is clearer, because with the normalization it is unnecessary to introduce fresh variables for the argument of the application.

2.1 Semantic domains for Eden

The semantic domains that are needed for defining the continuation semantics for Eden are listed in Figure 2.

As it was motivated in the introduction, the meaning of a program is a continuation, or a state transformer. To deal with the non-determinism present in Eden, we consider sets of states (for a given set S, $\mathcal{P}_f(S)$ is the set of all the finite parts of S), so that a continuation transforms an initial state into a set of possible final states. In the present case a state ($s \in \mathbf{State}$) will consist of two items:

- An *environment* ($\rho \in \mathbf{Env}$) mapping identifiers to values. Environments here are analogous to stores when dealing with imperative variables. Following [Sto77], we do not consider the option of separating the environment from the state, because no evaluation in Eden produces irreversible changes

Cont	$= \mathbf{State} \to \mathbf{SState}$	continuations
$\kappa \in$ **ECont**	$= \mathbf{EVal} \to \mathbf{Cont}$	expression continuations
$s \in$ **State**	$= \mathbf{Env} \times \mathbf{SChan}$	states
$S \in$ **SState**	$= \mathcal{P}_f(\mathbf{State})$	set of states
$\rho \in$ **Env**	$= \mathbf{Ide} \to (\mathbf{Val} + \{\text{undefined}\})$	environments
$\mathcal{C}h \in$ **SChan**	$= \mathcal{P}_f(\mathbf{Chan})$	set of channels
Chan	$= \mathbf{IdProc} \times \mathbf{CVal} \times \mathbf{IdProc}$	channels
CVal	$= \mathbf{EVal} + \{\text{unsent}\}$	communication values
$\upsilon \in$ **Val**	$= \mathbf{EVal} + (\mathbf{IdProc} \times \mathbf{Clo}) + \{\text{not\_ready}\}$	values
$\varepsilon \in$ **EVal**	$= \mathbf{Abs} \times \mathbf{Ides}$	expressed values
$\alpha \in$ **Abs**	$= \mathbf{Ide} \to \mathbf{Clo}$	abstraction values
$\nu \in$ **Clo**	$= \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Cont}$	closures
$I \in$ **Ides**	$= \mathcal{P}_f(\mathbf{Ide})$	set of identifiers
$p, q \in$ **IdProc**		process identifiers

Fig. 2. Semantic domains for Eden-core

in the state, so that the preservation and restoration of the environment is easily practicable.

- A *set of channels* ($\mathcal{C}h \in$ **SChan**). Processes in Eden-core are not isolated entities, but they communicate through unidirectional, one-value channels. Each channel is represented by a triple ⟨producer,value,consumer⟩. The value can be either the expressed value that has been communicated, or unsent if there has been no communication through the channel.

The domain **Val** of values includes final denotational values (or *expressed values* $\varepsilon \in$ **EVal**), and closures ($\nu \in$ **Clo**), i.e. "semi"-evaluated expressions. The evaluation of a closure may imply the creation of new processes. In order to build the process system topology it is necessary to know which is the father of each newly created process. This is the reason for associating a process identifier to the closure. The special value not_ready indicates that the corresponding closure is currently being evaluated, and it is used to detect self-references.

Abstractions values ($\alpha \in$ **Abs**) are the only type of expressed values in this calculus. Each abstraction is represented by a function from identifiers to closures, together with the list of its free variables, that have to be evaluated before communicating the abstraction value through some channel.

A closure is a function that depends on (1) the process where the closure will be evaluated, and (2) the expression continuation denoting the rest of the program.

2.2 Evaluation function for Eden

Distributing the computation in parallel processes requires indicating the process where an expression is to be evaluated. Notice that the process identifier is not necessary to distinguish its variables from those of other processes —each process owns different identifiers—, but to be able to determine the parent in possible process creations. The semantic function for evaluating expressions has the following signature:

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

After evaluating the expression, the continuation obtained by instantiating the expression continuation with the value produced for the expression carries on with the computation.

The definition of the evaluation function for Eden-core is detailed in Figure 3. We use the operator $\oplus$ to express the extension/update of environments, like for instance in $\rho \oplus \{x \mapsto \nu\}$, and $\oplus_{ch}$ in the case of the set of channels of a state:

$$\mathcal{E}[\![x]\!]p\,\kappa = \mathsf{force}\,x\,\kappa$$

$$\mathcal{E}[\![\backslash x.E]\!]p\,\kappa = \kappa\langle \lambda x.\mathcal{E}[\![E]\!], \mathsf{fv}(\backslash x.E)\rangle$$

