Parallel Processing Letters

AN OPERATIONAL SEMANTICS FOR THE PARALLEL LANGUAGE EDEN

MERCEDES HIDALGO-HERRERO
Dpt. Didáctica de las Matemáticas, Universidad Complutense de Madrid
C/ Rector Royo-Villanova s/n, Madrid, 28040, Spain
mhidalgo@edu.ucm.es

and

YOLANDA ORTEGA-MALLÉN
Dpt. Sistemas Informáticos y Programación, Universidad Complutense de Madrid
Avda. Complutense s/n, Madrid, 28040, Spain
yolanda@sip.ucm.es

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

The functional parallel language Eden—suitable for the description of parallel and concurrent algorithms in a distributed setting—is an extension of Haskell with a set of coordination features. In this paper we present a formal operational semantics for the *kernel* of Eden, or more precisely, for a λ-calculus widened with explicit parallelism and potentially infinite communication channels. Eden overrides the lazy nature of Haskell on behalf of parallelism. This interplay between laziness and eagerness is accurately described by the semantics proposed here, which is based on Launchbury's natural semantics for lazy evaluation, and is expressed through a two-level transition system: a lower level for the local and independent evaluation of each process, and an upper one for the coordination between all the parallel processes in the system. As processes are created either under demand or in a speculative way, different scheduling strategies are possible—ranging from a minimal one that only allows the main thread to evolve, to a maximal one that evolves in parallel every active binding.

1. Introduction

When designing the programming language Eden[1,2], it was our purpose to make profit of computing distributed environments, which are more and more common and available, while staying in the realm of functional programming. There are basically three alternatives for the exploitation of the parallelism inherent to functional programs:

- to automatically parallelize programs (implicit parallelism);

- to add to the program some indications on how the parallelism can be derived by the compiler (controlled parallelism);
- to extend a functional language with constructions for the creation of parallel processes and communication between them (explicit parallelism).

R. Loogen discusses at length in [3] the parallelization of declarative languages, and provides a good overview of the work done following the three approaches mentioned above. We claim that the third alternative is the most appropriate for parallel programming in a distributed setting, because the programmer can have a greater control of the granularity—and thus obtain a better task distribution—through explicit process definitions. Therefore, this has been the approach adopted for Eden. With respect to communication (and synchronization), in the absence of common shared memory, message passing is the best choice but, in order to keep more closely to the declarative relish, explicit sending/receive formulas (like in [4,5]) are avoided, and implicit communication through streams [6] is considered. Moreover, to be able to describe not only pure *transformational* programs, but also *reactive* systems, a restricted form of non-determinism for merging two or more streams into a single one is introduced, so that a process may attend input coming from several channels without being hold up in any of them.

Although lazy evaluation implies normal order evaluation, and avoids repeated computations through reduction sharing, a parallel system where every task is strictly done under demand seems slightly promising. It is better to have some amount of work done in advance, so that the results are ready when they are needed; even if this means producing some *speculative* computation. Thus, Eden combines lazy function application with eager process creation and eager evaluation of communication values; this interplay between laziness and eagerness cannot be well understood without a precise formal semantics. It is for this purpose that we propose an operational semantics in the line of Launchbury's *natural* semantics for lazy evaluation [7], which provides an accurate model of sharing by viewing closures as name/expression bindings, which are collected in a heap representing the program space. In our case, a distributed system will be represented by a collection of separate heaps—one for each process in the system—while distinct variables to represent communication channels are introduced. Unlike to Launchbury's semantics, ours needs small-step transitions to model parallelism in a synchronous way, i.e. single reductions are local and independently carried out at each process and then combined before proceeding to the next step. As it is usual for parallel and concurrent languages, our semantics embodies two levels of transition systems: one lower level to handle the *local* behavior of processes, and an upper level to describe *global* effects on the process system.

An operational semantics—also based on Launchbury's work—for parallel lazy evaluation is presented in [8]. Although we have taken profit from some of the ideas given there (like labelling bindings for modelling thread states), there are great differences in form and concept between that work and ours. On the one hand, the semantics given in [8] is applied to a simple subset of the language GpH [9] without any notion of process or communication, and without any form of non-determinism; it just models a single heap with several active threads which compete for the available processors. On the other hand, the authors are much concerned with extracting some metrics for comparing programs in terms of time and parallelism, instead of with showing the effects of mixing lazy and strict evaluation.

In [10] we have already presented an operational semantics for a λ-calculus just extended with process creation and single-value communication, which focuses on the exact amount of speculative computation produced. With respect to this first attempt, the present work offers a much more elaborated model, and introduces significant extensions such as stream communication and non-determinism.

In the next section, we introduce a small language including Eden's coordination features. The semantic model is described in section , while section is devoted to the definition of the operational semantics, whose transitions rules are described in sections and , while section describes how the model reflects the degree of speculation. We conclude with a brief reflection on what has been achieved, and give some hints of future work.

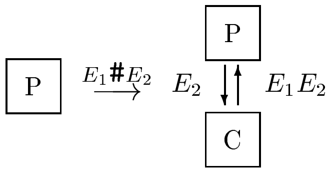
2. Mini-Eden

Eden extends the lazy functional language Haskell [11] by a coordination model, which is derived from the approach of stream-based communication [6]. The semantics described in this paper considers a subset of Eden consisting of an untyped λ-calculus extended with recursive lets and some coordination features: process creation, non-determinism, streams for communication, and dynamic channels, as defined in figure 1, where $\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$ stands for $\texttt{let}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ E$.

Syntax			**Restricted Syntax**	
E	$::=$	x	x	variable
	$\mid$	$E_1 E_2$	Ex	application
	$\mid$	$E_1 \# E_2$	$x \# E$	process creation
	$\mid$	$\lambda x.E$	$\lambda x.E$	λ-abstraction
	$\mid$	$\Lambda[x_1 : x_2].E_1 [\!] E_2$	$\Lambda[x_1 : x_2].E_1 [\!] E_2$	Λ-abstraction
	$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	local declaration
	$\mid$	$E_1 \bowtie E_2$	$x_1 \bowtie x_2$	non-determinism
	$\mid$	$\texttt{new}(y, x)\ \texttt{in}\ E$	$\texttt{new}(y, x)\ \texttt{in}\ E$	channel creation
	$\mid$	$x\,!\,E_1\ \texttt{par}\ E_2$	$x\,!\,E_1\ \texttt{par}\ E_2$	channel connection
	$\mid$	L		list
L	$::=$	$\texttt{nil}$	$\texttt{nil}$	empty list
	$\mid$	$[E_1 : E_2]$	$[x_1 : x_2]$	non-empty list

Fig. 1. Mini-Eden syntax

The evaluation of a Mini-Eden expression branches in several parallel processes by using the #-operator for applying abstractions to their arguments. Therefore, when evaluating an expression $E_1 \# E_2$, the new process (the child) is fed by its creator (the parent) by sending the value for E_2 via an input channel, and returns the value for $E_1 E_2$ (to its parent) through an output channel, as the following diagram illustrates:



Both communications are evaluated speculatively, that is, their evaluation is initiated although there is no actual demand for their values. This is a point where our language combines laziness and eagerness: while application is non-strict, there is an *eager* demand for communication values. Moreover, when #-expressions are found at top-level, *eager process creation* takes place. The following example illustrates this point:

Example 1 Laziness vs. eagerness.

