Using Maude and its strategies for defining a framework for analyzing Eden semantics

Mercedes Hidalgo-Herrero[a] Alberto Verdejo[b]
Yolanda Ortega-Mallén[b]

[a] *Departamento de Didáctica de las Matemáticas, Universidad Complutense de Madrid*

[b] *Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid*

Abstract

Eden is a parallel extension of the functional language Haskell. On behalf of parallelism Eden overrides Haskell's pure lazy approach, combining a non-strict functional application with eager process creation and eager communication. We desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design. In this paper we show how to implement in Maude the operational semantics of Eden in such a way that semantic rules can be modified easily. Moreover, other semantic features can be implemented by means of parameterized modules that allow to instantiate in different ways several parameters of the semantics but without modifying the semantic rules.

Keywords: Operational semantics, parallel functional languages, Eden, rewriting logic, Maude, rewrite strategies.

1 Introduction

It is well-known that functional languages offer great possibilities for parallel programming [10], ranging from a completely implicit parallelism —for instance an automatic parallelization— to an explicit parallelism where the programmer distributes the computation among a set of communicating processes that even may be located by the programmer himself at designated processors. The parallel language Eden lies more closely to this latter approach, extending Haskell [15] with coordination features for creating processes with stream-based communication.

Haskell is a lazy language, i.e. it adopts normal order evaluation, avoiding repeated computations by sharing reductions. The lazy approach restricts the exploitation of parallelism because expressions are evaluated only under demand. Therefore, Eden overrides the pure lazy approach, combining a non-strict functional application with eager process creation and eager evaluation of communication values. This may produce *speculative* computation, i.e. the calculation of results that

[1] Research supported by MCyT Spanish project *MIDAS* (TIC200301000).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

may never be used. The amount of speculative computation produced during the evaluation of an Eden program is variable, depending on the number of processors, the speed of basic operations, etc. This interplay between laziness and eagerness is precisely established by Eden's operational semantics [6,11]. Moreover, this semantics defines two extreme degrees of speculative computation: minimal and maximal.

Although there exists a stable implementation of Eden [8,17] on top of the Glasgow Haskell Compiler (GHC) [4], we desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design. For this purpose, it is extremely useful to have a framework where Eden's operational semantics can be easily programmed and that provides mechanisms to reflect with small effort changes in the semantics. Rewriting logic [14] and Maude [3] are excellent candidates for this aim. First, Eden's syntax can be represented literally. Second, Eden's operational semantics rules can be represented in Maude quite literally in most cases, so keeping the representation distance as short as possible. Third, since Maude specifications are executable, we directly get an implementation of Eden where program examples can be executed and analyzed. Finally, a recently proposed *strategy* language [13] for Maude can be used to control in every desired way the application of semantic rules.

In this paper we show how to implement in Maude the operational semantics of Eden in such a way that two main objectives are possible: (1) the semantic rules can be modified in an easy manner so that in a near future we can investigate with different possibilities, and (2) several measures —parallelism, speculative computation, communications, etc.— can be taken by changing some parameters of the semantics (which is defined in a parameterized module) without modifying the semantic rules.

From the point of view of Eden, this is the first step towards a framework where Eden expressions can be evaluated according to different semantics in order to be compared and analyzed. From the point of view of the implementation of operational semantics in Maude, this work constitutes another step in a continued effort to represent semantics for more complex languages. The simplest concurrent language we have considered is Milner's CCS in [18], that does not require any strategies. This is not the case with Cardelli and Gordon's Ambient Calculus that we have tackled in [16]. However, the use of strategies in the latter solves problems different from the ones we consider in this paper, where we take into account that Eden's semantics inherently depends upon an order of application of the rules, thus exploiting the strategy language expressiveness.

The rest of the paper is organized as follows. First we present a brief introduction to Maude; for a complete treatment we refer the reader to the Maude manual [3]. Section 3 gives an overview of Eden and implements its kernel syntax, while Section 4 is devoted to the operational semantics and its implementation in Maude. In Section 5 we extend our framework in order to be able to obtain measures from the computations. The last section presents our conclusions and outlines future work.

2 Visiting Maude

In Maude the state of a system is formally specified as an algebraic data type by means of an equational specification. Maude uses a very expressive version of equational logic, namely *membership equational logic* [2]. In this kind of specifications we can define new types (by means of the keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; equations (`eq`) that identify terms built with these operators; and memberships (`mb`) $t : s$ stating that the term t has sort s. Both equations and memberships can be conditional. Conditions are formed by a conjunction (written `/\`) of equations and memberships. Equations are assumed to be confluent and terminating, that is, we can use the equations from left to right to reduce a term t to a unique (modulo the operator attributes as associativity, commutativity, and identity) canonical form t' that is equivalent to t, i.e. they represent the same value.

The *dynamic* behavior of a system is specified by rewrite rules of the form

$$t \longrightarrow t' \ \textit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

that describe the local, concurrent transitions of the system. That is, when part of a system matches the pattern t and the conditions are fulfilled, it can be transformed into the corresponding instance of the pattern t'.

Maude modules can be *parameterized* with one or more parameters, each of which is expressed by means of one theory that defines the interface of the module, that is, the structure and properties required of an actual parameter.

