Chapter 8

A Distributed Operational Semantics for a Parallel Functional Language

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

Abstract: We present an operational semantics for a functional parallel language with explicit process creation and implicit message-passing communication. The semantics is based on a distributed memory model and is effective for investigating the interplay between laziness and eagerness in the language, as well as for measuring speculative parallelism.

8.1 INTRODUCTION

The combination of the parallel programming and the functional languages worlds has been proved fruitful and remains still alive and active, as gives evidence the recent recompilation by K.Hammond and G.Michaelson [HM99]. Concerning parallel functional languages, research has been mostly devoted to their implementation and performance evaluation, while formal semantical issues have been mainly relegated to a secondary attention. There is no much wonder about this fact, considering that many proposals were done at the level of *implicit* parallelism, where the system tries to exploit the inherent parallelism and is, therefore, semantically transparent. This transparency property also holds for so-called *controlled* parallelism approaches, like para-functional programming, evaluation strategies or skeletons. But there is still a third level of parallel functional programming: functional languages with an *explicit* notion of a process and including constructs for

[1]Dept.Didáctica de las Matemáticas, Facultad de Educación.Universidad Complutense de Madrid, Spain; Email: `mhidalgo@eucmax.sim.ucm.es`

[2]Dept.Sistemas Informáticos y Programación, Facultad CC.Matemáticas. Universidad Complutense de Madrid, Spain; Email: `yolanda@sip.ucm.es`

defining process topologies. This kind of languages are practical to parallelise *transformational* systems —limited to compute a final result from some given input— as well as for describing *reactive* systems, i.e. those maintaining some interaction with the environment. In general, explicit parallelism provides a language with additional expressive power and, therefore, requires some extension of the semantics as well. Different approaches have been proposed for giving semantics to these concurrent functional languages. Prominent is the work around Concurrent ML (CML) [PR97]. Most of these proposals [DB97, FH99, PR97] derive from research in the field of process algebra, where the keypoint is the observational behaviour of process systems, in terms of their external communications.

Turning back to parallel programming and transformational systems, there is no external observational behaviour except for the input-output relation, and the main concern is the speed-up of the underlying algorithm. It is the target of the present work to obtain a semantic framework helpful for the comparison of parallel programs in terms of required resources, but keeping in an adequate high level. Our interest focuses on lazy languages. More exactly, we aim at defining a semantics for Eden [BLOMP97], an extension of Haskell [PH99] with explicit process definition and creation, and implicit message-passing communication. In Eden there is no shared memory; this makes the language specially suitable for being implemented in a distributed setting. The drawback is that a closure may be evaluated more than once by different processes. Moreover, in order to propitiate parallelism, Eden overrules laziness by instantiating processes in a speculative way. Therefore, we are greatly interested in obtaining a semantic model suitable for studying the interplay between laziness and eagerness in the language, and for measuring the amount of speculative computation produced during program execution.

We extend and adapt the operational semantics for parallel lazy evaluation given in [BFKT00], which is based on Launchbury's *natural* semantics for lazy evaluation [Lau93], where closures are modeled as name/expression bindings which are collected in a heap representing the program space. In [BFKT00] bindings are used to model threads, which are executed in the available processors, but sharing a unique heap. However, following the distributed nature of Eden, in our extension each process is represented by a separate heap, and distinct variables are introduced to represent communication channels.

In the next section we introduce a kernel language based on Eden. The operational semantics and its transitions are described in Sections 8.3 and 8.4, while in Section 8.5 we describe how to extend the semantic model to extract the degree of speculative parallelism in a program. We conclude with a brief outlook to further extensions of the present model.

8.2 A KERNEL LANGUAGE

For the presentation of our semantics we shall consider a very simple subset of the Eden language, consisting of the untyped λ-calculus extended with numbers, recursive lets and *process instantiations*, as given in figure 8.1.

Kernel language	Restricted syntax	
$e ::= n$	n	number
$\mid x$	x	variable
$\mid e_1 e_2$	$e\ x$	application
$\mid e_1 \# e_2$	$x \# e$	instantiation
$\mid \lambda x.e$	$\lambda x.e$	λ-abstraction
$\mid \mathtt{let}\ x_1 = e_1$	$\mathtt{let}\ x_1 = e_1$	
$\dots$	$\dots$	local declaration
$x_n = e_n\ \mathtt{in}\ e$	$x_n = e_n\ \mathtt{in}\ e$	

FIGURE 8.1. Kernel language and restricted syntax

Unlike the kernel language presented here, Eden introduces different syntactic constructs for defining functions and *process abstractions*, i.e. abstract schemes for processes. Although it is useful in practice to make such a distinction, semantically we can identify both concepts as λ-abstractions and distinguish between (function) application and (process) instantiation. At execution time, an instantiation expression $e_1 \# e_2$ originates the creation of a process to evaluate the application expression $e_1 e_2$. Two channels are established between parent and child: one for communicating the value of e_2 from the parent to the child, and a second one for communicating the result value of the application from the child to the parent.