Let us consider the following expression:

$$\texttt{let}\ x = (\lambda x_1.\lambda x_2.x_1)\#2, y = (\lambda y_1.3 * y_1)\#4, z = (\lambda z_1.z_1)8\ \texttt{in}\ xz,$$

where we have introduced numeric constants and operators for clarity.

Provided an unlimited number of processors, the evaluation of this local declaration produces the following steps:

(i) Introduction of local variables x, y and z in the main process environment.
(ii) Creation of two processes (children of the main process) one to apply the first λ-abstraction to 2, obtaining in this way the abstraction for the `let` body, and the other one to multiply 3 and 4.
(iii) Demand on the variable x.
(iv) Evaluation of applications $(\lambda x_1.\lambda x_2.x_1)2$ and $(\lambda y_1.3 * y_1)4$ (although the latter has not been really demanded, i.e. it is not needed for the evaluation of another expression) in the newly created processes.
(v) Evaluation, in parallel with the evaluations in (iv), of the application $(\lambda x_2.2)z$ in the main process, resulting the value 2.

It can be observed that z is not demanded by the functional application and, in concordance with the lazy evaluation, it is not evaluated. However, although y would not be demanded in a pure lazy setting, a new process is created and its output expression is evaluated. □

Eden forsakes laziness in another aspect: there is no reduction sharing between processes, because these are closed entities communicating through channels, and any other form of (implicit) information sharing is not allowed. This makes the language suitable for distributed environments, but may lead to repeated computations. Moreover, due to race conditions, the amount of repeated work may vary on different computations of the same program, as explained in the next example:

Example 2 The following expression, in the framework of a process, may lead to evaluate x before copying it to the child:

$$\texttt{let}\ z = \lambda z_1.z_1, y = z\#(x\#3), x = (\lambda x_1.x_1)(\lambda x_2.x_2)\ \texttt{in}\ x.$$

The evaluation of this expression demands x, but it also creates a first child process bound to y. Consequently, the inport for this process is demanded as well. This inport is a process creation where the x binding has to be copied to the second child, but this copy may take place either before or after x has been evaluated in the parent. □

The communication of more than one value through a single channel is achieved by using streams of normal form values, i.e. lists which are element-wise produced and sent to the consumer. In order to deal with lists, we have introduced in Mini-Eden Λ-abstractions: a construction which combines function definition and pattern-matching. The application of Λ-abstractions is strict in its argument, which has to be a list, and proceeds by evaluating the first expression when the argument list is empty, or by evaluating the second one in the other case.

We introduce a restricted form of non-determinism ($E_1 \bowtie E_2$) for merging two streams into a single one. The resulting list is obtained by pasting together the elements in the order they are produced by the respective processes.

Communication goes, so far, in a hierarchical manner from parent to child, and from child to parent. For a greater flexibility in communication, we include in our language the possibility of creating channels dynamically ($\texttt{new}(y, x)\,\texttt{in}\,E$). The name of the channel will be sent from one process to another, until some process *uses* it ($z\,!\,E_1\,\texttt{par}\,E_2$), establishing the connection with the process which generated that dynamic channel. Further attempts to use the same dynamic channel will abort the program execution.[a]

Following [7], we first assume a general renaming of variables which transforms an expression E into a renamed one $\hat{E}$. This allows our semantics to consider heaps in which all bound variables are distinct. Secondly, to simplify the transition rules in the operational semantics, the language is normalized to a restricted syntax (see figure 1) consistent with the non-strict nature of each construction: the second expression in applications and the first one in process creations are required to be variables; to make non-empty lists shareable, they are decomposed in head and tail variables; in order to properly culminate the laziness of the $\bowtie$ operator (no demand on its arguments) and the eager process creation and stream evaluation, this operator is also restricted to have variables as arguments. For instance, the expression $((\lambda y.y)\#(\lambda x.x)E_2) \bowtie ((\lambda z.z)\#[(\lambda t.t) : \texttt{nil}])$ is normalized to

$$\begin{array}{l}\texttt{let}\ y_1 = (\texttt{let}\ x_1 = \lambda y.y\ \texttt{in}\ x_1\#(\texttt{let}\ x_2 = E_2\ \texttt{in}\ (\lambda x.x)x_2)),\\ \quad\ \ y_2 = (\texttt{let}\ x_3 = \lambda z.z\ \texttt{in}\ x_3\#(\texttt{let}\ x_4 = \lambda t.t, x_5 = \texttt{nil}\ \texttt{in}\ [x_4 : x_5])\\ \texttt{in}\ y_1 \bowtie y_2.\end{array}$$

3. A Distributed Model

The evaluation of a Mini-Eden expression requires, in general, the creation of several parallel processes. Each process will, in turn, encompass a set of independently executing threads—each thread is devoted to the production of one output of the process—that share the process data.

Usually, the evaluation of an expression ends up when a weak head normal form (whnf) value ($W \in \mathit{Val}$) is reached. However, except for abstractions, the values

[a] Trying to connect a dynamic channel with two different producers is considered a program flaw, because this situation might introduce a different kind of non-determinism.

to be communicated between processes ($CV \in CVal$) have to be in normal form. Consequently, values are classified in three syntactic categories, as it is expressed in the following table:

$CV ::= \lambda x.E \mid \Lambda[x_1 : x_2].E_1 [\![E_2 \mid \mathsf{ch}(p,x) \mid LV$	communication values
$LV ::= \mathtt{nil} \mid [CV : LV]$	list values
$W ::= CV \mid [E_1 : E_2]$	whnf values

The value $\mathsf{ch}(p,x)$ represents a dynamic channel name, where p is the name of the creator process (the consumer), and x is the input variable associated to the channel in p. Lists never appear directly in normal form because the head and tail are shared (see the restricted syntax given in figure 1).

In order to observe the sharing—or not sharing—of reductions (see [7]) under this mixture of lazy and eager evaluation, we model the evaluation state by a heap of bindings of expressions to variables, representing closures. As we assume no shared memory between processes, we distribute the global heap into separate heaps, each corresponding to one process in the system. We will then denote a process by $\langle p, H\rangle$, where p is the process name and H is the bindings heap. Following [8], each binding is considered a potential thread, and gets associated a label indicating the thread state: $x \overset{\alpha}{\mapsto} e$, where $\alpha ::= I|A|B$ corresponds, respectively, to *Inactive* (either not yet demanded or already completely evaluated), *Active* (or demanded) and *Blocked* (demanded but waiting for the value of another binding).

In the following sections, in order to avoid writing multiple similar transition rules, we allow a binding to appear with several labels, corresponding to the different possibilities admitted by the rule. Thus, if for instance $x \overset{IAB}{\mapsto} E$ appears in the left side of a rule, and $x \overset{ABA}{\mapsto} E'$ in the right one, it means that the thread corresponding to closure $x \mapsto E$ becomes active in the case it was either inactive or blocked, while it becomes blocked if it was previously active. Besides, notation $H + \{x \overset{\alpha}{\mapsto} E\}$ means that the heap H is extended with the binding $x \overset{\alpha}{\mapsto} E$, while $H : x \overset{\alpha}{\mapsto} E$ means that the binding for x is the one which guides the application of the corresponding rule. The set $dom(H)$ contains the left-side variables of a heap H.