Rewrite rules need be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions. We have defined a strategy language for Maude that can be used to control how rules are applied to rewrite a term [13]. The simplest strategies are the constants `idle`, which always succeeds by doing nothing, and `fail`, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control by means of search expressions how the rewrite conditions are solved. An operation `top` to restrict the application of a rule just to the *top* of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (`;`), union (`|`), and iteration (`*` for 0 or more iterations, `+` for 1 or more, and `!` for a "repeat until the end" iteration). Another strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy. The language provides a (`x`)`matchrew` combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten.

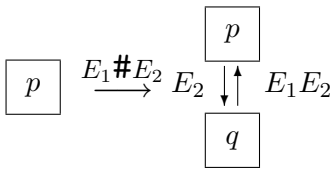
3 A quick excursion to Eden

Eden [11] extends the non-strict functional language Haskell with a set of *coordination* features to control parallel evaluation of processes. Coordination in Eden is based on two principal concepts: *explicit definition of processes* and *implicit stream-based communication*, i.e. there are not communication primitives such as *send* and *receive*. As well as there is a distinction between function definition and function application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes. Moreover, *nondeterminism* is introduced in Eden by means of a predefined process abstraction which is used to instantiate nondeterministic processes that fairly merge several input streams into a single output stream.

For the purpose of this paper we just concentrate on Eden's essentials, which are captured by the untyped λ-calculus whose abstract syntax is given next, where $x \in \mathit{Var}$ represents identifiers and $E \in \mathit{Exp}$ represents expressions:

$E ::= x$		identifier
	$\lambda x.E$	λ-abstraction
	$E_1\, E_2$	application
	$E_1 \texttt{\#} E_2$	process creation
	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	local declaration

When evaluating the expression $E_1\texttt{\#}E_2$ inside a process p, a new child process q is created together with two communication channels. The child is fed with the value of E_2 via the input channel by its parent process p. Process q evaluates $E_1\ E_2$ and returns the result (to its parent) via the output channel. The following diagram illustrates this:



The language is normalized to a restricted syntax where all subexpressions, except for the body of λ-abstractions, are replaced by variables defined in `let`-expressions. This guarantees that subexpressions are shared, and are evaluated at most once. We also assume a general renaming of variables for avoiding name clashes during expression evaluation.

For instance, the evaluation of the following expression

$$\begin{array}{ll} \texttt{let} & x_0 = x_1\texttt{\#}x_1, \\ & x_1 = \lambda x.x, \\ & x_2 = 1, \\ & x_3 = x_4\, x_0, \\ & x_4 = x_5\, x_2, \\ & x_5 = \lambda y.(\lambda z.z) \\ \texttt{in} & x_3 \end{array}$$

gives place to a process creation: the main process evaluates x_3 while the child computes the application $x_1\, x_1$. In order to do that, the child process needs the

value of x_1 twice:

(i) for obtaining the λ-abstraction: the definition is copied to the child's heap, and

(ii) for getting the argument: the parent communicates the value to the child.

By the end of the evaluation of the application, the resulting value is sent back to the parent process.

3.1 Representation in Maude

We define in Maude the syntax of the kernel of Eden given above. We use sorts and subsorts to represent the different syntactic categories and their relations. Having different sorts allows us to concrete the patterns used in rewrite rules by using (Maude) variables of the most appropriate sort. We have sorts for ordinary variables (`Std`), for channels (`Cha`) and for the union of both sets (`Var`). We also use two sorts for distinguishing between expressions that are in weak head normal form (`Whnf`) and those that are not (`NonWhnf`). Both are Eden expressions (`Exp`).

```
sorts Std Cha Var Whnf NonWhnf Exp LetBind LetBinds .
subsorts Std Cha < Var < NonWhnf .
subsorts Whnf NonWhnf < Exp .
subsort LetBind < LetBinds .
```

We define constructors for building expressions. For each constructor, the most concrete sort is used as the result sort; for example, a λ-expression `\_._` is a weak head normal form, whereas an application `__` (empty syntax) is not. Strings are used as variable identifiers.

```
op s : String -> Std .
op c : String -> Cha .
op \_._ : Std Exp -> Whnf .
op __ : Exp Exp -> NonWhnf .
op let_in_ : LetBinds Exp -> NonWhnf .
op _#_ : Exp Exp -> NonWhnf .
op _=_ : Std Exp -> LetBind .
op nil : -> LetBinds .
op __ : LetBinds LetBinds -> LetBinds .
```

4 Operational semantics

In this section we describe an operational semantics in the style of [1], which in turn is based on Launchbury's *natural* semantics for lazy evaluation [9].

It is our purpose just to describe the structure of the semantics and to present some of the transition rules focusing on how they have been implemented in Maude, but neither to explain nor to justify their definition. For a more detailed overview of Eden's semantics, the reader is referred to [11]; a version extended with streams for communication, dynamic channels and nondeterminism can be found in [6]. Correctness proofs, examples and applications are gathered in [5].

4.1 A two-level transition system

A process is represented by a pair $\langle p, H\rangle$, where p is a process identifier and H is the heap collecting the variable-to-expression bindings that model the closures corresponding to the process evaluation state. Each binding is considered a potential thread to be executed by the available processors, so that a label indicates the thread state: $x \overset{\alpha}{\mapsto} E$, where $\alpha ::= I|A|B$ corresponds to *Inactive* (either not yet demanded or already completely evaluated), *Active* (or demanded), and *Blocked* (demanded but waiting for the value of another binding), respectively. Channel identifiers can appear on either side of a binding: on the left-hand side they represent outports; while on the right-hand side they denote inports.