The language combines laziness and eagerness. Namely, while application is non-strict, there is an eager demand on the output produced by processes, implying eager process instantiation. Actually, in $e_1 \# e_2$ the first expression, e_1, is lazily evaluated by the child, while e_2 is eagerly evaluated by the parent.

Following [Lau93], we first assume a renaming of variables which transforms an expression e into a renamed one $\hat{e}$. This allows the semantics to consider heaps in which all bound variables are distinct. Secondly, to simplify the transition rules in the semantics, the language is normalised to a restricted syntax consistent with the non-strict nature of each construction, i.e. the second expression in applications and the first one in instantiations are required to be variables.

8.3 A DISTRIBUTED MODEL

The evaluation of an expression in our kernel language will, in general, require the creation of several parallel processes. Each process will, in turn, encompass a set of independently executing threads, each devoted to the production of one output of the process. In a distributed setting, we assume no shared memory. Each process, together with all the necessary data, will be allocated to some processor, which will be shared between all the existing threads in the process. From our semantic point of view, the distinction between processors is completely irrelevant. Therefore, we ideally suppose the existence of a different processor for each process. The limitation in the number of processors will be reflected in a bound on the number of processes effectively executing at each instant.

Each process is represented by a separate heap of bindings of expressions to variables, corresponding to closures. We will then denote a process by $\langle p, H \rangle$, where p is the process name and H is the bindings heap. Following [BFKT00], each binding is considered a potential thread and gets associated a label indicating the thread state:

$$x \overset{\alpha}{\mapsto} e$$

where $\alpha ::= I|B|R|A$, corresponding, respectively, to *Inactive* (either not still initiated or already finished), *Blocked* (waiting for the value of another binding), *Runnable* (ready to evaluate when there is an avalaible processor) and *Active* (currently evaluating).

In the following sections, 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.

For representing communication channels between processes, a set of channel identifiers $\mathcal{C}$ is introduced. Communication in the kernel language is unidirectional, from parent to child or viceversa. Taking the point of view of the child, we consider $\mathcal{C} = \mathcal{I} \cup \mathcal{O}$, where $\mathcal{I}$ corresponds to *input* channels, i.e from parent to child, while $\mathcal{O}$ corresponds to *output* channels, i.e from child to parent. In the following, we will use $x, y, z \in \mathit{Var}$ (ordinary variables), $i \in \mathcal{I}$, $o \in \mathcal{O}$ and $ch \in \mathcal{C}$ (channels), while $\theta, \rho \in \mathit{Var} \cup \mathcal{C}$.

8.4 THE TRANSITION SYSTEM

Similar to the majority of operational semantics for concurrent or parallel functional languages, our semantics involves two levels of transitions. At the lower level, each process evolves local and independently evaluating ordinary expressions such as β-reduction or `let`-binding. All these local evolutions are considered to be simultaneous and get entwined in a parallel step. At a higher level are the interactions between the processes in the system, namely process creation and communication. With these two kinds of transition rules we stablish a reduction sequence from an initial configuration to a final configuration:

$$\langle p_0, \{main \overset{A}{\mapsto} e\}\rangle \implies^* \langle p_0, H + \{main \overset{I}{\mapsto} v\}\rangle, \langle p_1, H_1\rangle, \ldots, \langle p_n, H_n\rangle$$

where *main* is the program identifier to be evaluated; value v is a weak head normal form (whnf) expression in the kernel language, that is $v ::= n \,|\, \lambda x.e.$; and notation $H + \{x \overset{\alpha}{\mapsto} e\}$ means that H is extended with the binding $x \overset{\alpha}{\mapsto} e$.

8.4.1 Local transitions

Local transitions, which are given in figure 8.2, model the reduction of an active binding/thread in the context of a single heap/process. This internal activity only has effect on the corresponding heap: bindings may be created, modified or blocked. In some cases a thread may become runnable, but no thread is activated at this local level. Activation corresponds exclusively to the scheduler at the system level (see subsection 8.4.2). These local rules express accurately how lazy evaluation progresses under demand and there is no novelty with respect to the semantics given in [BFKT00].