$$\begin{aligned}
&\mathcal{E}[\![x_1 \,\$\, x_2]\!]p\,\kappa = \mathcal{E}[\![x_1]\!]p\,\kappa' \\
&\quad \texttt{where } \kappa' = \lambda\langle \alpha, I\rangle.\lambda s.(\alpha\, x_2)\,p\,\kappa\,s
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![x_1 \,\$\#\, x_2]\!]p\,\kappa = \mathsf{forceFV}\,x_1\,\kappa' \\
&\quad \texttt{where } \kappa' = \lambda\langle \alpha, I\rangle.\lambda s.(\alpha\, x_2)\,q\,\kappa''\,s \\
&\qquad\qquad q = \mathsf{newIdProc}\,s
\end{aligned}$$

$$\boxed{\begin{aligned}
&\kappa''_{min} = \lambda\langle \alpha', I'\rangle.\lambda s'.\texttt{case } (\rho'\,x_2) \texttt{ of} \\
&\qquad \langle \alpha'', I''\rangle \in \mathbf{EVal} \longrightarrow S_d \oplus_{ch} \{\langle q, \langle \alpha, I\rangle, p\rangle\ , \langle p, \langle \alpha'', I''\rangle, q\rangle\} \\
&\qquad \texttt{otherwise} \longrightarrow S_c \oplus_{ch} \{\langle q, \langle \alpha, I\rangle, p\rangle\ , \langle p, \text{unsent}, q\rangle\} \\
&\qquad \texttt{endcase} \\
&\quad \texttt{where } (\rho', \mathit{Ch}') = s' \\
&\qquad\quad S_c = \mathsf{mforceFV}\,I'\,s' \\
&\qquad\quad S_d = \bigcup_{s_c \in S_c} \mathsf{mforceFV}\,I''\,s_c
\end{aligned}}$$

$$\boxed{\begin{aligned}
&\kappa''_{max} = \lambda\langle \alpha', I'\rangle.\lambda s'. \bigcup_{s_c \in S_c} \mathsf{forceFV}\,x_2\,\kappa_c\,s_c \\
&\quad \texttt{where } S_c = \mathsf{mforceFV}\,I'\,s' \\
&\qquad\qquad \kappa_c = \lambda\varepsilon''.\lambda s''.\{s'' \oplus_{ch} \{\langle q, \langle \alpha, I\rangle, p\rangle, \langle p, \varepsilon'', q\rangle\}\}
\end{aligned}}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{let } \{x_i = E_i\}_n \texttt{ in } x]\!]p\,\kappa = \lambda\langle \rho, \mathit{Ch}\rangle.\mathcal{E}[\![x]\!]p\,\kappa'\,\langle \rho', \mathit{Ch}\rangle \\
&\quad \texttt{where } \{y_1, \ldots, y_n\} = \mathsf{newIde}\,n\,\rho \\
&\qquad\qquad \rho' = \rho \oplus \{y_i \mapsto \langle \mathcal{E}[\![E_i[y_1/x_1, \ldots, y_n/x_n]]\!], p\rangle \mid 1 \le i \le n\}
\end{aligned}$$

$$\boxed{\kappa'_{min} = \kappa}$$

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

$$\mathcal{E}[\![x_1 [\!] x_2]\!]p\,\kappa = \lambda s.(\mathcal{E}[\![x_1]\!]p\,\kappa\,s) \cup (\mathcal{E}[\![x_2]\!]p\,\kappa\,s)$$

Fig. 3. Evaluation function for Eden-core

The evaluation of an identifier "forces" the evaluation of the value bound to that identifier in the given environment. The function force is defined in Figure 4 and is explained later.

The evaluation of a λ-abstraction produces the corresponding expressed value, that is a pair formed by a function which given an identifier returns a closure, and the set of free variables of the syntactic construction.

In the case of *lazy application*, the evaluation of the argument, x_2, is delayed until it is effectively demanded; the expression continuation, κ, given for the evaluation of the application is modified, κ' to reflect this circumstance. This κ' is the one supplied for the evaluation of the variable corresponding to the application abstraction, x_1. Therefore, once the abstraction value is obtained, it is applied to the argument variable, x_2, and the corresponding closure is evaluated.

The evaluation of a *parallel application* $x_1 \,\$\#\, x_2$ produces the creation of a new process, q. The following actions take place:

- Creation of a new process q (where q is a fresh process identifier).
- Creation of two new channels, $\langle q, \_, p\rangle, \langle p, \_, q\rangle$, between parent p and child q.
- Evaluation of x_1, to obtain the abstraction value, together with its free variables, $\langle \alpha, I\rangle$. The function $\mathsf{forceFV}$, given in Figure 4, is used for this purpose. In Eden, processes do not share memory. Therefore, the heap where the new process is going to be evaluated must contain every binding related to the free variables of the value abstraction corresponding to the process body. All this information is evaluated in the parent and then "copied" to the child.
- Evaluation of x_2. As we have explained in the introduction, speculative parallelism ranges from a minimum to a maximum. In the case of Eden, we can define a *minimal semantics* where argument x_2 is evaluated only if needed, while in a *maximal semantics* this evaluation always takes place. The value for x_2 has to be communicated (from the parent to the child) together with all the information concerning its free variables. The function $\mathsf{forceFV}$, given in Figure 4, is used again for evaluating x_2 and its free variables.
- Communication (of the value for x_2) from parent p to child q. Under a minimal semantics it may not occur this communication.
- Evaluation of the application $x_1\, x_2$ in the new process q.
- Communication (of the value for $x_1\, x_2$) from child q to parent p . Before communication, the free variables have to be evaluated too; the auxiliary function $\mathsf{mforceFV}$ (see Figure 4) is invoked.