In the following, we will use $x, y, z \in \mathit{Std}$ for ordinary variables, $c \in \mathit{Chan}$ for channels, $\theta \in \mathit{Var} = \mathit{Std} \cup \mathit{Chan}$, $W \in \mathit{Whnf}$ for weak head normal forms, and p and q for process identifiers.

The model of evaluation is represented by a sequence of systems —a system is a set of parallel processes— regulated by the transition rules. Some of the bindings in a heap are executed in parallel, sharing the data of the corresponding process; but bindings in different processes can only share information through process communication. The semantics needs small-step transitions to model parallelism in a synchronous way, in the sense that single reductions are local and independently carried out at each process and then combined before proceeding to the next step. The semantics reflects the distinction between the two sub-languages (computation and coordination) that configure Eden, so that it consists of a two-level transition system: the lower level handles local effects within processes, while the upper level describes the effects global to the whole system, like process creation and data communication.

4.2 Representing the transition system in Maude

A thread is built with a variable, a thread state (of sort `TState`), and an expression. A heap is a set of threads: `none` represents the empty heap, a thread represents a singleton heap, `subsort Thread < Heap`, and the union `_+_` of heaps builds them. The union constructor is declared to be associative, commutative, and with the empty heap as the identity element; pattern matching will take place modulo these properties. Finally, the process constructor has four arguments: a string corresponding to the process identifier; a heap; and two counters: one represents the number of children of this process and the other indicates the maximum number used to build new variables (incremented when renamings are needed because of the generation of new variables). There is also a union operator `__` (with empty syntax) for building systems.

```
sorts TState Thread Heap Process System .
subsort Thread < Heap .
subsort Process < System .
ops A I B : -> TState .
op _|-_->_ : Var TState Exp -> Thread .
op none : -> Heap .
op _+_ : Heap Heap -> Heap [assoc comm id: none] .
op <_,_,_,_> : String Heap Nat Nat -> Process .
```

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

$$\text{if } E \notin \mathit{Whnf},\ H + \{x \overset{IAB}{\longmapsto} E\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{AAB}{\longmapsto} E, \theta \overset{B}{\mapsto} x\} \qquad \textbf{(demand)}$$

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

$$\text{if } E \notin \mathit{Whnf},\ H + \{x \overset{IAB}{\longmapsto} E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{AAB}{\longmapsto} E, \theta \overset{B}{\mapsto} x\, y\} \qquad \textbf{(app-demand)}$$

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

$$H : \theta \overset{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}\ \texttt{in}\ x \longrightarrow H + \{y_i \overset{I}{\mapsto} E_i\sigma\}_{i=1}^{n} + \{\theta \overset{A}{\mapsto} \sigma(x)\} \qquad \textbf{(let)}$$

$$\text{where } \mathit{fresh}(y_i)\ (1 \le i \le n) \text{ and } \sigma := [y_1/x_1, \ldots, y_n/x_n]$$

Fig. 1. Local transition rules

```
op empty : -> System .
op __ : System System -> System [assoc comm id: empty] .
```

We have defined several auxiliary operations needed by the semantics: substitution, renaming of variables in a heap, normalization, etc. They are defined structurally by means of equations and using the `owise` (otherwise) Maude attribute. For the complete Maude code we refer the reader to [7].

4.3 Local process evolution

Local transitions express the reduction of an active thread in the context of a single process. This internal activity affects only the corresponding heap. The evaluation of an expression terminates when it reaches a whnf value ($W \in \mathit{Whnf}$). Local transitions take the form $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$, which is read as "the evaluation of the *active thread* $\theta \overset{A}{\mapsto} E$ transforms the heap $H + \{\theta \overset{A}{\mapsto} E\}$ into H'". In Figure 1 we show the local rules expressing how lazy evaluation progresses under demand. We avoid writing multiple similar transition rules by allowing a binding to appear with several labels, corresponding to the different possibilities admitted by the rule. Thus, $x \overset{IAB}{\longmapsto} E$ on the left-hand side of rule **(demand)**, and $x \overset{AAB}{\longmapsto} E$ on the right-hand side means that the thread corresponding to the closure $x \mapsto E$ becomes active in the case it was inactive, and remains active or blocked otherwise.

In Maude we represent the semantic rules as rewrite rules. There are several ways of mapping inference systems into rewriting logic [12]. In the structural operational semantics case, judgements typically have the form of some kind of transition $P \to Q$ between states, so that it makes sense to map directly this transition relation between states to a rewriting relation between terms representing the states. Thus, an inference rule of the form

$$\frac{P_1 \to Q_1 \quad \ldots \quad P_n \to Q_n}{P_0 \to Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \mathit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n.$$

In this way the semantic rules become (conditional) rewrite rules: the transition

in the conclusion becomes the main rewrite of the rule, and the transitions in the premises become rewrite conditions [18].