(value)

$$\langle p, H + \{x \overset{I}{\mapsto} v, \theta \overset{A}{\mapsto} x\}\rangle \longrightarrow \langle p, H + \{x \overset{I}{\mapsto} v, \theta \overset{A}{\mapsto} \hat{v}\}\rangle$$

(blocking)

$$\frac{e \notin Val}{\langle p, H + \{x \overset{IBR}{\mapsto} e, \theta \overset{A}{\mapsto} x\}\rangle \longrightarrow \langle p, H + \{x \overset{RBR}{\mapsto} e, \theta \overset{B}{\mapsto} x\}\rangle}$$

(blackhole)

$$\langle p, H + \{x \overset{A}{\mapsto} x\}\rangle \longrightarrow \langle p, H + \{x \overset{B}{\mapsto} x\}\rangle$$

(β-reduction)

$$\langle p, H + \{\theta \overset{A}{\mapsto} (\lambda y.e)x\}\rangle \longrightarrow \langle p, H + \{\theta \overset{A}{\mapsto} e[x/y]\}\rangle$$

(application)

$$\frac{\langle p, H + \{\theta \overset{A}{\mapsto} e\}\rangle \longrightarrow \langle p, H' + \{\theta \overset{\alpha}{\mapsto} e'\}\rangle}{\langle p, H + \{\theta \overset{A}{\mapsto} e\,x\}\rangle \longrightarrow \langle p, H' + \{\theta \overset{\alpha}{\mapsto} e'\,x\}\rangle}$$

(let)

$$\langle p, H + \{\theta \overset{A}{\mapsto} \mathtt{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e\}\rangle \longrightarrow$$
$$\longrightarrow \langle p, H + \{x_1 \overset{I}{\mapsto} e_1, \ldots, x_n \overset{I}{\mapsto} e_n, \theta \overset{A}{\mapsto} e\}\rangle$$

FIGURE 8.2. Local transition rules

The restrictions on the kind of variables to be considered in each rule are the outcome of some properties on channel variables. For instance, an application can never be over a channel (it is always over a variable introduced in the restricted syntax), and a variable in a heap can never be bound to a channel belonging to the corresponding process, i.e. $\theta \overset{\alpha}{\mapsto} ch \in H \implies ch \notin dom(H)$, where $dom(H)$ contains all the variables at the left hand side of the bindings in H. The latter also implies that autoreferences of channels can never happen. Lastly, the new bindings introduced by the (let) rule never correspond to channels, because they are not program variables.

8.4.2 Global transitions

At an upper level, we have global transitions between process systems, i.e. sets of processes. The tasks to be done at the system level are listed below:

- Process creation.
- Interprocess communication.
- Unblocking of threads.
- Activation/deactivation of threads.

In general, these tasks imply multiple single steps involving at most two processes, which are continuedly applied until they cannot longer be employed. If we denote a process system by S, and † represents the name of a rule, for each single step rule $S \xrightarrow{\dagger} S'$ we can define a multi step rule

$$S \stackrel{\dagger}{\Longrightarrow} S'$$

satisfying:

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

Process creation

New processes are created when executing process instantiations. Related to this activity we present two single step rules in figure 8.3.

(instantiation)

$$\begin{array}{c} \text{if } \mathtt{nc}(x,H) = \emptyset \\ \mathit{fresh}(i,o,y) \\ (S, \langle p, H + \{\theta \stackrel{\alpha}{\mapsto} x\#e\}\rangle) \\ \xrightarrow{ins} (S, \langle p, H + \{\theta \stackrel{B}{\mapsto} o, i \stackrel{R}{\mapsto} e\}\rangle, \langle c, \eta(\mathtt{nh}(x,H)) + \{o \stackrel{R}{\mapsto} x\,y, y \stackrel{B}{\mapsto} i\}\rangle) \end{array}$$

(blocking instantiation)

$$\begin{array}{c} \text{if } \mathtt{nc}(x,H) \neq \emptyset \\ (S, \langle p, H + \{\theta \stackrel{\alpha}{\mapsto} x\#e\}\rangle) \xrightarrow{bIns} (S, \langle p, H + \{\theta \stackrel{B}{\mapsto} x\#e\}\rangle) \end{array}$$

FIGURE 8.3. Single steps for global transitions: Process creation.

A main characteristic of our language is top-level instantiation which originates speculative parallelism by applying the (instantiation) rule not only to active (i.e. demanded) bindings.

When creating a new process, the thread evaluating the instantiation expression (at the *parent* side) is blocked on a fresh output channel $o \in O$, 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 $i \in I$, which is served by a new thread in the parent (communication from parent to child).