Let $\mathcal{C}han$ be the set of channel identifiers; as communication in our calculus is unidirectional—from parent to child or viceversa—we split $\mathcal{C}han$ into two disjoint sets: $\mathcal{C}han = \mathcal{I}n \cup \mathcal{O}ut$ where, taking the point of view of the child, $\mathcal{I}n$ corresponds to *input* channels, i.e from parent to child, while $\mathcal{O}ut$ corresponds to *output* channels, i.e from child to parent.

In the following, we will use $x, y, z \in Var$ for "ordinary variables", and $i \in \mathcal{I}n$, $o \in \mathcal{O}ut$ or $ch \in \mathcal{C}han$ for channels, while $\gamma \in Var \cup \mathcal{C}han$. A special kind of variables is introduced for indicating evaluation to normal form ($\mathcal{NF}$), so that we will consider $w \in \mathcal{NF}$, $\nu \in Var \cup \mathcal{NF}$, and $\theta, \rho \in Var \cup \mathcal{C}han \cup \mathcal{NF}$.

4. The Transition System

Similarly to many operational semantics given for concurrent or parallel functional languages, our semantics involves two levels of transitions. At the lower level, each process evolves local and independently carrying out functional computations like β-reduction or `let`-binding. All these local evolutions are considered to be simultaneous, so that they get entwined in a parallel (global) step (see section). At a higher level do take place the interactions between the processes in the system, namely process creation and communication.

4.1. Local Transitions

The transitions rules given in this section model the reduction of an active thread in the context of a single process. This internal activity has effect only on the corresponding heap: bindings may be created, modified, blocked or activated.

Local transitions take the form $H : x \stackrel{A}{\mapsto} E \longrightarrow H'$, which is read as "the evaluation of the active thread $x \stackrel{A}{\mapsto} E$ transforms the heap $H + \{x \stackrel{A}{\mapsto} E\}$ into H'".

$$H + \{x \stackrel{I}{\mapsto} W\} : \theta \stackrel{A}{\mapsto} x \;\longrightarrow\; H + \{x \stackrel{I}{\mapsto} W, \theta \stackrel{A}{\mapsto} \hat{W}\} \qquad \textbf{(value)}$$

$$H + \{w \stackrel{I}{\mapsto} LV\} : \theta \stackrel{A}{\mapsto} w \;\longrightarrow\; H + \{w \stackrel{I}{\mapsto} LV, \theta \stackrel{A}{\mapsto} \hat{LV}\} \qquad \textbf{(NF value)}$$

$$\text{if } E \notin \mathit{Val} \qquad H + \{\nu \stackrel{IAB}{\mapsto} E\} : \theta \stackrel{A}{\mapsto} \nu \;\longrightarrow\; H + \{\nu \stackrel{AAB}{\mapsto} E, \theta \stackrel{B}{\mapsto} \nu\} \qquad \textbf{(demand)}$$

$$H : \nu \stackrel{A}{\mapsto} \nu \longrightarrow\; H + \{\nu \stackrel{B}{\mapsto} \nu\} \qquad \textbf{(blackhole)}$$

$$H : \theta \stackrel{A}{\mapsto} (\lambda y.E)x \;\longrightarrow\; H + \{\theta \stackrel{A}{\mapsto} E[x/y]\} \qquad \textbf{(β-reduction)}$$

$$\frac{H : \theta \stackrel{A}{\mapsto} E \;\longrightarrow H' + \{\theta \stackrel{\alpha}{\mapsto} E'\}}{H : \theta \stackrel{A}{\mapsto} E\, x \;\longrightarrow\; H' + \{\theta \stackrel{\alpha}{\mapsto} E'\, x\}} \qquad \textbf{(application)}$$

$$H : \theta \stackrel{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}\ \texttt{in}\ E \;\longrightarrow\; H + \{x_1 \stackrel{I}{\mapsto} E_1, \ldots, x_n \stackrel{I}{\mapsto} E_n, \theta \stackrel{A}{\mapsto} E\} \qquad \textbf{(let)}$$

Fig. 2. Local transition rules: Basic λ-calculus

Basic λ-calculus. The first set of rules, given in figure 2, express accurately how lazy evaluation progresses under demand. Notice that a λ-abstraction can never be directly applied over a channel (the normalization phase introduces an intermediary program variable). Channel self-references are imposible, because a variable in a heap cannot get bound to a channel belonging to the same corresponding process, i.e. $\theta \stackrel{\alpha}{\mapsto} ch \in H \implies ch \notin dom(H)$.

(Λ-demand)
if $E \notin \mathit{Val}$

$$H + \{x \stackrel{IAB}{\mapsto} E\} : \theta \stackrel{A}{\mapsto} (\Lambda[y_1 : y_2].E_1 [\![E_2)x \;\longrightarrow\; H + \{x \stackrel{AAB}{\mapsto} E, \theta \stackrel{B}{\mapsto} (\Lambda[y_1 : y_2].E_1 [\![E_2)x\}$$

(B-reduction empty list)

$$H + \{x \stackrel{I}{\mapsto} \texttt{nil}\} : \theta \stackrel{A}{\mapsto} (\Lambda[y_1 : y_2].E_1 [\![E_2)x \;\longrightarrow\; H + \{x \stackrel{I}{\mapsto} \texttt{nil}, \theta \stackrel{A}{\mapsto} E_1\}$$

(B-reduction non-empty list)

$$H + \{x \stackrel{I}{\mapsto} [z_1 : z_2]\} : \theta \stackrel{A}{\mapsto} (\Lambda[y_1 : y_2].E_1 [\![E_2)x] \;\longrightarrow\; H + \{x \stackrel{I}{\mapsto} [z_1 : z_2], \theta \stackrel{A}{\mapsto} E_2[z_1/y_1, z_2/y_2]\}$$

Fig. 3. Local transition rules: Λ-abstraction

Λ-abstraction. The treatment of Λ-abstractions (see figure 3) is similar to that of λ-abstractions, except for that application of the former is blocked until the argument is in whnf. Then "B-reduction" discriminates between the empty list and the non-empty list cases and, in the latter, variables y_1 and y_2 are instantiated to the head and tail of the list, respectively. It is supposed that Λ-abstractions are applied exclusively to expressions which will evaluate to lists, or there will never be an applicable rule.

