semantics of languages based on such lazy strategies rises the need of developing good debugging tools. Currently, there are debugger models for pure functional languages [25],[14],[18]. Moreover, the interest of functional logic programming for practical applications and the different implementations existing for them [3],[15, 16],[23],[27], [6],[12], [20] have motivated the development of good debugging tools as we can find in [13],[8]. However, although in the last years methods to translate lazy rewriting and narrowing into Prolog have been developed in such a way that Prolog computation rules simulate the lazy strategy, there are no existing debugger tools for this kind of strategies.

In this paper we present a debugging model and an environment -BabLog- for a programming language which combines lazy functional and logic programming. The computation of BabLog programs is based on lazy narrowing under a particular control regime *demand driven (dds)* presented in [19] which determines an order for evaluating arguments and applying rules in computing head normal forms of total function applications. Such order is driven by the concept of *demanded* arguments which will be defined in subsection 3.1. Moreover, the debugging model is also appropriate for reflecting BabLog goal computations executed under other lazy strategies which could be expressed similarly to the *dpt* strategy.

Our debugging model is based on Byrd's box model [4] for logic programs, incorporating new kinds of boxes which reflect lazy narrowing, and giving extra information in some box ports in order to make easier the chase of goal computations. From the user point of view, several facilities similar to those existing in most Prolog debuggers have been incorporated. On the other hand, the environment includes a compiler which translates a program into Prolog clauses following the *dds* and *naïve* strategies and tools enabling to measure the time and number of calls to hnf needed for computing one or more solutions of a goal.

The organization of this paper is as follows. In section 2 we introduce the syntax of BabLog. Section 3 explains the computational semantics of the *dds* strategy, giving an algorithm which reflects how to compute head normal forms of total function applications. Section 4 contains the debugging model of BabLog and explains the possibility of applying this model to other lazy strategies. Section 5 presents our actual implemented environment. Some conclusions are finally drawn in section 6.

2 BabLog Syntax

The syntax of BabLog is similar to that of the functional logic programming language Babel [15],[22] which includes a type system with parametric polymorphism in the style of ML [21] [7], facilities to define higher order functions using *curryfication* and *partial application* in order to achieve higher order programming without using λ–abstractions in the line of Miranda [26].

Let $< DC, DF >$ be a signature with the ranked alphabet $DC = \bigcup_{n \in N} DC^n$ of data constructor symbols, and the disjoint ranked alphabet $DF = \bigcup_{n \in N} DF^n$ of function symbols. The following syntactic domains are distinguished:

- *Variables* $X, Y, Z \in Var$
- *Terms* $s, t, .. \in Term$:

$$
\begin{array}{lcll}
t & ::= & X & \% \; X \in Var \\
 & | & c & \% \; c \in DC^0 \\
 & | & (c\, t_1 \ldots t_n) & \% \; c \in DC^n, t_i \in Term, i \in \{1, \ldots, n\}, n \geq 1,
\end{array}
$$

- *Expressions* $e, l, r, .. \in Expr$:

$$
\begin{array}{lcll}
e & ::= & t & \% \; t \in Term \\
 & | & c & \% \; c \in DC \\
 & | & f & \% \; f \in DF \\
 & | & (e_1 e_2) & \% \; \text{'application'}, e_1, e_2 \in Expr
\end{array}
$$

In the following, letters $c, d, ..$ are used for constructors, $t, s, ..$ for terms, $e, l, r, ..$ for expressions and $f, g, h, ..$ for function symbols.

2.1 BabLog programs

A BabLog program consists of a finite set of function and predicate definitions and declarations of the constructed types. Type declarations occurring in a BabLog program must be declared before being used. A function symbol $f \in DF^n$ is defined by a finite set of rules which are conditional equations. Each *defining rule for* f must have the form:

$$\underbrace{f\, t_1 \ldots t_n}_{lhs} \;:=\; \overbrace{\underbrace{\{l_1 = r_1, \ldots, l_n = r_n \rightarrow\}}_{guard (optional)}\; e}^{rhs}$$

where $t_i \in Terms$, $e, l_i, r_i \in Expr$. A BabLog goal has the same structure than a *guard*.

Function definitions must obey certain natural conditions such as *left linearity*, *local determinism* and *nonambiguity*; see [9]. Function symbols must also have main type. The syntax for predicate definitions is an extension of Prolog one, but predicates are handled as boolean functions. As an example, a legal BabLog program is given by the following defining rules and type declarations:

```
data nat      := zero|(suc nat).
data list A   := [ ] | [A |(listA)].

fun foo_great                      :  nat → nat → bool.
foo_great zero Y                   :=  false.
foo_great (suc X) zero             :=  true.
foo_great (suc (suc X)) (suc Y)    :=  foo_great (suc X) Y.

fun or          :  bool → bool → bool.
or X true       :=  true.
or true Y       :=  true.
or false false  :=  false.

fun map         :  (A → B) → list A → list B.
map F [ ]       :=  [ ].
map F [X|Xs]    :=  [F X| map F Xs].
```

A goal for this program is:

$$map \;\; (foo\_great\; (suc\; (suc\; (suc\; zero)))) \; [zero, (suc\; (suc\; zero))] \;=\; L$$

for which we expect $L = [true, true]$ as computed answer.

3 Operational Semantics under *dds*

The Demand Driven Strategy *(dds)* is a computational strategy for lazy narrowing which was first specified in [19] as a translation into Prolog. We have adapted this strategy in order to support BabLog programs. As we said in subsection 2.1, a BabLog goal G is a sequence of equalities between expressions, thus *to solve* G will consist of solving sequentially the equalities occurring in G. Before commenting the way in which an equality is solved, we define the set $Sfree$ of safe free variables of an expression - which corresponds to the free variables occurring in its shell [19]-. as follows:

$$
\begin{array}{lcll}
Sfree(X) & := & \{X\} & \% \ X \in Var \\
Sfree(c\ e_1 \ldots e_n) & := & \bigcup_{i\in\{1,\ldots,n\}} Sfree(e_i) & \% \ e_i \in Expr, c \in DC \\
Sfree(f\ e_1 \ldots e_n) & := & \bigcup_{i\in\{1,\ldots,n\}} Sfree(e_i) & \% \ e_i \in Expr, f \in DF^m, n < m \\
Sfree(f\ e_1 \ldots e_n) & := & \emptyset & \% \ e_i \in Expr, f \in DF^n
\end{array}
$$

The first step for computing an equality $e = l$ between two expressions is to calculate the head normal forms of l and e (Hl, He respectively). If some of the previous computations fail, then the equality fails. Otherwise, the equality $He = Hl$ must be solved, by following the rules:

1. $X = Y$, $X, Y \in Var$, succeeds producing the binding $\{X/Y\}$.
2. $X = (c\ e_1 \ldots e_n)$, $X \in Var$, $c \in DC$ succeeds producing the following binding $\{X/(c\ t_1\ \ldots\ t_n)\}$ if:
   - $X \notin Sfree(c\ e_1 \ldots e_n)$.
   - The sequence of equalities $X_1 = e_1, \ldots, X_n = e_n$ succeeds producing the bindings $\{X_i/t_i\}$, where X_i are fresh variables.
3. $(c\ e_1 \ldots e_n) = X$ is analogous to rule 2.
4. $(c\ e_1 \ldots e_n) = (c\ r_1 \ldots r_n)$ succeeds if $e_1 = r_1, \ldots, e_n = r_n$ succeeds.

The case of partial applications of a function symbol is computed analogously by considering the function symbol as a constructor symbol [9],[11].

3.1 Computation of head normal forms

We will use $hnf(e)$ to denote the head normal form of an expression e [2]. $hnf(e) = e$ holds except for total function applications for which program rules must be applied. In order to avoid repeated evaluations of arguments, the *dds* strategy applies defining rules based on the notion of *demanded positions* in such a way that the order in which rules are applied can vary from their application order. We will return over this point in later examples.