As processes can only communicate through channels, and there is no common shared heap, all bindings needed for the evaluation of the free variables in the process abstraction must be copied from the parent to the child heap. Moreover, in order to keep all the names distinct, even if they belong to different heaps, we rename (η) the copied closures. The function $\mathtt{nh}(e,H)$ collects all the bindings in H that are reachable from e. However, when the evaluation of the process abstraction depends on a value to be communicated from some other process, the instantiation is delayed. The detection of dependency on channel values is achieved by the function `nc`. We combine both functions in a single function `nec` which returns a pair of sets:

$$\mathtt{nec}(e,H) = \langle \mathtt{nh}(e,H), \mathtt{nc}(e,H) \rangle$$

The definition for `nec` is given in figure 8.4, where the union symbol $\cup$ must be understood as pairwise set union.

$$
\begin{array}{lcl}
\mathtt{nec}(n,H) & = & \langle \emptyset, \emptyset \rangle \\
\mathtt{nec}(ch,H) & = & \langle \emptyset, \{ch\} \rangle \\
\mathtt{nec}(x,H) & = & \begin{cases} \langle \emptyset, \{ch\} \rangle & \text{if } x \overset{\alpha}{\mapsto} ch \in H \\ \langle \{x \overset{I}{\mapsto} e\}, \emptyset \rangle \cup \mathtt{nec}(e,H) & \text{if } x \overset{\alpha}{\mapsto} e \in H \wedge e \notin C \\ \langle \emptyset, \emptyset \rangle & \text{otherwise} \end{cases} \\
\mathtt{nec}(e\,x,H) & = & \mathtt{nec}(e,H) \cup \mathtt{nec}(x,H) \\
\mathtt{nec}(x\#e,H) & = & \mathtt{nec}(x,H) \cup \mathtt{nec}(e,H) \\
\mathtt{nec}(\lambda x.e,H) & = & \mathtt{nec}(e,H)
\end{array}
$$

$$\mathtt{nec}(\mathtt{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e, H) = \bigcup_{e=1}^{n} \mathtt{nec}(e_i,H) \cup \mathtt{nec}(e,H)$$

FIGURE 8.4. Collecting bindings and detecting channel dependencies

The instantiation can be carried out only when all the bindings needed to evaluate the process abstraction do no longer depend on channel values. In the other case, the thread must be blocked in order not to waste processor time, but there is no need of an *unblocking* instantiation rule. The corresponding multi step rules are combined in a single rule:

$$\stackrel{pc}{\Longrightarrow} = \stackrel{bIns}{\Longrightarrow} \circ \stackrel{ins}{\Longrightarrow}$$

Interprocess communication

When the value to be communicated corresponds to a λ-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 λ-abstraction. Similar to the case of process creation, this copy can only take place if there is no dependency on other channels. The renaming (η) for the heap is applied to the value passed too. A renaming of the bound variables of the λ-abstraction is also needed. The rules for communication are given in figure 8.5. Notice that after the communication of the value the channel *ch* disappears.

(value communication)

$$\text{if } \mathtt{nc}(v, H_p) = \emptyset$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} v\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle)$$
$$\overset{vCom}{\longrightarrow} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathtt{nh}(v, H_p)) + \{\theta \overset{R}{\mapsto} \eta(\hat{v})\}\rangle)$$

(blocking communication)

$$\text{if } \mathtt{nc}(v, H) \neq \emptyset$$
$$(S, \langle p, H + \{ch \overset{\alpha}{\mapsto} v\}\rangle) \overset{bCom}{\longrightarrow} (S, \langle p, H + \{ch \overset{B}{\mapsto} v\}\rangle)$$

FIGURE 8.5. Single steps for global transitions: Communication.

The communication phase comprises the repeated application of the two rules:

$$\overset{com}{\Longrightarrow} = \overset{bCom}{\Longrightarrow} \circ \overset{vCom}{\Longrightarrow}$$

Scheduling

In a setting with limited resources, namely processors, the set of active threads at each instant is determined by a sort of global scheduling policy, defined by the rules given in Figure 8.6, where e^{θ} denotes an expression that is immediately blocked on θ, i.e. one of the following: $\theta \mid \theta x$.

Before deactivating those threads which have reached a whnf, we have to release all the threads which were blocked on them. The (unblocking) rule is applied only to threads blocked on ordinary variables, i.e. not on communication channels, being its effect the transformation of blocked threads into runnable; the value will be bound later by the application of the local (value) rule given in the previous section; by contrast, a communication value will be (instantaneously) copied at the heap of the receiving process, as has been explained in the previous subsection.

Thereafter, runnable threads are activated under some restrictions. For instance, a process may include in its heap many runnable threads but, at each moment, at most one thread can be active, because all the threads belonging to a same process share heap and processor, and they do not migrate to other processors. Formally, let $H^A = \{\theta \overset{A}{\mapsto} e \in H\}$ be the set of active threads in a heap H, then for each process $\langle p, H\rangle \in S$ we have always $|H^A| \in \{0, 1\}$.