Streams. We must distinguish the evaluation of a list from the evaluation of a stream, because the former is lazy whereas the latter is eager, i.e. when evaluating a stream its head is reduced until a communication value is reached, while in the case of a list, the complete evaluation of each element in the list is not forced:

(stream demand)

$$\text{if } E \notin Val \qquad H + \{\nu_1 \overset{IAB}{\mapsto} E\} : ch \overset{A}{\mapsto} [\nu_1 : \nu_2] \;\longrightarrow\; H + \{\nu_1 \overset{AAB}{\mapsto} E, ch \overset{B}{\mapsto} [\nu_1 : \nu 2]\}$$

Merge. The $\bowtie$-operator builds the resulting list with all its elements in normal form. In the expression $x_1 \bowtie x_2$ neither x_1 nor x_2 are demanded. Instead, it is intended that each x_i $(i \in \{1, 2\})$ is bound to some output stream, which will be eagerly evaluated by another process. As soon as one of the two argument streams obtains a normal form value at its head, this value can become part of the result list. The following rules, where z is a fresh variable, model this behavior:

(left merge)

$$\text{if } (\texttt{rc}\ W\ H) \qquad \begin{array}{c} H + \{x_1 \overset{I}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow \\ \longrightarrow H + \{x_1 \overset{I}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} [y_1 : z], z \overset{A}{\mapsto} y_2 \bowtie x_2\} \end{array}$$

(right merge)

$$\text{if } (\texttt{rc}\ W\ H) \qquad \begin{array}{c} H + \{x_2 \overset{I}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow \\ \longrightarrow H + \{x_2 \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} [y_1 : z], z \overset{A}{\mapsto} x_1 \bowtie y_2\} \end{array}$$

where the function rc (ready for communication) tests whether a whnf value is in normal form:

$$\texttt{rc}\ W\ H = \begin{cases} \texttt{rc}\ E_1\ H \wedge \texttt{rc}\ E_2\ H & \text{if } W \equiv [x_1 : x_2] \\ & \quad \text{where } H = H' + \{x_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} E_2\} \\ \texttt{true} & \text{otherwise} \end{cases}$$

The merger is finished when both streams become empty.

(merge termination)

$$H + \{x_1 \overset{I}{\mapsto} \texttt{nil}, x_2 \overset{I}{\mapsto} \texttt{nil}\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow H + \{x_1 \overset{I}{\mapsto} \texttt{nil}, x_2 \overset{I}{\mapsto} \texttt{nil}, \theta \overset{A}{\mapsto} \texttt{nil}\}$$

The $\bowtie$-operator gets blocked if its arguments are not streams or their heads are not ready for communication (see figure 4).

Dynamic channels. A new channel is created when evaluating an expression $\texttt{new}(y, x)\ \texttt{in}\ E$, which declares a local variable (x) in the scope of the expression that will consume the communications (E). This variable is associated to the output of the channel, whose producer will be determined posteriorly. Therefore, x is initially bound to the special value $\not\leftrightarrow$ (disconnected), and the corresponding thread is blocked. The variable representing the name of the channel (y) is bound to a

(blocking merge 1)

$$\text{if } E_1, E_2 \notin \mathit{List} \vee (E_1 \notin \mathit{List} \wedge E_2 \equiv \mathtt{nil}) \vee (E_2 \notin \mathit{List} \wedge E_1 \equiv \mathtt{nil})$$

$$H + \{x_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 2)

$$\text{if } \neg(\mathtt{rc}\ E_1\ H) \wedge (E_2 \notin \mathit{List} \vee E_2 \equiv \mathtt{nil})$$

$$H + \{x_1 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 3)

$$\text{if } (E_1 \notin \mathit{List} \vee E_1 \equiv \mathtt{nil}) \wedge \neg(\mathtt{rc}\ E_2\ H)$$

$$H + \{x_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 4)

$$\text{if } \neg(\mathtt{rc}\ E_1\ H) \wedge \neg(\mathtt{rc}\ E_2\ H)$$

$$\langle p, H + \{x_1 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} [y_1' : y_2'], y_1' \overset{\alpha}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2\rangle \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha}{\mapsto} [y_1 : y_2], y_1 \overset{\alpha}{\mapsto} E_1, x_2 \overset{\alpha}{\mapsto} [y_1' : y_2'], y_1' \overset{\alpha}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

Fig. 4. Local transition rules: Blocking merge

special semantic expression $\mathsf{ch}(\mathit{self}, x)$;[b] that can be sent through other channels until one of the receiving processes connects the channel and becomes the producer:

(channel creation)

$$H : \theta \overset{A}{\mapsto} \mathtt{new}(y, x)\ \mathtt{in}\ E \longrightarrow H + \{\theta \overset{A}{\mapsto} E, y \overset{I}{\mapsto} \mathsf{ch}(\mathit{self}, x), x \overset{B}{\mapsto} \langle\rangle\}$$

The evaluation of a `par` expression has to be blocked until the channel name expression is obtained:

(channel name demand)

$$H + \{x \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\,!\,E_1\ \mathtt{par}\ E_2 \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\,!\,E_1\ \mathtt{par}\ E_2\}$$

Normal form. As explained in section , only normal forms can be communicated. Therefore, each element of a stream has to be evaluated to normal form. In the case of the communication element being a list, it is bound to a new special (fresh) variable $w \in \mathcal{NF}$. Only lists associated to variables of this kind are evaluated to normal form:

(normal form demand)

$$H + \{x_1 \overset{I}{\mapsto} [\nu_1 : \nu_2]\} : ch \overset{A}{\mapsto} [x_1 : \nu'] \ \longrightarrow\ H[w/x_1] + \{w \overset{A}{\mapsto} [\nu_1 : \nu_2], ch \overset{B}{\mapsto} [w : \nu']\}$$

To evaluate a list to normal form, head and tail must be both reduced to normal form; fresh variables w_1 and w_2 are introduced:

(head demand)

$$H : w \overset{A}{\mapsto} [\nu_1 : \nu_2] \ \longrightarrow\ H + \{w \overset{B}{\mapsto} [w_1 : w_2], w_1 \overset{A}{\mapsto} \nu_1, w_2 \overset{I}{\mapsto} \nu_2\}$$

4.2. Global Transitions

[b] *self* denotes the name of the process where this rule is being applied, i.e. the owner of the heap.

At an upper level, we have global transitions between process systems (S), i.e. sets of processes. A global transition takes the general form:

$$\{\langle p, H_p \rangle\}_{p \in S} \xrightarrow{\diamond} \{\langle p, H'_p \rangle\}_{p \in S \cup S'}$$

where each heap H_p (associated to a process p in S) is transformed to H'_p, while new processes (in S') may be created. In each rule the processes involved in the transformation are highlighted.

The tasks to be done at the system level are: process creation, channel connection, interprocess communication and state management (thread unblocking and deactivation). In general, these tasks imply multiple single steps, each involving at most two processes. Let S be a process system, and $\diamond$ the name of a rule, for each single-step rule $S \xrightarrow{\diamond} S'$ we can define a multi-step rule $S \xRightarrow{\diamond} S'$ satisfying:

1. $S \xrightarrow{\diamond}^{*} S'$ and,
2. there is no S'' such that $S' \xrightarrow{\diamond} S''$.

The application of a single-step rule $\diamond$ to a particular binding in a particular process, may enable the application of that same rule $\diamond$ to other bindings—in the same or in other processes—but it cannot disable applications of $\diamond$ which were enabled before the former application.