Before evaluating the body of a local declaration, all the local variables, x_i, (with fresh identifiers, y_i, to avoid name clashes) are to be incorporated in the environment. Each new local variable, y_i, is bound to the proper closure, $E_i[y_j/x_j]$, which is obtained from the corresponding expression in the declaration. Closures are paired with the process identifier, p, where the declaration is being evaluated. When a local variable is defined as a parallel application, this must be evaluated too (in the case of a maximal semantics).

Finally, the *choice operator* comprises the possibility of evaluating either of its two subexpressions, x_1 and x_2.

The auxiliary functions for "forcing" the evaluation are defined in Figure 4. The action of force depends on the value bound to the identifier that is to be "forced". In the case of an expressed value, the given expression continuation is applied to it in order to continue with the computation. In the case of a

$$
\begin{array}{l}
\mathsf{force} :: \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\, x\, \kappa = \lambda\langle\rho, \mathcal{C}h\rangle.\mathtt{case}\ (\rho\, x)\ \mathtt{of} \\
\quad \varepsilon \in \mathbf{EVal} \longrightarrow \kappa\, \varepsilon\, \langle\rho, \mathcal{C}h\rangle \\
\quad \langle p, \nu\rangle \in (\mathbf{IdProc} \times \mathbf{Clo}) \longrightarrow \nu\, p\, \kappa'\, s' \\
\qquad \mathtt{where}\ \kappa' = \lambda\varepsilon'.\lambda\langle\rho', \mathcal{C}h'\rangle.\kappa\, \varepsilon'\, \langle\rho' \oplus \{x \mapsto \varepsilon'\}, \mathcal{C}h'\rangle \\
\qquad\qquad\quad s' = \langle\rho \oplus \{x \mapsto \mathrm{not\_ready}\}, \mathcal{C}h\rangle \\
\quad \mathtt{otherwise} \longrightarrow \mathsf{wrong}
\end{array}
$$

$$
\begin{array}{l|l}
\mathsf{forceFV} :: \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Cont} & \mathsf{mforceFV} :: \mathbf{Ides} \to \mathbf{Cont} \\
\mathsf{forceFV}\, x\, \kappa = \mathsf{force}\, x\, \kappa' & \mathsf{mforceFV}\, \emptyset = \lambda s.\{s\} \\
\quad \mathtt{where}\ \kappa' = \lambda\langle\alpha, I\rangle.\lambda s'. \bigcup_{s'' \in S''} \kappa\, \langle\alpha, I\rangle\, s'' & \mathsf{mforceFV}\, (\{x\} \cup I) = \lambda s. \bigcup_{s' \in S'} \mathsf{mforceFV}\, I\, s' \\
\qquad\quad S'' = \mathsf{mforceFV}\, I\, s' & \quad \mathtt{where}\ S' = \mathsf{forceFV}\, x\, id_\kappa\, s
\end{array}
$$

Fig. 4. Auxiliary semantic functions for Eden-core

closure, this has to be evaluated and the result is bound in the environment to the identifier. During the evaluation of the closure, the identifier is bound to not_ready; if an identifier bound to not_ready is ever forced, this indicates the presence of a self-reference. After the evaluation of the closure, the expression continuation is applied to the obtained expressed value and the modified state. Of course, forcing an “undefined” location or a “not_ready” variable is a mistake.

The difference between force and forceFV lies in the scope of evaluation. The former only evaluates the closure associated to the variable, while the latter propagates the evaluation to the free variables of that closure, and to the ones corresponding to the closures of these free variables, and so on. The multiple demand for a set of identifiers is carried out by mforceFV.

In the definition of mforceFV, id_κ represents the identity for expression continuations, defined as $id_\kappa = \lambda\varepsilon.\lambda s.\{s\}$.

3 GpH

GPH (Glasgow Parallel Haskell) introduces parallelism via the annotation `'par'`; the expression e_1`'par'`e_2 indicates that e_1 and e_2 may be evaluated in parallel, returning the value obtained for e_2. Sequential composition is possible in GPH using `'seq'`; so that e_1`'seq'`e_2 evaluates e_1 and, only after obtaining the corresponding value, if any, the evaluation proceeds with e_2. In Figure 5 the syntax for GPH-core is given, which includes identifiers, λ-abstractions, lazy functional application, local declaration of variables, and both sequential and parallel composition.

3.1 Semantic domains for GpH

The semantic domains that we need for GPH-core are given in Figure 6. As GPH has neither notion of process nor of communication, the program state consists only of the environment, and the rest of semantic domains are simplified; for instance, expressed values are just abstraction values, without the set of free variables.