The local transition rules (as those given in Figure 1) are translated quite literally. We introduce two new constructors for representing heaps. The first one, `_:_`, is already used in the semantic rules in order to separate the leading thread —that which is going to evolve— from the rest of the heap. The second one, `_&_`, is used in the right-hand side of rewrite rules in order to separate the modified threads from the unmodified ones because this separation will be useful later. Actually, the rule **(demand)** puts together three transition rules, one for each possible state of the thread consulted in the heap. The rewrite rule `demand` given below represents the three semantic rules at the same time, by using a variable `T` of sort `TState` and auxiliary operations for detecting if the thread is modified or not. Notice how the variable `NW` of sort `NonWhnf` is used to ensure the condition $E \notin \mathit{Whnf}$ in the semantic rule. The rule `let` also uses auxiliary operations to build the new heap on the right. This rule rewrites a process instead of only a heap because, due to the renaming, the fourth argument has to be incremented.

```
rl [value] : H + X |- I -> W : Theta |- A -> X
          => H + X |- I -> W & Theta |- A -> W .

rl [demand] : H + X |- T -> NW : Theta |- A -> X
           => H + nmd(X |- T -> NW) & md(X |- T -> NW) + Theta |- B -> X .

rl [blackhole] : H : X |- A -> X
              => H & X |- B -> X .

rl [app-demand] : H + X |- T -> NW : Theta |- A -> X Y
         => H + nmd(X |- T -> NW) & md(X |- T -> NW) + Theta |- B -> X Y .

rl [beta-reduction] : H + X |- I -> \ Z . E : Theta |- A -> X Y
                   => H + X |- I -> \ Z . E & Theta |- A -> E [Y / Z] .

rl [let] :  < p, H : Theta |- A -> let LBS in X, N, M >
   => < p, H & letBindsToHeap(Theta, LBS, X, newvars(p, M, numvars(LBS))),
        N, M + numvars(LBS) > .
```

4.4 Local parallelism

Local evolutions —corresponding to the local transition rules— are considered to occur simultaneously, entwined in a parallel step. The rule given in Figure 2 expresses the evolution of parallel threads inside a process, where $\mathcal{ET}(S)$ is the set of active threads in the system S that are allowed to evolve. $H^{(i,1)}$ is the part of H that remains unchanged during the application of the corresponding local rule, while $K^{(i,2)}$ contains the bindings from $H^{(i,2)}$ that have been modified. It is guaranteed that there is no interference among local transitions. There are several possibilities for defining $\mathcal{ET}(S)$, depending on the number of available processors, the allowed degree of speculative computation, the priority given to some threads, etc. Maude modularity, by means of parameterized modules, is very useful to implement and then compare different scheduling strategies, as we will see in Section 4.6.

The rule **(local parallel)** is quite "abstract." First, it has a variable amount

$$\frac{\{H^{(i,1)} + H^{(i,2)} : \theta^i \overset{A}{\mapsto} E^i \longrightarrow H^{(i,1)} + K^{(i,2)} \quad \text{s.t. } H = H^{(i,1)} + H^{(i,2)} + \{\theta^i \overset{A}{\mapsto} E^i\} \text{ and } \theta^i \overset{A}{\mapsto} E^i \in \mathcal{ET}(S) \cap H\}_{i=1}^{n}}{H \xrightarrow{par}_S (\cap_{i=1}^{n} H^{(i,1)}) \cup (\cup_{i=1}^{n} K^{(i,2)})}$$

where $n = |\mathcal{ET}(S) \cap H|$

Fig. 2. **(local parallel)** rule

of premises, depending on the number of threads returned by $\mathcal{ET}(S)$; and second, it makes separations of the heaps distinguishing between modified and unmodified threads. For its implementation, we have solved the second problem by modifying the right-hand side of local rules with the `_&_` operator. To deal with the first problem we have devised several approaches; we show here the one which represents the resolution of premises, and the calculation of intersections and unions in the right-hand side of the conclusion *step by step*, by means of rewrite rules. We have chosen this form because it is similar to its mathematical presentation, it simplifies the strategies needed, and it is more efficient.

We consider the following three rewrite rules as the basic steps of an algorithm that implements the **(local parallel)** rule. The rule `extend` adds to the process three arguments: the first one is the set of variables associated with threads that have to evolve (new variable `VS`, explained below), the second represents the (partial) evaluation of the intersection of unmodified threads (initially the whole heap), and the third represents the (partial) evaluation of the union of modified threads (initially the empty heap). The rule `parallel-step` performs the main step of the algorithm, by solving one premise each time. It is a conditional rewrite rule: the first two conditions (which are matching equations) extract the thread corresponding to the variable `Theta` from the heap `H`, and the third (rewrite) condition represents the premise in **(local parallel)** corresponding to the variable `Theta`. This last condition has to be solved by using one of the local transition rules. Notice that the heap `H` is kept unmodified because it is used in the resolution of each premise, and the variable `Theta` is removed from the set `VS`. Finally, the rule `contract` removes the extra arguments from a process, and performs a final union of heaps.

```
rl [extend] :  < p, H, N, M > => < p, H, VS, H, none, N, M > .
crl [parallel-step] :  < p, H, Theta . VS, H', K, N, M > =>
                       < p, H, VS, int(H',H1), K + K1, N', M' >
   if Theta |- T -> E := lookUp(Theta, H) /\ H1-2 := filter(Theta, H) /\
      < p, H1-2 : Theta |- T -> E, N, M > => < p, H1 & K1, N', M' > .
rl [contract] :  < p, H, mt, H', K, N, M > => < p, H' + K, N, M > .
```