Given a BabLog program P, let P_f be the defining rules of f in P. We present an algorithm (DDS) to transform P_f into a new unique defining rule $f\ X_1 \ldots X_n := EA$, where $EA \in ExprAux$ and $ExprAux$ is defined by:

[2] An expression may have several head normal forms. In pure functional languages that is not possible.

$$
\begin{array}{lcll}
EA & ::= & \{b \rightarrow\}\ e & \% \ e \in Expr, b \text{ is a guard} \\
 & | & \textit{case } e \textit{ of} & \% \ e \in Expr \\
 & & \quad c_1\ X_{11} \ldots X_{1r_1} : EA_1 & \% \ c_1 \in DC^{r_1} \\
 & & \quad \square\ c_2\ X_{21} \ldots X_{2r_2} : EA_2 & \% \ c_2 \in DC^{r_2} \\
 & & \quad \ldots\ldots & \\
 & & \quad \square\ c_n\ X_{n1} \ldots X_{nr_n} : EA_n & \% \ c_n \in DC^{r_n} \\
 & & \textit{endcase} & \\
 & | & \textit{seq } EA_1 \square EA_2 \square \ldots \square EA_m \textit{ endseq} & \% \ EA_i \in ExprAux,\ i \in \{1, \ldots, m\}
\end{array}
$$

The set $ExprAux$ contains three kind of basic expressions. The first one refers to the *rhs* of some rule of P_f, the second one is a case distinction over the head normal form of e -being e the expression referenced by the operator *case*-. Each construction occurring in a *case* alternative represents a possible unification between $hnf(e)$ and such construction. Finally, the third one represents alternatives which will be tried sequentially -as a Prolog disjunction-.

◇ Preliminary notions

Before explaining the algorithm it is necessary to define some notions.

- A *call pattern* cpt is any function application $f\ t_1 \ldots t_n$ where $f \in DF^n$, $t_i \in Terms$. If $t_i \in Var$, for all $i \in \{1, \ldots, n\}$ then $f\ t_1 \ldots t_n$ is called a *generic call pattern*.
- Let cpt be a call pattern and let l be the *lhs* of a defining rule. We say that l *matches* cpt iff l is an instance of cpt via some term-substitution (necessarily linear).
- $pos(e)$ denotes the set of positions of the expression e. It is defined inductively by:
  $pos(e) = \{\epsilon\},\ if\ e \in Var$
  $pos(e) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n, p \in pos(e_i)\},\ if\ e \equiv \varphi\ e_1 \ldots e_n, \varphi \in DF \cup DC$
- Let cpt and R be a call pattern and a set of defining rules respectively, for some function symbol $f \in DF$.
  $lhs(cpt, R) = \{lhs \mid lhs := rhs \in R \text{ and } lhs \text{ matches } cpt\}$
- Let $cpt{=}f\ t_1 \ldots t_n$ be a call pattern, and let R be a subset of the set of defining rules of f. If $u \in pos(cpt)$, we will say:
  - u *is demanded* by $l \in lhs(cpt, R)$ iff l has a constructor at position u.
  - u is *demanded in* R iff u is demanded by some $l \in lhs(cpt, R)$.
  - u is *uniformly demanded* in R iff $lhs(cpt, R)$ is not empty and u is demanded by every $l \in lhs(cpt, R)$.

  as an example, let R be the set of defining rules for the function symbol *foo_great* defined in section 2.1. ($foo\_great\ X\ Y$) has position 1 uniformly demanded - and thus demanded- in R by the constructors $\{zero, suc\}$, while position 2 is demanded but not uniformly demanded in R.

3.2 The DDS algorithm

Let f be a function symbol and P_f its defining rules in P. We define:

$$DDS(f, P_f) := DDS(f\ X_1 \ldots X_n, P_f, \{1, .., n\}).$$

A call to $DDS(cpt, R, VP)$ where cpt is an instance of $f\ X_1 \ldots X_n$, R [3] is a subset of P_f, and $VP = pos(cpt)$ returns an expression $EA \in ExprAux$ by following the steps:

1. Some position in VP is uniformly demanded in R. Let u be the first less - following the lexicographic order- uniformly demanded position in R. Let X be the variable at position u in cpt. Let $c_1, \ldots, c_n$ -taken in textual order- be the constructors occurring at position u of the lhs of the rules in R such that $c_i \neq c_j$ for $i \neq j$, $i, j \in \{1, \ldots, n\}$. Let r_j be the arity of c_j, for all $j \in \{1, \ldots, n\}$. For each c_j build the following subsets of R, call pattern and VP:
   - $R_j = \{lhs := rhs \in R \textit{ taken in textual order} \mid c_j \textit{ occurs at position } u \textit{ in } lhs\}$
   - $cpt_j = cpt[X/c_j\ X_{j1} \ldots X_{jr_j}]$ where $X_{j1}, \ldots, X_{jr_j}$ are fresh variables
   - $VP_j = (VP - \{u\}) \cup \{u.1, .., u.r_j\}$ where '$-$' denotes the set difference

   The call to $DDS(cpt, R, VP)$ returns a case distinction:

   case X *of* % *demanded position* u
   $c_1\ X_{11} \ldots X_{1r_1}$: $DDS(cpt_1, R_1, VP_1)$
   $\square\ c_2\ X_{21} \ldots X_{2r_2}$: $DDS(cpt_2, R_2, VP_2)$
   .
   .
   $\square\ c_n\ X_{n1} \ldots X_{nr_n}$: $DDS(cpt_n, R_n, VP_n)$
   endcase

   Note that each construction occurring in a *case* alternative represents a possible unification between the head normal form of the expression occurring at position u in cpt and such construction. If the unification succeeds, bindings over the original cpt variables can be produced.

2. No position in VP is demanded in R. Then $DDS(cpt, R, VP)$ returns a sequence of alternatives:

   seq
   $\{b_1 \rightarrow\}\ e_1$ % *by* r_{l1}
   $\square$ $\{b_2 \rightarrow\}\ e_2$ % *by* r_{l2}
   .
   .
   $\square$ $\{b_m \rightarrow\}\ e_m$ % *by* r_{lm}
   endseq

   where r_{li} has the form $f\ t_{i1} \ldots t_{in} := \{b_i \rightarrow\}\ e_i$ and have been taken in the textual order of R. If $m = 1$ then the operator *seq* can be omitted.

3. Some position in VP is demanded, but not uniformly demanded in R. Let $u_1, \ldots, u_k$ be those positions in VP which are demanded in R, taken in lexicographic order. We make the following partition of R:

[3] R will be composed of rules whose lhs are instances of cpt.

- For each j, $1 \leq j \leq k$: R_j is the subset of R composed of those rules whose lhs matches cpt, demand the position u_j and are not in any R_i, $i < j$ [4].
- Let *Reduce* be the subset of R consisting of those rules not demanding positions in R.

The call to $DDS(cpt, R, VP)$ returns:

seq
$DDS(cpt, R_1, VP)$ % *go to step* 1
$\square$
$DDS(cpt, R_2, VP)$ % *go to step* 1
$\square$
.
.
$\square$
$DDS(cpt, R_k, VP)$ % *go to step* 1
$\{\square\ DDS(cpt, Reduce, VP)\}$ % *if* $Reduce \neq \emptyset$ *go to step* 2.
endseq

Example 1

Let *or* the function symbol defined in subsection 2.1. After computing $DDS(or, P_{or})$, the new defining rule for *or* has the form:

or A B := *seq*
case A of
true : *true* % *by* or_2
$\square$ *false* : *case B of*
false : *false* % *by* or_3
endcase
endcase
$\square$
case B of
true : *true* % *by* or_1
endcase
endseq

The steps followed by *dds* strategy in computing the head normal forms of (*or X true*) are represented in this new rule for *or*. The computed solutions would be:

1) *true binding X to true*
2) *true without producing bindings*

Note that if the defining rules for *or* had been applied in their definition order, the computed head normal forms for (*or X true*) would have been obtained just in the reverse order to the followed one by the *DDS* algorithm.

[4] This condition avoids repeated solutions.

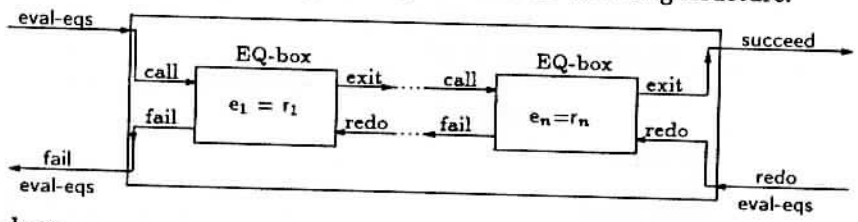
4 BabLog Debugging Model

Our debugging model for lazy narrowing is based on Byrd's box model. The standard *box-oriented* debugger for logic programs associates a box to each literal call during the computation process. The boxes are defined by the well-known four ports: *call, redo, exit, fail.* The first one is used when a literal must be proved for the first time. It is possible to leave the box through the *exit* port (if the literal is successfully proved) or the *fail* port (if the literal proof fails). If the literal proof succeeds and it is necessary to find another proof for it, the box is entered again through the *redo* port.