$E ::= x$	identifier
$\mid\ \backslash x.E$	λ-abstraction
$\mid\ x_1 \,\$\, x_2$	lazy application
$\mid\ \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration
$\mid\ x_1\ \texttt{'seq'}\ x_2$	sequential composition
$\mid\ x_1\ \texttt{'par'}\ x_2$	parallel composition

Fig. 5. GPH-core syntax

Moreover, GPH does not introduce non-determinism. Consequently, a continuation is a state transformer which takes an environment as its argument and yields another environment.

$\mathbf{Cont}$	$= \mathbf{Env} \to \mathbf{Env}$	continuations
$\kappa \in \mathbf{ECont}$	$= \mathbf{EVal} \to \mathbf{Cont}$	expression continuations
$\rho \in \mathbf{Env}$	$= \mathbf{Ide} \to (\mathbf{Val} + \{\text{undefined}\})$	environments
$v \in \mathbf{Val}$	$= \mathbf{EVal} + \mathbf{Clo} + \{\text{not\_ready}\}$	values
$\varepsilon \in \mathbf{EVal}$	$= \mathbf{Abs}$	expressed values
$\alpha \in \mathbf{Abs}$	$= \mathbf{Ide} \to \mathbf{Clo}$	abstraction values
$\nu \in \mathbf{Clo}$	$= \mathbf{ECont} \to \mathbf{Cont}$	closures

Fig. 6. Semantic domains for GPH-core

3.2 Evaluation function for GpH

The signature for the evaluation function $\mathcal{E}$ for GPH-core is similar to the one given for Eden-core, but without process identifiers:

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

whose definition is detailed in Figure 7.

$$\mathcal{E}[\![x]\!]\kappa = \mathsf{force}\, x\, \kappa$$

$$\mathcal{E}[\![\backslash x.E]\!]\kappa = \kappa(\lambda x.\mathcal{E}[\![E]\!])$$

$$\mathcal{E}[\![x_1 \,\$\, x_2]\!]\kappa = \mathcal{E}[\![x_1]\!]\kappa' \quad \texttt{where } \kappa' = \lambda\varepsilon.\lambda\rho.\varepsilon\, x_2\, \kappa\, \rho$$

$$\mathcal{E}[\![x_1\ \texttt{'seq'}\ x_2]\!]\kappa = \mathcal{E}[\![x_1]\!]\kappa' \quad \texttt{where } \kappa' = \lambda\varepsilon.\lambda\rho.\mathcal{E}[\![x_2]\!]\kappa\, \rho$$

$$\mathcal{E}[\![x_1\ \texttt{'par'}\ x_2]\!]\kappa = \mathcal{E}[\![x_2]\!]\kappa' \quad \texttt{where } \kappa' = \lambda\varepsilon.\lambda\rho.\kappa\, \varepsilon\, \rho_{par},\ \rho_{par} = \mathsf{par}\, x_1\, \rho$$

$$\begin{aligned}\mathcal{E}[\![\texttt{let}\ \{x_i = E_i\}_n\ \texttt{in}\ x]\!]\kappa &= \lambda\rho.\mathcal{E}[\![x]\!]\kappa\, \rho' \\ \texttt{where } \{y_1, \ldots, y_n\} &= \mathsf{newIde}\, n\, \rho \\ \rho' &= \rho \oplus \{y_i \mapsto \mathcal{E}[\![E_i[y_1/x_1, \ldots, y_n/x_n]]\!] \mid 1 \le i \le n\}\end{aligned}$$

Fig. 7. Evaluation function for GPH-core

The definition of the evaluation function for identifiers, λ-abstractions, and lazy applications, as well as the auxiliary function force (in Figure 8) is similar to the definition given for Eden-core, but much simpler because process identifiers, channel sets and free variables are removed.

As local declaration does not introduce parallelism, its evaluation just extends the environment with the new variables bound to the corresponding closures.

In a *sequential* composition x_1 `'seq'` x_2 we have to respect the order of evaluation: firstly, x_1 is evaluated, and then its expression continuation evaluates x_2 only if the evaluation of x_1 has given rise to an expressed value.

In a *parallel* composition x_1 `'par'` x_2, x_2 is always evaluated, but the evaluation of x_1 takes place only if there are enough resources. Therefore, in a minimal semantics x_1 is not evaluated, while in a maximal semantics x_1 is evaluated in parallel. This is expressed by using the function par (see Figure 8).