The application of these three rules has to be controlled. First of all, the rule `extend` is applied by providing the concrete value for variable `VS`, namely the variables in $\mathcal{ET}(S) \cap P$, where P is the process being rewritten and S is the whole system[2]; then, the rule `parallel-step` is applied as many times as possible, i.e. once for each thread in $\mathcal{ET}(S) \cap P$; and finally, the rule `contract` has to be applied. The following strategy `-par->`, that receives as argument a set of variables, corresponds to this concrete application of the rules. It represents the relation $\xrightarrow{par}_S$ in

[2] This intersection is computed when the strategy `-par->` is called from strategy `=par=>` bellow.

the semantics (defined in Figure 2).

```
sop -par-> : VarSet .
seq -par->(ActVS:VarSet) = extend[VS:VarSet <- ActVS:VarSet] ;
                           ( parallel-step ! ) ; contract .
```

An alternative way of implementing the relation $\xrightarrow{par}_S$ would put more control in the strategy, making it to traverse the set of evolvable variables and applying a local rule to each of these variables (by means of other strategies). Although in this case the rule `parallel-step` would be simplified, the approach presented before has proved to be more efficient by doing the rewrite rules more powerful, and by simplifying the strategies.

4.5 Global system evolution

At an upper level we define global transitions between process systems represented by sets of processes. A global transition takes the general form:

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

where each heap H_p (associated to a process p in the system S) is transformed to H'_p, while new processes (in S') may be created. The diamond $\diamond$ is a place-holder for the name of the rule.

4.5.1 Parallel

Now we consider the parallel evolution of processes within a system S:

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

This rule has a variable number of premises, one for each process in the system S. Each premise makes the corresponding process to evolve exactly once through the transition $\xrightarrow{par}_S$. We implement this rule by means of the strategy `=par=>` that applies the strategy `-par->` to each process in a system. This strategy is recursive and it terminates when the rest of the system (represented by the variable `S:System` below) is empty. The strategy `=par=>` receives as argument the variables corresponding to the threads returned by the function $\mathcal{ET}$ applied to the whole system. Strategy `-par->` is called with the set of evolvable variables of process `P`, calculated by function `inters`.

```
sop =par=> : VarSet .
seq =par=>(VS:VarSet) = if (match empty) then idle
        else (matchrew P:Process S:System by
                P:Process using -par->(inters(P:Process, VS:VarSet)),
                S:System using =par=>(VS:VarSet) ) fi .
```

4.5.2 Multi-step rules

After each process has internally evolved, the following tasks have to be done at the system level: process creation, 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 ($\diamond \neq par$), for each single-step rule $S \xrightarrow{\diamond} S'$ we define a multi-step rule $S \overset{\diamond}{\Longrightarrow} S'$ satisfying: $S \xrightarrow{\diamond}{}^{*} S'$ and, there is no S'' such that $S' \xrightarrow{\diamond} S''$. The

$$(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}\rangle) \xrightarrow{pc} (S, \langle p, H + \{\theta \overset{B}{\mapsto} c_1, c_2 \overset{A}{\mapsto} y\}\rangle, \langle q, \eta(\mathsf{nh}(x, H)) + \{c_1 \overset{A}{\mapsto} \eta(x)\, z, z \overset{B}{\mapsto} c_2\}\rangle)$$
$$\text{if } \mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset$$

q, z, c_1, c_2 are fresh identifiers and substitution η replaces all variables by fresh ones

Fig. 3. **(process creation)** rule

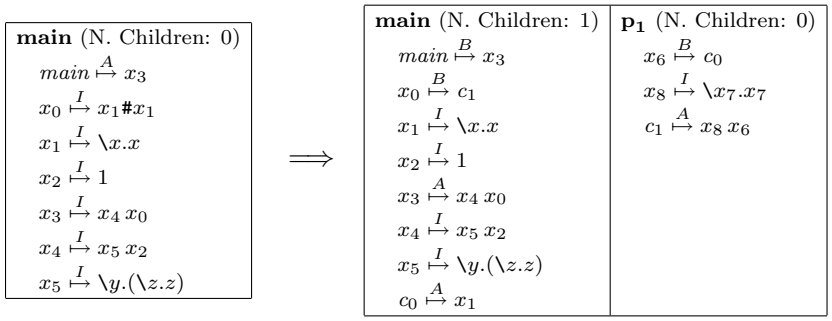
application of a single-step rule $\diamond$ to some binding in some process may enable the application of the same rule $\diamond$ to other bindings —in the same or in other processes— but it can never disable applications of rule $\diamond$ which were enabled before the former application.

Single-step rules are implemented in Maude as rewrite rules, while the relations $\overset{\diamond}{\Longrightarrow}$ are built by means of strategies. Afterwards, these relations are combined through more strategies.

4.5.3 Process creation

The initial heap of a child process contains all the bindings that are needed for the evaluation of the dependent variables in the process body; these are copied from the parent to the child heap by the function nh (*needed heap*): $\mathsf{nh}(x, H)$ collects all the bindings in H that are reachable from x. A renaming η with fresh variables is applied to avoid name clashes. A process creation (see $\xrightarrow{pc}$ rule in Figure 3) is blocked if there is some dependency on values that have to be communicated. The function nf (*needed free*) collects the dependencies derived from the free variables.

