avoid these disadvantages is the *demand driven strategy* presented in [11]. The idea is roughly as follows: instead of naively trying the defining rules in textual order and restarting the evaluation of argument expressions for each rule, they look for suitable argument expressions that can be evaluated first and then be used for all rules.

For this particular strategy and a concrete language, we outline an abstract machine design. It is based on the narrowing machine presented in [6, 9, 10, 12]. As a "natural" extension, we have modified and extended the original instruction set in order to be able to reflect the *demand driven strategy*. Local determinism is detected, leading to an early deletion of choice points that increases the space efficiency, in the line of [12]. An optimization based on ideas from [2] has been incorporated: instead of having static choice point assignment for rule selection, a choice point is dynamically created only when an unbound variable is matched, and then it controls all alternative bindings of that variable. Deterministic computations automatically avoid unnecessary choice point creation. A classification of the different functional, logic, and narrowing abstract machines is presented in [8], where it is also established that mixed paradigms admit efficient implementations.

The rest of this paper is organized as follows: a higher order functional logic language is presented in Section 2. Section 3 outlines the lazy narrowing abstract machine design. The demand driven strategy for lazy narrowing is presented in Section 4: each type of demand is illustrated by means of examples, as well as the compilation of programs and goals.

2 A Functional Logic Language

For our presentation we use a simple functional logic language (SFL for short) which is based on conditional rewrite rules and encompasses the expressive power of several more concrete languages, e.g. K-LEAF [3] and BABEL [13]. It is a higher order (similar to that presented in [5]), polymorphically typed language and uses lazy narrowing as evaluation mechanism. We assume expressions to be well typed w.r.t. types declared for constructors and function symbols. For simplicity, types will not be mentioned explicitly.

2.1 SFL Syntax

We assume a higher order signature $\langle DC, FS\rangle$ with the ranked alphabet $DC = \bigcup_{n\in\mathbb{N}} DC^n$ of *constructor symbols* and the disjoint ranked alphabet $FS = \bigcup_{n\in\mathbb{N}} FS^n$ of *function symbols*. Given a countably infinite set Var of *variables*, we distinguish the syntactic domains given in Figure 1.

As usual, we assume that application associates to the left and omit parenthesis accordingly. For function symbols $f \in FS^n$ we consider *defining rules*, which must be *left linear* conditional equations of the following form:

$$\underbrace{f\ p_1 \ldots p_n}_{lhs} = \underbrace{e}_{body} \underbrace{\Longleftarrow l_1{=}{=}r_1, \ldots, l_m{=}{=}r_m}_{optional\ condition}.$$

Patterns $p_1, p_2, \ldots \in Pat$:	p	::=	X	% $X \in Var$
		|	$c\ p_1 \ldots p_n$	% $c \in DC^n$
Terms $t_1, t_2, \ldots \in Term$:	t	::=	X	% $X \in Var$
		|	$c\ t_1 \ldots t_k$	% $c \in DC^n, 0 \le k \le n$
		|	$f\ t_1 \ldots t_k$	% $f \in FS^n, 0 \le k < n$
Expressions $e_1, e_2, \ldots \in Exp$:	e	::=	X	% $X \in Var$
		|	c	% $c \in DC^n, n \ge 0$
		|	f	% $f \in FS^n, n \ge 0$
		|	$(e_1\ e_2)$	% *Application*

Figure 1: Syntactic Domains

where $p_i \in Pat$ $(1 \le i \le n)$ and $e, l_i, r_i \in Exp$ $(1 \le i \le m)$. The body e is any expression such that $vars(e) \subseteq vars(f\ p_1 \ldots p_n)$. Operationally, such equations will be used as *conditional rewrite rules*. The sign '==' in conditions stands for *strict equality*, meaning that a condition "$l_i{=}{=}r_i$" must be satisfied by narrowing l_i, r_i into unifiable terms. The *condition* may contain extra variables that do not occur on the left hand side of the rule.

An *SFL-program* consists of a finite set of defining rules for the function symbols in FS, satisfying a non-ambiguity condition to avoid semantic overlapping between rules (*termination* is not required). For a complete description of the non-ambiguity condition, see [4, 13].

Goals for SFL-programs are exactly as rule conditions. They are solved by narrowing. The evaluation of an expression e to yield a value can be triggered by a goal "$\Longleftarrow e == R$", being R a new variable.

Higher order logic variables are not allowed (presently), i.e. higher order variables may occur in the left hand side of rules, but are forbidden to occur as extra variables in conditions, or in goals.

Example 1 Let $CS^0 = \{true, false, 0, [\,]\}$, $CS^1 = \{s\}$, $CS^2 = \{[.|.]\}$ and $FS^2 = \{member, funor, sum, map\}$. A legal SFL-program is given by the following defining rules:

$$member\ X\ [Y|Ys] = true \Longleftarrow X == Y. \qquad (MEMBER_1)$$
$$member\ X\ [Y|Ys] = true \Longleftarrow member\ X\ Ys == true. \qquad (MEMBER_2)$$

$$map\ F\ [\,] = [\,]. \qquad (MAP_1)$$
$$map\ F\ [X|Xs] = [(F\ X)|(map\ F\ Xs)]. \qquad (MAP_2)$$

$$sum\ 0\ Y = Y. \qquad (SUM_1)$$
$$sum\ (s\ X)\ Y = (s\ (sum\ X\ Y)). \qquad (SUM_2)$$

$$
\begin{array}{llllll}
funor & true & X & = & true. & (FUNOR_1) \\
funor & false & false & = & false. & (FUNOR_2) \\
funor & X & true & = & true. & (FUNOR_3)
\end{array}
$$

A goal for this example is

$$\Longleftarrow funor\ (member\ X\ (map\ (sum\ X)\ [0]))\ false == true.$$

for which we may expect $\{X \backslash 0\}$ as the first computed answer. In the rest of the paper we refer to this program as "the running example".

3 A Lazy Narrowing Abstract Machine

The design is based on that presented by Loogen in [9, 10]. The underlying narrowing machine is a combination of a particular reduction machine with mechanisms for unification and backtracking based on the Warren's Abstract Machine ([1, 15]). The set of machine instructions has been modified and extended to be able to incorporate the demand driven strategy for lazy narrowing.

3.1 Components

Basically, the machine architecture is the same described in [9]. Hence, we make a brief description of its components.

The **program store** (ps) is a code area where the abstract machine code corresponding to a program is stored. Data representation is managed via a graph or a **heap** (hp) structure. Rather than using argument registers as it is done in the WAM, the **data stack** (ds) allows data manipulation: it stores heap addresses corresponding to arguments or to results of function calls. The **control stack** (st) is the central component of the machine, and it is structured as a double linked list consisting of environments and choice points. Actual arguments and local variables for a function call are stored into an *environment* frame. Also stored are the previous environment pointer and the return address (to where it will return after a successful evaluation of the function call). In case of backtracking, the necessary information to restore the state of the machine is saved into a *choice point* frame. A detailed description of these frames structure is given below.

Environment Frame		*Choice Point Frame*	
n	arity	tds	top of data stack
A1	1st argument	nds	number of saved data stack entries
	...	sd_1	1st saved data stack entry
An	n-th argument		...
k	number of local variables	sd_{nds}	nds-th saved data stack entry
lv_1	1st local variable	tt	top of trail
	...	shp	saved heap pointer
lv_k	k-th local variable	sbp	previous choice point pointer
sep	previous environment pointer	badr	backtrack address
ra	return address		

Control Stack Frames

The **trail** (tr) is used to keep note not only of variable heap addresses to undo bindings, but also to keep note of suspension nodes to undo updatings, in order to be able to backtrack without loosing information. Some pointers are provided to facilitate the access to the machine components: IP points to the next instruction in the program store to be executed, HP points to the next free position in the heap, EP points to the current environment allocated in the control stack, BP points to the topmost choice point in the control stack. The control stack length is determined by $max\{\texttt{EP}, \texttt{BP}\} = lg(\texttt{st})$.

Data are represented by means of *heap nodes*. We have extended the *constructor nodes* representation, in order to admit partial applications. We have also added a natural specialization of constructor nodes: *constant nodes*.

Variable nodes <VAR,a> representing a logical variable bound to the heap address a, while <VAR,?> represents an unbound variable.

Constructor nodes <CONSTR,c,a_1:...:a_m,n-m> with $c \in DC^n$, $n > 0$, a_i $(1 \leq i \leq m)$ being the heap addresses of the first m arguments and n-m ≥ 0 being the number of remaining arguments to obtain a totally applied constructor.

Constant nodes <CST,c> with $c \in DC^0$ represents a constant, i.e. a constructor of arity zero.

Function nodes <FUN,f,a_1:...:a_l,n-l,k> with $f \in DF^n$, $k \in \mathbb{N}$ representing the number of local variables, a_i $(1 \leq i \leq l)$ being the heap addresses of the first l arguments and n-l ≥ 1 being the number of missing arguments to obtain a totally applied function. Function nodes represent function partial applications.

Suspension nodes <SUSP,ca,lv_1:...:lv_k,rs>. They contain the suspension code address ca , the environment lv_i $(1 \leq i \leq k)$ needed during its code execution, and the place to keep note of the result heap address rs after a successful evaluation. They are necessary to represent delayed evaluations.

Hole nodes <HOLE> are used during unification to keep place for the remaining arguments, and as a flag while evaluating conditions.

The state of the machine will always be given by a tuple of the form:

(IP, hp, HP, ds, st, EP, BP, tr).

The transition to the *next state* is determined by the instruction pointed by IP.

3.2 Machine Instructions

The main modifications performed to the machine reside in the set of machine instructions (as well as in the compilation schemes). Due to the limited space in the present paper, their specification has been omitted. Instead, we have chosen to devote our attention to the place where the strategy is reflected: the compilation of programs.

4 The Demand Driven Strategy

We take here the *demand driven strategy* (DDS, for short) presented in [11]. The idea is roughly as follows: instead of trying the defining rules in textual order and restarting the evaluation of argument expressions for each rule, they search for suitable argument expressions that can be evaluated first and then be used for all rules. The authors specified DDS as a Prolog translation, whereas we perform a translation to machine code.

4.1 Preliminaries

Definition 1 A *call pattern* is any linear expression of the form $f\ p_1 \ldots p_n$, where f is a function symbol and p_i are patterns. A *generic call pattern* is any call pattern of the form $f\ X_1\ \ldots\ X_n$, where X_i are n different variables.

Let π be a call pattern and let l be the *lhs* of a defining rule. We say that l *matches* π iff l is an instance of π via some (necessarily linear) pattern-substitution. Moreover, l is a *variant* of π iff this pattern-substitution is a variable renaming.

Let $vpos(\pi)$ and $cpos(\pi)$ denote the set of *variable and constructor positions* in a call pattern π, respectively. Let π be a call pattern which is matched by the *lhs* of at least one defining rule in a given SFL-program P. Let $\mathcal{R}$ be the set of rules from P whose *lhs* match π. Let $lhs(\mathcal{R})$ be the set of all *lhs* of the rules from $\mathcal{R}$. Let u belong to $vpos(\pi)$. We say that:

1. u is *demanded by the lhs* l iff l has a constructor at position u.
2. u is *demanded by* $\mathcal{R}$ iff u is demanded by some l in $lhs(\mathcal{R})$.
3. u is *uniformly demanded by* $\mathcal{R}$ iff u is demanded by every l in $lhs(\mathcal{R})$.
4. u is *demanded with priority by* $\mathcal{R}$ iff u is uniformly demanded and the same constructor symbol appears at position u in all l in $lhs(\mathcal{R})$.

We make an additional distinction w.r.t. [11] that is well-suited for our purposes: the concept of *priority demand*. It reflects the need of evaluating the expression appearing at this position to a term headed by the (same) constructor symbol demanded by all the rules. With a position *uniformly demanded*, we don't know a priori which constructor symbol will head the resulting term.

4.2 Program Compilation

In order to handle potentially infinite data structures, the evaluation to *head normal form* (*hnf* for short) is used: only those subexpressions necessary to decide a unification will be evaluated. When something is demanded by a group of rules, we need to ensure that the evaluation has been performed in order to proceed with the unification. As it can be a suspended form, we have to initiate its evaluation to *hnf*.


```
member X1 X2 =
  let X2 = [X2.1|X2.2] in
    try
      true <== X1 == X2.1
    □ true <== member X1 X2.2 == true
    endtry
  endlet

map X1 X2 =
  case X2 of
    []          → []
  □ [X2.1|X2.2] → [(X1 X2.1)|(map X1 X2.2)]
  endcase

sum X1 X2 =
  case X1 of
    0          → X2
  □ (s X1.1)   → (s (sum X1.1 X2))
  endcase
```

```
funor X1 X2 =
  or
    case X1 of
      true  → true
    □ false → let X2 = false in
                false
              endlet
    endcase
  □ let X2 = true in
      true
    endlet
  endor
```

Figure 2: Example Intermediate Transformation

Given an SFL-program P, the code generation is done separately for each function symbol defined by the program. We start observing the kind of demand determined by the function defining rules, and guided by it we produce the corresponding machine code. The Figure 2 shows an intermediate transformation of our running example that might help to understand the process. The notation used in this figure reflects the demand imposed by the rules as follows:

DEMAND		SYMBOL
u is demanded with priority	.....	let $X_u = \ldots$
u is uniformly demanded	.....	case $X_u \ldots$
u is demanded	.....	or ...
u is not demanded	.....	try ...

Some position is demanded with priority
We start initiating evaluation at these positions. The idea is to give priority to the evaluation of everything forced to have the same 'shape'. This is the case for the second argument of *member* in our running example. It is clear that regardless of the goal's form, both rules will require evaluation of the second argument to a term headed by the constructor associated to the non-empty list (or to a variable). Hence, before deciding which rule is to be applied, we initiate the evaluation to *hnf* of all those subexpressions appearing at *demanded with priority* positions.

The first instructions generated for *member* are:

```
LOAD 1          POP-ARGS
MATCHVAR 1      LOAD 1
LOAD 2          INITIATE 2
MATCHVAR 2      RESET-ENV(2,3) ...
```

This group of instructions performs the task of leaving an environment frame with the second argument evaluated to *hnf*. At this point, code generation will continue. There are no more demanded positions. We will see later how to manage this situation.

Some position is uniformly demanded but without priority

As above, we initiate evaluating to *hnf* the subexpression appearing at the leftmost such a position and we perform a case distinction on the result. In our running example, *sum* demands uniformly position 1. Let us reproduce its first lines of code:

```
LOAD 1                IF-VAR(1, cp1)      cp1 :  TRY-ME-ELSE cp2
MATCHVAR 1            CHECK(1, 0, l1)      l1 :  code for (SUM1)
LOAD 2                CHECK(1, s, l2)     cp2 :  TRUST-ME-ELSE-FAIL
MATCHVAR 2                                 l2 :  code for (SUM2)
POP-ARGS
INITIATE 1
LOAD 2
RESET-ENV(2,2)
```

The first argument has to be initiated. Afterwards a choice point is dynamically created only if the result is a variable. Otherwise, we jump to the corresponding instructions. Observe that function *map* is exactly in the same situation, with the only difference that it is the second argument which appears at a *uniformly demanded* position. Hence, similar code will be generated for it.

Some position is demanded, but no one is uniformly demanded

The idea is to split the rules into the ones demanding this position and the rest of them. The structure that will be generated is:

```
          TRY-ME-ELSE alt
          Demanding rules
alt :     TRUST-ME-ELSE-FAIL
          Non-demanding rules
```

In our running example, *funor* corresponds to this situation. The first argument is demanded by the first two rules but not by the third one. A choice point is necessary. As it can be seen in Figure 2, the splitting yields to consider the first two rules separately from the third. Now, the first argument appears at a *uniformly demanded* position w.r.t. the first group, and the second one has the second argument at a *demanded with priority* position. The complete code for *funor* is:


```
TRY-ME-ELSE alt      cp1 :  TRY-ME-ELSE cp2      alt :  TRUST-ME-ELSE-FAIL
LOAD 1                l1 :  LOAD 1                      LOAD 1
MATCHVAR 1                  MATCHCST true               MATCHVAR 1
LOAD 2                      LOAD 2                      LOAD 2
MATCHVAR 2                  MATCHVAR 2                  MATCHVAR 2
POP-ARGS                    POP-ARGS                    POP-ARGS
INITIATE 1                  CNODE true                  LOAD 1
LOAD 2                      RETURN                      INITIATE 2
RESET-ENV(2,2)       cp2 :  TRUST-ME-ELSE-FAIL          RESET-ENV(2,2)
IF-VAR(1,cp1)         l2 :  LOAD 1                      LOAD 1
CHECK(1,true,l1)            MATCHCST false              MATCHVAR 1
CHECK(1,false,l2)           LOAD 2                      LOAD 2
                            MATCHVAR 2                  MATCHCST true
                            POP-ARGS                    POP-ARGS
                            LOAD 1                      CNODE true
                            INITIATE 2                  RETURN
                            RESET-ENV(2,2)
                            LOAD 1
                            MATCHCST false
                            LOAD 2
                            MATCHCST false
                            POP-ARGS
                            CNODE false
                            RETURN
```

No position is demanded

• If there is only one rule, one only needs to apply it. Being the code for rule (SUM_1) easier to generate, we include the instructions for rule (SUM_2):

```
          LOAD 1                % Left hand side:
          MATCHCONSTR(s,1)      % sum (s
          MATCHVAR 1.1          %       X)
          LOAD 2
          MATCHVAR 2            %          Y =
          POP-ARGS              % Right hand side:
          SUSPEND lab           %    suspension
          LOAD 2                %    Y
          LOAD 1.1              %    X
          FNODE(sum,2,2,0)      %    sum
          APPLY                 %    (sum X)
          APPLY                 %    ((sum X) Y)
          UPDATE                % Susp := ((sum X) Y)
lab:      NODE(s,1,0)           % s
          APPLY                 % (s Susp)
          RETURN                % .
```

• In case of having at least two rules, their (common) *lhs* has been processed as a common part for all the rules (as we did above for *member*). This case is identified

in Figure 2 by "try". Here, the new feature is the way conditions are translated. We try to satisfy the first rule condition. If we obtain a success, the result is *true* (given by the first rule); if not, we try to satisfy the second rule condition and so on. If both conditions are not satisfiable, the rules are not applicable and a failure has to be produced.