(unblocking)

$$(S, \langle p, H + \{x \overset{AR}{\mapsto} v, \theta \overset{B}{\mapsto} e^x\}\rangle) \overset{unbl}{\longrightarrow} (S, \langle p, H + \{x \overset{AR}{\mapsto} v, \theta \overset{R}{\mapsto} e^x\}\rangle)$$

(deactivation)

$$(S, \langle p, H + \{x \overset{AR}{\mapsto} v\}\rangle) \overset{deact}{\longrightarrow} (S, \langle p, H + \{x \overset{I}{\mapsto} v\}\rangle)$$

(preemption)

$$\text{if } \langle \theta, p\rangle \in \mathtt{pm}(S) \wedge \langle \rho, p\rangle \notin \mathtt{pm}(S)$$

$$(S, \langle p, H + \{\theta \overset{R}{\mapsto} e, \rho \overset{A}{\mapsto} e'\}\rangle) \overset{pree}{\longrightarrow} (S, \langle p, H + \{\theta \overset{A}{\mapsto} e, \rho \overset{R}{\mapsto} e'\}\rangle)$$

(activation)

$$\text{if } |H^A| = 0 \wedge \sum_{\langle q,K\rangle \in S} |K^A| < \text{number of processors}$$

$$(S, \langle p, H + \{\theta \overset{R}{\mapsto} e\}\rangle) \overset{act}{\longrightarrow} (S, \langle p, H + \{\theta \overset{A}{\mapsto} e\}\rangle)$$

FIGURE 8.6. Single steps for global transitions: Scheduling

Moreover, because we do not consider a fair share of the processor between runnable threads, and 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`. In general, for $\langle p, H\rangle \in S$ and $\theta \in \mathit{dom}(H)$ we define

$$\mathtt{pre}(\theta, \langle p, H\rangle, S) = \begin{cases} \{\langle \theta, \langle p, H\rangle\rangle\} & \text{if } \theta \overset{AR}{\mapsto} e \in H \\ \mathtt{pre}(\rho, \langle p, H\rangle, S) & \text{if } \theta \overset{B}{\mapsto} e^{\rho} \in H \\ \bigcup_{ch} \mathtt{pre}(ch, \langle q, K\rangle, S) & \text{if } \theta \overset{B}{\mapsto} e_{ch} \in H \ \wedge ch \in \mathit{dom}(K) \end{cases}$$

where e_{ch} denotes an expression depending on channel $ch \in \mathcal{C}$, i.e. one of the following: $ch \mid x\#e$ with $ch \in \mathtt{nc}(x, H)$.

Let $\langle p_0, H_0\rangle$ be the initial process —H_0 is the unique heap in a system S which contains the variable *main*— then $\mathtt{pm}(S) = \mathtt{pre}(\mathit{main}, \langle p_0, H_0\rangle, S)$ contains all the non-blocked threads "demanded" by *main*. Notice that $\mathtt{pm}(S)$ may contain more than one pair and then it may be the case that there are not enough free processors to activate all of them. Moreover, it may happen that some process containing a "preference" thread has already active some "speculative" thread. In this situation, we deactivate the latter in order to activate the former, as it is expressed by the (preemption) rule in figure 8.6. However, for the moment we do not consider a fully preemptive scheduling policy, where a processor is taken from one process to be given to another process.

Finally, we try to activate as many threads as possible, but giving priority to those which were demanded by the main process, and maintaining only one active thread per process. This is expressed by the following definition:

$$S \overset{act}{\Longrightarrow} S'$$

satisfying:

1. $S = S_0 \xrightarrow{act} S_1 \xrightarrow{act} \ldots \xrightarrow{act} S_k = S'$;
2. $\forall i \in \{0..k-1\}$
$(\exists \langle \theta, \langle p, H \rangle \rangle \in \mathtt{pm}(S_i) : |H^A| = 0) \Rightarrow (\mathtt{pm}(S_i) \cap S_i^A \subset \mathtt{pm}(S_{i+1}) \cap S_{i+1}^A)$, and
3. there is no S'' such that $S' \xrightarrow{act} S''$.

Where $S^A = \{\langle \theta, \langle p, H \rangle \rangle \mid \langle p, H \rangle \in S \wedge \theta \overset{A}{\mapsto} e \in H\}$ represents the set of active threads in a process system S. The second restriction in the definition above states that no speculative thread will ever be activated as long as there is some preference thread candidate for activation, that is runnable and with no other active thread belonging to the same process.

It is also guaranteed that, at each moment, at least one of the threads demanded by the initial process will be active, i.e. $\mathtt{pm}(S) \cap S^A \neq \emptyset$. In this way, in systems where only one processor is available, processes could still be speculatively instantiated, but they would be activated only when their output would be really demanded.