In order to provide a debugging model for lazy narrowing similar to the standard box-oriented one, we have introduced new kinds of boxes which reflect the different computation steps (narrowing, lazy_unifications,etc..) given by the *dds strategy* for solving goals. On the other hand our debugging model incorporates several facilities to allow the user to follow the computation of a goal in an easy way. Each box will have an associated number to indicating the number of entered boxes which are still to be proved. All ports for the same box will have the same associated number that the box has. Moreover each box is identified by a label which provides information about the kind of computation being executed. Leaving a box through the *fail* port means that the process represented by such box is completely finished.

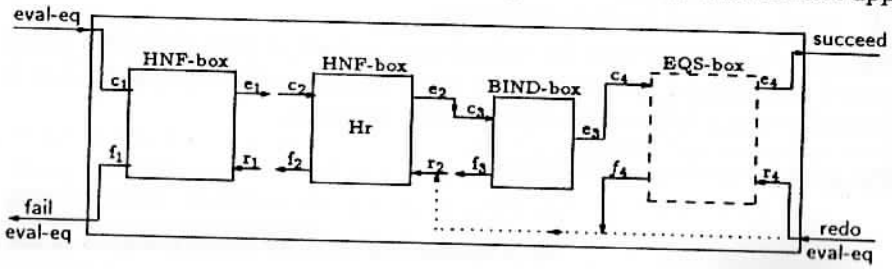
◇ Eqs-box

An *eqs-box* for a goal $e_1 = r_1, \ldots, e_n = r_n$ contains n boxes which correspond to the computation of each equality $e_i = r_i$, and has the following structure:



◇ Eq-box

An *eq-box* for an equality $e = r$ contains two *hnf-boxes* for computing the head normal forms (He, Hr respectively) of e and r, a *bind-box* for checking the compatibility of He and Hr, producing the corresponding bindings and, if either He or Hr are not variables, then it also contains an *eqs-box* for solving the new sequence of generated equalities -as it was said in section 3-. The structure of an *eq-box* is shown in the figure below, where the box represented by discontinuous lines can not appear.



In order to simplify the trace shown to the user, we have removed the *bind-box* contained in an *eq-box* showing the goal variable bindings when an *eq-box* is left with success.

◇ Hnf-box

This box corresponds to the computation of head normal forms. The *hnf-box* associated to the computation of head normal forms of variables, partial applications or expressions with a constructor in their head is very simple, since these expressions are in head normal form. Thus their *hnf-box* will be entered through the call port and they will be left immediately through the exit port giving the same expression as unique result.

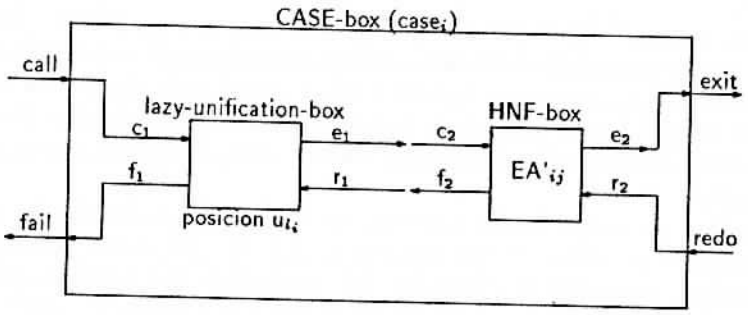
The *hnf-box* associated to the computation of head normal forms for total function applications is based on the expression $EA \in ExprAux$ returned by the DDS algorithm, presented in subsection 3.1, applied to the function symbol. Let $EA \in ExprAux$ be the result of computing $DDS(f, P_f)$ where $f \in DF^n$. The new defining rule for f is: $f\ X_1 \ldots X_n\ :=\ EA$. The steps given by the *dds* strategy for computing the head normal form of any total function application $f\ e_1 \ldots e_n$, $e_i \in Expr$ are represented in $EA' \in ExprAux$, where $EA' = (EA)[X_i/e_i]$. The general structure for EA' has the form below:

$EA' \ :=$ *seq*
case e_{l_1} *of* % *case*$_1$, *demanded position* u_{l_1}
c_{11} : EA'_{11}
...
□ c_{1n_1} : EA'_{1n_1}
endcase
...
□
case e_{l_k} *of* % *case*$_k$, *demanded position* u_{l_k}
c_{k1} : EA'_{k1}
...
□ c_{kn_k} : EA'_{kn_k}
endcase
□
seq r_{s_1} □ r_{s_2} □ ... □ r_{s_m} *endseq* % *reduce*
endseq

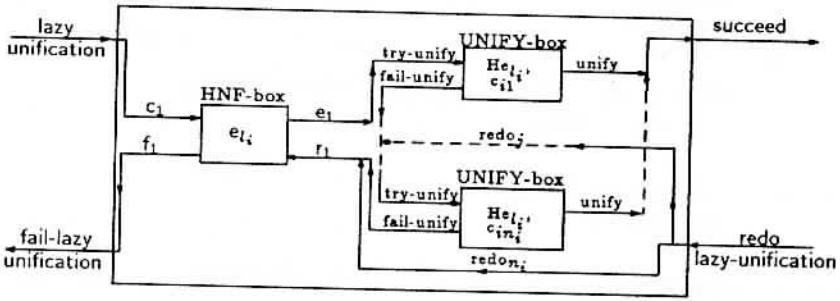
In order to simplify notation, we have used c_{ij} to represent the construction $c_{ij}\ X_1 \ldots X_{r_{ij}}$, $c_{ij} \in DC^{r_{ij}}$, where $X_1 \ldots X_{r_{ij}}$ are fresh variables. The *hnf-box* for $f\ e_1 \ldots e_n$ is based on EA' and thus contains k *case-boxes* -one for each case alternative- and a *reduce*-box corresponding to the set of applying rules $r_{s_1}, \ldots, r_{s_m}$. As all information offered by these boxes to the user is contained in their internal boxes, we omit their structures, showing only the contents of their internal boxes.

Inside the $i-th$ case alternative -labeled with *case*$_i$- two processes can be distinguished. The first one refers to the evaluation to head normal form of e_{l_i} and the possible unifications between $hnf(e_{l_i})$ and the constructions $c_{ij}\ X_1 \ldots X_{r_{ij}}$, for

each $j \in \{1, .., n_i\}$. The second one refers to the computations represented by EA'_{ij}. Thus, a *case-box* contains a *lazy-unification*-box which represents the first mentioned process, and a *hnf-box* for representing the second one. The structure of a *case-box* is given below:

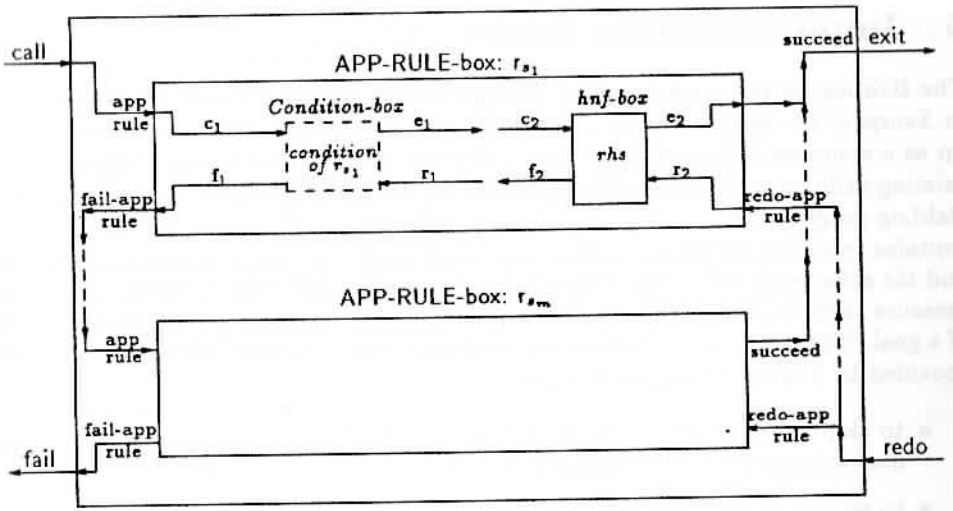


As it can be observed in the next figure, the *lazy-unification-box* for position u_{l_i} contains a *hnf-box* for computing $hnf(e_{l_i})$ and as many *unify-boxes* as constructors demanding u_{l_i}. A *unify-box* computes the unification between two expressions, so these boxes have no *redo* port. If the unification between $hnf(e_{l_i})$ and $c_{ij}\ X_1 \ldots X_{r_{ij}}$, for some $j \in \{1, .., n_i\}$ succeeds, then the *hnf-box* associated to the expression EA'_{ij} is entered. If the *lazy-unification-box* must be entered again through its corresponding *redo* port then the unification between $hnf(e_{l_i})$ and $c_{ij+1}\ X_1 \ldots X_{r_{ij+1}}$ is tried. If $j = n_i$ then the *hnf-box* associated to e_{l_i} is entered through its redo port in order to find another alternative proof.