In our running example, *member* corresponds to this last situation; the remaining code for it is:

```
        LOAD 1                  % Common left hand side:
        MATCHVAR 1
        LOAD 2
        MATCHCONSTR([·|·],2)
        MATCHVAR 2.1
        MATCHVAR 2.2
        POP-ARGS                % member X1 [X2.1|X2.2] =
        TRY-ME-ELSE rul2        % Try first rule condition:
        INIT-GVARS(3,0)
        INITIATE 1
        INITIATE 2.1
        CALL-EQ                 % <== X1 == X2.1
        JUMP-TRUE res1
        FAIL
res1:   CNODE true              % First rule result
        RETURN
rul2:   TRUST-ME-ELSE-FAIL      % Try second rule condition:
        INIT-GVARS(3,0)
        LOAD 2.2
        LOAD 1
        FNODE(member,2,3,0)
        APPLY
        APPLY                   % <== member X1 X2.2
        JUMP-TRUE res2          % == true
        FAIL
res2:   CNODE true              % Second rule result
        RETURN
```

Note: Due to the lack of space in the present paper, it has been impossible to detail the machine instructions specification as well as other issues. The interested reader may find them in [14], which will be sent on request.

4.3 Goal Compilation

The code generated when our goal

$$\Longleftarrow funor\ (member\ X\ (map\ (sum\ X)\ [0]))\ false == true.$$

is compiled is:


```
        CNODE false             % false
        SUSPEND s1              % susp1
        SUSPEND s2              % susp2
        CNODE 0
        CNODE []
        NODE([·|·],2,2)         % [0]
        LOAD 1                  % X
        FNODE(sum,2,2,1)        % sum X
        FNODE(map,2,3,0)        % map
        APPLY                   % map (sum X)
        APPLY                   % map (sum X) [0]
        UPDATE                  % susp2 := map (sum X) [0]
s2 :    LOAD 1                  % X
        FNODE(member,2,3,0)     % member
        APPLY                   % member X
        APPLY                   % member X susp2
        UPDATE                  % susp1 := member X susp2
s1 :    FNODE(funor,2,2,0)      % funor
        APPLY                   % funor susp1
        APPLY                   % funor susp1 false
        JUMP-TRUE end           % == true
        FAIL
end :   CNODE true              % Result: true (identifying a success)
        RETURN                  % End of the goal's execution
```

The goal execution begins with $funor\ susp_1\ false$, which represents a call to *funor* with a suspended form as first argument and the constant $false$ as second argument. The only applicable rule is $(FUNOR_1)$. Following its code, we need to initiate evaluation to *hnf* of the first argument ($susp_1$). This requires a call to *member*, which needs evaluating its second argument ($susp_2$) to hnf. This suspended form represents a delayed call to *map*. The only applicable rule is (MAP_2). The result is $[(sum\ X\ 0)|(map\ (sum\ X)\ [])]$. Having initiated this argument, we try to apply the first rule of *member*: after evaluating $(sum\ X\ 0)$ it binds X to 0 and returns the result *true*. Hence, we follow with the application of $(FUNOR_1)$, which also returns *true*. This is equal to *true* and goal evaluation has finished with the binding $\{X\backslash 0\}$ as *computed answer*. Backtracking would offer other alternative *computed answers*.

5 Conclusions and Future Work

In this paper our aim has been to illustrate the incorporation of the demand driven strategy into the abstract machine design. The new abstract machine supports an extension of the original language, and its set of instructions has been modified and extended. The strategy is reflected in the sequence of machine instructions generated by the compilation of a program.

We are currently engaged in the implementation of this work. The incorporation of some optimizations in the machine architecture as well as in the code generation is also planned for the future.

6 Acknowledgements

I am grateful to Mario Rodríguez Artalejo and to Teresa Hortalá González for many valuable discussions and for giving me the chance to participate in their research team.

References

[1] H. Aït Kaci: *Warren's Abstract Machine, A Tutorial Reconstruction*, Logic Programming Series, The MIT Press 1991.

[2] M.M.T. Chakravarty, H.C.R. Lock: *The implementation of Lazy Narrowing*, Proceedings PLILP'91, LNCS 528, Springer Verlag 1991, 123–134.

[3] 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.

[4] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*, Proceedings CSL'92, LNCS 702, Springer Verlag 1993, 216–230.

[5] J.C. González-Moreno: *Programación Lógica de Orden Superior con Combinadores*, PhD Thesis, Dpto. de Informática y Automática, Universidad Complutense de Madrid (UCM), Spain, 1994.

[6] W. Hans, R. Loogen, S. Winkler: *On the Interaction of Lazy Evaluation and Backtracking*, Proceedings PLILP'92, LNCS 631, Springer Verlag 1992, 355–369.

[7] M. Hanus: *The Integration of Functions into Logic Programming: A Survey*, Journal of Logic Programming Vols. 19–20, Special Issue: "Ten Years of Logic Programming", 1994, 583–628.

[8] H.C.R. Lock: *The implementation of Functional Logic Programming Languages*, PhD Thesis, Technical University of Berlin, 1992. Also available as GMD Report 208, Oldenbourg Verlag, München.

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

[10] R. Loogen: *Relating the Implementations Techniques of Functional and Functional Logic Languages*, New Generation Computing, Vol. 11, 1993, 179–215.

[11] R. Loogen, F. J. López-Fraguas, M. Rodríguez-Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Proceedings PLILP'93, LNCS 714, Springer Verlag 1993.

[12] R. Loogen, S. Winkler: *Dynamic Detection of Determinism in Functional Logic Languages*, Proceedings PLILP'91, LNCS 528, Springer Verlag 1991, 335–346.

[13] 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.

[14] E. Ullán-Hernández: *A Lazy Narrowing Abstract Machine*, Technical Report DIA 3-95, Dpto. de Informática y Automática, UCM, Spain.

[15] D.H.D. Warren: *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International Artificial Intelligence Center, October 1983.

Exploiting Expression- and Or-Parallelism for a Functional Logic Language

W. Hans, St. Winkler*

RWTH Aachen, Lehrstuhl für Informatik II, D-52056 Aachen, Germany
e-mail: {hans,winkler}@zeus.informatik.rwth-aachen.de

F. Sáenz*

Universidad Complutense de Madrid, Dept. Informática y Automática
E-28040 Madrid, Spain, e-mail: fernan@eucmvx.sim.ucm.es

Abstract

We discuss a parallel implementation for a functional logic language which exploits or- and transparent expression-parallelism. Or-parallelism is well-known from the context of logic programming, whereas the so-called expression-parallelism is a natural extension of and-parallelism to functional logic languages. The proposed model aims primarily at distributed architectures but runs on both shared and distributed memory models. We present some results for a scalable architecture with up to 64 processor elements with distributed memory as well as for a bus-based system.

Keywords: Functional Logic Programming, Expression Parallelism, Or-Parallelism

1 Introduction

Declarative programming languages offer a high degree of (implicit) parallelism. Functional logic languages as Babel are instances of the declarative programming paradigm. They extend functional programming languages with principles taken from logic programming. Apart from the unification parallelism, which can be integrated into a finer layer within the unification process, expression-parallelism and or-parallelism seem to be worthwhile to be exploited transparently on parallel architectures. During the last years numerous approaches have been proposed for exploiting parallelism in logic programs. Most of them rely on a special hardware platform, such as shared memory multiprocessors or switch-based machines [1, 3, 7, 18, 19]. It has been shown that almost linear speed-ups can be obtained on these machines,

*This work was supported by the Spanish PRONTIC project TIC92-0793-C02-01 and by the German DFG-grant In 20/6-1.

and even super-linear speed-ups when failing computations are early detected. But their efficiency collapse when the shared memory model is simulated at low level in a distributed environment.

As far as parallel systems with loosely-coupled memories are concerned, often stack-copying and re-computation models for or-parallelism are cited in the literature [2, 8]. These models rely on the transfer of the complete machine state, but have the disadvantage that sophisticated schedulers have to be introduced in order to minimize the copying and the re-computation costs. In our model we follow a process model in which each processor can be seen as an interpreter. Each processor has to execute a certain piece of the overall computation, which has been delegated by a parent processor.

The process model fits naturally to expression-parallelism, too. The progressive approach of [7] implementing independent and-parallelism is intended for a shared memory architecture and adopts the WAM [17] in an efficient way. Later, [16] proposed an extension of this approach for a distributed machine. But their proposal lacks in any capabilities of multiprocessing which yield severe restrictions concerning the abilities for goal scheduling. Our model integrates multiprocessing that seems to be a crucial feature, when dealing with scalable architectures with many processors.

Furthermore, the functional aspects of Babel offer more opportunities for parallelization. Babel programs are applied with several syntactic restrictions that ensure some determinative characteristics. The compositional style of programming allows a natural encoding of the problem and yields a clear data-flow. In contrast, the predicative style (e.g. of Prolog) demands the introduction of further variables and artificial flattening to express the data-flow. This hampers the automatic detection of the intended data-flow. Generally, the functional style allows a better abstract mimicry of the concrete computation, a better determinism detection, and a more powerful parallelization.

The organization of the paper is the following. The next section gives a brief introduction of the functional logic programming language Babel. Section 3 and 4 describe the two kinds of parallelism and explain how expression-parallelism is derived and exploited. The latter section explains how to reduce the amount of synchronizing constrains and to increase the process granularity. Section 5 presents the extensions of a stack-based sequential abstract machine to a parallel one that also fits for a distributed memory model. The runtime results, which are given in Section 6, show the behaviour of the abstract machine on a distributed system.

2 Babel

Babel is a functional logic language with a constructor discipline and a polymorphic type system. It has a functional syntax and uses narrowing for evaluation. For the sake of clarity, we will consider the first order subset of Babel with the leftmost innermost narrowing strategy applied.

A Babel program consists of a finite set of function definitions and can be queried

with a goal expression. Each function f is a finite sequence of defining rules, where each rule has the form:

$$\underbrace{f(t_1,\ldots,t_n)}_{\text{left hand side } (lhs)} \quad := \quad \underbrace{\underbrace{\{B \rightarrow\}}_{\text{optional guard}} \quad \underbrace{M}_{\text{body}}}_{\text{right hand side } (rhs)} .$$

with a Boolean expression B, and an arbitrary expression M. Babel functions represent mathematical functions, i.e. for each tuple of (ground) arguments, there is at most one result. This is ensured by special syntactic restrictions [12].
Terms t and Expressions M are defined as follows:

t	$::= X$	% variable
	$\mid c(t_1 \ldots t_n)$	% data constructor
M	$::= X$	% variable
	$\mid \varphi(M_1,\ldots,M_n)$	% φ is a n-ary function or constructor symbol
	$\mid B \rightarrow M_1\{\square M_2\}$	% if B then M_1 else undefined $\{$ else $M_2\}$

The operational semantics of Babel is based on narrowing [14].

3 Expression-Parallelism

Independent expression-parallelism consists of the evaluation of expressions in parallel if they are independent and improves the evaluation along one computation path. It is well suited to speed up complex deterministic computations. Independence means the lack of any shared unbound variables that may lead to binding conflicts when at least two computations instantiate them in an incompatible manner. One sufficient condition for the independence of two expressions e and e′ is $\mathrm{var}(e) \cap \mathrm{var}(e') = \emptyset$ and their lexicographical independence, i.e. neither is a subexpression of the other expression. Though the maximal parallelism under expression-parallelism could be exploited it is not worth because of the overhead due to the needed dynamic scheduling, being natural to impose several restrictions [7].

The independence condition allows the semi-intelligent search space pruning if one sibling of the parallel evaluation fails. As this failure can not be influenced by some other siblings, their computation can be reset.

We have developed an automatic parallelization tool which generates CGEs (Conditional Graph Expressions). CGEs are used to express parallelization conditions for independent expression-parallelism. CGEs involve dynamic tests at runtime that are desirable to reduce. The first stage of the tool covers this dynamic test reduction by using abstract interpretation techniques to infer information about independence, i.e., the modes of functions and sharing information of the variables in the rules [6]. Modes describe whether some arguments of functions are used ground as input arguments or unbound as output arguments. Sharing describe potential dependencies between the variables appearing in the rules.

A program transformer is eventually supplied in a second stage of the parallelizing tool with the independence information in order to express the implicit

parallelism by special parallel annotations. For instance, the expression e containing two subexpressions e′ and e″ is transformed into the following expression "letpar X = e′, Y = e″ in e", provided the analyzer determines the independence of e′ and e″. The evaluation of e is delayed and synchronized with the successful termination of both parallel computations. We have added annotations for the parallel non-strict boolean connectives to Babel, i.e. conjunction and disjunction, with their own adjusted semantics. The parallel system can execute the operands of these connectives in parallel, speculatively.

In a final stage, the parallelizing tool translates the annotated program into a sequence of abstract machine instructions.

Detection of deterministic computations is done both statically and dynamically. The former when inferring ground instantiations of function calls and the latter because the so-called dynamic cut [11] is embodied in the parallel system.

4 Or-Parallelism

Or-parallelism consists of the parallel evaluation of the rules whose left hand sides unify with a given function application. According to the corresponding rules, several branches of the search tree can be explored in parallel. Unlike a sequential execution of the program, the or-parallel execution causes binding conflicts due to multiple bindings within the different branches of the search tree. This kind of parallelism seems to be well suited for that class of functional logic programs which includes non-deterministic computations.

Process-based Or-Parallelism

In our model we follow a modified approach of the process-based or-parallelism [5]. Herein, or-processes are generated for each applicable program rule. The process execution takes place after the set of applicable rules is fixed by a complete indexing procedure [11]. This guarantees that the unification can be performed at low costs by the parent process so that each or-process is merely responsible for computing the rule body and for delivering the solutions with the answer substitution.

Process execution takes place in three phases. First, argument terms are copied into the initial environment of a new process replacing all global variables by local copies whose identification is noted in a special import list. Second, during the evaluation of the rule's body, the model is not faced with the binding conflicts since all data accesses occur on local copies. And finally, the execution succeeds with a backward synchronization with the parent by returning the solution and the answer substitution which can be derived from the import list and the local variable bindings.

Communication between processes takes place via messages. After their successful computation the results are embedded in a success message and sent back to the parent process. If the parent requests further solutions, it simply sends a backtrack

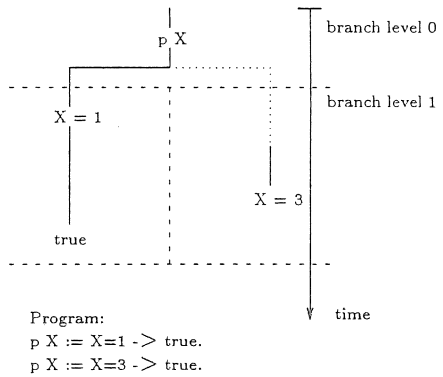


Figure 1: Retroactive parallelism example where the first branch is embedded within the parent process: The global variable X is bound to 1 with time stamp 1. After the unification within the first branch, the second branch is picked up by an idle processor . Nevertheless, X is treated as a free variable because the branch originates with level 0.

message to the child. This procedure can be repeated until the search space of the child is exhausted and the process terminates with a failure message.

This concept with the parallel execution of or-processes on different processor elements makes the process model well suited for distributed memory machines. Nevertheless, the process model has several drawbacks:

- There exist synchronization constraints concerning the process scheduling. Ideally, idle processors run the processes. But if there are not enough processors available, the processes are executed locally. On the other hand, the process model and the binding problem prevent a more flexible control regime.
- Process generation is expensive. Especially the local computation enforces too much overhead relative to the sequential computation. At least the expensive argument copying needs to be avoided by embedding the local computation of some branches within the process that created the branch point.

In order to solve the abovementioned problems we have extended the simple process model by introducing *retroactive parallelism* and *process inheritance*.