Unblocking, deactivation, preemption and activation, are combined in the following scheduling rule:

$$\xRightarrow{sched} = \xRightarrow{act} \circ \xRightarrow{pree} \circ \xRightarrow{deact} \circ \xRightarrow{unbl}$$

8.4.3 System evolution

The execution of a program is done by "big" computation steps consisting of the local evolution of each process in the system in parallel with the others, followed by a global housekeeping of the system. This is expressed by the transition

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

Parallel computation. The (parallel) rule, given below, combines the local transitions of each active thread at each process/heap —at most one per process— while processes without active thread are kept at the same state.

$$\textbf{(parallel)} \quad \text{if } S^A = \emptyset$$
$$\frac{\{\langle p, H_p + \{\theta_p \overset{A}{\mapsto} e_p\}\rangle \longrightarrow \langle p, H'_p \rangle\}}{S \cup \{\langle p, H_p + \{\theta_p \overset{A}{\mapsto} e_p\}\rangle\} \xRightarrow{par} S \cup \{\langle p, H'_p \rangle\}}$$

System management. The tasks relative to the management of the system at the global level, which have been explained in the last subsection, are organized in three consecutive phases:

$$\xRightarrow{sys} = \xRightarrow{sched} \circ \xRightarrow{pc} \circ \xRightarrow{com}$$

8.5 SPECULATIVE PARALLELISM

As it is pointed out in the introduction, and it has been shown in the semantics (see Subsection 8.4.2), Eden overrides laziness by instantiating processes in a speculative way. We will measure the amount of speculative work done during a program execution, in terms of the number of instantiated processes whose output value has never been used for the evaluation of the main process. For this purpose, we associate to each process a speculation accumulator to take account of the speculative parallelism generated by the process through its children. Therefore, a process will be denoted by $\langle p,s,H\rangle$, where s indicates the number of descendants which have not already contributed to the output towards the parent of p. We also add the possibility of decorating the binding labels in the form $\theta \overset{\alpha/\sigma}{\mapsto} v$, to indicate that the evaluation of v is, for the moment, speculative —the value v has been communicated to the process where this binding belongs, but the latter has still not consumed it—. In this notation σ indicates the number of descendants created for the evaluation of v.

The local rules given in Section 8.4 are still valid, if we just consider that whenever a binding carries some speculation information this is conserved by the binding, even if the label is changed. The only exception is the (value) rule, because speculation information disappears when a process effectively "consumes" a value. This is expressed by the following rule:

(speculative value)

$$\langle p,s,H+\{x \overset{I/\sigma}{\mapsto} v, \theta \overset{A}{\mapsto} x\}\rangle \longrightarrow \langle p,s,H+\{x \overset{I}{\mapsto} v, \theta \overset{A}{\mapsto} \hat{v}\}\rangle$$

Most global rules are still sound too. Only those concerning process instantiation and communication need to be modified. To begin with, we have to initialize the speculation accumulator of each process. Except for the main one, we consider every process to be speculative. Therefore, the initial configuration is $\langle p_0, 0, \{main \overset{A}{\mapsto} e\}\rangle$, but when instantiating a child process we have $\langle c, 1, \eta(\mathtt{nh}(x,H)) + \{o \overset{R}{\mapsto} x\,y, y \overset{B}{\mapsto} i\}\rangle$. Remember that, by the definition of the function nh, bindings are copied as inactive and they do not carry any speculation information.

Now we have to distinguish between communication from parent to child and communication from child to parent, because the speculation information goes from child to parent, but not the other way around. In the parent, the thread blocked at the communication channel o will become runnable and will have associated the speculation accumulated by the child except for the speculation associated to those bindings in the child that have never been used and, therefore, have not contributed to produce the communicated value. Moreover, the parent increments its global speculation account by the same amount of speculation which is passed to the binding, that is, the communication speculation (*sc*), whereas in the child the speculation is decreased to that one which is registered in the child bindings not used so far.