Call ports for *unify-boxes* contain extra information about the rules being tried, while fail ports show the number -according to textual order- of the rule which has been discarded.

Finally, the *reduce-box* associated to the alternative labeled with *reduce* will contain m *app-rule-boxes* which represent the different applications of defining rules for the symbol function f. If the rule to be applied has a condition, i.e. it has the form: $f\ e_1 \ldots e_n := b \rightarrow e$, it will be necessary to solve b before evaluating e to head normal form. If the rule has no condition, then the head normal form of its *rhs* must be computed immediately. The boxes for *reduce* and *appl-rule* are given in the figure below:



As conditions have the same structure that a goal, the *condition-box* is analogous to an *eqs-box* although their identifiers are different. Permitting different identifiers to represent the same process helps the user to distinguish clearly the execution steps.

4.1 Debugging other lazy narrowing strategies

Our debugging model can be applied to another existing strategy of translation into Prolog for BabLog. Such strategy is based on the *naïve* regime control presented in [19]. The application of our debugging model to this strategy is possible due to the fact that computations of head normal forms of total function applications can be represented by an expression $EA \in ExprAux$. For more details about the *naïve* strategy, see [19].

Let us suppose that $f \in DF^n$ is defined by the following set of defining rules:

$$\begin{array}{lcll} f\ t_{11} \ldots t_{1n} & := & \{b_1 \rightarrow\}\ e_1 & \%\ f_1 \\ f\ t_{21} \ldots t_{2n} & := & \{b_2 \rightarrow\}\ e_2 & \%\ f_2 \\ \ldots & & & \\ f\ t_{m1} \ldots t_{mn} & := & \{b_m \rightarrow\}\ e_m & \%\ f_m \end{array}$$

If we apply separately the DDS algorithm to each defining rule of f, we obtain:

$$\begin{array}{lcl} DDS(f, \{f_1\}) & := & EA_1 \\ DDS(f, \{f_2\}) & := & EA_2 \\ \ldots & & \\ DDS(f, \{f_m\}) & := & EA_m \end{array}$$

where $EA_i \in ExprAux$. With the former auxiliary expressions we build a new defining rule for f:

$$\begin{array}{lcl} f\ X_1 \ldots X_n & := & seq \\ & & EA_1 \square \ldots \square\ EA_m \\ & & endseq \end{array}$$

in such a way that this rule contains all the information needed to compute the head normal forms of any total function application for f.

5 Implementation Issues

The BabLog environment has been implemented in *Bimprolog* and can be executed in *Sunsparc* 1+ stations, under *SunOs* 4.1.1 or higher. The environment is thought up as a command interpreter by means of which the user can access to the different existing utilities, such as compilation -following *dds* or *naïve* strategies- and loading of BabLog programs, debugging of goal computations, etc.. Moreover, the environment contains two kind of evaluation modes: *the eval mode* -for computing all kind of goals- and *the solve mode* -for computing only boolean goals-. In both modes it is possible to measure the time and number of calls to hnf needed to compute one or more solutions of a goal. The debugger in the BabLog environment presents facilities similar to those provided by Prolog debuggers. It allows:

- to skip from a *call* port to an *exit* port -belonging to the same box- skipping over subcomputations inside the box.
- to set spy points on defined functions, to consult the current spy points and to remove existing spy points.
- to consult the current bindings of the variables of a goal being computed.
- to leave the debugger mode aborting the computation of a goal, or to leave the debugger mode showing the solutions obtained for the goal.

Moreover, some *call* and *exit* ports contain some extra information which helps the user to follow the computation of a goal. For instance, the *call* port of each *unify-box* will show the possible rules available for the evaluation of a total function application. In this way, when all the *lazy-unification-boxes* have finished with success, the last *unify-box* appearing in the last *lazy-unification-box* will contain the set of possible rules which will be tried sequentially. The *call* port for each *lazy-unification-box* will show the position in the goal which is going to be evaluated to head normal form and the set of constructors demanding that position.

In order to simplify the trace shown to the user, all ports for *case* and *reduce boxes* are not shown since the information they can give is contained in their internal boxes. Moreover, ports for *hnf-boxes* inside a *case-box* are omitted by showing in *exit* ports of lazy-unification-boxes the new aspect of the original total function application after the lazy unification.

Now, we present an example which shows some of the steps shown by the debugger in computing the goal $(foo\_great\ suc\ (suc\ zero)\ B\ =\ H)$. In this example it can be observed that demanded positions inside the same argument are not evaluated sequentially from left to right.

[0] $EVAL-EQS:\ foo\_great\ (suc\ (suc\ zero))\ B\ =\ H$
[1] $EVAL-EQ:\ foo\_great\ (suc\ (suc\ zero))\ B\ =\ H$
[2] $EVAL-HNF:\ foo\_great\ (suc\ (suc\ zero))\ B$
[3] $LAZY-UNIFICATION:\ (suc\ (suc\ zero))$
$-position\ 1\ demanded\ by\ \{zero, (suc\ X)\}-$
[4] $EVAL-HNF:\ (suc\ (suc\ zero))$
[4] $HNF >>\ (suc\ (suc\ zero))$

[4] $TRY-UNIFY:\ (suc\ (suc\ zero))\ with\ zero$
$-trying\ apply\ rule\ 1\ of\ foo\_great-$
[4] $FAIL-unify:\ rule\ 1\ of\ foo\_great\ not\ applicable$
[4] $TRY-UNIFY:\ (suc\ (suc\ zero))\ with\ (suc\ X)$
$-trying\ apply\ rule\ 2,3\ of\ foo\_great-$
[4] $UNIFY >>\ \{(suc\ (suc\ zero)) \rightarrow\ (suc\ (suc\ zero))\}$
[3] $SUCCEED >>\ foo\_great\ (suc\ (suc\ zero))\ B$ $-new\ call\ pattern-$
[3] $LAZY-UNIFICATION:\ B$
$-position\ 2\ demanded\ by\ \{zero, (suc\ Y)\}-$ "s" % skip
[3] $SUCCEED >>\ foo\_great\ (suc\ (suc\ zero))\ zero$ $-new\ call\ pattern-$
[3] $APP-RULE:\ rule\ 2\ to\ (foo\_great\ (suc\ (suc\ zero))\ zero)$
[4] $EVAL-HNF:\ true$
[4] $HNF >>\ true$
[3] $SUCCEED >>$
[2] $HNF >>\ true$
[2] $EVAL-HNF:\ H$
[2] $HNF >>\ H$
[1] $SUCCEED >>\ \{B \rightarrow\ true,\ H \rightarrow\ true\}$
[0] $SUCCEED >>\ \{B \rightarrow\ true,\ H \rightarrow\ true\}$

If we wish to obtain another solution:

[0] $REDO-eval-eqs:\ foo\_great\ (suc\ (suc\ zero))\ true\ =\ true$
[1] $REDO-eval-eq:\ foo\_great\ (suc\ (suc\ zero))\ true\ =\ true$
[2] $REDO-eval-hnf:\ true$
[2] $FAIL-eval-hnf:$
[2] $REDO-eval-hnf:\ foo\_great\ (suc\ (suc\ zero))\ true$
[3] $REDO-app-rule:\ rule\ 2\ to\ (foo\_great\ (suc\ (suc\ zero))\ zero)$ "s"
[3] $FAIL-app-rule:\ \{no\ more\ rules\ for\ applying\ to$
$(foo\_great\ (suc\ (suc\ zero))\ zero)\}$
[3] $REDO-lazy-unification:\ zero$
$-position\ 2\ demanded\ by\ \{zero, (suc\ Y)\}-$ "s"
[3] $SUCCEED >>\ foo\_great\ (suc\ (suc\ zero))\ (suc\ Y)$
$-new\ call\ pattern-$
[3] $LAZY-UNIFICATION:\ (suc\ zero)$
$-position\ 1.1\ demanded\ by\ \{(suc\ Z)\}-$ "s"
[3] $SUCCEED >>\ foo\_great\ (suc\ (suc\ zero))\ (suc\ Y)$
$-new\ call\ pattern-$
[3] $APP-RULE:\ rule\ 3\ to\ foo\_great\ (suc\ (suc\ zero))\ (suc\ Y)$ "s"
[3] $SUCCEED >>$
[2] $HNF >>\ true$
[2] $EVAL-HNF:\ H$
[2] $HNF >>\ H$
[1] $SUCCEED >>\ \{B \rightarrow\ (suc\ zero), H \rightarrow\ true\}$
[0] $SUCCEED >>\ \{B \rightarrow\ (suc\ zero), H \rightarrow\ true\}$

To compute the second head normal form of $(foo\_great\ (suc\ (suc\ zero))\ B)$ it has been necessary to evaluate to head normal form positions $1, 2 and 1.1$. Note that position 1.1 (within the first argument) has been evaluated to *hnf* after position 2 (corresponding to the second argument) has been. This means that a *left-to-right* evaluation order is not always followed.

6 Conclusions and Related Work

We have presented a debugging model for BabLog -a lazy functional logic programming language- which reflects faithfully the computational semantics for the *Demand Driven control regime*. Lazy evaluation delays the computation of an expression until it is needed, hence the sequence of evaluations is usually quite hard to predict. The *dds* strategy establishes its own order in evaluating arguments and applying rules, based on the lazy narrowing. Our debugger helps the user to trace and correct BabLog programs and to understand the sophisticated operational semantics of the *dds* strategy by showing aditional information about goal computations. The environment also contains tools to measure the execution time and number of calls to hnf needed for computing a goal and compare the *dds* strategy efficience with other existing lazy strategies for lazy narrowing. Moreover, the presented debugging model is suitable for other lazy narrowing strategies whose operational semantics can be expressed similarly to the *dds* strategy.

Acknowledgement

We thank Mario Rodríguez-Artalejo for many valuable discussions. We are grateful to J. Mateos-Lago for his careful reading of this paper and also to Lourdes Araujo-Serna and J.C. Gónzalez-Moreno for their help about LaTeX.

References

[1] S. Antoy: *Lazy Evaluation in Logic*, Symp. on Programming Language Implementation and Logic Programming 1991, LNCS 528, Springer Verlag 1991, 371-382.

[2] S. Antoy: *Definitional Trees*, Int. Conf. on Algebraic and Logic Programming (ALP) 92, LNCS 632, Springer Verlag 1992, 143–157.

[3] P.G. Bosco, C. Cecchi, and C. Moiso: *An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing.* In proc. Sixth International Conference on Logic Programming (Lisboa), MIT press 1989, pp. 318–333.

[4] L. Byrd: *Understanding the control flow of Prolog programs.* In proc. of the Whorkshop on Logic programming, December 1980.

[5] P.H. Cheong: *Compiling lazy narrowing into Prolog*, Technical Report 25, LIENS, 1990, to appear in: Journal of New Generation Computing.

[6] M.M. Chakravarty, H.C.R. Lock: *the implementation of lazy narrowing.* In proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, pp. 123–134. Springer LNCS 528, 1991.

[7] L. Damas, R. Milner: *Principal type schemes for functional programs*, ACM Symp. on Principles of Programming Languages, 1982, pp. 207–212.

[8] A. Gil-Luezas, F.J. López-Fraguas, M.T. Hortalá-Gónzalez: *Babel, Manual de Utilización.* thecnical report DIA 92/15. UCM, December 1992.

[9] J.C. González Moreno, M.T. Hortalá González, M. Rodríguez Artalejo: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*, Procs. Computer Science Logic CSL'92, LNCS 702,1993, pp. 216-230.

[10] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: KERNEL-LEAF: : *A Logic plus Functional Language*, journal of Computer and System Sciences, Vol. 42,No. 2, Academic Press 1991, 139–185.

[11] J.C. Gónzalez-Moreno: *A correctness proof for Warren's HO into FO translation.* In procs. GULP'93, pp. 569–583,1993.

[12] M. Hanus: *efficient implementation of narrowing and rewriting*, In proc. Int. Workshop on Processing Declarative knowledge, Springer LNAI 567, 1991, pp. 344–365.

[13] M. Hanus, B. Josephs: *A Debugging Model for Functional Logic programs.* In procs. PLILP'93, Springer LNCS 714, 1993. pp. 28–43.

[14] Cordelia V. Hall, John T. O'Donnell: *Debugging in applicative languages.* Journal of Lisp and Symbolic Computation, 1,1 1988

[15] H . Kuchen,R. Loogen, J.J. Moreno Navarro, M. Rodríguez-Artalejo: *Graph Based Implementation of a Functional Logic Language*, European Symp. on Prog. ESOP 1990, LNCS 432, Springer Verlag 1990, pp. 71–84.

[16] H . Kuchen,R. Loogen, J.J. Moreno Navarro, M. Rodríguez-Artalejo: *Lazy Narrowing in a graph machine.* Conf. on Algebraic and Logic Programming (ALP 90), LNCS 463, Springer Verlag 1990, pp. 298-317.

[17] J.A. Jiménez Martín, J. Mariño Carballo, J.J. Moreno Navarro: *Efficient Compilation of Lazy Narrowing into Prolog*, LOPSTR 92, LNCS, Springer Verlag 1992.

[18] G. Lapalme, M. Latendresse: *A Debugging Environment for Lazy Functional Languages.* Journal of Lisp and symbolic computation, Vol. 42,pp. 271–287, Kluwer Academic Publishers, 1992.

[19] R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP) 93, LNCS, Springer Verlag 1993.

[20] R. Loogen: *From reduction machines to narrowing machines.* TAPSOFT 91,CCPSD, LNCS 494, Springer Verlag 1991, pp. 438–457.

[21] R. Milner: *A theory of type polimorphism in programmimg*, J. Computer an System Sciences 17, 1978.

[22] J.J. Moreno Navarro, M. Rodríguez Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming Vol. 12, North Holland 1992 191–223.

[23] A. Muck: *Compilation of Narrowing.* In proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming. Springer LNCS 456, 1990. pp. 16–29.

[24] S. Narain: *A technique for doing lazy evaluation in logic*, The Journal of Logic Programming, 3, 1986, 259-276.

[25] C. Runciman, and I. Toyn: *Adapting combinator and SECD machines to display snapshots of functional computations*, New Generation Computing Vol. 4 1986. pp. 339–363.

[26] D.A. Turner: *Miranda: A non-strict functional language with polymorphic types*, ACM Conf. on Functional Languages and Computer Architecture 1985, LNCS 201, Springer 1985, pp. 1–16.

[27] D. Wolz: *Design of a compiler for lazy pattern driven narrowing.* In recent Trends in Data Type Specification, Springer LNCS 534, 1990. pp. 362–379.

eta

Everything buT Assignment*

Vincenzo Ambriola Giovanni A. Cignoni Laura Semini

Dipartimento di Informatica
Università di Pisa
corso Italia 40, 56100 Pisa

Abstract

We define *eta*, a new programming language that combines multiple tuple space, object oriented, and logic programming models. Multiple tuple spaces provide a powerful model for the development of large systems as a collection of communicating components. The combination of multiple tuple spaces with logic programming results in a paradigm that well integrates modularity and declarativity.

The object oriented model brings new insights to enhance tuple space languages without affecting their basic characteristics. In the language *eta* we integrate these three models: *eta* is a multiple tuple space logic language that borrows the programming style of the object oriented paradigm; supplies a reliable communication policy; enhances control allowing mutual exclusion and priority.