Let us consider again the expression given as example in Section 3. After the application of the **(let)** local rule, the resulting heap is the one shown in the left-hand side of the following picture; and the **(process creation)** rule generates the structure in the right-hand side:



where c_7 is the inport of the child whereas c_8 is the outport; both are internal variables not defined by the programmer since communications in Eden are implicit.

The following rule implements in Maude the $\xrightarrow{pc}$ rule. It uses auxiliary functions to rename the heap copied into the child, and to build new variables and channels.

```
crl [pc] : < p, H + Theta |- T -> X # Y, N, M >
        => < p, H + Theta |- B -> c1 + c2 |- A -> Y,N + 1, M + 1 >
           < q, H' + c1 |- A -> (searchVar(X,VVL) Z) + Z |- B -> c2, 0, M' >
if nf(X, H + Theta |- T -> X # Y) = none /\  q :=  childName(p, N) /\
   c2 := c(newvar(p, M)) /\ c1 := c(newvar(q, 0)) /\
   Z := s(newvar(q, 1))  /\ < H',VVL,M' > := renH(nh(X,H),q,2) .
```

	EC	IC	BIE
1	yes	yes	parent
2	yes	no	parent
3	yes	yes	child
4	yes	no	child
5	no	yes	child
6	no	no	child
7	no	yes	parent
8	no	no	parent

Fig. 4. Evaluation alternatives for defining nf

When designing Eden there was great discussion about how to distribute computation between a process and its children. In the one extreme the parent would advance as much work as possible, so that every dependent variable of the instantiation body should be bound to a whnf before creating the child process. But this may lead to a poor parallelization, where a process has to do too much computation before delegating work to a helping process. In the other —we could say the "laziest"— extreme the parent would pass on all the work to its offspring, so that for a normalized expression $x\#y$, the argument y would be evaluated by the parent, while the body x as well as the application, $x\,y$, would be evaluated by the newborn child. This may lead to repeated calculations, because certain subexpressions may get evaluated independently by several children of the same parent. But this can be easily avoided by the programmer, by forcing the evaluation in the parent of these common subexpressions. The latter option has been adopted for Eden and its actual implementation, and this has been reflected in the operational semantics presented in [11]. In this regard, feasible combinations are gathered in Figure 4, where EC (evaluation before copy) stands for the option of evaluating all the bindings before being copied to the initial heap of the newly created process (or the consumer in case of a communication); IC (instantiation copy) represents the copy of bindings from one process to another corresponding to process instantiations; and BIE (body instantiation evaluation) comprises the alternatives for the evaluation of an instantiation body: either by the parent process, or by the child.

We can represent the different approaches in our semantics just by modifying the equational definition of the function nf. Each definition is specified in a different module that then is used to instantiate the parameterized module defining the semantics, which has as a parameter a theory requiring a function nf.

The relation $\overset{pc}{\Longrightarrow}$ is implemented in Maude as the following strategy, that iterates the application of the rule pc as many times as possible:

```
sop =pc=> .
seq =pc=> = pc ! .
```

$$(S,\ \langle p, H_p + \{c \overset{\alpha}{\mapsto} W\}\rangle,\ \langle q, H_q + \{\theta \overset{B}{\mapsto} c\}\rangle) \quad \overset{com}{\longrightarrow}$$
$$(S,\ \langle p, H_p\rangle,\ \langle q, H_q + \{\theta \overset{A}{\mapsto} \eta(W)\eta(\mathtt{nh}(W, H_p))+\}\rangle)$$

if $\mathtt{nf}(W, H_p) = \emptyset$ and η introduces fresh names for all variables.

Fig. 5. **(process communication)** rule

4.5.4 Communication

The rule for value communication between processes is given in Figure 5. When the value to be communicated corresponds to an abstraction, it is mandatory to copy —from the producer's heap to the consumer's heap— all the bindings needed for the evaluation of the dependent variables in the abstraction. Again, this copy can only take place if the abstraction does not depend on pending communications (the discussion corresponding to the two first columns in Figure 4 can also be applied in this case), a renaming substitution (η) is applied to the transferred heap, and bound variables are replaced by fresh variables.

Although a communication may enable additional ones, this never leads to an infinite number of communications (in one system step). Besides, the order of communications is not relevant, because variables that are already bound to values are not affected by communications. Hence, the image of the corresponding multi-step rule $\overset{com}{\Longrightarrow}$, which carries out every possible communication, is well defined.

The rule for value communication is easily implemented in Maude:

```
crl [com] :
    < p, Hp + ch |- T -> W, N, M > < c, Hq + Theta |- B -> ch, N', M' >
 => < p, Hp, N, M >
    < c, Hq + H' + Theta |- A -> (msubs(W',VVL)),N',N2 >
if nf(W, Hp) = none /\
   < H',VVL,N1 > := renH(nh(W, Hp),c,M') /\
   < W',N2 > := renL(W,c,N') .
```

When communicating a value it is mandatory to copy —from the producer's heap to the consumer's heap— all the bindings needed for the evaluation of the dependent variables in the value. This copy can only take place if the value does not depend on pending communications.

4.5.5 Scheduling

Once all the enabled process creations and communications have been done, the following tasks have to be achieved:

- Unblocking bindings depending on a variable bound to a whnf value meanwhile (*wUnbl*).
- Deactivating bindings to values in whnf (*deact*).
- Blocking process creations that could not be executed (*bpc*).
- Demanding bindings needed for pending process creations and/or communications (*pcd* and *vComd*).