Process creation. New processes are created when evaluating #-expressions, by applying the following rule:

$$\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textbf{(process creation)}\\
\text{if } \neg\mathtt{d}(x, H) \qquad\qquad\qquad\qquad\qquad \mathit{fresh}(c, y, i, o), \mathit{freshrenaming}(\eta)\\
(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\mathtt{\#}E\}\rangle) \xrightarrow{pc} (S, \langle p, H + \{\theta \overset{B}{\mapsto} o, i \overset{A}{\mapsto} E\}\rangle, \langle c, \eta(\mathtt{nh}(x, H)) + \{o \overset{A}{\mapsto} \eta(x)\, y, y \overset{B}{\mapsto} i\}\rangle)
\end{array}$$

As mentioned before, processes are eagerly created when they are found at the top-level, i.e. when a variable in a heap is directly bound to a #-expression, even if that binding is not active (i.e. demanded).

When creating a process, the thread evaluating the #-expression (at the *parent* side) is blocked on a fresh output channel $o \in \mathcal{O}ut$, corresponding to the initial thread in the new *child* process c. Correspondingly, the child process gets a thread which is blocked on a new input channel $i \in \mathcal{I}n$, that is served by a new thread in the parent (communication from parent to child).

As there is no common shared heap, all bindings needed for the evaluation of the free variables in the process body must be copied from the parent to the child heap, using for this purpose the function `nh` (needed heap) defined in figure 5, so that $\mathtt{nh}(E, H)$ collects all the bindings in H that are reachable from E. Moreover, in order to keep all the names distinct, even if they belong to different heaps, we rename—by means of η—the copied closures. However, when the evaluation of the process body depends on a value to be communicated from some other process, the process creation is delayed until the necessary communications and instantiations have taken place. The detection of pending communications is achieved by the function `d` (dependencies) defined in figure 6, and only when this function returns `false` the collect of bindings is invoked (`nh`).

$$\mathtt{nh}(\nu, H) = \begin{cases} \{\nu \overset{I}{\mapsto} E\} \cup \mathtt{nh}(E, H) & \text{if } \nu \overset{\alpha}{\mapsto} E \in H \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl}
\mathtt{nh}(ch, H) & = & \emptyset \\
\mathtt{nh}(\mathtt{nil}, H) & = & \emptyset \\
\mathtt{nh}(E\,x, H) & = & \mathtt{nh}(E, H) \cup \mathtt{nh}(x, H) \\
\mathtt{nh}(x\mathtt{\#}E, H) & = & \mathtt{nh}(x, H) \cup \mathtt{nh}(E, H) \\
\mathtt{nh}(\lambda x.E, H) & = & \mathtt{nh}(E, H) \\
\mathtt{nh}(\Lambda[x_1 : x_2].E_1 [\![E_2, H) & = & \mathtt{nh}(E_1, H) \cup \mathtt{nh}(E_2, H) \\
\mathtt{nh}(x_1 \bowtie x_2, H) & = & \mathtt{nh}(x_1, H) \cup \mathtt{nh}(x_2, H) \\
\mathtt{nh}(\mathtt{new}(y, x)\,\mathtt{in}\,E, H) & = & \mathtt{nh}(E, H) \\
\mathtt{nh}(z\,\mathtt{!}\,E_1\,\mathtt{par}\,E_2, H) & = & \mathtt{nh}(z, H) \cup \mathtt{nh}(E_1, H) \cup \mathtt{nh}(E_2, H) \\
\mathtt{nh}([E_1 : E_2], H) & = & \mathtt{nh}(E_1, H) \cup \mathtt{nh}(E_2, H)
\end{array}$$

$$\mathtt{nh}(\mathtt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \mathtt{in}\,E, H) = \bigcup_{i=1}^{n} \mathtt{nh}(E_i, H) \cup \mathtt{nh}(E, H)$$

Fig. 5. Collecting bindings

$$\mathtt{d}(\nu, H) = \begin{cases} \mathtt{false} & \text{if } \nu \overset{\alpha}{\mapsto} E \notin H \\ \mathtt{true} & \text{if } \nu \overset{\alpha}{\mapsto} E \in H \text{ with } E \equiv ch \vee E \equiv E_1\mathtt{\#}E_2 \vee E \equiv \nleftrightarrow \vee E \equiv \mathsf{ch}(p, x) \\ \mathtt{d}(E, H) & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl}
\mathtt{d}(\mathtt{nil}, H) & = & \mathtt{false} \\
\mathtt{d}(E\,x, H) & = & \mathtt{d}(E, H) \vee \mathtt{d}(x, H) \\
\mathtt{d}(x\mathtt{\#}E, H) & = & \mathtt{d}(x, H) \vee \mathtt{d}(E, H) \\
\mathtt{d}(\lambda x.E, H) & = & \mathtt{d}(E, H) \\
\mathtt{d}(\Lambda[x_1 : x_2].E_1 [\![E_2, H) & = & \mathtt{d}(E_1, H) \vee \mathtt{d}(E_2, H) \\
\mathtt{d}(x_1 \bowtie x_2, H) & = & \mathtt{d}(x_1, H) \vee \mathtt{d}(x_2, H) \\
\mathtt{d}(\mathtt{new}(y, x)\,\mathtt{in}\,E, H) & = & \mathtt{d}(E, H) \\
\mathtt{d}(z\,\mathtt{!}\,E_1\,\mathtt{par}\,E_2, H) & = & \mathtt{d}(z, H) \vee \mathtt{d}(E_1, H) \vee \mathtt{d}(E_2, H) \\
\mathtt{d}([E_1 : E_2], H) & = & \mathtt{d}(E_1, H) \vee \mathtt{d}(E_2, H)
\end{array}$$

$$\mathtt{d}(\mathtt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \mathtt{in}\,E, H) = \bigvee_{i=1}^{n} \mathtt{d}(E_i, H) \vee \mathtt{d}(E, H)$$

Fig. 6. Detecting pending communications

The process creation rule introduces new bindings, but it modifies only the one corresponding to the #-expression. As a consequence, the creation of one process cannot disable the creation of other processes. By contrast, it may bring new top-level #-expressions. For instance, in an expression like $x\mathtt{\#}(y\mathtt{\#}\lambda z.z)$ the created process will be fed with $y\mathtt{\#}\lambda z.z$, thus implying the creation of a new process for evaluating $y(\lambda z.z)$. Even then, the number of processes that can be created in one multi-step is always finite: only expressions with a similar form to the one given in the previous example can generate new top-level process creations, but, as expressions are finite, the number of # in them is finite too. Besides, the number of bindings in the system is finite as well. Therefore, an infinite process creation loop is impossible, and the corresponding multi-step rule $\overset{pc}{\Longrightarrow}$, which carries out every possible process creation, is well defined.

Communication. The rules concerning communication are given in figure 7, distinguishing between streams and one-single value communications.