Keywords Concurrency, multiple tuple spaces, logic programming, object orientation.

1 Introduction

Tuple space concurrent languages exploit, for cooperation among processes, the shared memory paradigm. The store (i.e. the tuple space) is a multiset of tuples representing the cooperation state. Processes communicate by reading and writing on it.

The first system truly based on the notion of a tuple space, logically shared among concurrent agents, is Linda [9, 8]. Initially conceived as a coordination language, Linda evolved into a well defined paradigm to model communication and synchronization among distributed processes, written in various languages (C, Fortran, Prolog). Linda provides primitive operations to read, update, and change a tuple space. In Linda

*This work has been partially founded by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo del CNR, Sottoprogetto 6

the tuple space is the medium for cooperation between processes that have their own private control and a local store.

Coordination in Linda resembles the model of concurrent constraint programming, where processes communicate and synchronize through the store. Indeed, there are many differences between the two models. For instance, shared variables, that provide a second medium of communication in the concurrent constraint model, are not present in Linda.

In Shared Prolog (SP) a tuple is a logical fact [5]. The interaction between the tuple space (called blackboard) and the agents is controlled by the unification algorithm. The presence of some configurations of the tuple space causes the agents to react with a transition. A transition (or action, or step) is an operation composed of a guarded access to the tuple space, a computation, and a write operation on the tuple space. Agents have a local store and a local control only within each single step.

In a far-reaching extension of the tuple space paradigm, systems are composed of a set (or hierarchy) of tuple spaces, instead of a single one. Tuples are sent through the system from one tuple space to another, exploiting explicit addressing. This extension orthogonally combines the shared memory and the message passing paradigms, providing a model for the modular development of large systems as collection of components. A tuple space is the local store of each component and components interact exchanging tuples. Languages based on multiple tuple spaces are Linda-3 [12], ESP [7], PoliS [10], and Paté [4].

In this paper we present a new language, called *eta*, that combines the multiple tuple space model with object orientation and logic programming following the ideas presented in [2]. We identify a component, i.e. a tuple space and its associated agents, with an object. As in the SP paradigm, we exploit unification for the interactions between the tuple space and the agents.

The object oriented paradigm supports the notions of information hiding, inheritance, and inter-objects interaction policies. This merge brings new insights on the multiple tuple space framework.

In Section 2 we discuss some features of existing multiple tuple space languages and we show how they can be improved. In Section 3 we present the language *eta* showing how it copes with some general notions of concurrent and object oriented languages. In Section 4 we supply the syntax and an informal semantics of *eta*. In Section 5 we analyse some features of *eta* with respect to the general object oriented model and in the Appendix we provide an example of *eta* programming.

2 Motivations

For the general characteristics of the languages based on multiple tuple spaces we refer to the literature [7, 4, 10, 12]. Here we focus on some features: program structure, protection mechanism, priority, and mutual exclusion. We show how the existing languages deal with these features and we propose an alternative solution.

Program structure
In ESP and Paté (PoliS and Linda-3 features are substantially equivalent) a program consists of a library of theories, each theory being the parametric definition of an agent behaviour. At run time, many theories are activated on each tuple space, most of the times at tuple space creation. At this purpose, an *activation goal* specifies: the tuple space name; its initial state; a final condition that, when satisfied, causes the termination of the tuple space itself (both optional); the theory instances corresponding to the activated agents.

This style of programming gives a high degree of freedom and allows the reuse of theory definitions in different contexts. It has at least two disadvantages: activation goals are dramatically cumbersome; it is unnatural to design theories out of any context, as theories describe the communication of a tuple space with the others.

The experience of three years of programming in these languages, within the Oikos project at the University of Pisa [1, 3], has proven the fallacy of this approach. The reuse of parametric theory definitions is a rare event. Almost always, theories are designed having in mind the tuple space on which they are successively activated.

We contend that it is more natural to treat a tuple space and the set of its associated agents as a unique programming entity. This idea leads to the notion of object.

Protection mechanism
One of the common features of multiple tuple space languages is that an agent can send tuples to any tuple space provided that its address is known. Furthermore, in Linda-3 each agent can freely access the tuples of any tuple space both in read and consume mode. These language design choices stem from the attempt to maintain the philosophy of (single) tuple space based languages, where the tuple space is a communication medium that allows the autonomous behaviour of (*uncoupled*) agents.

As a consequence, an agent cannot check whether either data will eventually reach their destination or communication failed, e.g. because of an error in the address. Similarly, it is not possible to prevent a tuple from reaching a tuple space. When a tuple reaches a tuple space where it is unexpected, i.e. in which it is ignored by the agents, the tuple is not processed and the sender will never receive an answer or an acknowledgement.

The interactions between different tuple spaces are de facto inspired by the message passing paradigm, and communication policies, appropriate for this paradigm, must be supplied.

Priority
In SP, as well as in ESP, an agent is composed of a set of guarded rules. At each step, one of the rules whose guard is satisfied is nondeterministically chosen and the corresponding action is performed. This amounts to say that an agent selects the action to be performed according to the don't care nondeterminism model of classical logic concurrent languages [14].

In Paté, control strategies such as sequencing and alternation of agent actions are declaratively expressed in terms of legal activation sequences, by path expressions [4].

However, the language is still missing a way of expressing a priority relation between rules.

Mutual exclusion
Agents operate on a tuple space concurrently accessing the same tuples. In some situations an agent need to lock a part of the tuple space. For instance, the agent may want to perform a sequence of actions on a "private space", or to guarantee a mutual exclusion between two or more actions of different agents. Mutual exclusion is especially needed when the predicate *all* is involved in the guard of an agent (see Paragraph 4.6 and the use of *all* in the example supplied in the appendix).

3 The language eta

The language *eta* is designed to build (logically and physically) distributed systems, composed of a collection of concurrent communicating objects. Object states consist of a passive component, the tuple space, and of an active component, a set of *threads*. The former is a multiset of logical facts (atoms), while the latter corresponds to agents and resembles the agents of a Paté system.

We present the characteristics of *eta* with respect to a general classification criterion, showing how it copes with the notions of dynamicity, communication, information hiding, modularity, and others.

3.1 Dynamicity

Dynamic languages allow the design of open systems, where it is possible to add to a system an entirely new component that was not even thought at system start up. This incremental growth should not compromise the behaviour of the system itself. In an open system, a component is added without halting the running system, compiling, and activating the new version.

In *eta*, where systems are composed of collections of objects, we achieve dynamic features with the definition and the activation of new objects at run time.

3.2 Communication

In *eta*, objects communicate by sending and receiving messages, using object names as addresses. The language is asynchronous in the sense that, after a send, objects do not stop waiting for an answer.

Messages are atoms that are added to the receiver object's state and not operation or procedure invocations like in most of the object oriented languages. This allows to model, at semantic level, the interactions between two objects as a *rendez-vous*: after sending a message, an object is blocked until either the message has reached the destination or a failure in the communication is acknowledged.

This synchronization constraint has a very low impact on the system parallelism and represents a sufficient condition to build reliable systems. Moreover, combined

with path expressions (see Paragraph 3.7), it forces the sender of a message to wait for an answer to its request. It is hence possible to program totally synchronous communications, yielding to a (potentially) synchronous language.

3.3 Information Hiding

An object provides an interface to the outside world in terms of the messages that both accepts as input by other objects and produces (and sends through the system). Any message sent to an object that does not consider it a possible input, is treated as an error, and the sender is acknowledged. In this way we prevent unexpected tuples from reaching a tuple space, as wanted. Moreover, as the interface is the only visible public part of an object, we can freely modify object behaviour, just keeping invariant the interface itself.

3.4 Modularity

A program in *eta* is a library of class definitions, where a class is, as usual, a parametric definition of objects. At run time, new objects are activated simply instantiating the parameters of a class. This one-to-one relationship between system modules - the physical objects composing a system- and programming modules -class definition instances- lifts the modular structure of the multiple tuple spaces paradigm at the programming level, simplifying the programmers work.

Moreover, modularity provides, at system level, a framework to develop large systems exploiting the client-server architecture, and modularity at programming level, coupled with inheritance, naturally supports reuse.

3.5 Reactiveness

Threads connected to an object react to given state configurations by reading and consuming some of the contained atoms. Then they carry out an internal computation and produce an output.