Retroactive Parallelism

Time stamps associated with each variable binding avoid the synchronization constraints. This technique has been borrowed from other models [19] where a special counter is incremented with each branch point allocation. In order to infer the original argument structure, the time stamp of the dereferenced variable is compared to

the branch level counter of the corresponding branch point. This mechanism enables retroactive parallelism and process generation on demand.

With the help of this scheme it is possible to integrate branches directly into the execution of the process which has created the branch point while the computation of the other branches is still suspended. More importantly, the process generation takes place only by incoming work requests of other processors on demand. Otherwise the original process is allowed to treat the branch point like an usual choice point without causing additional overhead.

Furthermore we are now in the position to renounce on the complete data transfer to the remote processors. The motivation is that the transfer of the complete data structures cannot be justified if a certain amount of the structure is not required by the child processor. [16] shows a detailed analysis of data locality.

Process Inheritance

Whenever a process can be executed locally, the sole computation of the rule body would be too restrictive. Therefore we allow a process to inherit the context of its parent process and all other former local ancestors. Now we can omit the backward synchronization with the parent process. The child shares all relevant data structures with its ancestors and performs an alternative search concurrently to the local ancestors. In other words this means a step toward the subtree-based approach of or-parallelism. The advantages for this modification are:

- We can avoid the direct synchronization with the local ancestors. The or-process is responsible to commit all these steps on its own.
- The process granularity grows when the exploitation of the whole subtree can be performed. Now remote processors can concentrate on that branches, which have probably enough work. Furthermore this makes the scheme even attractive for shared memory architectures where all relevant data are held in the global memory space.
- All data are accessible by the children and the original argument structure can be inferred by the use of the time stamp technique avoiding copying costs.

The memory space of a process can be divided into two parts. A private part represents the local search space of the process. The process is allowed to commit bindings for each variable originating from that part whereas the global part is write protected. The process has to record each instantiation of a 'global' variable in its own import list. Since each variable of the global part might be bound during the execution, the import list is implemented as a hash window which can be seen as an optimized version of a binding list. Clearly, the process also inherits the bindings of its ancestors. Instead of copying all the former bindings in its hash window, the hash window is simply linked into the chain of its forerunners via pointers. As a consequence, the variable dereferencing operation might become a non-constant time operation, but in practice the additional overhead seems to be low.

Scheduling Work

Sometimes the execution of only the right hand side of a rule on a remote processor may have fine granularity. Therefore a processor is allowed to suspend certain program branches with coarse granularity and keep this work available for remote processor which resume the corresponding computation later on. So a processor can provide enough work for other processors and might pick up other work locally at some other branch points.

Combining Expression- and Or-parallelism

During the evaluation process expression- and or-parallelism might appear. In principle the behaviour of both types of processes is equal. They work on local copies of the data in case of remote computation. Local expression-children are allowed to perform their private variable bindings directly and might bind variables from the shared part likewise in their own hash windows. On the other hand or-processes are still allowed to inherit the context of its ancestors.

When or-parallelism under expression-parallelism comes into play, there is the risk of recomputing some branches. Therefore a cross product node can be introduced that combines all solutions from the lower branches in order to prevent their recomputation. This is also the common technique of other systems that combine these two parallelism paradigms. [19, 2].

5 The Parallel System

We have designed a parallel system which exploits expression-or-parallelism in a distributed environment. Each node in the system is an abstract machine which runs the compiled code. The coarse structure of each abstract machine is sketched in figure 2. This figure shows primarily the partition into a communication unit and a narrowing unit. The former unit maps the logical complete connected topology onto the physical topology. It performs the necessary routing tasks statically if two communicating processors are only indirectly connected.

All the other activities of the abstract machine are performed by the narrowing unit. This includes multiprocessing as well as the message passing.

Meshed Stack Architecture

The narrowing unit consists of different memory areas. Among them, the graph stores the data structures and the runtime stack stores the control structures. Both structures are known from the sequential stack based abstract machine for Babel [10] which in turn inherits the concepts behind the WAM [17]. The representation of the runtime stack is slightly extended towards a meshed stack organization in order to support multiprocessing. Several processes may share the stack resource i.e., they allocate new entries on top of the stack, even if they do this in an interleaved fashion. Storage reclamation is performed by a garbage collector. But immediate storage

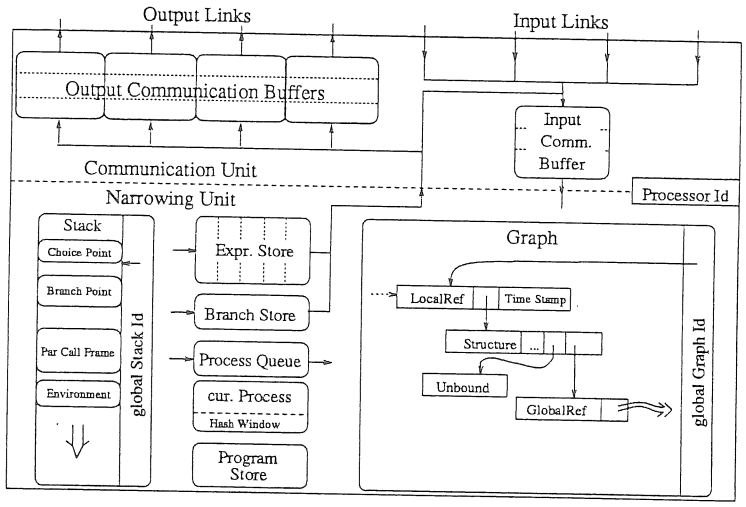


Figure 2: The Abstract Machine

reclamation remains possible if the topmost structure is the one to be deallocated. First observations show that this happens really often.

The graph structure was extended from the stack representation towards the more general heap representation. This generalization retains the speed of structure allocation but requires a general garbage collector, too.

Aside from the structures known from sequential implementations, several buffers are introduced to manage the local processes, to store separately the subexpressions that can be executed remotely, and to store the or-branches ready for remote execution.

Multiprocessing

Multiprocessing is advantageous to minimize computation locks due to communication delays. Even in a shared memory environment, the single-processing decision imposes severe restrictions to the load distribution when pure stack based architectures are considered [15]. Otherwise, the correctness in the presence of backtracking becomes endangered.

The expression store (LIFO) is intended to hold the expression-parallel siblings to be distributed, while the branch store (FIFO) holds the or-parallel siblings. The support of local process execution is an important optimization to reduce the overhead of process generation that involves the expensive creation of several structures for their management. This includes the preparation of a new set of registers, the extension of the process queue, and some arrangements in case of exhausted evaluations. The support of local process creation requires the introduction of one further small control structure in the runtime stack [7] that is mainly used to separate the

stack areas of different local (≡virtual) processes. Actually, no further process is generated. During forward computation, necessary context switches are controlled by the code. Only when backtracking happens, these special choice points enforce the needed switches.

Our observations have shown that many subexpressions put into the expression store are locally evaluated.

Parallel evaluation is performed following the fork&join paradigm where the annotated expressions are spawned in parallel (fork). Instead of purely synchronizing, the spawning process participates in the parallel evaluation and fetches locally some subexpression yielding nearly sequential speed. In order to control the parallel (virtual) child processes, a so-called parallel call frame is generated.

5.1 Load and Data Distribution

Load distribution is an important topic in order to achieve good speed-ups. But it should introduce very low overhead. Mainly, we follow a passive, decentralized strategy in which each idle processor asks several other processors for available work.

As a starting point we chose the sole work request from the neighbours. Hereafter, we enrich these requests with some information about the local load which is determined from the occupancy of the different memory areas. At each processor this information gives coarse information about the load on its neighbour and drives the selection order of the requested processors. We extend this set of neighboured processors and support the routing of work requests in accordance to the noted loads up to a certain distance accelerating the load distribution. I.e., if a work request encounters a processor without available work, it determines its best suited neighbour and bypasses the work request. This gives a more intelligent load balancing strategy without the introduction of a global scheduling mechanism.

As the creation of distributable work introduces some overhead, the engine switches between a parallel and a serial mode [16] which is triggered by the local load and remote work requests. The spawning of parallel work happens in the parallel mode.

Furthermore, data distribution involves some overhead. Our experiments confirm the observation in [16] that the exchange of small data nodes introduces too much overhead. Therefore, not only small data nodes but greater data packages are communicated on data requests.

5.2 Distributed Garbage Collection

Declarative programs, e.g. functional logic programs, are memory hogs that undoubted require garbage collectors. We embodied specific garbage collectors for the runtime stack and the graph. Both garbage collectors are hybrid applying a proposal of [9] that distinguishes between local and remote references. Remote references are managed via weightened reference counting [4] and local references are dealt with a variant of the Morris algorithm [13].

6 Results

The parallel system is mapped both onto a loosely-coupled Transputer network with up to 64 processor elements and onto the tightly-coupled Sequent Symmetry with 6 processor elements. The emulating system is implemented in C. We have investigated the behaviour of the parallel systems for both the expression-parallel context and the or-parallel one.

In order to test the implementation, we measured the efficiency of the implementation on one processor. A comparison with [16] which provides a parallel emulator implementation for Prolog in a similar environment, shows that we get double speed for equivalent programs. Furthermore, we observed much lower parallelism overhead following their 2-mode approach. For instance, the slightly modified Fibonacci function nfib which counts the function calls, and the quick-sort program show a parallelization overhead below 5% for our emulator. A comparison of our implementation with a Transputer implementation of K-Leaf [3] shows a similar behaviour for the Fibonacci function fib(22) and the 8-queens problem.

program	2	4	8	16	32	48
f-queen 11	1.97	3.93	7.85	15.49	30.95	45.54
nfib 30	1.97	3.95	7.82	14.67	26.56	35.80
Hanoi 22	1.99	3.98	7.94	14.62	26.66	33.94

The first table shows the speed-ups for some typical functional programs. These programs focus on the expression-parallelism and are well suited to measure the impacts of the independence restrictions. We considered the following deterministic programs: a functional version of the classical n-queens problem with 11 queens, the slightly modified Fibonacci, and the towers of Hanoi examples. Here we studied different topologies with up to 48 processors. The queens program which is the most complex among them shows the best speed-up. Even with 48 processors it reaches nearly linear speed-up. Roughly 5% of the computational power of the Transputer system is lost.

program	2	4	8	16	32	60
queens12	1.98	3.93	7.84	15.57	28.62	50.83
naive-sort9	1.98	3.95	7.85	15.68	31.36	46.95
puzzle	1.97	3.93	7.79	14.41	25.71	38.08
knapsack	1.95	3.90	5.00	6.75	9.78	14.77

In contrast, the second table shows the speed-ups we got for some nondeterministic search problems. These or-parallel programs are the n-queens problem in the logical version, the naive-sort, the knapsack and a puzzle problem. Again, the results show that considerable speed-ups are possible. Note that the examples show almost linear speed-ups concerning topologies with up to 16 processors and often even with 32 processors. Break-ins can be observed when the system is scaled beyond the frontier of processor elements for nowadays shared memory architectures.

Program	func-queen8	quicksort3000	queens9	naive-sort8	knapsack
Sequent	5.57	2.64	5.72	5.78	5.77
Transputer	4.91	2.46	5.41	5.62	4.69

A comparison in the third table with the speed-ups on the Sequent Symmetry (assuming 6 processors) shows that further improvements can be gained on systems with shared memory if expensive argument copying can be avoided (quicksort) and whenever the granularity decreases so that the available work can be better distributed in the bus-based system.

7 Conclusion and Future Work

In this paper we have sketched a parallel system for the functional logic programming language Babel exploiting both expression- and or-parallelism transparently. This system aims at machines with loosely coupled memory which are known to introduce severe communication costs concerning remote data access. In spite of these restrictions, the result section has shown the applicability of a process based approach when the load distribution is attached with granularity consideration, speculative data distribution, and slight extensions to promote the available work. The combined approach of expression-or-parallelism has been only investigated by an outline of the intended approach because the cross-product remains to be implemented.

Aside of this extension, our future work deals with the adjustments of the parallel system to other distributed machines. Currently, we are porting the Transputer implementation to the Fujitsu AP1000 which has more powerful Sparc-processors at each node. The new runtime results may give new insights for this kind of parallelism.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *North American Conference on Logic Programming*, pages 757–776. MIT Press, 1990.

[2] L. Araujo and J.J. Ruz. PDP: Prolog Distributed Processor for Independent AND/OR Parallel Execution of Prolog. In *Workshop on Parallelism and Implementation Technologies*, 1994.

[3] G.P. Balboni, P.G. Bosco, C. Cecchi, R. Melen, C. Moiso, and G. Sofi. *Parallel Computers. Object-Oriented, Functional, Logic*, chapter 7: Implementation of a Parallel Logic + Functional Language. John Wiley & Sons, 1990.

[4] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE*, number 259 in LNCS, pages 176–187, 1987.

[5] J. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publisher, 1987.

[6] W. Hans and St. Winkler. Abstract Interpretation of Functional Logic Languages. Technical Report AIB 92–43, RWTH Aachen, 1992.

[7] M.V. Hermenegildo. *An Abstract Machine based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, The University of Texas at Austin, 1986.

[8] P. Kacsuk. OR-parallel Prolog on Distributed Memory Systems. In *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 453–463. Springer-Verlag, 1994.

[9] D.R. Lester. An Efficient Distributed Garbage Collection Algorithm. In *PARLE*, volume 365 of *LNCS*, pages 207–223. Springer Verlag, 1989.

[10] R. Loogen. Stack-based Implementation of Narrowing. In *CCPSD, Tapsoft, LNCS 494*. Springer-Verlag, 1991.

[11] R. Loogen and St. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. In *Symposium on Programming Language Implementatio and Logic Programming (PLILP)*, Passau, 1991. Springer-Verlag.

[12] J.J. Moreno and M. Rodríguez-Artalejo. BABEL: A Functional and Logic Programming Language Based on a Constructor Discipline and Narrowing. In *Conference on Algebraic and Logic Programming, LNCS 343*, pages 223–232. Springer-Verlag, 1988.

[13] F. Lockwood Morris. A Time– and Space–Efficient Garbage Compaction Algorithm. *Communications of the ACM*, 21(8):662–665, 1978.

[14] U.S. Reddy. Narrowing as the Operational Semantics of Functional Programs. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Society Press, July 1985.

[15] F. Sáenz, J.J. Ruz, W. Hans, and St. Winkler. A Stack-based Machine for Parallel Execution of Babel Programs. In Hoon Hong, editor, *Parallel Symbolic Computation*, volume 5 of *Lecture Notes Series in Computing*, pages 336–345, 1994.

[16] A. Verden and H. Glaser. An AND–Parallel Distributed Prolog Executor. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, Series in Parallel Computing, chapter 7, pages 143–158. Wiley, 1992.

[17] D. H. D. Warren. An Abstract Prolog Instruction set. 309 Technical Note, SRI International, 1983.

[18] D. H. D. Warren. Or-Parallel Execution Models of Prolog. *Datsoft'87*, pages 243–257, 1987.

[19] H. Westphal, P. Robert, J. Chassin de Kergommeaux, and J.-C. Syre. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In The IEEE Computer Society Press, editor, *Proceedings - 1987 Symposium on Logic Programming*, pages 436–448. IEEE, September 1987.

An Effective Algorithm for Compiling Pattern Matching Keeping Laziness

Pedro Palao and Manuel Núñez
Departamento de Informática y Automática
Universidad Complutense de Madrid
fax: (34-1) 394 4607 ph: (34-1) 394 4468
e-mail: {ecceso,manuelnu}@eucmvx.sim.ucm.es

Abstract

In this paper we present an algorithm for compiling functions defined by pattern matching with side conditions, which can be easily implemented. As previous approaches, this algorithm has a complete set of rules. The generated code has no backtracking and side conditions are evaluated at most once, which represents an advantage with respect to most of the previous algorithms. This algorithm is parameterized by a *subroutine*, which can be chosen such that the result of the compilation fulfills certain properties. In this paper we choose a subroutine that keeps the *laziness* of a list of patterns. But in contrast with previous algorithms (characterization algorithms), this algorithm does not previously determine if the pattern is lazy or not; our algorithm works with any kind of patterns, generating a *lazy* code if the pattern is lazy and even finding the lazy subpatterns of a non lazy pattern. A fundamental concept in order to apply this subroutine is the concept of *distinguisher* of a pattern, which indicates if a *column* of a list of patterns must be chosen to *expand* with it the algorithm.

Keywords: Functional Programming, Pattern Matching, Laziness, Compilation.

1 Introduction

Pattern Matching has been widely studied in the theory of Term Rewriting. This problem can be stated as: given a list of terms $p_1, \ldots, p_n$ and a term t, find whether t is an instance of any of the p_i. The most straightforward algorithm to solve this problem is checking t against each p_i, but this solution is not acceptable because the running time depends on the number of terms in the list. Several algorithms have been developed to solve this problem more efficiently (see [HO82], [Grä91]).