(single value communication)

$$\text{if } \neg\mathtt{d}(CV, H_p) \qquad \mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} CV\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{vCom}{\longrightarrow}$$
$$\overset{vCom}{\longrightarrow} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathtt{nh}(CV, H_p)) + \{\theta \overset{A}{\mapsto} \eta(\widehat{CV})\}\rangle)$$

(head-stream communication)

$$\text{if } \neg\mathtt{d}(W, H) \wedge (\mathtt{rc}\ W\ H_p) \qquad \mathit{fresh}(y_1, y_2), \mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} [\nu : x], \nu \overset{I}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{sCom}{\longrightarrow}$$
$$\overset{sCom}{\longrightarrow} (S, \langle p, H_p + \{ch \overset{A}{\mapsto} x, \nu \overset{I}{\mapsto} W\}\rangle,$$
$$\langle c, H_c + \eta(\mathtt{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{A}{\mapsto} \eta(W), y_2 \overset{B}{\mapsto} ch\}\rangle)$$

Fig. 7. Single steps for global transitions: Communication.

When the value to be communicated corresponds to an abstraction, it is mandatory the copy—from the producer's heap to the consumer's heap—of all the bindings needed for the evaluation of the free variables in the abstraction. Similarly to the case of process creation, this copy can take place only if there is no dependency on pending communications. The renaming (η) for the heap is applied to the passed value too. A renaming of the bound variables of the abstraction is also needed.

In the case of streams, whenever its head is ready for communication (`rc` is `true`), the consumer will receive the corresponding value composed with a fresh variable, which is bound to the tail of the stream. The stream is closed when it finally reaches the empty list status, which is dealt with by the single value communication rule.

The communication of a value—either a single one or the head of a stream—may give rise to more communications. Both rules in figure 7 have to be applied intertwined and repeatedly until no communication is possible, so that we define the multi-step rule for communication as $\overset{Com}{\Longrightarrow} = (\overset{sCom}{\Longrightarrow} \circ \overset{vCom}{\Longrightarrow})^*$.

Although a communication may enable additional ones, this never leads to an infinite number of communications (in one system step) because the number of communication channels in the system is finite at every instant, and list values—bound to channels in the heaps—are finite.

Variables that are already bound to values are not affected by "external" communications, so that the application of a communication rule to a channel does not prevent communication through other channels.

Threads unblocking. When a binding finally reaches its value, the threads blocked on it are unblocked and the binding is deactivated. The corresponding rules are gathered in figure 8, where E_B^θ denotes an expression that is immediately blocked on θ, i.e. one of the following: $\theta \mid \theta\, x \mid (\Lambda[y_1 : y_2]E_1 \| E_2)\theta$; while $E_{\bowtie}^x$ stands for merge expressions involving variable x, i.e. $x \bowtie x'$ or $x' \bowtie x$.

There are two cases for unblocking a $\bowtie$-expression: either there is a communication value at the head of one of the merged streams (unblocking merge 1), or both streams have been closed down, i.e. they are bound to `nil` (unblocking merge 2).

When deactivating bindings, we must distinguish between whnfs and normal forms. In order to complete the evaluation to normal form of a list, the evaluation of the tail has to be forced (tail demand), although it is not demanded until the head

(WHNF unblocking)

$$(S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{B}{\mapsto} E^x_B\}\rangle) \overset{wUnbl}{\longrightarrow} (S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{A}{\mapsto} E^x_B\}\rangle)$$

(NF unblocking)

$$(S, \langle p, H + \{w \overset{A}{\mapsto} LV, \theta \overset{B}{\mapsto} w\}\rangle) \overset{nfUnbl}{\longrightarrow} (S, \langle p, H + \{w \overset{A}{\mapsto} LV, \theta \overset{A}{\mapsto} w\}\rangle)$$

(unblocking merge 1)

$$(S, \langle p, H + \{x \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} CV, \theta \overset{B}{\mapsto} E^x_{\bowtie}\}\rangle) \overset{mUnbl1}{\longrightarrow}$$
$$\overset{mUnbl1}{\longrightarrow} (S, \langle p, H + \{x \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{I}{\mapsto} CV, \theta \overset{A}{\mapsto} E^x_{\bowtie}\}\rangle)$$

(unblocking merge 2)

$$(S, \langle p, H + \{x_1 \overset{AI}{\mapsto} \mathtt{nil}, x_2 \overset{AI}{\mapsto} \mathtt{nil}, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}\rangle) \overset{mUnbl2}{\longrightarrow}$$
$$\overset{mUnbl2}{\longrightarrow} (S, \langle p, H + \{x_1 \overset{AI}{\mapsto} \mathtt{nil}, x_2 \overset{AI}{\mapsto} \mathtt{nil}, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}\rangle)$$

(tail demand)

$$\text{if } \neg(\mathtt{rc}\ W\ H) \quad \begin{array}{l} (S, \langle p, H + \{w \overset{B}{\mapsto} [w_1 : w_2], w_1 \overset{I}{\mapsto} W, w_2 \overset{I}{\mapsto} y\}\rangle) \overset{tDem}{\longrightarrow} \\ \overset{tDem}{\longrightarrow} (S, \langle p, H + \{w \overset{B}{\mapsto} [w_1 : w_2], w_1 \overset{I}{\mapsto} W, w_2 \overset{A}{\mapsto} y\}\rangle) \end{array}$$

(WHNF deactivation)

$$(S, \langle p, H + \{\gamma \overset{A}{\mapsto} W\}\rangle) \overset{wDeact}{\longrightarrow} (S, \langle p, H + \{\gamma \overset{I}{\mapsto} W\}\rangle)$$

(NF deactivation)

$$\text{if } CV \notin \mathit{List} \vee CV \equiv \mathtt{nil} \qquad (S, \langle p, H + \{w \overset{A}{\mapsto} CV\}\rangle) \overset{nfDeact}{\longrightarrow} (S, \langle p, H + \{w \overset{I}{\mapsto} CV\}\rangle)$$

Fig. 8. Single steps for global transitions: Waking-up threads

is ready for communication; otherwise, undesirable parallelism would be introduced in the system.

The iteration and combination of the rules in figure 8 gives rise to the rule:

$$\overset{Unbl}{\Longrightarrow} = \overset{nfDeact}{\Longrightarrow} \circ \overset{wDeact}{\Longrightarrow} \circ \overset{tDem}{\Longrightarrow} \circ \overset{mUnbl2}{\Longrightarrow} \circ \overset{mUnbl1}{\Longrightarrow} \circ \overset{nfUnbl}{\Longrightarrow} \circ \overset{wUnbl}{\Longrightarrow} .$$

The number of blocked threads is finite, and it is not increased by unblocking or deactivation. This guarantees the finiteness of *Unbl*.

Channel connection. A process p that has received some dynamic channel name $\mathsf{ch}(c, x)$ (bound to z) becomes its producer by evaluating an expression $z\,!\,E_1\ \mathtt{par}\ E_2$:

(channel connection)

$$(S, \langle p, H_p + \{z \overset{I}{\mapsto} \mathsf{ch}(c,x), \theta \overset{AB}{\mapsto} z\,!\,E_1\ \mathtt{par}\ E_2\}\rangle, \langle c, H_c + \{x \overset{B}{\mapsto} \diamond, y \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle) \overset{conn}{\longrightarrow}$$
$$\overset{conn}{\longrightarrow} (S, \langle p, H_p + \{\theta \overset{A}{\mapsto} E_2, o \overset{A}{\mapsto} E_1, z \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle, \langle c, H_c + \{x \overset{B}{\mapsto} o, y \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle)$$

A new outport o is created and bound to the expression E_1, and the variable x is bound to this outport. Meanwhile, the evaluation of the whole expression continues with E_2. Connection is only possible with disconnected channels.