Threads behaviour is described by rules: rules have a name and are composed of a guard, a body, and a postcondition.

name : *guard*
| *body*.
postcondition

The rule name is a term that uniquely identifies the rule within the object. The guard is a sequence of read (*Read-guard*) and input conditions (*In-guard*): the former is a compound goal on the object state, the latter defines the set of atoms to be removed from the state. A guard is satisfied when two conditions are satisfied: there exists a substitution that unifies a subset of the tuples contained in the object state with the rule guard; the atoms specified by the *In-guard* occur in the object state and can be removed. The *body* is an atomic Prolog goal, evaluated with respect to the Prolog program associated to the object (see Section 4). The *postcondition* is a list

of atoms to be written in the object state, or activation requests, or atoms to be sent to other objects. The commit operator '|' separates the precondition from the body. The commit operator has a semantics similar to that of classical logic concurrent languages. If the rule is fired, i.e. the commit is taken, then no backtracking is possible. This is often called don't care nondeterminism [14].

3.6 Declarativity

In *eta* the interaction between the state and the threads of an object is controlled by the unification algorithm. Unification enhances the expressive power of the language as complex conditions on the tuple space can be expressed in the rules in a declarative and concise manner.

Variables scope plays an important role in the language: variables appearing as object parameters bind the variables occurring in the rules and in the object interface. Viceversa, the scope of any variable appearing in a rule, and not in the class parameters, does not extend out of the rule itself. Finally, bindings in the Prolog program are limited to the single clauses, as usual.

3.7 Control

Path expressions have been introduced in Paté as a mechanism to constrain a wild non-determinism in rules activations [4]. In *eta* they are extended to specify critical regions as well as sequencing, alternation, and rule priority.

A path expression is similar to a regular expression. It specifies a set of paths over the alphabet of rule names extended with ϵ, the empty path expression and $\{\gamma_1, \ldots \gamma_n, \ldots\}^{\{+,-\}}$, special terms used to mark critical regions. According to this form of control, a rule can fire when both its firing satisfies the path expression and its guard is satisfied by the current state.

3.8 Parallelism

The language *eta* provides two grains of parallelism: fine grained parallelism among the threads of an object and coarse grained parallelism among the objects of an *eta* system. Each form of parallelism refers to a precise cooperation model: shared memory and message passing.

These *eta* features, both with respect to parallelism grain and to cooperation models, look like multitasking implementations commonly used in last generation operating systems, as lightweight and heavyweight processes in Mach [15]. In *eta* these features are independent from any language implementation: they are formalized by language constructs rather than achieved by system calls.

3.9 Reliability

Communication between objects can fail: the addressee object may not exist or it may not consider an incoming atom as a possible input. The dynamic characteristics

of *eta* make not possible to detect these failures at compile time: they are handled by the run time support of the language.

At the language level, it is possible to supply, in rule postconditions, default atoms to be written in the object state, in case of failure of the body evaluation; failure of the communication; failure in the activation of a new object.

3.10 Inheritance

In *eta* a class definition can inherit from another one, exploiting an *isa* declaration. We adopt an extensional mode of inheriting for the object interfaces, while we allow overriding for the object internal behaviour, uniformly with the encapsulation aspects of the language. An object can send and receive only the messages that belong to a superset of those that are sent and received by an object of the superclass, while the reactions to an identical message can be different.

4 Syntax and Semantics

In *eta* a program is a collection of object class definitions, where a class definition has the following structure:

Class	*class_name* [**isa** *class_name*]
input	$\{atom\}^+$
output	$\{free\ atom\}^+$
contents	$\{free\ atom\}^+$
initial	$\{atom\}^*$, $\{thread\}^+$
critical	$\{region\_name : \{atom\}^+\}^+$
thread	$\{thread\_name : path\_expression\}^+$
rules	$rule_1$
	$\vdots$
	$rule_n$
with	*prolog_program*
end	

In case of class definition by inheritance, all these parts are optional, while at least *thread*, *rules*, and one between *input*, *output*, and *contents* must be specified when defining an entirely new class.

4.1 Class_name

A class name is a term. The functor identifies the class, while the arguments are free variables that are instantiated when an object is activated. Variables appearing as parameters in a class name can bind variables appearing in all the parts but the Prolog program.

4.2 Input, output, contents

These parts specify the atoms that an object of the class can accept as input from other objects, contain during the execution, or send to other objects, respectively. Input, output, and contents of an object can be statically derived from rule definitions. However, we contend that their explicit declaration is a readable interface of the object itself. In *contents* we allow only free atoms, as we want to statically verify if they meet rule postconditions.

4.3 Initial

Initial specifies the sequence of atoms that represent the object initial state and the sequence of the initial threads , i.e. those activated with the object.

4.4 Critical

A critical region is identified by a name that ranges over the alphabet $\{\gamma_1, \ldots, \gamma_n, \ldots\}$ and specifies a set of atoms. Atoms specified in a critical region cannot be consumed by other threads than the one who entered the critical region itself.

4.5 Thread

A thread consists of a name and a path expression. The thread path expression specifies critical regions, sequencing, alternation, and rule priority, according to the syntax given below. We use three syntactic categories: *Exp*, *Rule*, and $\gamma_i^{\{+,-\}}$, where Rule denotes a rule name, *Exp* a path expression, and γ_i ranges over the alphabet of the critical region names.

$Exp ::$	$=$	ϵ	*empty*
	$\mid$	*Rule*	*rule name*
	$\mid$	*Exp Exp*	*sequencing*
	$\mid$	$Exp \mid Exp$	*choice*
	$\mid$	$Exp > Exp$	*priority*
	$\mid$	Exp^*	*iteration*
	$\mid$	$[Exp]$	*optionality*
	$\mid$	(Exp)	*precedence*
	$\mid$	$Rule\{Exp; Exp; Exp; Exp\}$	*failure*
	$\mid$	$\gamma_i^+\ Exp\ \gamma_i^-$	*critical region*

The grammar generates the set of path expressions over the alphabet of rule names extended with $\{\gamma_1, \ldots, \gamma_n, \ldots\}^{\{+,-\}}$ and ϵ, the empty path expression. A path expression is similar to a regular expression, with the following exceptions: the priority operator modifies choice allowing to express a priority between two alternative paths; the optionality construct is a shorthand for $Exp \mid \epsilon$; the failure operator takes into account the possibly different outcomes of a rule evaluation (success or failure of the body and of the communication).

Every rule name must appear at least in one of the threads of the object, and can appear more than once in each thread.

A thread terminates when it reaches the end of one of the (finite) paths described by its path expression. When all the threads of an object are terminated, the object itself terminates.

4.6 Rules

Rule syntax is indeed more complex than the one we have given in Paragraph 3.5, where we skipped many details, such as error handling and primitives. A rule takes the form:

name : *read_guard* {*in_guard*}
| *body*.
out
; **body** $fail_{body}$
data $fail_{data}$
target $fail_{target}$

Rule guard
Both *read_guard* and *in_guard* are sequences of atoms. *Read_guard* may contain primitives like $=$, $?=$, $\backslash=$, $==$, $<$, $>$, *is*, *var*, ... derived from Prolog, to express relations between the different facts composing a guard, and primitives of *eta*:

- *self(O)*, *father(O)*. Variable O is bound to the object name or to the name of the object creator, respectively.
- *not A* succeeds iff A does not unify with any atom of the state. In case of success, *not A* does not produce any substitution.
- *all(A, L)*, *all(A, L, G)*. In the first case L is bound to the list of all the atoms of the object state that unify with A, while in the second case atoms in L must also satisfy goal G.

The last two primitives can appear in an *In-guard* too. In this case the selected atoms are removed from the state. For guard evaluation we refer to [11].

Some syntactic sugar: in the precondition we use curly brackets to denote tuples removed (*In_guard*), in the postcondition the same brackets denote tuples to be written.

Rule body
Rule body is an atomic Prolog goal. It is instantiated by the substitution computed by guard evaluation and then evaluated with respect to the Prolog program accordingly to Prolog usual semantics. In case of success the first computed answer substitution binds the variables of the postcondition, otherwise no substitution is computed, the postcondition is skipped, and $fail_{body}$ is written in the object state.