s_p	p (main)	s_{c_1}	c_1	s_{c_2}	c_2
0	$main \overset{A}{\mapsto} \texttt{let}\ x = \lambda z.4,\ w = x\#y,$ $y = (\lambda t.t)3\ \texttt{in}\ x\#y$				
0	$main \overset{A}{\mapsto} x\#y,\ x \overset{I}{\mapsto} \lambda z.4,$ $w \overset{I}{\mapsto} x\#y,\ y \overset{I}{\mapsto} (\lambda t.t)3$				
0	$main \overset{B}{\mapsto} o_m,\ i_m \overset{A}{\mapsto} y,\ y \overset{I}{\mapsto} (\lambda t.t)3,$ $x \overset{I}{\mapsto} \lambda z.4,\ w \overset{B}{\mapsto} o_s,\ i_s \overset{R}{\mapsto} y$	1	$x \overset{I}{\mapsto} \lambda z.4,\ u \overset{B}{\mapsto} i_m,$ $o_m \overset{A}{\mapsto} xu$	1	$x \overset{I}{\mapsto} \lambda z.4,\ u' \overset{B}{\mapsto} i_s,$ $o_s \overset{A}{\mapsto} xu'$
0	$main \overset{B}{\mapsto} o_m,\ i_m \overset{B}{\mapsto} y,\ y \overset{A}{\mapsto} (\lambda t.t)3,$ $x \overset{I}{\mapsto} \lambda z.4,\ w \overset{B}{\mapsto} o_s,\ i_s \overset{R}{\mapsto} y$	1	$x \overset{I}{\mapsto} \lambda z.4, u \overset{B}{\mapsto} i_m$ $o_m \overset{A}{\mapsto} (\lambda z.4)u$	1	$x \overset{I}{\mapsto} \lambda z.4,\ u' \overset{B}{\mapsto} i_s,$ $o_s \overset{A}{\mapsto} (\lambda z.4)u'$
0	$main \overset{B}{\mapsto} o_m,\ i_m \overset{R}{\mapsto} y,\ y \overset{I}{\mapsto} 3,$ $x \overset{I}{\mapsto} \lambda z.4,\ w \overset{B}{\mapsto} o_s,\ i_s \overset{R}{\mapsto} y$	1	$x \overset{I}{\mapsto} \lambda z.4, u \overset{B}{\mapsto} i_m$ $o_m \overset{A}{\mapsto} 4$	1	$x \overset{I}{\mapsto} \lambda z.4,\ u' \overset{B}{\mapsto} i_s,$ $o_s \overset{A}{\mapsto} 4$
$1+1$	$main \overset{A/1}{\mapsto} 4,\ i_m \overset{R}{\mapsto} y,\ y \overset{I}{\mapsto} 3,$ $x \overset{I}{\mapsto} \lambda z.4,\ w \overset{R/1}{\mapsto} 4,\ i_s \overset{R}{\mapsto} y$	0	$x \overset{I}{\mapsto} \lambda z.4, u \overset{B}{\mapsto} i_m$	0	$x \overset{I}{\mapsto} \lambda z.4,\ u' \overset{B}{\mapsto} i_s,$
$1+1$	$main \overset{I/1}{\mapsto} 4,\ i_m \overset{R}{\mapsto} y,\ y \overset{I}{\mapsto} 3,$ $x \overset{I}{\mapsto} \lambda z.4,\ w \overset{R/1}{\mapsto} 4,\ i_s \overset{R}{\mapsto} y$	0	$x \overset{I}{\mapsto} \lambda z.4, u \overset{B}{\mapsto} i_m$	0	$x \overset{I}{\mapsto} \lambda z.4,\ u' \overset{B}{\mapsto} i_s,$

TABLE 8.1. Speculative example

Let $\texttt{SH}(H)$ be the sum of the speculation associated to the bindings in a heap H, that is, $\texttt{SH}(H) = \sum\{\sigma \mid \theta \overset{\alpha/\sigma}{\mapsto} e \in H\}$. We replace the (value communication) rule by the following two rules:

(parent $\rightsquigarrow$ child)

$$(S, \langle p, s_p, H_p + \{i \overset{\alpha}{\mapsto} v\}\rangle, \langle c, s_c, H_c + \{\theta \overset{B}{\mapsto} i\}\rangle)$$
$$\overset{p\rightsquigarrow ch}{\longrightarrow} (S, \langle p, s_p, H_p\rangle, \langle c, s_c, H_c + \eta(\mathtt{nh}(v, H_p)) + \{\theta \overset{R}{\mapsto} \eta(\hat{v})\}\rangle)$$

(child $\rightsquigarrow$ parent)

$$(S, \langle p, s_p, H_p + \{\theta \overset{B}{\mapsto} o\}\rangle, \langle c, s_c, H_c + \{o \overset{\alpha}{\mapsto} v\}\rangle)$$
$$\overset{ch\rightsquigarrow p}{\longrightarrow} (S, \langle p, s_p + sc, H_p + \eta(\mathtt{nh}(v, H_c)) + \{\theta \overset{R/sc}{\mapsto} \eta(\hat{v})\}\rangle, \langle c, \texttt{SH}(H_c), H_c\rangle)$$

where $sc = s_c - \texttt{SH}(H_c)$.

For instance, being the main program $\texttt{let}\ x = \lambda z.4,\ w = x\#y,\ y = (\lambda t.t)3\ \texttt{in}\ x\#y$, and considering an unbounded number of processors, we obtain the reduction sequence shown in Table 8.1.