The corresponding rules are given in Figure 6 and they are easily readable in the Maude implementation:

WHNF unblocking:

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

WHNF deactivation:

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

blocking process creation:

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

demanding process creation:

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

demanding communication:

$$(S, \langle p, H + \{c \overset{I}{\mapsto} W\}\rangle) \overset{vComd}{\longrightarrow} (S, \langle p, H + \{c \overset{I}{\mapsto} W, x \overset{A}{\mapsto} E\}\rangle) \quad \text{if } x \overset{I}{\mapsto} E \in \mathtt{nf}(W, H)$$

Fig. 6. Rules for scheduling

```
crl [wUnbl] :  < p, H + X |- A -> W + Theta |- B -> E, N, M >
           =>  < p, H + X |- A -> W + Theta |- A -> E, N, M >
 if X = blockedOn(E) .

rl [deact] :  < p, H + Theta |- A -> W, N, M >
          =>  < p, H + Theta |- I -> W, N, M > .

crl [bpc] :  < p, H + Theta |- T -> X # Y, N, M >
         =>  < p, H + Theta |- B -> X # Y, N, M >
 if T =/= B .

crl [pcd] :  < p, H + Theta |- B -> X # Y, N, M >
         =>  < p, H + Theta |- B -> X # Y + Z |- A -> E, N, M >
 if  Z |- I -> E + H' := nf(X, H).

crl [vComd] :  < p, H + ch |- I -> W, N, M >
           =>  < p, H + ch |- I -> W + X |- A -> E, N, M >
 if  X |- I -> E + H' := nf(W, H) .
```

Their iteration and sequential composition produce a new global rule $\overset{unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow}; \overset{deact}{\Longrightarrow}; \overset{bpc}{\Longrightarrow}; \overset{pcd}{\Longrightarrow}; \overset{vComd}{\Longrightarrow}$ that is combined with the other two rules explained before to obtain the global transition $\overset{sys}{\Longrightarrow} = \overset{comm}{\Longrightarrow}; \overset{pc}{\Longrightarrow}; \overset{unbl}{\Longrightarrow}$. The following Maude strategies define both relations:

```
sop =unbl=> .
seq =unbl=> = =wUnbl=> ; =deact=> ; =bpc=> ; =pcd=> ; =vComd=> .
sop =sys=> .
seq =sys=> = =com=> ; =pc=> ; =unbl=> .
```

4.5.6 Transition system step

Finally, each transition step of the system is defined as $\Longrightarrow = \overset{par}{\Longrightarrow}; \overset{sys}{\Longrightarrow}$. In Maude, the following strategy allows to compute a transition step. It will be applied to the whole system S that is being evolved, and first it applies strategy `=par=>` by passing as argument the set of evolvable threads returned by $\mathcal{ET}(S)$.

```
sop ==> .
seq ==> = (matchrew S:System by S:System using =par=>(ET(S:System))
          ) ; =sys=> .
```

4.6 Speculative parallelism

In any concrete implementation the evaluation of an Eden 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 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 main thread, the latter would correspond to having an unlimited set of processors for evaluating the output of every generated process. It is also possible to reflect in the semantics the distribution of a *limited number of processors* among the active threads following different rules, for instance: randomly among the threads, or fairly distributing the processors among the threads, or even giving priority to the demands of the main thread and distributing the rest of the processors among the other threads.

Once again, the facilities and modularity of Maude allow us to produce an implementation where to experiment different alternatives by selecting the appropriate definition of the functions `nf` and `ET`. These functions can be defined in different ways, thus obtaining different semantics for Eden. In the present implementation we have put each definition in a different Maude module. By instantiating the module defining the semantics rules with a module with a concrete definition of `nf` and a module with a concrete definition of `ET`, we obtain a complete specification of Eden.

5 Computation measures

In this section we show how our framework can be extended in order to perform measurements over the computations, such as work done, degree of parallelism, amount of communications, and so on. Modularity, particularly the separation between rules and strategies, is again an useful instrument because the necessary changes do not imply to modify the already implemented semantic rules.

First of all, the term being rewritten is extended with the actual values of the measures. One possible way to do that is by means of a set of attributes together with their values. One of these attributes contains the Eden system (`Sys`), that will be rewritten by the semantics rules shown in the previous sections. Here we show some examples of attributes.

```
sorts Attr AttrSet .
subsorts Attr < AttrSet .
op nilAS : -> AttrSet .
op __ : AttrSet AttrSet -> AttrSet [assoc comm id: nilAS] .
op Sys : System -> Attr .
op Work : Nat -> Attr .       --- Number of evolved active threads
op NumProc : Nat -> Attr .    --- Number of processes
op MaxPar : Nat -> Attr .     --- Maximum thread parallelism
```

```
op AvPar : Nat -> Attr .      --- Average thread parallelism
op AvProcPar : Nat -> Attr .  --- Average process parallelism
```

Then, rewrite rules have to be defined to describe the modification of these measures. For example, the following rule `addPC` increments by one the number of processes, and the rule `addET` updates the total work that has been done as well as the maximal thread parallelism. These updates are determined by the variable `CardET` that will be instantiated by a strategy.

```
rl [addPC] : NumProc(N) => NumProc(N + 1) .
rl [addET] : MaxPar(Max) Work(W) =>
             MaxPar(max(Max, CardET)) Work(W + CardET) .
```