In this paper we restrict ourselves to the study of pattern matching in the implementation of functional programming languages. In functional programming, this problem has some specific features; for example, it is usual to add some strategy in order to decide which of the possible p_i such that t is an instance of p_i is chosen (usually the first top–down). There are also specific algorithms (see [Aug85], [Wad87], [Lav88], [Sch88]) for functional programming.

But previous algorithms usually do not deal with *laziness*. Intuitively speaking, laziness means that a value is only computed when it is needed in order to evaluate an expression. In order to know whether a value matches a pattern, this value must be evaluated to *head normal form*. Then, a lazy language must take care about how

pattern matching is performed, evaluating arguments as less as possible to determine if the argument matches the pattern. Nevertheless, almost all the implementations of recent functional languages do not consider techniques which perform pattern matching in a lazy way.

In this paper we propose an algorithm with the following features:

- Each of the arguments is parsed at most once in order to determine which pattern matches (therefore, there is no backtracking in the compiled code).
- Side conditions, which may be very hard to evaluate, are just tried at most once, and only when it is not possible to distinguish by patterns. Previous algorithms usually do not deal with side conditions.
- One expects, that in a lazy language, the order of evaluation over the argument structure is performed such that the function may diverge (at this point) only if it diverges with any order of evaluation. Our algorithm deals with this topic (laziness), but in contrast with other algorithms ([Lav87, Lav88]), it does not previously characterize if the pattern is lazy or not (see definition 4).

The remainder of the paper is organized as follows. Section 2 introduces preliminary definitions. Section 3 gives the bulk of our algorithm for compiling functions defined by pattern matching. In Section 4 we introduce a subroutine, that combined with the previous algorithm, keeps laziness. In Section 5 we give some outlines for the implementation of the algorithm. Finally, Section 6 presents our conclusions.

2 Definitions

Definition 1 Let Σ be a finite ranked alphabet which is the disjoint union of alphabets Σ_n ($\Sigma = \biguplus_{n\in\mathbb{N}} \Sigma_n$). We consider a set of variable symbols Σ_X such that $\Sigma_X \subseteq \Sigma_0$. $\alpha \in \Sigma_n$ means that α has *arity* n. The set of Σ-terms is defined inductively as the least set such that if $\alpha \in \Sigma_n$, and $t_1, t_2, \ldots, t_n \in \Sigma$-terms, then $\alpha\, t_1\, t_2 \cdots t_n \in \Sigma$-terms. Given a Σ–term $t = \alpha\, t_1\, t_2 \cdots t_n$, we will denote t_i by $t[i]$ (for $1 \le i \le n$), and α by $t[0]$. Given $x \in \Sigma_X$, x is denoted by $x[0]$, while $x[i]$ ($i > 0$) denote fresh variables. ✡

Definition 2 A *pattern* is a tuple of Σ-terms. We say that a pattern is *linear* if there is no variable which appears twice in the same pattern. An *instance* of a pattern is a tuple of terms which can be obtained from the pattern by replacing all the variables by any values. Let σ be a function which maps variables into terms. We call *substitution* the extension of σ as a morphism from terms to terms. ✡

Note that if a term is an instance of a pattern, then there exists a substitution σ which yields the term as image of the pattern. From now on, when we consider lists of patterns, we will suppose that all the patterns (tuples) have the same length.

Definition 3 Let $[p_1, \ldots, p_m]$ be a list of patterns. We say that a tuple of terms t *matches* p_i, if p_i is the first pattern of the list such that t is an instance of p_i. We say that an algorithm that decides if t matches p_i exploring t (starting from the root of each argument in t, and comparing the symbols encountered with the symbols in the corresponding part of the pattern) is a *matching strategy*. Note that, with this definition, there is at most one p_i in P such that t matches p_i. ✡

Definition 4 We say that a matching strategy $\mathcal{E}$ for a list of patterns P is *lazy* if for any t such that $\mathcal{E}$ diverges exploring t, then any other strategy diverges exploring t. We say that a list of patterns P is *lazy* if there is a lazy strategy for P. ✡

Example 1 Let f and g be the functions that follow:

$$\begin{array}{llll} f & [\,] & [\,] & = exp_1 \\ f & x & y & = exp_2 \end{array} \qquad \begin{array}{llll} g & [\,] & [\,] & = exp_1 \\ g & x & y{:}ys & = exp_2 \end{array}$$

There is no lazy strategy for $P = [([\,],[\,]),(x,y)]$. Consider the call $f\ (a_1{:}a_2)\ \Omega$ (where Ω is a divergent argument). If the evaluation starts with the first argument, the value exp_2 is obtained, while if the evaluation starts with the second argument, a divergent computation is produced. But $f\ \Omega\ (a_1{:}a_2)$ diverges starting with the first argument, while starting with the second one, returns exp_2.

On the other hand, there is a lazy strategy for $Q = [([\,],[\,]),(x,y{:}ys)]$: the second argument is evaluated before the first one. Nevertheless, lazy functional languages like Gofer or Miranda give a divergent computation in the evaluation of $g\ \Omega\ (a_1{:}a_2)$. ✡

Definition 5 We say that a function is *defined by pattern* if

- Its definition is a list of triples (pattern, expression, side condition).
- Its value, when applied to an argument t, is obtained finding the triple (p_i, exp_i, con_i) such that t matches p_i by a substitution σ and con_i holds for this substitution, and then evaluating the result of applying σ to exp_i. If there is no such a p_i, then an error message is produced. ✡

Definition 6 Let P be a list of patterns $[p_1, \ldots, p_m]$, where $p_j = (p_{j1}, \ldots, p_{jn})$, and let $q = (q_1, \ldots, q_k)$ be a pattern. The next notation will be used:

$$\begin{array}{lcl} [p_i]_{i=1}^{n} & = & [p_1, \ldots, p_n] \\ q\,!\,i & = & q_i \\ q\,\bar{!}\,i & = & (q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k) \\ P\,!\,i & = & (p_1\,!\,i, \ldots, p_n\,!\,i) \ \text{(column } i\text{)} \\ P\,\bar{!}\,i & = & [p_1\,\bar{!}\,i, \ldots, p_n\,\bar{!}\,i] \end{array} \qquad \begin{array}{lcl} \#Var_i & = & \text{Number of variables in } P\,!\,i \\ \#C_i & = & \text{Number of } \textit{root} \text{ occurrences} \\ & & \text{of the constructor } C \text{ in } P\,!\,i \\ & & \\ Cons_i & = & \{C_i \mid \#C_i \neq 0\} \end{array}$$

✡

Definition 7 Let $P = [(p_{k1}, \ldots, p_{kn})]_{k=1}^{m}$. We define the *constructor selection function* for a constructor C in column i ($1 \le i \le n$), denoted by γ, as the function which satisfies the following conditions:

- γ: $1..(\#C_i + \#Var_i) \rightarrow 1..m$
- $\gamma(j) < \gamma(j+1)$, $\forall j (1 \le j < \#C_i + \#Var_i)$
- $p_{\gamma(j)\,i}[0] \in \Sigma_X \cup \{C\}$, $\forall j (1 \le j \le \#C_i + \#Var_i)$, i.e. $p_{\gamma(j)\,i}$ is either a variable or a term $t = C\ t_1 \cdots t_k$. ✡

Lemma 1 Given a list of patterns P, a constructor C and a column i, the definition of γ for the constructor C in column i is unique. ✡

Intuitively speaking, given a column and a constructor C, γ considers the terms which either have the constructor C in the root or are variables, preserving the ordering in the list of patterns.

Example 2 Let $P = [(x{:}xs, y{:}ys), (x{:}xs, y{:}ys), (xs, [\,]), ([\,], ys)]$. Then, for ":" and the column 2, we have: $\gamma{:}\,1..3 \rightarrow 1..4$, and $\gamma(1) = 1,\ \gamma(2) = 2,\ \gamma(3) = 4$. For "[ ]" and the column 2, we have: $\gamma{:}\,1..2 \rightarrow 1..4$, and $\gamma(1) = 3,\ \gamma(2) = 4$. ✡

Definition 8 Let P be a list of patterns $[(p_{k1}, \ldots, p_{kn})]_{k=1}^{m}$. We define the *variable selection function* for a column i $(1 \leq i \leq n)$, denoted by γ^X, as the function which satisfies the following conditions:

- $\gamma^X{:}\,1..\#Var_i \rightarrow 1..m$
- $\gamma^X(j) < \gamma^X(j+1),\ \forall j(1 \leq j < \#Var_i)$
- $p_{\gamma^X(j)\,i}$ is a variable, $\forall j(1 \leq j \leq \#Var_i)$. ✡

Lemma 2 Given a list of patterns P and a column i, the definition of γ^X for the column i is unique. ✡

Definition 9 Let $P = [(p_{k1}, \ldots, p_{kn})]_{k=1}^{m}$ be a list of patterns and $t = (t_1, \ldots, t_n)$ be a tuple of terms. We define the *subpattern of P generated by t*, denoted by $\mathcal{S}_t(P)$, as the list of patterns defined as:

1. If $\forall i(t_i \in \Sigma_X)$, then $\mathcal{S}_t(P) = P$.
2. If $\exists j(t_j \notin \Sigma_X)$, then $\mathcal{S}_t(P) = \mathcal{S}_{t'}(P')$ where $C = t_j[0]$, r is the arity of C, $s = \#Var_j + \#C_j$, $t' = (t_1, \ldots, t_{j-1}, t_j[1], \ldots, t_j[r], t_{j+1}, t_n)$, and $P' = [(p_{\gamma(k)\,1}, \ldots, p_{\gamma(k)\,j-1}, p_{\gamma(k)\,j}[1], \ldots, p_{\gamma(k)\,j}[r], p_{\gamma(k)\,j+1}, \ldots, p_{\gamma(k)\,n})]_{k=1}^{s}$ ✡

Lemma 3 Let P be a list of patterns, and t be a term. If $\mathcal{S}_t(P)$ is not a lazy list of patterns then P is not a lazy list of patterns. ✡

3 The Algorithm

In this section we present a formal description of our algorithm. Our algorithm has a function defined by pattern as argument and returns an expression which, considered as a tree, has in the internal nodes a **case** clause over a simple pattern or an **if** clause over any of the side conditions. The *leaves* of the tree are the expressions that define the function.

Although the algorithm only works with linear patterns, it can be generalized to non linear patterns in the usual way, changing the repeated variables for new ones and adding an equality condition to the side condition.

We will specify the algorithm as a function *compile* which has a function defined by pattern as argument and returns the tree expression.

Definition 10 Let $f = [((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m}$ be a function defined by pattern. We define the function *compile* as

$$compile\ f = match\ (u_1, \ldots, u_n)\ f$$

where $u_1, \ldots, u_n$ are fresh variables indicating the length of the patterns $\bar{v}_k$. ✡

The rest of the section is devoted to define the function *match*. This function is inductively defined, with the property that in recursive calls, the ordering of the triples in the original definition is preserved. We give a complete set of rules; some of them are similar to those in [Wad87] (Empty Rule, Variable-Column Rule) while others are specific for our algorithm.

3.1 Base Rules

There are two base cases: when the list of variables is empty (first argument), and when the list of triples is empty (second argument).

When the list of variables is empty, all the expressions are equally acceptable, because there are no patterns. We use the side condition in order to know which of the expressions is chosen. The algorithm must keep the order in the function, and for this reason this case has to be compiled with a sequence **if** $\cdots$ **elsif** $\cdots$ **else**, finishing with a *failure* clause (used if no condition evaluates to true).

Rule 1 (Empty Rule)

$$match\ (\,)[((\,), exp_1, con_1), ((\,), exp_2, con_2), \cdots, ((\,), exp_m, con_m)] =$$

$$\textbf{if}\ con_1\ \textbf{then}\ exp_1\ \textbf{elsif}\ con_2\ \textbf{then}\ exp_2\ \cdots\ \textbf{elsif}\ con_m\ \textbf{then}\ exp_m\ \textbf{else No Match}$$ ✡

Another base case appears when the list which defines the function is empty. That means that the pattern is not exhaustive, and a run–time error must be produced. This error is not a fault with a *backtracking jump* like in [Aug85], but indicates that the function argument matches no pattern (considering side conditions) in the definition of the function.

Rule 2 (Fail Rule) $match\ (u_1, \ldots, u_n)\ [\,] = \textbf{No Match}$ ✡

3.2 Inductive Rules

Now we consider the inductive cases. There are two rules which simplify the call to *match*, and another rule which is used if none of the previous rules can be applied (Default Rule).

The first rule can be applied when the first triple has a pattern only with variables. Then, an **if** is generated with the condition con_1, substituting the variables appearing in the pattern with the values $(u_1, \ldots, u_n)$, the expression exp_1 in the **then** part (applying the same substitution) and, in the **else** part, the result of the rest of compilation.

Rule 3 (Variable-Row Rule) If $v_{11}, \ldots, v_{1n}$ are variables, then

$match\ (u_1, \ldots, u_n)[((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m}\ =$

$\textbf{if}\ con_1[u_1/v_{11}, \ldots, u_n/v_{1n}]\ \ \textbf{then}\ exp_1[u_1/v_{11}, \ldots, u_n/v_{1n}]$
$\qquad\qquad\qquad\qquad\ \ \textbf{else}\ match\ (u_1, \ldots, u_n)[((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=2}^{m}$ ✡

The second rule is applied if there exists a column in the list of patterns such that only variables appears in this column. This can be solved by removing this variable, and then, performing a renaming both in the expressions and in the side conditions.

Rule 4 (Variable-Column Rule) If there exists a *column* i such that all the terms are variables, then

$$\begin{array}{l} match\ (u_1, \ldots, u_i, \ldots, u_n)[((v_{k1}, \ldots, v_{ki}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m} = \\ match\ (u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n) \\ \qquad [((v_{k1}, \ldots, v_{k\,i-1}, v_{k\,i+1}, \ldots, v_{kn}), exp_k[u_i/v_{ki}], con_k[u_i/v_{ki}])]_{k=1}^{m} \end{array}$$ ✡

If none of the previous rules can be applied, then the *Default Rule* is applied. This rule expands with a column (the algorithm which chooses this column will be presented in section 4). Intuitively speaking, if the i-th argument of the function is evaluated enough to find a constructor in the root, then we can discard most of the patterns of the i-th column. We only have to consider the patterns which have the previously obtained constructor in the root, and the patterns which are variables. In the former case, the subexpressions of the argument still have to be compared with the constructor arguments, but in the latter this comparison is not necessary; in these triples, new variables are needed (according to the arity of the considered constructor). This is shown in the following

Example 3 Consider the call to the *match* function

$$\begin{array}{l} match\ \ (u_1, u_2) \\ \qquad [((x{:}xs,\ y{:}ys),\ \ x : \text{merge}\ xs\ (y{:}ys),\ \ x \leq y), \\ \qquad\ ((x{:}xs,\ y{:}ys),\ \ y : \text{merge}\ (x{:}xs)\ ys,\ \ x > y), \\ \qquad\ ((\ xs,\ \ [\ ]\ ),\ \ \ \ \ \ \ \ xs,\ \ \ \ \ \ \ \ true\ ), \\ \qquad\ ((\ [\ ],\ \ ys\ ),\ \ \ \ \ \ \ \ ys,\ \ \ \ \ \ \ \ true\ )] \end{array}$$

and suppose that the second column is chosen to expand with. The case expression has two entries: one for "[ ]" and another one for ":". In the first entry, the third triple (since it has "[ ]" in the root) and the fourth one (because it is a variable) are placed. The first, second and fourth ones appear in the second entry. The result is

$$\begin{array}{lcl} \textbf{case}\ u_2\ \textbf{of} & & \\ \quad [\,] & \Rightarrow & match\ \ (u_1) \\ & & \qquad [((xs),\ \ xs,\ \ true), \\ & & \qquad\ (([\,]),\ \ u_2,\ \ true)] \\ \quad w_1{:}w_2 & \Rightarrow & match\ \ (u_1, w_1, w_2) \\ & & \qquad [((x{:}xs, y,\ ys),\ \ x : \text{merge}\ xs\ (y{:}ys),\ \ x \leq y), \\ & & \qquad\ ((x{:}xs, y,\ ys),\ \ y : \text{merge}\ (x{:}xs)\ ys,\ \ x > y), \\ & & \qquad\ ((\ [\,],\ \alpha_1{,}\alpha_2),\ \ \ \ \ \ \ \ u_2,\ \ \ \ \ \ \ \ true\ )] \end{array}$$

where $w_1, w_2, \alpha_1, \alpha_2$ are fresh variables. ✡

Rule 5 (Default rule) If the column i is chosen to expand with, then

$$\begin{array}{lcl} match\ (u_1, \ldots, u_i, \ldots, u_n)[((v_{k1}, \ldots, v_{ki}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m} = & & \\ \textbf{case}\ u_i\ \textbf{of} & & \\ \quad C^1\bar{w}_1 & \Rightarrow & match\ (u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n) +\!\!+\ \bar{w}_1 \\ & & \qquad [(v_{\gamma_k^1}\bar{\imath}i +\!\!+ (v_{\gamma_k^1 i}[1], \ldots, v_{\gamma_k^1 i}[n_1]), exp'_{\gamma_k^1}, con'_{\gamma_k^1})]_{k=1}^{s^1} \\ \quad \ldots & & \qquad\qquad \ldots \\ \quad \ldots & & \qquad\qquad \ldots \\ \quad C^r\bar{w}_r & \Rightarrow & match\ (u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n) +\!\!+\ \bar{w}_r \\ & & \qquad [(v_{\gamma_k^r}\bar{\imath}i +\!\!+ (v_{\gamma_k^r i}[1], \ldots, v_{\gamma_k^r i}[n_r]), exp'_{\gamma_k^r}, con'_{\gamma_k^r})]_{k=1}^{s^r} \\ \textbf{otherwise} & \Rightarrow & match\ (u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n) \\ & & \qquad [(v_{\gamma_k^x}\bar{\imath}i, exp_{\gamma_k^x}[u_i/v_{\gamma_k^x i}], con_{\gamma_k^x}[u_i/v_{\gamma_k^x i}])]_{k=1}^{s^x} \end{array}$$

where
$$\left\{\begin{array}{lcl} Cons_i & = & \{C^1, C^2, \ldots, C^r\},\ s^x = \#Var_i,\ s^a = s^x + \#C_i^a\ \ (1 \leq a \leq r) \\ \gamma_b^a & = & \gamma(b), \text{ where } \gamma \text{ is the selection function for } C^a \text{ in column } i \\ \gamma_b^x & = & \gamma^X(b), \text{ where } \gamma^X \text{ is the variable selection function for column } i \\ n_a & = & \text{arity of the constructor } C^a\ \ (1 \leq a \leq r) \\ exp'_l & = & \left\{\begin{array}{ll} exp_l & , \text{ if } \neg Var(v_{l\,i}) \\ exp_l[u_i/v_{l\,i}] & , \text{ otherwise} \end{array}\right. \\ con'_l & = & \left\{\begin{array}{ll} con_l & , \text{ if } \neg Var(v_{l\,i}) \\ con_l[u_i/v_{l\,i}] & , \text{ otherwise} \end{array}\right. \end{array}\right.$$

and for any a, b such that $1 \leq a \leq r$, $1 \leq b \leq s^a$, and such that $v_{\gamma_b^a i}[0] \in \Sigma_X$ (i.e. $v_{\gamma_b^a i}$ is a variable), $v_{\gamma_b^a i}[1], \ldots, v_{\gamma_b^a i}[n_a]$ are fresh variables. With this condition we ensure that patterns remain linear and that there is no name capture. ✡