Multiple connections—of different channels—are carried out by using the corresponding multi-step rule $\overset{Conn}{\Longrightarrow}$.

System evolution. After each individual process has evolved through local transitions, the global system evolution is represented by

$$\stackrel{sys}{\Longrightarrow} = \stackrel{Conn}{\Longrightarrow} \circ \stackrel{Unbl}{\Longrightarrow} \circ \stackrel{pc}{\Longrightarrow} \circ \stackrel{Com}{\Longrightarrow}$$

that is, first of all, all pending communications are carried out, afterwards new processes are created. Next, unblocking (and deactivation) of threads takes place. Finally, dynamic channels are connected.

At the highest level, each transition step in the system is defined by

$$\Longrightarrow = \stackrel{sys}{\Longrightarrow} \circ \stackrel{par}{\Longrightarrow}$$

where ($\stackrel{par}{\Longrightarrow}$), collecting the synchronous local evolution of every process in the system, is defined in the next section.

4.3. Scheduling Considering Speculation

Evaluation of a program may give rise to different computations. The exact amount of speculative parallelism depends on the number of available processors, the scheduler decisions and the speed of basic instructions. Hence, the execution of a program may range from reducing the speculation to the minimum—only what it is effectively demanded is computed—to expanding it to the maximum—every speculative computation is carried out— While the former would be equivalent to executing the program on a single processor with the scheduler giving priority to the demand originated by the initial process, the latter would correspond to having an unlimited set of processors for evaluating the output of every generated process. Moreover, if a reduction sequence for a program expression E is defined as a—finite or infinite—sequence of configurations

$$\langle p_0, \{main \stackrel{A}{\mapsto} E\}\rangle \Longrightarrow^* \langle p_0, H + \{main \stackrel{I}{\mapsto} W\}\rangle, \langle p_1, H_1\rangle, \ldots, \langle p_n, H_n\rangle \Longrightarrow^*$$

then, in the minimal semantics, the final configuration is the first one where the main variable becomes inactive, while in the case of the maximal semantics, the execution continues until every process that has been created is finished or get blocked, i.e. there is no active thread in the system.

Each process evolves internally by performing local transitions (see section), so that the local evolution of a process p is determined by the set of parallel (active) threads in H_p (the bindings heap for p) that are allowed to progress (evolutionary threads, $EvTh(H_p)$):

$$\textbf{(parallel-p)}\quad \frac{\{H_p^{(i,1)} + H_p^{(i,2)} : \theta_p^i \stackrel{A}{\mapsto} E_p^i \longrightarrow H_p^{(i,1)} + K_p^{(i,2)} | \theta_p^i \stackrel{A}{\mapsto} E_p^i \in EvTh(H_p)\}_{i=1}^{n_p}}{\langle p, H_p\rangle \stackrel{par}{\Longrightarrow}_p \langle p, (\cap_{i=1}^{n_p} H_p^{(i,1)})[\cup_{i=1}^{n_p} K_p^{(i,2)}]\rangle}$$

where $H_p = H_p^{(i,1)} + H_p^{(i,2)} + \{\theta_p^i \stackrel{A}{\mapsto} E_p^i\}$, $n_p = |EvTh(H_p)|$, $H_p^{(i,1)}$ is the part of H_p that remains unchanged after the application of the corresponding local rule, and $K_p^{(i,2)}$ represents the modifications in H_p. The notation $H[K]$ denotes a heap H modified by the bindings in K, that is, $H[K] = \{\nu \stackrel{\alpha}{\mapsto} E \in H \mid \nu \notin dom(K)\} \cup K$.

For the (parallel-p) rule to be sound, it is necessary that each active thread can evolve in parallel without interference with other active threads in the same process. This is expressed by the following proposition:

Proposition 1 *Given two distinct variables θ_1 and θ_2, and a heap $H = H^{(i,1)} + H^{(i,2)} + \{\theta_i \overset{A}{\mapsto} E_i\}$, $i \in \{1,2\}$ so that $H^{(i,1)} + H^{(i,2)} : \theta_i \overset{A}{\mapsto} E_i \longrightarrow H^{(i,1)} + K^{(i,2)}$, $i \in \{1,2\}$. Then:*

1. *If $\theta' \overset{IB}{\mapsto} E' \in K^{(1,2)}$ then $\theta' \notin dom(K^{(2,2)})$*
2. *If $\theta' \overset{A}{\mapsto} E' \in K^{(1,2)}$ then either $\theta' \notin dom(K^{(2,2)})$ or $\theta' \overset{A}{\mapsto} E' \in K^{(2,2)}$* □

The proof, by induction on the applied rule, is not included.

The parallel evolution of all the processes in a system S is defined by the rule

(parallel)
$$\frac{\{\langle p, H_p\rangle \overset{par}{\Longrightarrow}_p \langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}$$

Minimal semantics. If no speculation is allowed, only what is ever demanded by the main computation is evaluated. Thus, in order not to get lost in speculative computation, we have to give preference to the work demanded from the main process. This work is represented by pairs of the form $\langle \mathit{variable}, \mathit{process}\rangle$ which are collected by the function `pre` (preference). In general, for $\langle p, H\rangle \in S$ and $\nu \in dom(H)$ we define

$$\mathtt{pre}(\nu, \langle p, H\rangle, S) = \begin{cases} \{\langle \nu, \langle p, H\rangle\rangle\} & \text{if } \nu \overset{A}{\mapsto} E \in H \\ \bigcup_\rho \mathtt{pre}(\rho, \langle p, H\rangle, S) & \text{if } \nu \overset{B}{\mapsto} E^\rho_B \in H \vee \nu \overset{B}{\mapsto} E^\rho_{\bowtie} \\ \bigcup_{ch} \mathtt{pre}(ch, \langle q, K\rangle, S) & \text{if } \nu \overset{B}{\mapsto} E_{ch} \in H \ \wedge ch \in dom(K) \end{cases}$$

where, E^ρ_B denotes an expression immediately blocked on ρ, and $E^\rho_{\bowtie}$ stands for $\bowtie$-expressions involving variable ρ (see "threads unblocking" in section), while E_{ch} denotes an expression depending on channel $ch \in \mathit{Chan}$, i.e. one of the following: $ch \mid x\#E$ with $ch \in \mathtt{nc}(x, H)$. The `nc` (needed channels) function collects all the channels which may be necessary for evaluating x, and its definition is the same as for `nh` except for the base cases:

$$\mathtt{nc}(\nu, H) = \begin{cases} \{ch\} & \text{if } \nu \overset{\alpha}{\mapsto} ch \in H \\ \mathtt{nc}(E, H) & \text{if } \nu \overset{\alpha}{\mapsto} E \in H \wedge E \notin \mathit{Chan} \wedge E \not\equiv E_1\#E_2 \wedge E \not\equiv \,\not\Leftrightarrow \wedge E \not\equiv \mathsf{ch}(p, x) \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathtt{nc}(ch, H) = \{ch\}$$

Let $\langle p_0, H_0\rangle$ be the initial process—H_0 is the unique heap in the system which contains the variable *main*—then $\mathtt{pm}(S) = \mathtt{pre}(\mathit{main}, \langle p_0, H_0\rangle, S)$ contains all the active threads "demanded" by *main*. This set may contain more than one thread for two reasons: firstly, if a thread is blocked on a $\bowtie$-expression, then it depends on two variables; secondly, a blocked process creation may be waiting for more than one communication.