Finally, we need to modify the strategies in order to apply these rules together with the semantics rules. We show below two of these new modified strategies. Strategy `=pc=>` now applies rule `addPC` after applying rule `pc` (process creation). And strategy `==>` updates the values of measures `MaxPar` and `Work` using the number of evolvable threads computed by expression `size(ET(S:System))`.

```
seq =pc=> = (pc ; addPC) ! .
seq ==> = (xmatchrew Sys(S:System) MaxPar(Max:Nat) Work(W:Nat)
             by S:System using =par=>(ET(S:System)) ,
                MaxPar(Max:Nat) Work(W:Nat) using
                    addET[CardET <- size(ET(S:System))]
          ) ; =sys=> .
```

We consider once again the example of Section 3. We have instantiated the semantics with two different definitions of function `ET` corresponding to the *minimal* semantics (the final configuration is the first one where the main variable becomes inactive), and the *maximal* semantics (the execution continues until there is no active thread in the system). The results are shown in the following table.

	Minimal	Maximal
Execution time/global steps	12	7
Total work done	12	10
Average thread parallelism	1	1.43
Maximal thread parallelism	1	3
Average process parallelism	1.92	1.875
Number of communications	2	2
Speed	1.71	

Notice that the total work done in the minimal semantics is greater than in the maximal one. This apparent contradiction is due to the fact that for the total work done we count the number of threads that have been activated. Consider the following situation: In the minimal semantics a variable x demands a communication value, but this value has not been evaluated yet, so that the thread corresponding to x becomes blocked. When the value is finally obtained, x is reactived, so that the binding for x has been active twice. By contrast, in the maximal semantics the needed value is evaluated previously; therefore, the thread that evaluates x gets the value and becomes deactivaded in only one step. Consequently, in this particular case, the work done (number of active threads that have evolved) is greater for the

minimal semantics.

The next step in our project will consist in studing representative examples that exploit the differences between the semantics, and consider other alternative definitions of function `ET`; thus, obtaining conclusions of the measurements.

6 Conclusions and future work

The conjugation of Eden operational semantics and Maude has proved to be fruitful because the characteristics of the latter meet Eden semantics implementation needs very faithfully. For instance, Maude rewrite rules mechanism has been an excellent tool for implementing the reduction steps in Eden semantics.

Furthermore, Maude modularity has helped to implement different language design decisions, or the scheduler options, which depend on the threads that are allowed to evolve at each step. We have been able to implement a prototype tool where the user can play with different parameters of the semantics.

Moreover, Maude high level of abstraction has allowed us to obtain an implementation of the rules very similar to the operational rules. Consequently, the code is exceptionally readable, in fact, its reading is almost equal to reading the original semantics. Maude not only has been useful in the semantic aspects, but also at the syntactical level: thanks to Maude operators we have defined the syntax in a very direct way. Besides, the existence of subsorts has facilitated the expression classification into variables, weak head normal forms, and so on.

Once this implementation is stable, this versatile interpreter is to be used for analyzing computations obtained by using different language design options. These analysis will be based on the measures mentioned above and on the computations themselves. The comparisons will focus on the efficiency, the duplication of work, the amount of speculation, the termination of computations, etc. Afterwards, the language will be extended with other features of Eden such as communication via streams, and nondeterminism.

Our examples have been executed using a prototype interpreter of the strategy language implemented at the Maude metalevel [13]. Due to the inefficiency of this interpreter, by now we have only been able to test our tool with small examples. Currently a direct implementation of the strategy language is being developed at the C++ level, at which the Maude system itself is implemented. This will make the strategy language a stable new feature of Maude, thus allowing a more efficient execution. Obviously, this new implementation will allow us to explore new more complex and interesting examples.

References

[1] 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)*, pp. 162–173, 2000.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, 2005. http://maude.cs.uiuc.edu/manual.

[4] Glasgow Haskell Compiler.
`http://www.haskell.org/ghc`.

[5] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. PhD thesis, Universidad Complutense de Madrid, 2004.

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

[7] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Looking for Eden through Maude and its strategies. Web page `http://maude.sip.ucm.es/eden`, 2006.

[8] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams become reality. In *Selected Papers 10th Int. Workshop on Implementation of Functional Languages (IFL'98)*, LNCS 1595, pp. 103–119. Springer, 1999.

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

[10] R. Loogen. Research Directions in Parallel Functional Programming. In K. Hammond and G. Michaelson, eds., *Programming Language Constructs*, pp. 63–92. Springer, 1999.

[11] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–445, 2005.

[12] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, eds., *Handbook of Philosophical Logic, Second Edition, Volume 9*, pp. 1–87. Kluwer, 2002.

[13] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, ed., *Proc. Fifth Int. Workshop on Rewriting Logic and its Applications, WRLA 2004*, ENTCS 117, pp. 417–441. Elsevier, 2004.

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

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

[16] F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, eds., *Proc. 6th Int. Workshop on Rule-Based Programming, RULE 2005*, ENTCS 147, pp. 135–161. Elsevier, 2006.

[17] F. Rubio. *Programación funcional paralela eficiente en Eden*. PhD thesis, Universidad Complutense de Madrid, 2001.

[18] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, eds., *Proc. Fourth Int. Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71, pp. 239–257. Elsevier, 2002.