Note that if any v_{ki} is a variable, then there is an occurrence corresponding to the k-th triple, for every entry $C^j\bar{w}_j$, and another one in the **otherwise** clause. The **otherwise** clause is used for the constructors which do not appear explicitly in the **case** expression. For this reason, if all the constructors of the type of the column i are in $Cons_i$ (i.e. column i is *exhaustive*), the **otherwise** clause may be removed (as in Example 3). A consequence of the Default Rule is that one expression may appear in different places. In Section 5, we show how a code without repetitions can be generated.

The compilation to a virtual machine of the **case** expression is usually done using a table of memory directions, which is indexed by the type constructors.

4 Choosing a *good* column

In the previous section, we have not given an algorithm which selects a column to expand with. Now, we give an algorithm that selects these columns keeping laziness. The idea is to find the columns which must be (necessarily) evaluated to determine if a term matches a pattern. Next, we introduce the concept of *distinguisher*.

Definition 11 Let $P = [p_1, \ldots, p_m]$ be a list of patterns. We say that p_i with $1 \leq i \leq m+1$ (for convenience p_{m+1} will be a pattern only with variables), is a *distinguisher* of P, if there is a tuple of terms t which is an instance of p_i and such that it is not an instance of any p_j for $1 \leq j < i$. We say that p_i is a *distinguisher of P for the column j* if $p_i \bar{1} j$ is a distinguisher of $P \bar{1} j$ and p_{ij} is a variable. ✡

Let us remark that there exists a distinguisher for a column which only has constructors in the root iff the rest of the columns are not exhaustive. This distinguisher would be in the *row* $m+1$, because there are no variables in that column.

Lemma 4 Let P be a list of patterns. P has a matching strategy not evaluating t_j in a list of arguments $t = (t_1, \ldots, t_n)$ iff P has a distinguisher for the column j.
Proof: Let t be a list of arguments such that it is not necessary to evaluate t_j, and let p_i be such that t matches p_i. Then, p_{ij} must be a variable; otherwise t_j would have to be evaluated in order to know whether it coincides with the constructor. Also, the fact that t does not match any p_k with $k < i$ implies that $t \bar{1} j$ does not match any $p_k \bar{1} j$ with $k < i$. Thus, $t \bar{1} j$ is the instance that makes p_i a distinguisher of P for the column j.

Now, suppose that p_i is a distinguisher of P for the column j. Then, p_{ij} is a variable and there is an instance t of $p_i \bar{1} j$ which is not an instance of $p_k \bar{1} j$ (for any k, $1 \leq k < i$). Then, a strategy which checks if an argument coincides with t in every column but in the j-th, is a strategy which does not evaluate the j-th column for the instance t. ✡

Lemma 5 Let P be a lazy list of patterns, and j be a column which has a distinguisher. Then, a lazy strategy cannot expand with column j.
Proof: Let $t = (t_1, \ldots, t_n)$ be a list of terms such that $t \bar{1} j$ is the instance with whom p_i is a distinguisher for the column j. Then, there is a strategy $\mathcal{E}$ that does not evaluate the column j when is applied to an argument a such that $a \bar{1} j = t \bar{1} j$. Thus, this strategy does not diverge when the argument $a = (t_1, \ldots, t_{j-1}, \Omega, t_{j+1}, \ldots, t_n)$ is applied. Easily follows that a lazy strategy does not diverge when the argument a is applied, and thus it can not be possible to expand with the column j. ✡

Definition 12 We define the set of *admissible columns* for a list of patterns P, denoted $\mathcal{A}(\mathcal{P})$, as the set of columns which have not a distinguisher. ✡

Corollary 1 Let P be a lazy list of patterns. Then $\mathcal{A}(\mathcal{P})$ is not empty. ✡

This corollary gives a necessary condition for a list of patterns to be lazy. The following example shows that "$\mathcal{A}(\mathcal{P}) \neq \emptyset \Rightarrow \mathcal{P}$ is a lazy list of patterns" (the reciprocal of Corollary 1) does not hold.

Example 4 Consider $P = [([\,],[\,]),([\,],y{:}ys),(x{:}xs,[\,]),(x{:}[\,],y{:}[\,]),(x{:}xs,y{:}ys)]$. P is exhaustive and all the terms have constructors in the root. Then, both columns are admissible ones (and thus $\mathcal{A}(\mathcal{P}) \neq \emptyset$). But the subpattern corresponding to the term $(a_1{:}b_1,\ a_2{:}b_2)$ is $[(x,[\,],y,[\,]),\ (x,xs,y,ys)]$, and (as we saw in function f of example 1) there is no lazy strategy for matching (b_1, b_2) with $[([\,],[\,]),\ (xs,ys)]$. By Lemma 3, P is not lazy because it has a subpattern which is not lazy. ✡

Lemma 6 Let P be a lazy list of patterns and let $A = \mathcal{A}(\mathcal{P})$. Then, for any $i \in A$, there is a lazy strategy that expands with the column i.
Proof: P is lazy implies that there exists a lazy strategy $\mathcal{E}$ for it. This strategy must expand with any of the columns in A (Lemma 5). Suppose that $i \in A$ is chosen to expand with, and consider $j \in A$ such that $j \neq i$. We know that $\mathcal{E}$ always has to evaluate the column j. Then, we consider a strategy which *interchanges* the points of evaluation of i and j. Obviously, this strategy is also lazy. ✡

Corollary 2 Let P be a lazy list of patterns. A pattern strategy is lazy iff this strategy expands with a column in $\mathcal{A}(\mathcal{P})$. ✡

After corollary 2, we can give our algorithm to choose a column to expand with.

Algorithm (Choose a column to expand with)
Let P be a list of patterns, and let A be the set of admissible columns. If $A \neq \emptyset$ we choose any one in the set A. If $A = \emptyset$ we choose any column (that means that the list of patterns is not lazy). ✡

This algorithm chooses a column keeping laziness (Lemma 6), and if the list of patterns is lazy, this choice gives a lazy strategy (Corollary 2). As we showed in Example 4, the idea of admissible column represents a characterization of *local* laziness over a part of the argument that it is been explored. Thus, our algorithm can isolate the part of the pattern where the problem (no laziness) appears, and it can compile the rest of the pattern in a lazy way.

Example 5 We will show that in Example 3 the second column is chosen by our algorithm. We must show that the second column is an admissible one, while the first one is not. This fact is because there is a distinguisher for the first column in the third row: the instance ([]) gives us the result. After third row, the first column is exhaustive and that is why it can not be any distinguisher for the second one. ✡

Theorem 1 Let P be a lazy list of patterns. Then, the algorithm in the previous section, parameterized with the algorithm above to choose a column, gives a lazy matching strategy.
Proof: The proof is done by induction over the rules of the algorithm. The base rules (Empty Rule and Fail Rule) give a lazy strategy because they do not perform any decision over the list of patterns. The result for the Variable–Row and Variable–Column rules is immediate by induction, since none of them evaluate any argument. The proof of laziness for the Default Rule can be done using Lemma 5 and by induction. ✡

5 Implementation

In this section we deal with some details of the implementation of the algorithm.

5.1 Calculation of the set of admissible columns

First, we suppose that all the columns are admissible, and for each column, we create a set of instances (initially empty). We go top-down over the list of patterns, looking for variables and updating the set of instances of each column with the instances of the rest of the pattern (for the pattern p_i and the column j, the rest of the pattern is $p_i \bar{1} j$). If one variable is found in one of the columns, which belongs to the set of admissible columns, we determine if the rest of the pattern is a distinguisher for this column (looking at the associated set). If it is a distinguisher, we remove this column from the set of admissible ones and we forget the associated set (which will not be needed any more). We repeat this process until the end of the list of patterns is reached or the set of admissible columns is empty. The update of the set of instances can be done *by request* instead of doing it for every pattern in the list of patterns; that is, the update is done only when a variable is found.

5.2 Improvements to the efficiency of the algorithm

Increasing the speed of the algorithm can be done when the Default Rule has to be applied. If there is a column which only has constructors, and the rest of the columns are exhaustive, then this column can be chosen keeping laziness, because it will be in the set of admissible columns. This is very effective and it is presented very often; for example, this technique can be applied to all the examples presented in [Lav87, Lav88].

When the set of admissible columns for a list of patterns has been calculated, the problem is which of them is chosen. The next heuristic rule can be applied to locally minimize the number of steps that the algorithm must perform: choose the column which minimizes the number of constructors multiplied by the number of variables. This rule tries to expand as less as possible the patterns which have variables, leaving the problem as small as possible. In a similar way, this heuristic rule can be applied if there is no admissible column. Another technique is to expand with all the admissible columns, and calculate the set of admissible columns for the new list of patterns. With this technique, we get that some calculations, which would be done several times, are done only once.

5.3 Duplication of code

Obviously, the code duplication is solved, sharing the expression among the different *branches* in the **case** tree where the same code appears. Let T be a triple $(patt, exp, con)$, such that one of the terms in *patt* is a variable x, and let us suppose that the Default Rule is applied, expanding with the column where this variable appears. The expression *exp*, after a renaming of x given exp', will be in a branch corresponding to each constructor (or **otherwise**). Also, x is replaced in *patt* by n new variables $x_1, \ldots, x_n$, where n is the arity of the constructor. But none of the new variables appears in exp', and that means that any substitution over them does not modify exp'. Only the changes over the rest of the pattern may modify it,

but always in the same form. Thus, all the leaves in the **case** tree which have as associated expression *exp*, after doing all the necessary substitutions, will have the same expression and the sharing will be total.

But it still remains the problem that the number of new variables x_i depends on the constructor in which *exp* is placed, and thus there are an amount of useless bindings for *exp* which depends on the *branch* where the expression *exp* was placed. This can be solved by adding a small code that reorganizes the bindings, or by compiling the expression in such a way that it can deal with these problems. Anyway, it depends on the model of machine in which the compilation is done. Also there exists the same problem for the code duplication for side conditions, but usually these codes are very small, and duplication can be better than sharing.

Related Work

In the framework of functional programming, the first proposed algorithms are [Aug85] and [Wad87]. They give a set of rules to compile functions defined by pattern. Some of our rules in section 3 are similar to those in [Wad87]. The difference is that we have a *default rule* while Wadler's algorithm has several rules to distinguish different cases. The main advantage of our algorithm (out of laziness) with respect to [Aug85] and [Wad87] is that ours has not backtracking in the compiled code (and thus each argument is parsed at most once). Wadler's algorithm has been widely used (for instance in Gofer).

[Grä91] presents a theoretical characterization of an algorithm. The generated code is equivalent to ours if we would always expand with the first column in the *default rule* (i.e. without taking care about laziness).

[Sch88] gives an algorithm that works with restrictions in the types. That means that it can compile functions defined over subtypes of a given type.

The algorithm proposed in [PB85] performs the match *bottom-up*. Our algorithm cannot work in this way, since a *bottom-up* strategy requires the whole evaluation of the expression that it is been matched, and this is against our objective of keeping laziness. Because the match is performed *bottom-up*, it can not work with infinite objects.

In [Lav87] a characterization to know whether a list of patterns is lazy or not is given, but it is very complex and it has a difficult implementation. [Lav88] presents an algorithm (based on [Lav87]), that simplifies that characterization, but it still needs to evaluate if a list of patterns is lazy. With our algorithm it is not necessary to do this characterization previously (which can be very hard to do) because it uses a set of complete rules which will find a lazy strategy (if there exists one). Even if there is no lazy strategy, the compiled code will be *better* than the one generated by the algorithms in [Grä91] or [Sch88], because we can use the local laziness of some subpatterns.

[SRR92] shows that there exist patterns which produce an exponential (in size) tree for any possible strategy. In a recent work, [Mar94] studies lazy algorithms but using backtracking. This approach have some problems, because there exist patterns such that the match leads to a sequential checking.

6 Conclusion

In this paper we have presented an effective algorithm for compiling pattern matching in functional programming languages. This algorithm has a complete set of rules to obtain a code that has no backtracking and that explores the arguments as good as possible in order to preserve laziness. These rules are easy to implement and allow a great variety of adjustment which can improve the generated code. We think that this work method is suitable for this kind of problems and it allows to refine the quality of an algorithm to contain new characteristics. In fact, our intention is to extend the algorithm so it can compile functions that are applied on a subset of its type (as it is done in [Sch88]).

Another advantage of working with a sequence of rules, opposite to give characterizations (as in [Lav88] and [Grä91]), is that in spite of the fact that the problem does not fulfill the property we are characterizing, the rules can be applied to pieces of it. Moreover, it is done without searching these pieces but applying the *best* possible rule in any moment. For instance, there are many patterns which are not lazy because of a small part of the parameters; i.e. there are certain arguments that have to be evaluated following a fixed ordering and there are others for whom this ordering is not necessary. An algorithm of characterization will discard these patterns whereas an algorithm of rules will be able to find most of this arrangement.

Acknowledgements

We would like to thank A. Gavilanes for his help in the first steps of this research.

References

[Aug85] L. Augutsson. Compiling pattern matching. *Functional Programming Languages and Computer Architecture'85, LNCS 201*, pages 368 – 381, 1985.

[Grä91] A. Gräf. Left-to-Right tree pattern matching. *Rewriting Techniques and Applications'91, LNCS 488*, pages 323 – 334, 1991.

[HO82] C.M. Hoffmann and M.J. O'Donell. Pattern matching in trees. *Journal of the ACM*, 29(1):68 – 95, 1982.

[Lav87] A. Laville. Lazy pattern matching in the ML language. *FST & TCS'87, LNCS 287*, pages 400 – 419, 1987.

[Lav88] A. Laville. Implementation of lazy pattern matching algorithms. *ESOP' 88, LNCS 300*, pages 298 – 316, 1988.

[Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Technical Report RR–2385, INRIA, 1994.

[PB85] P. W. Purdom and C.A. Brown. Fast many-to-one matching algorithms. *Rewriting Techniques and Applications'85, LNCS 202*, pages 407 – 416, 1985.

[Sch88] Ph. Schnoebelen. Refined compilation of pattern-matching for functional languages. *Science of Computer Programming*, 11:133 – 159, 1988.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *ICALP'92, LNCS 623*, 1992.

[Wad87] P. Wadler. Efficient compilation of pattern-matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice Hall International, 1987.