To calculate the number of created processes whose produced value has never been used in the reduction of the *main* thread, we have to consider the speculation associated to those bindings belonging to the initial process which have not been used, together with the speculation accumulated by the rest of processes. For n processors, let $\langle p_0, s, H_0 + \{main \overset{I/\sigma}{\mapsto} v\}\rangle, \langle p_1, s_1, H_1\rangle, \ldots, \langle p_m, s_m, H_m\rangle$ be the final configuration of the reduction sequence. The total speculation is obtained using the following formula:

$$SP(n) = \mathtt{SH}(H_0) + \sum_{k=1}^{m} s_k$$

Then, in the previous example, according to this formula, the speculation is $SP(\infty) = 1 + (0+0)$. But for the 1-processor case the process ch_2 will never be evaluated and, as a consequence, it will never pass the speculation to the *main* process, resulting $SP(1) = 0 + (0+1)$.

8.6 CONCLUSIONS AND FUTURE WORK

The present distributed operational semantics is parameterised over the number of processors (see rule (activation) in figure 8.6). As in [BFKT00], we have employed labelled name/expression bindings to simulate closures. The computation state is given by a set of binding heaps, where each heap represents a process. We have modelled lazy evaluation except for top level instantiations and value production for communication, and we have achieved the measurement of the speculative parallelism by using accumulators in each process and labels in the bindings.

The main goal for developing the operational semantics presented here was to offer a way to observe —from a theoretical point of view— cost properties of programs such as the amount of speculative parallelism, the grade of parallelism, and so on, and to allow the programmer to analyse formally the consequences of some program decisions. This is particularly interesting for languages like Eden, where speculative parallelism is controled by the programmer, in contrast with other approaches like [Mat93] where it is done automatically by the compiler, as an optimization for the program execution. Besides, we intend to use the semantics to infer other properties inherent to parallel and concurrent programs. For instance, we can detect deadlock situations involving communications. For example, the definition $z = x\#z$ in a heap $H = \{z \stackrel{R}{\mapsto} x\#z, x \stackrel{\alpha}{\mapsto} \lambda w.w, \dots\}$, would be evaluated generating the sequence:

$$(\langle p, \{z \stackrel{R}{\mapsto} x\#z, x \stackrel{\alpha}{\mapsto} e\}\rangle) \longrightarrow (\langle p, \{z \stackrel{B}{\mapsto} i, o \stackrel{R}{\mapsto} z\}\rangle, \langle ch, \{i \stackrel{R}{\mapsto} x\,y, y \stackrel{B}{\mapsto} o, x \stackrel{\alpha}{\mapsto} \lambda w.w\}\rangle)$$
$$\longrightarrow^* (\langle p, \{z \stackrel{B}{\mapsto} i, o \stackrel{B}{\mapsto} z\}\rangle, \langle ch, \{i \stackrel{B}{\mapsto} y, y \stackrel{B}{\mapsto} o, x \stackrel{\alpha}{\mapsto} \lambda w.w\}\rangle)$$

from which the loop of references $y \stackrel{B(ch)}{\mapsto} o \stackrel{B(p)}{\mapsto} z \stackrel{B(p)}{\mapsto} i \stackrel{B(ch)}{\mapsto} y$ can be constructed.

We also expect to prove some theoretical properties of our semantics. For instance, by unifying a set of heaps to a singleton, and by reducing process instantiation to the GPH construction `par`, we can relate our semantics to the one presented in [BFKT00], which — for the 1-processor case — has already been proved to be equivalent to the natural semantics of Launchbury [Lau93].

One value channels is a very restrictive form of communication. Our next step is to extend the present semantics to handle communication streams, which will be represented as lists. A further extension is allowing a process to be instantiated with a tuple of input and a tuple of output channels. This implies initiating several parallel threads in the child process (as well as in the parent) to compute each output independently. We will also consider the introduction of other Eden features like dynamic channels, as well as some form of non-determinism.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish project CICYT-TIC97-0672 and the Spanish-British Acción Integrada HB1999-0102. We wish to thank Phil Trinder for his helpful suggestions which led to improvements in this paper.

REFERENCES

[BFKT00] Clem Baker-Finch, David King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *ICFP'00*, Montreal, Canada, September 2000.

[BLOMP97] 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.

[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. Flemming Nielson. Springer-Verlag, 1997.

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

[HM99] K. Hammond and H. A. Michaelson (editors). *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of ACM Principles of Programming Languages*, Charleston, 1993.

[Mat93] James S. Mattson Jr. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. PhD thesis, University of California, San Diego, 1993.

[PH99] S. Peyton Jones and J. Hughes (editors). Report on the Programming Language Haskell 98. URL http://www.haskell.org, February 1999.

[PR97] P. Panangaden and J. Reppy. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 2: The Essence of Concurrent ML, pages 5–30. Monographs in Computer Science. Ed. Flemming Nielson. Springer-Verlag, 1997.

Contents