The minimal semantics is obtained by defining $\mathit{EvTh}_{\min}(H_p) = H_p \cap \mathit{EvTh}_{\min}$ for each $\langle p, H_p\rangle \in S$, where $\mathit{EvTh}_{\min} = \mathtt{pm}(S)$. Processes are still speculatively created, but their bodies are evaluated only if their output is really demanded. As $EvTh_{\min}$ contains only threads demanded by the main expression, when *main* is bound inactive to a whnf value, this set is empty and the computation ends.

Maximal semantics. At each step, every active thread in the system evolves in parallel. More formally, if we define the set of active threads in a heap H as $H^A = \{\theta \overset{A}{\mapsto} E \mid \theta \in dom(H)\}$, then $EvTh_{\max}(H) = H^A$.

Considering that the initial expression is well formed—errors such as $[y_1 : y_2]\, y$ or an attempt to connect twice the same dynamic channel, cannot appear in the evaluation—then, whenever a thread is active, there is a local rule for the evolution of such a thread. This fact is formalized in the following proposition:

Proposition 2 *Let $E_0 \notin Val$ be a well formed closed expression and let $\langle p_0, \{main \overset{A}{\mapsto} E_0\}\rangle \Longrightarrow^* (S, \langle p, H + \{\theta \overset{A}{\mapsto} E\}\rangle)$ be a sequence of configurations. If E is not a #-expression,*[c] *then there exists H' so that $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$ in process p.* □

The proof, by structural induction on E, is not included.

The following example illustrates the difference between the minimal and the maximal semantics.

Example 3 Let $P \equiv \texttt{let}\ x = y\texttt{\#}\lambda t.t,\ y = \lambda z.z\ \texttt{in}\ y$, whose evaluation generates a speculative process.

In both semantics, the speculative process is created. However, in the minimal one the evaluation of the child output is delayed because it is not necessary for the evaluation of the main expression, and the final system is:

$$\langle main, \{main \overset{I}{\mapsto} \lambda z.z, x \overset{B}{\mapsto} o, y \overset{I}{\mapsto} \lambda z.z\}\rangle, \langle p, \{o \overset{A}{\mapsto} y\, w, w \overset{I}{\mapsto} \lambda t.t, y \overset{I}{\mapsto} \lambda z.z\}\rangle.$$

By contrast, the maximal semantics does not restrict the threads which will evolve and, thus, the evaluation of the child output is not delayed:

$$\langle main, \{main \overset{I}{\mapsto} \lambda z.z, x \overset{I}{\mapsto} \lambda t.t, y \overset{I}{\mapsto} \lambda z.z\}\rangle, \langle p, \{w \overset{I}{\mapsto} \lambda t.t, y \overset{I}{\mapsto} \lambda z.z\}\rangle.$$

Let us now consider $Q \equiv \texttt{let}\ x = y\texttt{\#}\lambda t.t,\ y = \lambda z.z\ \texttt{in}\ x$. The evaluation of Q yields the same final system in both semantics:

$$\langle main, \{main \overset{I}{\mapsto} \lambda z.z, x \overset{I}{\mapsto} \lambda t.t, y \overset{I}{\mapsto} \lambda z.z\}\rangle, \langle p, \{w \overset{I}{\mapsto} \lambda t.t, y \overset{I}{\mapsto} \lambda z.z\}\rangle.$$

Notice also that processes P and Q are equivalent under the maximal semantics. □

The previous example has shown how, in the absence of speculative computations, the output of both semantics generate the same final system. However, when at least one of the created processes is not needed, minimal and maximal semantics produce different systems for the same program.

5. Conclusions and Future Work

[c] Notice that this kind of expressions evolve through global rules.

We have given a semantic model for most significant constructions of the parallel functional language Eden, namely lazy function application, eager process creation and value production for communication via streams, a non-deterministic operator for merging input streams, and dynamic channels.

The mixture of lazy and eager evaluation has been accurately modelled through the following features:

- Labelled name/expression bindings to simulate closures as in [8].
- Independent processes (represented by binding heaps) to model the computation state.
- Two levels of transitions: local and global.
- Different degrees of speculative computation.

Although we have considered unlimited resources, the semantics can be easily adapted to model a system with a limited number of processors, just by defining a different set of evolutionary threads (*EvTh*) for the rule (parallel-p) in section .

So far, we have just given a transition system that simulates the execution of programs,[d] obtaining for each process a set of possible computations. Our next task is to identify properties over computation sets that will be useful for the establishment of equivalences between processes. For instance, we conjecture that in the absence of the merger operator, the final value obtained by the different computations of a program will be unique, although the speculative work may be different.

The main goal for developing the operational semantics presented here was to offer a way to obtain some metrics of parallelism from a theoretical point of view, and to allow the programmer to formally analyze the consequences of some program decisions. This is particularly interesting for languages like Eden, where speculative parallelism is controlled by the programmer, in contrast with other approaches like [12] where it is done automatically by the compiler, as an optimization for the program execution. As it is shown in [10], by introducing accumulators in each process and special labels in the bindings, we can modify the rules in the transition system for measuring the speculative parallelism.

The semantics can also be employed to establish the usefulness of some program transformations which have been developed for Eden [13], and to prove properties of skeletons defined in this language [14].

We are actually working on a denotational semantics for mini-Eden, where continuations are used to model process creation and communications as side effects [15] in a distributed environment. A preliminary version—without streams and non-determinism—can be found in [16]. We intend to obtain some fully-abstractness results of the denotational semantics with respect to the operational one.

Acknowledgements

This work has been supported by the Spanish project CICYT-TIC2000-0738 and the Spanish-British *Acción Integrada* HB1999-0102.

[d]Presently we are implementing this transition system, to obtain a kind of interpreter for mini-Eden, which will be very useful for the derivation and analysis of behavioural properties.

References

[1] 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.

[2] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

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

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

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

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

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

[8] 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.

[9] 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.

[10] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for a parallel functional language. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 89–102. Intellect, 2000.

[11] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98. URL http://www.haskell.org, February 1999.

[12] J. S. Mattson Jr. and W. G. Griswold. Local speculative evaluation for distributed graph reduction. In *Proceedings of Glasgow Workshop of Functional Programming*, pages 185–102. Springer WCS, 1993.

[13] 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.

[14] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 4: Parallelism Abstractions in Eden. Eds. F. A. Rabhi and S. Gorlatch. Springer, 2002.

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

[16] M. Hidalgo-Herrero and Y. Ortega-Mallén. Towards a denotational semantics for Eden. In *Draft Proceedings of the Third Scottish Functional Programming Workshop*, pages 45–54. University of Stirling, Scotland, 2001.