A Prolog implementation of KEM

Alberto Artosi*, Paola Cattabriga, Guido Governatori****

**Dipartimento di Filosofia*
Università di Bologna
via Zamboni,38, 40126 Bologna (Italy)
governat@cirfid.unibo.it

***CIRFID*
Università di Bologna
via Galliera, 3, 40121 Bologna (Italy)
paola@cirfid.unibo.it

Abstract In this paper, we describe a Prolog implementation of a new theorem prover for (normal propositional) modal and multi–modal logics. The theorem prover, which is called KEM, arises from the combination of a classical refutation system which incorporates a restricted ("analytic") version of the cut rule with a label formalism which allows for a specialised, logic–dependent unification algorithm. An essential feature of KEM is that it yields a rather simple and efficient proof search procedure which offers many computational advantages over the usual tableau–based proof search methods. This is due partly to the use of linear 2–premise β rules in place of the branching β rules of the standard tableau method, and partly to the crucial role played by the analytic cut (the only branching rule) in eliminating redundancy from the search space. It turns out that KEM method of proof search is not only computationally more efficient but also intuitively more natural than other (e.g. resolution–based) methods leading to simple and easily implementable procedures (two KEM Theorem Prover–like systems have been implemented: an LPA interpreter on Macintosh, and a Quintus compiler on Sun-Sparcstation) which make it well suited for efficient automated proof search in modal logics.

1 An overview of KEM

KEM [AG94, Gov95] is a tableau–like modal proof system based on D'Agostino and Mondadori's [DM94] classical refutation system KE. The basic feature of KEM is that it uses KE rules in combination with a label unification scheme constituted of (1) a label formalism, and (2) a specialised, logic–dependent unification algorithm. The label formalism arises from two (non empty) sets $\Phi_C =$

$\{w_1, w_2, \cdots\}$ and $\Phi_V = \{W_1, W_2, \cdots\}$ respectively of constant and variable world–simbols through the following definition: a world–label is either (i) an element of the set Φ_C, or (ii) an element of the set Φ_V, or (iii) a path term (k', k) where (iiia) $k' \in \Phi_C \cup \Phi_V$ and (iiib) $k \in \Phi_C$ or $k = (m', m)$ where (m', m) is a label. Intuitively, we may think of a label $i \in \Phi_C$ as denoting a (given) world, and a label $i \in \Phi_V$ as denoting a set or worlds (any world) in some Kripke model. A label $i = (k', k)$ may be viewed as representing a path from k to a (set of) world(s) k' accessible from k (according to the appropriate accessibility relation. For any label $i = (k', k)$ we shall call k' the *head* of i, k the *body* of i, and denote them by $h(i)$ and $b(i)$ respectively. Notice that these notions are recursive: if $b(i)$ denotes the body of i, then $b(b(i))$ will denote the body of $b(i)$, $b(b(b(i)))$ will denote the body of $b(b(i))$; and so on. We shall call each of $b(i)$,$b(b(i))$, etc., a *segment* of i. Let $s(i)$ denote any segment of i (obviously, by definition every segment $s(i)$ of a label i is a label); then $h(s(i))$ will denote the head of $s(i)$. For any label i, we shall define the length of i, $l(i)$, as the number of world–symbols in i (obviously $l(s(i))$ will denote the lenght of $s(i)$). We shall call a label i *restricted* if $h(i) \in \Phi_C$, otherwise we shall call it *unrestricted.*

KEM's label unification scheme involves two kinds of unifications, respectively "high" and "low" unifications. "High" unifications are meant to mirror specific accessibility constraints. They are used to build "low" unifications which account for the full range of conditions governing the appropriate accessibility relation. Let $\Im$ denote the set of labels. A substitution is defined in the usual way as a function $\Phi_V \longrightarrow \Im^-$ where $\Im^- = \Im - \Phi_V$. For two labels i, k and a substitution σ, if σ is a unifier of i and k, then we shall say that i and k are σ-unifiable. We shall (somewhat unconventionally) use $(i, k)\sigma$ to denote both that i and k are σ-unifiable and the result of their unification. On this basis we can define several specialised, logic–dependent notions of σ "high"(or σ^L-) unification. In particular, the notion of two labels i, k being σ^K-, σ^D-, and σ^T-unifiable is defined in the following way:

$$
\begin{array}{lcl}
(i,k)\sigma^K & = & (i,k)\sigma \iff \\
 & (i) & \text{at least one of } i \text{ and } k \text{ is restricted, and} \\
 & (ii) & \text{for every } s(i),\ s(k),\ l(s(i)) = l(s(k)),\ (s(i), s(k))\sigma^K \\
 & & \\
(i,k)\sigma^D & = & (i,k)\sigma \\
 & & \\
(i,k)\sigma^T & = & (s(i),k)\sigma \iff \\
 & & l(i) > l(k), \text{ and} \\
 & & \forall h(s(i)) : l(s(i)) \geq l(k), (h(s(i)), h(k))\sigma = (h(i), h(k))\sigma \text{ or} \\
(i,k)\sigma^T & = & (i,s(k))\sigma \iff \\
 & & l(k) > l(i), \text{ and} \\
 & & \forall h(s(k)) : l(s(k)) \geq l(i), (h(i), h(s(k)))\sigma = (h(i), h(k))\sigma.
\end{array}
$$

In what follows we shall concentrate on KEM method for dealing with the B logics. To deal with these logics we need an appropriate notion of "reduction" of

(intuitively something like the deletion of "irrelevant" steps from the path represented by) a label i. Formally, the *B-reduction*, $r_B(i)$, of a label i is defined to be a function $r_L : \Im \rightarrow \Im$ determined as follows:

$$
r_B(i) = \begin{cases} b(b(i)) & i \text{ unrestricted and either } l(i)) \leq 3 \text{ or } b(i) \text{ restricted} \\ (h(i), r_B(b(i))), & i \text{ restricted} \end{cases}
$$

The notion of σ "low" (or σ_L-) unification for the B logics ($L = KB, DB, B$) can now be defined as follows:

$$
(i,k)\sigma_{KB} = (r_B(i,k))\sigma^K \quad (i,k)\sigma_{DB} = (r_B(i,k))\sigma^D \quad (i,k)\sigma_B = \begin{cases} (r_B(i.k))\sigma^D \\ (r_B(i,k))\sigma^T \end{cases}
$$

where $r_B(i,k)$ denotes either $r_B(i)$ or $r_B(k)$ or both.

The full set of KEM inference rules is constituted of (i) 1–premise α rules (the familiar linear branch–expansion rules of the tableau method) and the usual ν and π rules for the modal operators (see **Alpha Elimination**, **Ni Elimination** and **Pi Elimination** in the KEM Algorithm Representation below); (ii) 2–premise (linear) β rules (see **Beta Elimination** below); and (iii) a 0–premise branching rule called PB (for Principle of Bivalence) which plays the role of the cut rule of the sequent calculus (see **PB1** and **PB2** below). Labels are manipulated, according to these rules, in such a way that (1) in all inferences via an α rule the label of the premise carries over unchanged to the conclusion; (2) in all inferences via a ν and π rule the label of premises is "updated" to an extended new (unrestricted or restricted) label; (3) in all inferences via a β rule the labels of the premises must be σ_L-unifiable, so that the conclusion inherits their unification; and (4) for the K logics, PB is applied only to already existing restricted labels. Closure of a branch follows from the occurrence of a pair of complementary formulas whose labels are σ_L-unifiable (let us call them σ_L-complementary).

2 Implementation

In this section we will briefly consider two main problems arising from the Prolog implementation of KEM. These problems are: (1) KEM's label unification scheme has some idiosyncratic features; for example it does not allow a variable to be substituted to another variable; and (2) KEM rules are essentially non–deterministic; in particular, PB is not an *analytic* rule.

The well–known difficulty to handle variables in lists and terms in Prolog, on one hand, and the unification theory and the necessity of recursively generating new costants and variables, on the other, have made necessary to define constants and variables as functions of the form w(N) and vw(N). Thus KEM Interpreter has Φ_C = {w(1), w(2), w(3),...} and Φ_V = {vw(1), vw(2), vw(3),...}. The labels are defined as binary terms. Let us consider a KEM label $(w_4, (W_3, (w_3, (W_2, w_1))))$.

Its KEM Interpreter equivalent is i(w(4),i(vw(3),i(w(3),i(vw(2),i(w(1),w(1)))))). The unification theory is completely redefined without using the built in Prolog predicate "unify". Labels are treated as binary terms and $(i,k)\sigma$, $(i,k)\sigma^L$, $(i,k)\sigma_L$ and $r_L(i)$ are defined, using functor and arg, as ternary predicates, where the first and the second argument are i,k and the third is their unification. (For a complete descritpion of KEM Prolog implementation see [Cat95]. The Interpreter is ftp available at ftp.cirfid.unibo.it.).

The KEM–Prolog Interpreter has been based on the notion of a canonical (deterministic) KEM-tree, see [AG94, Gov95]. A KEM-tree is said to be *canonical* if it is generated by applying the rules of KEM in the following fixed order: first the 1-premise rules, then the 2-premises rules, and finally the 0-premises rule (PB). As proved in [AG94, Gov95] a KEM-tree is closed iff the corresponding canonical KEM-tree is closed, and canonical KEM-trees always terminate. Notice that in a canonical KEM-treee PB is applied only to *unanalysed* or *unfulfilled* β formulas (see **Beta Elimination** below). This allows much of the characteristic redundancy generated by the standard tableau branching rules to be eliminated from the search space.

The basic data structures ([DP94, PC94]) are provided by two sets Δ, Λ of unanalysed and analyzed formulas respectively. In Prolog Δ and Λ are lists. The Interpreter starts with the list Δ of input formulas and $\Lambda = \emptyset$, and simulates the rules of KEM by analysing and moving formulas from Δ to Λ. Each rule application produces subformulas which are added to Δ. The rules of KEM are applied until an application of the branch–closure rule (see **Closure** below) succeeds or Δ is empty. In the first case Γ is closed and unsatisfiable, in the second Γ is completed and satisfiable. The KEM algorithm runs as follows.

KEM Algorithm Representation

Δ is the list of the unanalysed formulas, Λ is the list of the analyzed formulas, "Labeltree" is a list of the generated labels, $\times$ denotes a closed branch

Analyse Literal: $? \; p \in \Delta \implies \Delta - p, \Lambda \cup p$

Closure: $? \; X, i$ and $X^C, k \in \Lambda$
$? \; (i,k)\sigma_L \implies$
$\times$

Ni Elimination: $? \; \nu, i \in \Delta \implies$
generate a new unrestricted label (i', i)
add (i', i) to Labeltree
$\Delta - \nu, i$
$\Delta \cup \nu_0, (i', i)$
$\Lambda \cup \nu, i$

Pi Elimination: $? \; \pi, i \in \Delta \implies$
generate a new restricted label (i', i)
add (i', i) to Labeltree

$\Delta - \pi, i$
$\Delta \cup \pi_0, (i', i)$
$\Lambda \cup \pi, i$

Alpha Elimination: $? \; \alpha, i \in \Delta \implies$
$\Delta - \alpha, i$
$\Delta \cup \alpha_1, i \cup \alpha_2, i$
$\Lambda \cup \alpha, i$

Beta Elimination: $? \; \beta, i \in \Delta$
$? \; \beta_1^C, k$ or $\beta_2^C, k \in \Delta \cup \Lambda$
$?(i,k)\sigma_L \implies$
$\Delta - \beta, i$
$\Delta \cup \beta_2^C, (i,k)\sigma_L$ or $\beta_1^C, (i,k)\sigma_L$
$\Lambda \cup \beta, i$

PB1: $? \; \beta, i \in \Delta$
$?$ not $(\beta_1^C, k : (i,k)\sigma_L) \in \Delta \cup \Lambda$
$?(i,m)\sigma_L$
(m is a restricted label in Labeltree) $\implies$

branch1	and	branch2
$\Delta - \beta, i$		$\Delta - \beta, i$
$\Delta \cup \beta_1, m$		$\Delta \cup \beta_1^C, m \cup \beta, i$
$\Lambda \cup \beta, i$		

PB2: $? \; \beta, i \in \Delta$
$?$ not $(\beta_2^C : k, (i,k)\sigma_L) \in \Delta \cup \Lambda$
$?(i,m)\sigma_L$
(m is a restricted label in Labeltree) $\implies$

branch1	and	branch2
$\Delta - \beta, i$		$\Delta - \beta, i$
$\Delta \cup \beta_2, m$		$\Delta \cup \beta_2^C, m \cup \beta, i$
$\Lambda \cup \beta, i$		

Modal Closure: $? \; X, i$ and $X^c, k \in \Lambda$
$?$ not $(i,k)\sigma_L$
$? \; m \in$ Labeltree, (m is a restricted label)
$? \; (i,m)\sigma_L$
$? \; (k,m)\sigma_L \implies$
$\times$

Modal Closure is an "hidden" application of PB to the the label which unifies with both the labels of the σ_L–complementary formulas.

We conclude by showing the KEM Prolog output of the characteristic axiom of B, i.e. $p \rightarrow \Box\Diamond p$.

```
::- kem(b, [~ (p-> $ (@ p))])
[i(w(1), w(1)): ~ (p-> $ (@ p))]
alpha elimination
[i(w(1), w(1)):p, i(w(1), w(1)): ~ ($ (@ p))]
```

```
literal
[i(w(1), w(1)): ~ ($ (@ p))]
pi elimination
[i(w(2), i(w(1), w(1))): ~ (@ p)]
ni elimination
[i(vw(1), i(w(2), i(w(1), w(1)))): ~ p]
literal
i(vw(1),i(w(2),i(w(1),w(1)))):~p, i(w(1),w(1)):p unify in  b
unsatisfiable in b in 10 msecs
N 1 yes
```

References

[ACG94] A. Artosi, P. Cattabriga and G. Governatori. An Automated Approach to Deontic Reasoning. In J. Breuker (ed.), *Artificial Normative Reasoning*, Workshop ECAI 1994: 132–145.

[AG94] A. Artosi and G. Governatori. Labelled Model Modal Logic. In *Proceedings of the CADE-12 Workshop on Automated Model Building*, 1994: 11–17.

[Cat95] P. Cattabriga. *Sistemi algoritmici indicizzati per il ragionamento giuridico*, PhD. Thesis, University of Bologna, 1995.

[DM94] M. D'Agostino and M. Mondadori. The Taming of the Cut. *Journal of Logic and Computation*, 4, 1994: 285–319.

[DP94] M. D'Agostino and J. Pitt. Private Communication, 1994.

[Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logic*, D. Reidel Publishing Company, Dordrecht, 1983.

[Gov95] G. Governatori. Labelled Tableaux for Multi-Modal Logics. In Peter Baumgartner, Reiner Hähnle and Joachim Posegga (eds.), *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Berlin, Springer Verlag, LNAI, 1995, 79–94.

[PC94] J. Pitt and J. Cunningham. Theorem Proving and Model Building with the Calculus KE. In *MEDLAR II, Esprit Basic research Project 6471*, Deliverable D IV.1.2-5P: 538–553.

Explicit Implementation of a Constraint Solving Mechanism in a Relational Programming System

Patrick BELLOT, Olivier CAMP, Christophe MATIACHOFF
Ecole Nationale Supérieure des Télécommunications
46 rue Barrault, 75634 Paris Cedex 13, France.
e-mail : bellot@inf.enst.fr

Abstract. The integration of logic and functional programming leads to new concepts such as relational programming and unknowns. Relational calculus can be considered as a generalization of functional calculus. A relational expression evaluates to several results instead of a single one as it is the case with a functional expression. The results are produced in a stream-like manner : when a result is computed, it is immediately passed to the continuation and the next result is computed only when the continuation is completed. Thus, a relation can virtually have an infinity of results. The unknowns are basically logic programming variables. They are used to denote objects which values are not known at the beginning of a computation. We have designed an implementation of logic programming with SLDNF semantic based on a translation from logic programs to relational programs with unknowns. In this framework, it appeared natural to try to handle constraint programming in our system. We show that the unification of two objects containing unknowns can be regarded as a constraint on these unknowns and that a solving mechanism can be coupled with the unification mechanism to allow the resolution of constraints in a very natural way. Therefore, we obtain a relational system which is as powerful as a constraint logic programming system. Moreover, the constraint solving mechanism is written in the relational language itself in order to bypass all the problems resulting from an external solver in a non-deterministic environment. This allows efficiency and flexibility. Any customized solving algorithm can be programmed this way. The real novelty of this article is the way the algorithms are implemented.

Keywords. logic programming, functional programming, constraints.

0 - Introduction

We give a brief and intuitive presentation of the concepts involved in this article: relations as a generalization of functions, how to compute relations, unknowns and constraints.

Relations. Computing using relations instead of functions is not entirely new, it has been introduced in [1,4,5,9,10,17,19,22,24,25]. A very efficient implementation was given [18] in the framework of the variableless functional programming language Graal [2,3]. [4,21] ported the concepts and the implementation in a Lisp environment named Miles. The idea was to provide some kind of oriented logic programming. If we consider a unary function f, it can be described by its graph $F=\{(x,y) / y=f(x)\}$. It has the property that if $(x,y)\in F$ and $(x,z)\in F$ then $y=z$. If such a set of pairs R does not have this property, it defines a binary relation $r(x,y)$ by $r(x,y)\Leftrightarrow (x,y)\in R$. Through a computational view, we have four possible processes: given some x_0 and y_0, check whether $r(x_0,y_0)$ holds ; given some x_0, find all the y such that $r(x_0,y)$ holds ; given some y_0, find all the x such that $r(x,y_0)$ holds ; find all the x and y such that r(x,y) holds. If the set R is not finite, these processes may give rise to infinite computations.