Rule postcondition
Rule postcondition consists of an *out* followed by the (optional) parts *body*, *data*, *target* each of them containing an atom. The *out* is a sequence of:

- atoms to be written on the object state;
- expressions of the kind atom@target whose meaning is that the atom must be sent to object target;
- thread activations;
- expressions of the kind **new**(O) whose meaning is that a new object O must be activated;
- expressions of the kind list(L) or list(L)@target whose meaning is that the atoms contained in list L must be written in the object state or sent to object target, respectively.

The parts *body*, *data*, *target* specify the atoms to be written in the object state in case of failure of the body evaluation (*body*); in case of a communication failure, caused by an interface violation (*data*) or by the non existence of the target object (*target*); in case of failure of an object activation because of the non existence of a corresponding class (*target*).

4.7 Isa

When a class inherits from another one, say A *isa* B, the parts *input*, *output*, *contents* and *initial* of the two classes are disjoint, with the meaning that in A we simply extend B. On the contrary, we allow overriding of rules, threads, critical regions, and Prolog clauses (predicate definitions), i.e. we can redefine them in A.

5 Discussion

Object orientation usually implies a structure over program data. In logic languages, object orientation leads to a structured knowledge base [6, 13]. An *eta* object has a static knowledge base, which defines the semantics of the rule bodies. The tuple space contents, which is dynamic and open to external inputs, affects, with rule definitions, the reactive behaviour of an object. Both these aspects belong to the definition of an object and exploit the advantages of the object oriented model, in particular of the inheritance.

The way to access an object in *eta* is, as usual, by sending messages through a rigid interface. Messages are atoms that are added to the receiver object's state, that is shared among concurrent threads. As a consequence, we miss the usual and full correspondence between the object interface, i.e. the set of exported methods, and its functional behaviour. However, from a programming point of view it is straightforward to define objects with this feature and, from a semantic point of view, it is not difficult to explicit this correspondence in the general case.

6 Conclusions and Future Works

The *eta* implementation is designed to exploit a network of Unix workstations. This choice has been motivated by the great diffusion and the acceptable price of distributed environments of this kind.

The realization of the *eta* run time support will benefit from the experience gained during the implementation of Paté and is more than an affordable task. The approach is to compile *eta* objects in programs written in a object oriented language (actually C++). Each *eta* object runs as a multithreaded Unix process. Evaluation of Prolog goals, interprocess communication, and multithreading are supported by custom predefined libraries.

As extensions of the language, we plan to introduce multiple inheritance and a notion of *eta* system that allows to statically define and check cooperation properties among objects.

Acknowledgements

We gratefully thank Roberto Deias and Angela Discenza for various helpful discussions.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software Process Enactment in Oikos. In *Proc. ACM SIGSOFT 90, 4th Symposium on Software Development Environments*, 1990.

[2] V. Ambriola, G. A. Cignoni, and L. Semini. A Proposal to Merge Object Orientation, Logic Programming, and Multiple Tuple Spaces. In *Proc. of the ICLP'94 Postconference Workshop on Process-based Parallel Logic Programming*, June 1994.

[3] V. Ambriola and C. Montangero. Modeling the Software Development Process. In V. Ambriola and G. Tortora, editors, *Proc. Advances in Software Engineering and Knowledge Engineering*, pages 41–72, Singapore, 1993. World Scientific Publishing Company.

[4] V. Ambriola and L. Semini. Control Specification in Tuple Space Based Languages. Technical Report 19-93, Dipartimento di Informatica, Università di Pisa, 1993.

[5] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[6] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In A. Voronkov, editor, *Proc. First Russian Conference on Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 102–113. Springer-Verlag, Berlin, 1991.

[7] A. Bucci, P. Ciancarini, and C. Montangero. A Distributed Logic Language Based on Multiple Tuple Spaces. In *Proc. Logic Programming Conference*, Tokio, 1991.

[8] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 5(2):97–107, 1989.

[9] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–459, 1989.

[10] P. Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. In *Proc. Sixth IEEE International Workshop on Software Specification and Design*, 1991.

[11] M. Gaspari. A Shared Dataspace Language and its Compilation. Technical Report UBLCS-94-5, Comp. Science Laboratory, Università di Bologna, 1994.

[12] D. Gelernter. Multiple Tuple Spaces in Linda. In *Proc. Parle 89*, volume 365 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.

[13] L. Monteiro and A. Porto. Contextual Logic Programming. In G. Levi and M. Martelli, editors, *Proc. Sixth International Conference on Logic Programming*, pages 284–302. The MIT Press, Cambridge, Mass., 1989.

[14] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[15] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. V. Young. Mach Threads and the Unix Kernel: The Battle for Control. In *Proc. USENIX Association Conference*, pages 185–197, Phoenix, 1987.

Appendix

The example describes a zoo.

Animals (*bears*) growl when hungry, wait some time (*PatienceTime*), and then either some food has arrived or they growl once more. After eating, if the food was enough with respect to their diet (*to_eat*(0)), they yawn and sleep, otherwise, they growl to have some more food.

Wardens take care of animals, preparing food when they growl. The time they spend in this operation depends on their *Laziness* and reduces if a warning (*hurry_up*) is received.

The zoo_management controls the work of the wardens, engages new wardens if they are not enough, and buys new animals.

```
Class     bear(Name, Diet, SleepTime, PatienceTime, ZooMan, Warden)

input     {food(honey, Quantity)}
output    {growl(Name),
           yawn(Name),
           animal(Type, Name, Id)}
contents  {to_eat(Amount)}
initial   {to_eat(Diet)},
          {startup}
thread    startup : init
          behaviour : ((eat > growl)* sleep awake)*

rules
          init :    self(Id)
                    |
                    {animal(bear, Name, Id)}@ZooMan
                    {animal(bear, Name, Id)}@Warden
                    {behaviour}

          eat :     {food(_, Quantity), to_eat(Amount)}
                    |eat(Amount, Quantity, NewAmount).
                    {to_eat(NewAmount)}

          growl :   to_eat(Amount), Amount > 0
                    |wait(PatienceTime).
                    {growl(Name)}@ZooMan
                    {growl(Name)}@Warden

          sleep :   {to_eat(0)}
                    |
                    {yawn(Name)}@ZooMan

          awake :   |sleep(SleepTime).
                    {to_eat(Diet)}

with      eat(Amount, Quantity, NewAmount) : -
              Amount > Quantity,
              NewAmount is Amount - Quantity,
              suspend(Quantity).
          eat(Amount, Quantity, 0) : -
              Amount ≤ Quantity,
              suspend(Quantity).
          wait(PatienceTime) : -
              suspend(PatienceTime).
          sleep(SleepTime) : -
              suspend(SleepTime).
end
```



```
Class     normalbear(Name, Diet, SleepTime, PatienceTime, ZooMan) isa bear

input     {food(meat, Quantity), food(vegetables, Quantity)}
end


Class     toonbear(Name, Diet, SleepTime, PatienceTime, ZooMan) isa bear

input     {food(sandwich, Quantity), food(cake, Quantity)}
end


Class     warden(Name, Rest, Laziness, ZooMan)

input     {animal(AName, Id), growl(AName), hurry_up}
output    {food(Food, Quantity), warden(Name, Id, 0)}
contents  {food(Food, Quantity), dislike(AName, Food), lazy(CLaziness)}
initial   {lazy(Laziness), food(banana, 6), food(honey, 4),
           food(meat, 8), food(sandwich, 6),
           food(vegetables, 6), food(cake, 8)},
          {startup}
thread    startup : init
          task : (work > hurry_up > rest)*

rules
          init :    self(Id)
                    |
                    {warden(Name, Id, 0)}@ZooMan
                    {task}

          work :    {growl(AName)}
                    animal(AName, Id), food(Food, Quantity),
                    not dislike(AName, Food),
                    lazy(CLaziness),
                    |prepare(Food, Quantity, CLaziness).
                    food(Food, Quantity)@Id
                    ;
                    data {dislike(AName, Food)}

          hurry_up :  {hurry_up, lazy(CLaziness)}
                    | speed_up(CLaziness, NLaziness).
                    {lazy(NLaziness)}
                    ;
                    body{lazy(CLaziness)}

          rest :    |suspend(Rest).

with      prepare(_, Quantity, Laziness) : -
              Time is Quantity × Laziness, suspend(Time).

          speed_up(CLaziness, NLaziness) : -
              CLaziness > 1, NLaziness is CLaziness - 1.
end
```