$$
\begin{array}{l}
\mathsf{force} :: \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\, x\, \kappa = \lambda\rho.\mathtt{case}\ (\rho\, x)\ \mathtt{of} \\
\quad \varepsilon \in \mathbf{EVal} \longrightarrow \kappa\, \varepsilon\, \rho \\
\quad \nu \in \mathbf{Clo} \longrightarrow \nu\, \kappa'\, \rho' \\
\qquad \mathtt{where}\ \kappa' = \lambda\varepsilon''.\lambda\rho''.\kappa\, \varepsilon''\, \rho'' \oplus \{x \mapsto \varepsilon''\} \\
\qquad\qquad\quad \rho' = \rho \oplus \{x \mapsto \mathrm{not\_ready}\} \\
\quad \mathrm{not\_ready} \longrightarrow \mathsf{wrong} \\
\mathtt{endcase}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{par} :: \mathbf{Ide} \to \mathbf{Cont} \\
\mathsf{par}_{\mathsf{min}}\, x = \lambda\rho.\rho \\
\mathsf{par}_{\mathsf{max}}\, x = \lambda\rho.\mathcal{E}[\![x]\!]\, id_\kappa\, \rho
\end{array}
$$

Fig. 8. Auxiliary semantic functions for GpH-core

4 pH

This section is devoted to pH (parallel Haskell), a successor of the Id dataflow language [Nik91], adopting the notation and type system of Haskell. pH is characterized by its implicit parallelism, the mixture of strictness and laziness, and the existence of updatable cells conveying *implicit synchronization*: I-structures are *single-assigment* data structures that help producer-consumer synchronization, while M-structures are *multiple-assigment* data structures that allow mutual exclusion synchronization, introducing side-effects and non-determinism.

Figure 9 shows the syntax for the pH-core, which includes identifiers, λ-abstractions, application —strict and non-strict—, local declaration of variables, and primitive operations to create, read from, and write to updatable cells.

I-cells dissociate the creation of a variable from the definition of its value, so that attempts to use the value of an I-cell are delayed until it is defined; but once an I-cell has been "filled" with a value, it can be neither emptied nor changed. On the other hand, a fetch over a M-cell empties the contents of that cell, so that any later query over it must wait until it is filled again. The behavior of both kinds of cells is summarized in Table 1.

Let us explain in more detail how updatable cells introduce side-effects and non-determinism:

$E ::= x$	identifier
$\mid \ \backslash x.E$	λ-abstraction
$\mid \ x_1 \,\$\, x_2$	lazy application
$\mid \ x_1 \,\$!\, x_2$	strict application
$\mid \ \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration
$\mid \ \texttt{iCell}\,x \mid \texttt{mCell}\,x$	cell creation
$\mid \ \texttt{Fetch}\,x \mid \texttt{Store}\,(x_1, x_2)$	cell operations

Fig. 9. pH-core syntax

	I-cell	**M-cell**
`cell`	empty I-cell creation	empty M-cell creation
`fetch`	I-cell reading error if empty	M-cell reading error if empty empty after reading
`store`	I-cell filling error if full	M-cell filling error if full

Table 1. I-cells and M-cells behavior

- A `Fetch` operation gives back the value stored in a M-cell, as a side-effect it empties the corresponding cell.
- Due to race-conditions, the evaluation of an expression may yield different values in different occasions. For an expression like

$$\texttt{let}\ t = \texttt{mCell}\ m\ , x = \texttt{Store}\ m\, v_1, y = \texttt{Store}\ m\, v_2, z = \texttt{Fetch}\ m\ \texttt{in}\ z$$

either the value of v_1 or the value of v_2 may be assigned to z.

4.1 Semantic domains for pH

Figure 10 shows the semantic domains that we need for formalizing the semantics of pH-core. Similarly to GpH and unlike Eden, pH does not have processes or communication channels; but, in order to model the separation of creation from value definition for updatable cells, a double binding-mechanism is needed: environments and *stores*. The *locations* of a store either contain some value or are undefined, while environments map identifiers to locations in the corresponding store.

Therefore, in the present case, the state of a program is represented by the global store ($\sigma \in$ **Store**). We have explained above how M-cells may introduce non-determinism; thus, continuations will transform a given store into a set of stores.

Similarly to the two previous approaches, the domain of values includes expressed values, closures and the special value not_ready. A new kind of values is included: (updatable) cells, that are distinguished by labels I and M. Each cell is either empty or it contains an expressed value.

Besides abstraction values —that in this case are mappings from locations to closures— the domain of expressed values includes the special value unit for expressions whose effect is not the production of a value, but the modification of the state (side-effects), like creating a cell, or storing a value in a cell.

$\mathbf{Cont}$	$= \mathbf{Store} \to \mathbf{SStore}$	continuations
$\kappa \in \mathbf{ECont}$	$= \mathbf{EVal} \to \mathbf{Cont}$	expression continuations
$\sigma \in \mathbf{Store}$	$= \mathbf{Loc} \to (\mathbf{Val} + \{\text{undefined}\})$	stores
$\Sigma \in \mathbf{SStore}$	$= \mathcal{P}_f(\mathbf{Store})$	set of stores
$\rho \in \mathbf{Env}$	$= \mathbf{Ide} \to \mathbf{Loc}$	environments
$\upsilon \in \mathbf{Val}$	$= \mathbf{EVal} + \mathbf{Clo} + \mathbf{Cell} + \{\text{not\_ready}\}$	values
$\varepsilon \in \mathbf{EVal}$	$= \mathbf{Abs} + \{\text{unit}\}$	expressed values
$\alpha \in \mathbf{Abs}$	$= \mathbf{Loc} \to \mathbf{Clo}$	abstraction values
$\nu \in \mathbf{Clo}$	$= \mathbf{ECont} \to \mathbf{Cont}$	closures
$\mathbf{Cell}$	$= \{\text{I}, \text{M}\} \times (\mathbf{EVal} + \{\text{empty}\})$	updatable cells
$l \in \mathbf{Loc}$		locations