Computing relations. Relational programming's aim is to describe and compute such processes. The basic idea is to give tools to specify functions that may have any number of results, even an enumerable infinity. This is done by introducing a new functional form (results A B). This form splits the evaluation into two processes. The results of this expression are the results of the evaluation of A followed by those of the evaluation of B. The joint use of this form and recursion may yield an infinity of results. The results are computed one at a time and immediately given to the continuation in a stream-like manner.

Unknowns. This sentence is true for all N whatever the truth of "N is even" is. But K denotes a hypothetic value that exists only if "N is even" is true. This sentence could be more formally rewritten in a logic syntax as : $\forall N$, "N is even" $\Leftarrow \exists K$ / N=2.K. Proving "214 is even" means finding K such that 214=2.K. From a computational point of view, the variable K exists at the beginning of the computation and continues to exist until the computation finds its value. Such a place-holder has been added to our relational programming system under the name of unknown. An unknown is the denotation of an object that is not known. It exists until the value of the object is determined.

Unification. Unknowns are the relational counterpart of variables in logic programming. Our system has a unification algorithm designed as a function (unify A B) that tries to unify A and B by giving values to the unknowns they contain. If it succeeds, the unknowns are valued to make A and B syntactically identical. If it fails, the evaluation process in which the unification appears is simply aborted and produces no result. The unification algorithm is classical [23], it is the only way to valuate an unknown.

Frozen expressions. One of the most obvious problems of our system is that functions can be applied to non-ground arguments, i.e. arguments which contain unknowns. Many cases occur but an interesting one is (+ *u 1) where *u is an unvalued unknown. This cannot lead to an error because the unknown may be valued later with a numerical value. Thus we decide that the result is a frozen expression denoted as #F(+ *u 1). The frozen expression will be computed as soon as the unknown is valued.

Constraints. If we unify a frozen expression such as #F(+ *u 1) with the value 9, we generate a constraint on *u, namely that *u can only be valued with 8 to preserve coherence. The system should be able to deduce. Further, we could imagine that a frozen expression #F(+ *x *y) is unified with 12 and that #F(* *x *y) is unified with 6. This cannot be handled by the unification since the solution can only be found by a global solving technique. That is where we need to introduce constraint solving methods.

1 - The relational system Miles

Miles is based on Common Lisp [26]. It is extended by the concept of relation. A relation is a function that can deliver zero, one or more results. These multiple results should not be confused with the multiple value results of Common Lisp. A result may be the collection of a finite number of values but a relation can have any number of results, even an infinity.

The form results. The form (results A B) allows to specify multiple results. The evaluation of this forms returns the results of the evaluation of A followed by those of the evaluation of B. Of course, if A has an infinity of results, the results of B will never be computed and delivered. Example : ? (results 1 2) → 1 → 2. The form results can have any

number of parameters. If it has no parameter, then it has literally speaking no result. We say that the computation branch fails. Example: ? (results 'a (results) 'b) → a → b.

Operational semantics. The results of a relation are computed in a stream-like manner. That is to say that as soon as a result is computed, it is given to the continuation and the next result will be computed only after the completion of the continuation.

Infinity of results. Combining the form results and recursion may produce relations with an infinity of results. For instance, the relation from delivers all the integers greater than its argument : ? (defun from (N) (results N (from (1+ N)))) → from

? (* 2 (from 20)) → 40 → 42 → 44 → ...

3 - Introducing constraints

The unification of two objects can be seen as an equality constraint on these objects. For instance, (letl (*x *y) (unify (list 1 2 *x 4) (cons *y '(2 3 4))) specifies that the lists (1 2 *x 4) and (*y 2 3 4) must be identical. This type of syntactic constraint on structures is called an active constraint because it can be actively solved instead of being memorized.

Passive constraints. The unification of two numerical frozen expressions cannot be solved by the unification algorithm. This kind of constraint is called a passive constraint. For instance, (letl (*x) (unify *x (+ *x 1))) is a passive constraint that can never be satisfied. If we consider the passive constraint (letl (*x *y) (unify *x (+ *x *y))), the unification has no way to know that *y must be unified with 0 for the constraint to hold.

The problem solver. It would be nice if passive constraints were processed the same way active constraints are. The unification algorithm is not able to do this because it has no semantical knowledge. That is why we provide a user defined function to be called when the unification is unable to process a constraint. This function is named problem-solver and is called with the two members of the unification.

When to call problem-solver? It must be recalled that whenever an indefinite object is used, the evaluator replaces it by its representative. A definite object is an object that is not indefinite but may contain indefinite objects. When the two members of the unification are definite objects, the problem solver is not called on these objects. Let us examine the other cases through some examples :

Unification of two unknowns *x and *y :

- If none of them freeze an expression : a link is created from one of the unknown to the other. The problem solver is not called.
- If only one of the unknown freezes an expression : the link is created as explained. Thus, it is assimilated to the renaming of a variable. The problem solver is not called.
- If both unknowns freeze expressions : if we assume that #F(* 2 *z) and #F(+ *x *y) have been previously unified. If *x and *y are now unified, the previous constraint must be written (* 2 *z)=(* 2 *x), hence *x=*y=*z. Thus, it is a case where the problem solver should be called.

Unification of an unknown *x and a definite value V :

- If the unknown does not freeze an expression : a link is created from the unknown to the value. The problem solver is not called.
- If the unknown freezes an expression : the value of the frozen expression is modified by the new value of *x. For instance, let *x freeze the expression #F(+ *x #F(+ *y 4))

that has previously been unified with 12. The unification of *x with 2 must also unify *y with 6. Thus, the problem solver should be called.

Unification of an unknown *x with a frozen expression F :

- If the unknown does not freeze an expression: only a link is created from the unknown to the frozen expression. The problem solver is not called.
- If the unknown freezes an expression : if we assume that F is #F(* -3 *y) and that *x freezes the expression #F(+ *x *y) that has previously been unified with 10, then the global solving of the two equalities (* -3 *y)=*x and (+ *x *y)=10 must unify *x with 15 and *y with -5. Typically, the problem solver should be called.

Unification of a frozen expression F with another frozen expression G :

- If we unify #F(+ *x *y) and #F(+ *z *x), the system must be able to deduce the unification of *y and *z. The problem solver should be called.

Unification of a frozen expression F and a definite object V :

- Let us assume we unify #F(+ *x 3) and 7, the system must be able to deduce the unification of *x and 4. The problem solver should be called.

Sequencing the operations. When the problem solver has to be called, the unification has three actions to perform: to create the directed links,to compute frozen expressions that have to be unfrozen, to process the new constraint with the problem solver. If we do the three points in this order, strange things may occur since the unfreezing of frozen expressions may produce constraints that would be processed before the one currently being processed. Therefore, we have chosen the order 3-1-2 to preserve the chronology of constraints settings.

Principles of the problem solver. The solver is designed to handle equality constraints on numerical expressions. It maintains all the currently defined constraints into a canonical form that will be described. It must respect some principles [14]. Incrementality. The constraints are added one by one. Given a system of constraints S and a new constraint C, the problem solver must be able to determine the solvability of the extended system $S \cup \{C\}$. Valuation of newly known unknowns. If the constraints allow to determine the values of some of the unknowns, they must be valued. Since these values are consequences of the current set of constraints, the problem solver should not be called in this case. Compatibility with backtracking. A constraint can be set by a unification. But if the system backtracks before the unification, the constraint must be unset and all the actions of the problem solver relative to this constraint must be undone.

4 - The problem solver for numerical equality constraints

In this section, we describe a problem solver for the linear numerical equality constraints. We will show how the non-linear numerical equality constraints are integrated in our problem solver. In our solver, we just process the numerical equality constraints, thus we assume that there is a first-level dispatcher which takes the arguments of the unifications that cannot be handled by the unification algorithm and dispatchs them to the appropriate solver. For instance:

```
(defun problem-solver (A B)
  (cond ((and (is-numerical A) (is-numerical B))
         (numerical-problem-solver A B))
        (....                                   ;; other cases
```


The resolution algorithm. The resolution algorithm is designed for linear equalities. It comes from the Gauss-Jordan algorithm for solving linear systems. Gauss-Jordan has been modified to be incremental, i.e. it receives the linear equations one by one and processes them immediately. Gauss-Jordan algorithm has been preferred to Gauss algorithm because Gauss-Jordan allows the value of the unknowns to be determined as soon as possible whereas Gauss computes them only when the system is completely determined. Knowing the value of an unknown sooner may help the system to abort a wrong computation branch.

Processing a non-linear constraint. If we get a non-linear equation, the Gauss-Jordan algorithm we use is unable to process it. Therefore, we must keep the constraint elsewhere and delay its processing until it becomes linear. This can only occur when its variables are unified with numbers. In this case, we use the following function:

```
(defun process-non-linear-constraint (E)
  (letl (*flag)
    (let ((L (list-of-the-unknowns E)))
      (mapc (lambda (u) (freeze (list u) 'unify (list *flag t))) L))
      (freeze (liste *flag) 'process-constraint (list E))))
```

When this code is executed, the expression (process-constraint E) is frozen by the unknown *flag. A code (unify *flag t) is frozen by each of the unknowns in E. Therefore, if an unknown in E is valued, the function process-constraint is applied to E. This function just checks whether the valuation of the unknown has linearized E. If it had become linear, it is integrated in the system of equations as any other linear constraint. Otherwise, it remains delayed.

5 - Conclusions

We have no real measure of efficiency. The results of some trials make us think that we are between 3 and 10 times slower than CLP(R). One order of magnitude is not amazing since we are comparing two versions of (roughly!) the same algorithm, one written in a compiled procedural language whereas the other is written in an interpreted functional language. The implementation of a solving algorithm is straight. That is to say that it does not care about backtracking. Backtracking is automatically handled by the system. More precisely, when receiving a new constraint, the solver processes it without worrying about undoing the processing. The implementation of a solver is flexible. The fact that it does not have to care about backtracking makes it easy to modify, to adapt or to improve by an ordinary Lisp programmer who does not know anything about the subtleties of a Prolog implementation. The implementation is open. In order to implement an incremental Simplex algorithm for numerical inequality constraints, we just had to write it in Lisp, modify the problem-solver function (see the beginning of section 4) to recognize such constraints and hand them over to the Simplex solver. The cooperation of the two solvers is automatically provided since they share the same environment.

In conclusion, Miles is a Lisp-based programming language that takes advantage of the flexibility, simplicity and freedom of Lisp. Miles is a powerful and interesting programming language standing by itself. The initial goal of Miles was to implement a Prolog interpreter. This has been done in [4,5]. Our constraint solving mechanism appears as a constraint solver for our implementation of Prolog. Miles has been

implemented [21] at Paris Scientific Centre on the IBM 370 family of computers under VM/CMS. The Lisp interpreter is classical. Lisp programs are not penalized by the extension. The efficiency of the Prolog implementation compares to the best interpreters of Prolog we know on this type of hardware and OS.

6 - References

[1] M. BELLIA, P. DEGANO — The call-by-name semantics of a clause language with functions —in Logic Programming, K.L. Clark & S.A. Tarnlud eds, Academic Press, 1982.

[2] P. BELLOT — Sur les sentiers du Graal - Etude, conception et réalisation d'un langage de programmation sans variable — Thesis, LITP report 86-62, University P. & M. Curie, Paris 6, 1986.

[3] P. BELLOT — Graal, a functional programming system with uncurryfied combinators and its reduction machine — ESOP'86, LNCS 213, pp. 82-98, B. Robinet ed, Saarbrücken, 1988.

[4] P. BELLOT et al.. — Miles, a new step toward the integration of logic and functions —JFLA'90, C. Queinnec ed., La Rochelle, 1990.

[5] P. BELLOT, R. LEGRAND — From logic to relational calculus — ILPS'93, Global Compilation Workshop, S. Michayov & W. Winsborough eds, Vancouver, 1993.

[6] O. CAMP — Les contraintes en programmation logico-fonctionnelle : application au langage Miles — Thèse de doctorat de l'Université Pierre et Marie Curie (Paris 6), France, octobre 1994.

[7] L.I. CHUMIN — Integration of passive constraints into G-Logis — Research report, Université de Technologie de Compiègne, France, 1989.

[8] V. CHVATAL — Linear programming — W.H. Freeman & Co, 1986

[9] K.L. CLARK, S. GREGORY — A relational language for parallel programming — FPLCA'81, pp. 171-178, Arvind & Dennis eds, Portsmouth, 1981.

[10] D. DEGROOT, G. LINDSTROM — Logic programming, functions, relations and equations —Prentice Hall, 1986.

[11] D. FRIEDMAN, C.T. HAYNES — Constraining control — Report 170, Indiana Univ., 1985.

[12] P. VAN HENTENRICK — Constraint satisfaction in Logic Programming — MIT Press, 1989.

[13] C. HOLZBAUR — Extensible unification as basis for the implementation of CLP languages — Proceedings of the 6th International Workshop on Unification, pp56-60, Boston University, Massachussets, 1993

[14] J. JAFFAR, S. MICHAYLOV — Methodology and implementation of a CLP system — 4th ICLP, 1987.

[15] J. JAFFAR, J-L. LASSEZ — From unification to constraints — Séminaire de Programmation en Logique du CNET-Lannion, 1988.

[16] R. LALEMENT — Logique, réduction, résolution — Masson ed, Paris, 1990.

[17] R. LEGRAND — Calcul relationnel et Programmation en Logique — Thesis, LITP report 88-72, University P. & M. Curie, Paris 6, 1987.

[18] R. LEGRAND — Extending functional programming towards relations — ESOP'88, LNCS 300, pp. 206-220, H. Ganzinger ed, Nancy, 1988.

[19] B.J. MACLENNAN — Introduction to relational programming — FPLCA'81, pp. 213-220, Arvind & J. Dennis eds, Portsmouth, 1981.

[20] C. MATIACHOFF — Une rationalisation sémantique de la programmation logico-fonctionnelle. Application à la traduction de programmes logiques — Thèse de doctorat de l'Université Pierre et Marie Curie (Paris 6), France, décembre 1994.

[21] E. PEROTTET — Miles, fonction et logique : un langage nouveau pour une algorithmique étendue — Thesis, University P.& M. Curie, Paris 6, 1990.

[22] R.J. POPLESTONE — Relational programming — in Machine Intelligence 9, J.E. Hayes & D. Michie & L.I. Mikulich eds, 1979.

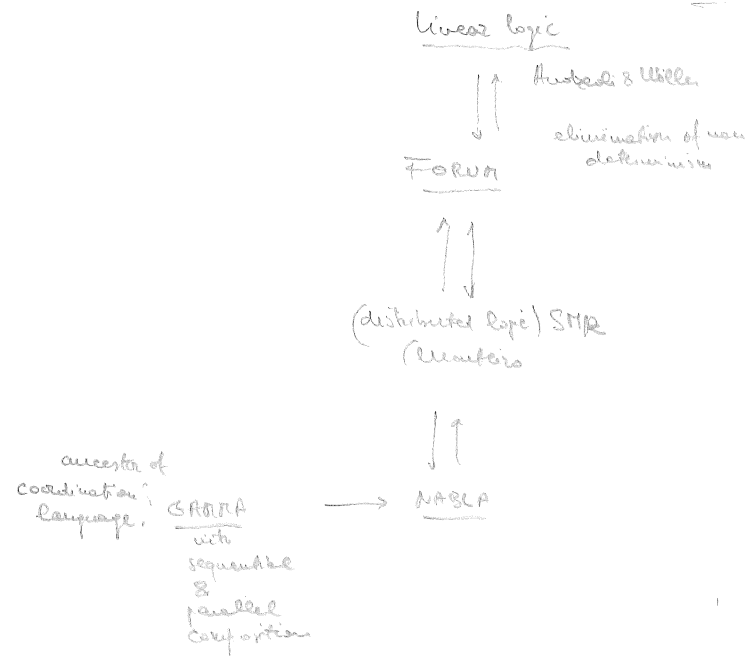
[23] J.A. ROBINSON — A Machine based on the Resolution Principle — in JACM 12, pp. 23-44, 1965.

[24] J.G. SANDERSON — A relational theory of computing — in LNCS 82, G. Goos & J. Hartmaniseds eds, Springer Verlag, 1980.

[25] G. SMOLKA — Fresh : a higher-order language based on unification — in [9].

[26] G.L. STEELE Jr — Common Lisp : the language — Digital Press, 1984.