Fig. 10. Semantic domains for pH-core

4.2 Evaluation function for pH

In addition to the expression continuation, the evaluation function $\mathcal{E}$ for pH-core needs an environment for determining the locations for the free variables in the expression. Its definition is shown in Figure 11 and its signature is:

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

The evaluation of an identifier forces the evaluation of the value stored in the corresponding location. The auxiliary function force (given in Figure 12) is very similar to those defined previously for Eden-core and GPH-core.

In the case of a λ-abstraction, the corresponding abstraction value is created and the expression continuation is applied to it.

Both, *lazy* and *strict* applications, force the evaluation of the expression which will yield the abstraction. The difference is that in lazy applications the argument is stored as a closure in a new location, to be evaluated only under demand; while in strict applications the evaluation of the argument is forced before the evaluation of the application.

We have three different actions concerning updatable cells:

Creation: An empty cell is stored in the location associated to the variable. The type of the cell is reflected in the cell label.

Query: If the corresponding cell is empty an error is raised. In the case of an M-cell, the cell is emptied after consulting its content.

Store: the value is obtained and placed in the corresponding cell. If the cell is already full an error is raised.

$$\mathcal{E}[\![x]\!]\rho\,\kappa = \mathsf{force}\,(\rho\,x)\,\kappa$$

$$\mathcal{E}[\![\backslash x.E]\!]\rho\,\kappa = \kappa(\lambda l.\mathcal{E}[\![E]\!](\rho \oplus \{x \mapsto l\}))$$

$$\begin{array}{l}
\mathcal{E}[\![x_1 \,\$\, x_2]\!]\rho\,\kappa = \mathcal{E}[\![x_1]\!]\rho\,\kappa' \\
\quad \texttt{where } \kappa' = \lambda\varepsilon.\lambda\sigma.\texttt{case } \varepsilon \texttt{ of} \\
\qquad\qquad \varepsilon \in \mathbf{Abs} \longrightarrow \varepsilon\, l\, \kappa\, \sigma' \\
\qquad\qquad\quad \texttt{where } l = \mathsf{freeloc}\ \sigma \\
\qquad\qquad\qquad\quad \sigma' = \sigma \oplus \{l \mapsto \mathcal{E}[\![x_2]\!]\rho\} \\
\qquad\qquad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad\quad \texttt{endcase}
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![x_1 \,\$!\, x_2]\!]\rho\,\kappa = \lambda\sigma. \bigcup_{\sigma_2 \in \Sigma_2} \kappa'\,(\sigma_2\,(\rho\,x_1))\,\sigma_2 \\
\quad \texttt{where } \Sigma_1 = \mathsf{force}\,(\rho\,x_1)\,id_\kappa\,\sigma \\
\qquad\qquad \Sigma_2 = \bigcup_{\sigma_1 \in \Sigma_1} \mathsf{force}\,(\rho\,x_2)\,id_\kappa\,\sigma_1 \\
\qquad\qquad \kappa' = \lambda\varepsilon.\lambda\sigma'.\texttt{case } \varepsilon \texttt{ of} \\
\qquad\qquad\quad \varepsilon \in \mathbf{Abs} \longrightarrow \varepsilon\,(\rho\,x_2)\,\kappa\,\sigma' \\
\qquad\qquad\quad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad\qquad \texttt{endcase}
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{let } \{x_i = E_i\}_n \texttt{ in } x]\!]\rho\,\kappa = \lambda\sigma.\mathcal{E}[\![x]\!]\rho'\,\kappa'\,\sigma' \\
\quad \texttt{where } \{l_1,\ldots,l_n\} = \mathsf{freeloc}\,n\,\sigma \\
\qquad\qquad \rho' = \rho \oplus \{x_1 \mapsto l_1,\ldots,x_n \mapsto l_n\} \\
\qquad\qquad \sigma' = \sigma \oplus \{l_1 \mapsto \mathcal{E}[\![E_1]\!]\rho',\ldots,l_n \mapsto \mathcal{E}[\![E_n]\!]\rho'\} \\
\qquad\qquad \kappa' = \lambda\varepsilon.\lambda\sigma''. \bigcup_{\sigma_d \in \Sigma_d} \kappa\,\varepsilon\,\sigma_d \\
\qquad\qquad\quad \texttt{where } \Sigma_d = \mathsf{decls}\,\{x_1,\ldots,x_n\}\,\rho'\,\sigma''
\end{array}$$

$$\mathcal{E}[\![\texttt{iCell}\,x]\!]\rho\,\kappa = \lambda\sigma.\kappa\,\mathrm{unit}\,(\sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{I},\mathrm{empty}\rangle\})$$

$$\mathcal{E}[\![\texttt{mCell}\,x]\!]\rho\,\kappa = \lambda\sigma.\kappa\,\mathrm{unit}\,(\sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{M},\mathrm{empty}\rangle\})$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{Fetch}\,x]\!]\rho\,\kappa = in_{\mathbf{Cont}}(x, \lambda\varepsilon.\lambda\sigma.\kappa\,\varepsilon\,\sigma') \\
\quad \texttt{where } \sigma' = \texttt{case } \sigma(\rho\,x) \texttt{ of} \\
\qquad\qquad\quad \langle \mathrm{I},\varepsilon'\rangle \longrightarrow \sigma \\
\qquad\qquad\quad \langle \mathrm{M},\varepsilon'\rangle \longrightarrow \sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{M},\mathrm{empty}\rangle\} \\
\qquad\qquad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad\qquad \texttt{endcase}
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{Store}\,(x_1,x_2)]\!]\rho\,\kappa = \mathcal{E}[\![x_2]\!]\rho\,\kappa' \\
\quad \texttt{where } \kappa' = \lambda\varepsilon.\lambda\sigma.(out_{\mathbf{Cont}}(x_1,\varepsilon,\kappa\,\mathrm{unit})\,\sigma') \\
\qquad\qquad \sigma' = \texttt{case } \sigma(\rho\,x_1) \texttt{ of} \\
\qquad\qquad\quad \langle \mathrm{I},\mathrm{empty}\rangle \longrightarrow \sigma \oplus \{(\rho\,x_1) \mapsto \langle \mathrm{I},\varepsilon\rangle\} \\
\qquad\qquad\quad \langle \mathrm{M},\mathrm{empty}\rangle \longrightarrow \sigma \oplus \{(\rho\,x_1) \mapsto \langle \mathrm{M},\varepsilon\rangle\} \\
\qquad\qquad\quad \texttt{otherwise} \longrightarrow \mathsf{wrong} \\
\qquad\qquad \texttt{endcase}
\end{array}$$

Fig. 11. Evaluation function for pH-core

There is a clear similitude between cells and channels: the filling of a cell resembles the action of communicating a value. On the other hand, the value of a cell can only be consulted if it is full. In a similar way, the reception of a value can only take place if the value has been sent. This point of view is denotationally expressed using the usual functions $out_{\mathbf{Cont}}$ and $in_{\mathbf{Cont}}$ (see [HI93]).

The evaluation of a local declaration is done in two phases: firstly, the environment and the store are widened with the information from the new local variables, and secondly, parallel threads are created to evaluate each variable. This is done by means of the auxiliary function decls, given in Figure 12.

In pH a program computation finishes only when every created thread has been completely evaluated. Therefore, we do not differentiate between minimal and maximal semantics (in fact, the definition given here corresponds to the maximal semantics).

$$
\begin{array}{l}
\mathsf{force} :: \mathbf{Loc} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\,\alpha\,\kappa = \lambda\sigma.\mathtt{case}\ (\sigma\,\alpha)\ \mathtt{of} \\
\quad \varepsilon \in \mathbf{Abs} \longrightarrow \kappa\,\varepsilon\,\sigma \\
\quad \nu \in \mathbf{Clo} \longrightarrow \nu\,\kappa'\,\sigma' \\
\qquad \mathtt{where}\ \kappa' = \lambda\varepsilon''.\lambda\sigma''.\kappa\,\varepsilon''\,\sigma'' \oplus \{l \mapsto \varepsilon''\} \\
\qquad\qquad \sigma' = \sigma \oplus \{l \mapsto \mathrm{not\_ready}\} \\
\quad \mathtt{otherwise} \longrightarrow \mathsf{wrong} \\
\mathtt{endcase}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{decls} :: \mathcal{P}_f(\mathbf{Ide}) \to \mathbf{Env} \to \mathbf{Cont} \\
\mathsf{decls}\,\emptyset\,\rho = \lambda\sigma.\{\sigma\} \\
\mathsf{decls}\,I\,\rho = \\
\quad = \lambda\sigma. \bigcup_{x \in I} (\bigcup_{\sigma_x \in \Sigma_x} \mathsf{decls}\,(I \setminus \{x\})\,\rho\,\sigma_x) \\
\qquad \mathtt{where}\ \Sigma_x = \mathcal{E}[\![x]\!]\rho\, id_\kappa\, \sigma
\end{array}
$$

Fig. 12. Auxiliary semantic functions for pH-core

5 Conclusions and future work

We have used continuations to model laziness, parallelism, non-determinism and side-effects in a denotational semantics.

The differences between the three approaches —explicit, semi-explicit and implicit parallelism— are reflected in the semantic domains, where only in the first case (explicit parallelism) some notion of process is needed. Moreover, the distributed (not shared memory) nature of Eden, complicates quite a lot the semantics, because bindings have to be copied from one process to the others, while in the other two cases there are no restrictions in sharing memory.

The explicit parallelism of Eden requires special domains for representing processes: process identifiers, **IdProc**, and sets of channels, **SChan**. By contrast, GpH and pH do not need these particular domains. However, not only the domains do vary, but also the definition of the semantic function differs. The explicit parallelism of Eden resides in the evaluation of `$#`, where the structure of a new process is created, i.e. the corresponding channels. A point where the differences clearly arise is in the`let`evaluation: GpH does nothing special because these new variables are evaluated only if they are demanded, consequently they are just added to the environment; pH evaluates in parallel all the local variables and simultaneously to the main expression; thanks to the expression continuation all these variables are evaluated. Eden only evaluates –if it is not the minimal case– the variables associated to processes creation; once again this task is developed by means of the expression continuation.

Obviously, GpH with no processes, no communications, no non-determinism, and no side-effects has the simplest semantics. However simple, this semantics can help to detect speculative parallelism by comparing the final environments under the minimal and under the maximal semantics. Even though, this probably can also be done without continuations.

The semantics presented in this paper allows to extract the degree of parallelism and the amount of speculative computation. For instance, in the case of Eden, a set of channels defines an oriented graph whose nodes correspond to process identifiers and edges are labelled with the communicated values. Therefore the number of nodes of this graph coincides with the number of processes in the corresponding system. By modifying the definition of the expression continuation for the parallel application, other approaches between the minimal and the maximal semantics are possible in this framework. The speculation degree can therefore be obtained as the difference between the number of nodes in a non-minimal graph and the number of nodes in the minimal one.

Concerning pH programs, a speculative location is one that has not been needed for obtaining the final result. We could define an alternative "minimal" expression continuation for the`let`expression that only introduces the closures in the store, but it does not evaluate it. Comparing the final store in the semantics given en Section 4 with the one derived from this new expression continuation we observe that the locations that do not longer appear in the new minimal version are speculative. Moreover, for a location that appears in both stores if it is bound to an expressed value or a cell in the non-minimal semantics but it is bound to a closure in the minimal, then this location is also speculative.

The analysis of the speculation in GpH follows the same ideas that we have outlined above for pH, but changing locations in the store by variables in the environment.

On the other hand, the abstraction level of the present denotational model does not permit to observe work duplication because this has to do with the copy of variables from one process to another.

One of our future tasks is to use the denotational semantics presented here to relate formally some notions concerning these languages. For instance, we intend to compare the behaviour of the communication channels of Eden with the cells of pH: an I-cell is like a channel where a thread writes some value that other threads can later read, but once this value has been set, the I-cell cannot be filled again; similarly, a channel is a one-use-only entity, although the reader of the value is unique in this case.

Acknowledgements: This work has been partially supported by the Spanish project CICYT-TIC2000-0738. We are very grateful to David de Frutos for his comments on earlier versions of Eden's semantics and to the APLAS'03 anonymous reviewers for their helpful criticisms.

References

[AAA+95] S. Aditya, Arvind, L. Augustsson, J. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, editor, *Haskell Workshop*, pages 35–49, La Jolla, Cambridge, MA, USA, YALEU/DCS/RR-1075, June 1995.

[BFKT00] C. Baker-Finch, D. King, and P. 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.

[BLOMP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden – the paradise of functional-concurrent programming. In *EUROPAR'96: European Conference on Parallel Processing*, pages 710–713. LNCS 1123, Springer, 1996.

[DB97] M. Debbabi and D. Bolignano. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 6: A semantic theory for ML higher-order concurrency primitives, pages 145–184. Monographs in Computer Science. Ed. F. Nielson. Springer, 1997.

[EM02] M. van Ekelen 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.

[FH99] W. Ferreira and M. Hennessy. A behavioural theory of first-order CML. *Theoretical Computer Science*, 216:55–107, 1999.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HI93] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. *Information and Computation*, 107:202–236, 1993.

[HOM02] 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.

[Jos86] M. B. Josephs. *Functional programming with side-effects.* PhD thesis, Oxford University, 1986.

[Jos89] M. B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP'77*, pages 993–998. Eds. B. Gilchrist. North-Holland, 1977.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *POPL'93*, Charleston, 1993.

[Loo99] R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs. Eds. K. Hammond and G. Michaelson. Springer, 1999.

[NA01] R. S. Nikhil and Arvind. *Implicit Parallel Programing in pH.* Academic Press, 2001.

[Nik91] R. S. Nikhil. Id (version 90.1) language reference manual. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge, MA, USA, 1991.

[Pey87] S. Peyton Jones. *Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pey03] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press, 2003.

[Rep92] J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University (Deparyment of Computer Scienqe), 1992.

[Rey98] J. C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[THM+96] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.