LINEAR LOGIC



A Linear Logic Programming Language with Parallel and Sequential Conjunction

Paola Bruscoli
Università di Ancona, Istituto di Informatica, via Brecce Bianche, 60131 Ancona, Italy
e-mail: paola@di.unipi.it web: http://www.di.unipi.it/~paola fax: +39 (71) 220 4474

Alessio Guglielmi
Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy
e-mail: guglielm@di.unipi.it web: http://www.di.unipi.it/~guglielm fax: +39 (50) 887 226

Abstract *In this paper we address the issue of understanding sequential and parallel composition of agents from a logical viewpoint. We use the methodology of abstract logic programming in linear logic, where computations are proof searches in a suitable fragment of linear logic. While parallel composition has a straightforward treatment in this setting, sequential composition is much more difficult to be obtained. We define and study a logic programming language,* SMR, *in which the causality relation among agents forms a series-parallel order; top agents are recursively rewritten by series-parallel structures of new agents. We show a declarative and simple treatment of sequentialization, which smoothly integrates with parallelization, by translating* SMR *into linear logic in a complete way. This means that we obtain a full two ways correspondence between proofs in linear logic and computations in* SMR*; thus we have full correspondence between the two formalisms. Our case study is very general per se, but it is clear that the methodology adopted should be extensible to other languages and orderings more general than the series-parallel ones.*

Keywords *Concurrency, abstract logic programming, linear logic.*

1 Introduction

Linear logic [4] is a powerful and elegant framework in which many aspects of concurrency, parallelism and synchronization find a natural interpretation. The difficulties of dealing with these issues within classical logic are overcome by the linear logic approach, mainly thanks to the "resource-orientation" of its multiplicative fragment. This roughly amounts to a good treatment of logical formulas as processes, or agents, in a distributed environment [2, 7]. The richness of the calculus and the deep symmetries of its proof theory make it an ideal instrument for purposes such as language design and specification, operational semantics, and it is certainly an interesting starting point for denotational semantics investigations. We are interested here in the "(cut-free) proof search as computation" paradigm, as opposed to the "cut-elimination as computation" one.

While the parallel execution of two agents $A \mid A'$ finds a natural understanding as $A \parr A'$ (or $A \otimes A'$ in a symmetrical interpretation), the same cannot be said for their *sequential* composition $A; A'$. Yet sequential composition is a very important expressive tool and theoretical concept. We can naively achieve sequential composition in an indirect way, through backchaining. This is not satisfactory for at least two reasons: because it is an unnatural form of encoding, and because backchaining is most naturally thought of, and dealt with, as a non-deterministic tool, while sequential composition is deterministic. A major problem one encounters when trying to express sequentialization is having to make use of "continuations," which are, in our opinion, a concept too distant from a clean, declarative, logical understanding of the subject.

In this paper we offer a methodology, through a simple and natural case study, which deals with sequentiality in a way which certainly does not have the flavor of continuations.

Sequentialization is achieved in linear logic by a controlled form of backchaining, whose non-determinism is eliminated by the linearity of the calculus (linear implication) and a declarative way of producing unique identifiers (universal quantification). In our case study these two mechanisms, together with the usual $\parr$ one, are embodied in a translation with a clear declarative meaning.

We introduce the language SMR (Sequential Multiset Rewriting) and give a translation of it into linear logic which is both correct and complete, thus fully relating the two formalisms. Computing in SMR is in the logic programming style: a goal of first order atoms (*agents*) has to be reduced to empty through backchaining by clauses, thus producing a binding for variables. Goals are obtained from agents by freely composing with the two connectives $\diamond$ (*parallel*) and $\triangleleft$ (*sequential*). Every top agent, *i.e.* every agent not preceded by other agents, can give birth to a new subgoal. The declarative meaning of $A \diamond A'$ is that we want to solve problems (to prove) *A and A'*; the meaning of $A \triangleleft A'$ is that we want to solve *A and then A'*. The simplest way to introduce synchronization in this framework is having clauses of the form $A_1, \ldots, A_h \leftarrow G_1, \ldots, G_k$. They state the simultaneous replacement of top agents $A_1, \ldots, A_h$ with goals $G_1, \ldots, G_k$, respectively. This framework has been studied by Monteiro in a more complex formal system called "distributed logic" [10, 11].

It is natural to associate hypergraphs to goals: nodes are agents and hyperarcs express the immediate sequentiality relationship among agents. Thus the hypergraph relative to $G = (A_1 \diamond A_2) \triangleleft A_3 \triangleleft (A_4 \diamond A_5)$ has the two hyperarcs $(\{A_1, A_2\}, \{A_3\})$ and $(\{A_3\}, \{A_4, A_5\})$. Let us associate to every agent A_i the empty agent $\circ_i$, whose declarative meaning is "agent in position i has been solved." A natural description in linear logic of the goal G is given by the formula $\Big((A_3 \multimap (\circ_1 \parr \circ_2)) \otimes ((A_4 \parr A_5) \multimap \circ_3) \otimes (\circ_4 \parr \circ_5)\Big) \multimap (A_1 \parr A_2)$. Here indices of agents have to be thought of as unique identifiers of the position of the agent in the goal. Now we need something more: since subgoals appear during the computation as an effect of resolutions, we need a mechanism to "localize" goal descriptions in linear logic, so as to fit them to the contingent goal dynamically. Again, a natural way to do that is describing G as $\forall i_1 i_2 i_3 i_4 i_5 . \Big(\Big((A_{i_3} \multimap (\circ_{i_1} \parr \circ_{i_2})) \otimes ((A_{i_4} \parr A_{i_5}) \multimap \circ_{i_3}) \otimes (\circ_{i_4} \parr \circ_{i_5})\Big) \multimap (A_{i_1} \parr A_{i_2})\Big)$. We do not really need $\otimes$ since $(A_1 \otimes \cdots \otimes A_h) \multimap A \equiv A_1 \multimap \cdots \multimap A_h \multimap A$. It turns out that this very simple-minded idea actually works. Moreover, the $\circ$ goal behaves as a unity for $\diamond$ and $\triangleleft$, as *true* does for *and* in classical logic. Since syntax (and operational semantics) may make somewhat opaque the declarativeness of hypergraphs, which consists essentially of the precedence relations, we shall establish strong bindings between a very declarative notion of normalization for goals and the computations as they are actually performed by the linear logic engine, showing their equivalence under suitable hypotheses.

SMR is a plain generalization of Horn clauses logic programming, using $\diamond$ instead of $\wedge$. As a matter of fact, considering clauses of the form $A \leftarrow A_1 \triangleleft \cdots \triangleleft A_h$, we grasp PROLOG's left-to-right selection rule, and of course many more selection rules and much greater control over the order of execution of goals are possible.

In order to link SMR to linear logic we use a fragment of FORUM [8], which is a presentation of linear logic from an abstract logic programming perspective [9]. Its choice is rewarding because FORUM puts under control a large amount of the non-determinism of linear logic, which is something in the direction we are pursuing. We refer the reader to the conclusions for a discussion of what we feel is the meaning of this contribution. This paper is

rather picky and technical. As a matter of fact, the technique presented works in principle, but the details turned out to be more important than expected. The conference format does not help, so, at least to have some more feeling with the language and its basic mechanisms, the reader is referred to [5] for a more relaxed exposition of an earlier attempt to define and specify SMR. Sect. 2 is devoted to preliminaries and FORUM, in sect. 3 we present SMR, its operational semantics and a study of the normalization properties of goals; then, in sect. 4, the translation into FORUM is shown and correctness and completeness are stated.

2 Basic Notions and Preliminaries

The first subsection fixes the notation for some usual preliminaries. In the second one a brief exposition of the fragment of FORUM we are interested in is given.

2.1 Notation and Basic Syntax

Let S and S' be sets: Then $S \setminus S'$ stands for their difference $\{ s \in S \mid s \notin S' \}$; $\mathsf{P}(S)$ stands for the set of subsets of S and $\mathsf{P}_\mathsf{f}(S)$ stands for the set of finite subsets of S; if h is a positive integer, S^h stands for the set $\underbrace{S \times \cdots \times S}_{h}$. Given $f : S \to S'$, let $\mathrm{dom} f = S$; if $S'' \subseteq S$ define $f(S'')$ as the set $\{ f(s) \mid s \in S'' \}$.

$\mathbb{N}$ is the set of the natural numbers $\{0, 1, 2, \ldots\}$. Given $h \in \mathbb{N}$, indicate with N_h the set $\{h, h+1, h+2, \ldots\}$; given $k \in \mathbb{N}$, indicate with N_h^k the set $\mathsf{N}_h \setminus \mathsf{N}_{k+1}$. Given $h, k \in \mathbb{N}$, if $h \leqslant k$ then $e|_h^k$ stands for "$e_h, \ldots, e_k$"; if $h > k$ then $e|_h^k$ and $(e|_h^k)$ stand for the empty object "".

Given a set S, indicate with S^+ the set $\bigcup_{i \in \mathsf{N}_1} S^i$ and with S^* the set $S^+ \cup \{\epsilon_S\}$, where $\epsilon_S \notin S^+$. ϵ_S is the *empty sequence* (*of* S) and at times we shall write ϵ or nothing instead of ϵ_S. If $s \in S$ then (s) and s denote the same object. On sequences is defined a *concatenation* operator $\|$, with unity ϵ. Given the (possibly infinite) sequence $Q = (s_1, s_2, \ldots)$ and given $f : S \to S'$, if $s_1, s_2, \ldots \in S$ define $f(Q)$ as $(f(s_1), f(s_2), \ldots)$.

In the rest of the paper, we shall frequently adopt the following convention: blackboard letters (as $\mathbb{P}$, the set of programs) denote sets whose generic elements shall be denoted by the corresponding italic letter (as P, a generic program). Therefore we shall often consider implicit such statements as $P \in \mathbb{P}$. Every newly introduced syntactic symbol or class of symbols shall be considered different or disjoint from the already introduced ones, except where the contrary is explicitly stated.

$\mathbb{x}$ denotes the set of *variables*, $\mathbb{p}$ the set of *predicates* and $\mathbb{A}$ denotes the set of *first order atoms*. Given a syntactical object F, $\lceil F \rceil$ denotes the set of free variables in F.

For substitutions the usual notation and conventions apply. Let $\mathbb{s}$ denote the set of *substitutions*, $\mathbb{p}$ the set of *renaming substitutions* and let $[]$ denote the *identity substitution*.

2.2 The FORUM$^{\parr \multimap \forall}$ Presentation of a Fragment of Linear Logic

The reader can find in [8] the details missing here. Methods are called this way after [1].

The set of *methods* $\mathbb{M}$ is the least set such that: 1) $\mathbb{A} \subset \mathbb{M}$. 2) If $M, M' \in \mathbb{M}$ then $(M \parr M') \in \mathbb{M}$ and $(M \multimap M') \in \mathbb{M}$. 3) If $M \in \mathbb{M}$ and $x \in \mathbb{x}$ then $(\forall x : M) \in \mathbb{M}$.

$\parr$ associates to the left and $\multimap$ associates to the right. Instead of $(\forall x_1 : (\ldots : (\forall x_h : M) \ldots))$ we shall write $(\forall x_1 \ldots x_h : M)$. Outermost parentheses shall be omitted whenever possible. If $h \leqslant k$ and $f : \mathsf{N}_h^k \to \mathbb{M}$, the notation $\parr_{i \in \mathsf{N}_h^k} f(i)$ stands for $f(h) \parr \cdots \parr f(k)$; given

Structural rules

$$\mathsf{I}\ \frac{}{\Psi;\ |\ A \vdash A;} \qquad \mathsf{L}\ \frac{\Psi;\ \Gamma \vdash \Lambda, A;\ \Xi}{\Psi;\ \Gamma \vdash \Lambda;\ A, \Xi} \qquad \mathsf{D_L}\ \frac{\Psi;\ \Gamma\ |\ M \vdash \Lambda;}{\Psi;\ M, \Gamma \vdash \Lambda;} \qquad \mathsf{D_C}\ \frac{M, \Psi;\ \Gamma\ |\ M \vdash \Lambda;}{M, \Psi;\ \Gamma \vdash \Lambda;}$$

Left rules

$$\parr_{\mathsf{L}}\ \frac{\Psi;\ \Gamma\ |\ M \vdash \Lambda; \quad \Psi;\ \Gamma'\ |\ M' \vdash \Lambda';}{\Psi;\ \Gamma, \Gamma'\ |\ M \parr M' \vdash \Lambda \curlyvee \Lambda';}$$

$$\multimap_{\mathsf{L}}\ \frac{\Psi;\ \Gamma \vdash \Lambda;\ M \quad \Psi;\ \Gamma'\ |\ M' \vdash \Lambda';}{\Psi;\ \Gamma, \Gamma'\ |\ M \multimap M' \vdash \Lambda \curlyvee \Lambda';}$$

$$\forall_{\mathsf{L}}\ \frac{\Psi;\ \Gamma\ |\ M[t/x] \vdash \Lambda;}{\Psi;\ \Gamma\ |\ \forall x : M \vdash \Lambda;}$$

Right rules

$$\parr_{\mathsf{R}}\ \frac{\Psi;\ \Gamma \vdash \Lambda;\ M, M', \Xi}{\Psi;\ \Gamma \vdash \Lambda;\ M \parr M', \Xi}$$

$$\multimap_{\mathsf{R}}\ \frac{\Psi;\ M, \Gamma \vdash \Lambda;\ M', \Xi}{\Psi;\ \Gamma \vdash \Lambda;\ M \multimap M', \Xi}$$

$$\forall_{\mathsf{R}}\ \frac{\Psi;\ \Gamma \vdash \Lambda;\ M[x/x'],\ \Xi}{\Psi;\ \Gamma \vdash \Lambda;\ \forall x' : M,\ \Xi}{}^{*}$$

*where $x \notin \lceil \Psi;\ \Gamma \vdash \Lambda;\ \forall x' : M,\ \Xi \rceil$.

Fig. 1—The FORUM$^{\parr \multimap \forall}$ fragment of FORUM.

$g\colon \mathsf{N}_h^k \to \aleph$, the notation $\bigparr_{i \in \mathsf{N}_h^k} g(i) : M$ stands for $\forall g(h) \ldots g(k) : M$. If $\bar{M} = (M|_1^h) \in \mathbb{M}^+$ then $\bigparr \bar{M}$ stands for $M_1 \parr \cdots \parr M_h$. If $\bar{x} = (x|_1^h) \in \aleph^\star$ then $\forall\, \bar{x} : M$ stands for $\forall x_1 \ldots x_h : M$ when $h > 0$, and for M when $h = 0$.

We adopt a special kind of sequents, made up from collections of methods with different structures imposed on them: sets, multisets and sequences. Sets are used to represent information as in classical logic: this is information which does not change during the computation; a program is represented as a set of methods. Multisets are used to represent the state of the computation, which, of course, changes as the computation goes ahead; here is where linear logic has its main usefulness. Sequences of atoms appear in our sequents as a way to limit the choice in the use of right inference rules; this ordering does not affect correctness and completeness. From the proof theory point of view, sets are places where weakening and contraction rules are allowed, while on multisets and sequences these rules are forbidden. In these sequents there is place for one method (which we call "focused") which drives the choice of left inference rules.

The set of *sequents* $\mathbb{\Sigma}$ contains elements of the form $(\Psi;\ \Gamma\ |\ M \vdash \Lambda;\ \Xi)$, where $\Psi \in \mathsf{P_f}(\mathbb{M})$ (the *classical context*), Γ is a finite multiset of methods (the *left linear context*), $M \in \mathbb{M} \cup \{\epsilon_{\mathbb{M}}\}$ (the *focused method*), $\Lambda \in \mathbb{A}^\star$ (the *atomic context*) and $\Xi \in \mathbb{M}^\star$ (the *right linear context*). Instead of $(\Psi;\ \Gamma\ |\ \epsilon_{\mathbb{M}} \vdash \Lambda;\ \Xi)$ we shall write $(\Psi;\ \Gamma \vdash \Lambda;\ \Xi)$. In the following Ψ, Γ, Ξ and Λ shall stand for, respectively, sets, multisets and sequences of methods and sequences of atoms.

We outline a sequent presentation of a fragment of the FORUM inference system. FORUM imposes a discipline (wrt full linear logic) on the non-deterministic bottom-up construction of proofs, thereby drastically reducing their search space. It turns out that FORUM is equivalent to linear logic, but proofs in FORUM are uniform (see [9]). Since FORUM is much closer to the computations we are interested in, it greatly helped us in finding the way to relate them to linear logic.

The inference system we shall use as an intermediate step from SMR to linear logic is FORUM$^{\parr \multimap \forall}$, meaning that $\parr$, $\multimap$ and $\forall$ are the only logical connectives this subsystem of FORUM deals with. FORUM$^{\parr \multimap \forall}$ is presented in fig. 1. There, with $\Lambda \curlyvee \Lambda'$, we represent any sequence of atoms obtained by an ordered merge of Λ and Λ'. For example, $(A_1, A_2) \curlyvee (A_3, A_4)$ may stand for (A_1, A_3, A_2, A_4) or (A_3, A_1, A_2, A_4) or ....

The link between FORUM$^{\parr \multimap \forall}$ and linear logic is established by the following proposition, which follows from the result in [8] and the cut-elimination theorem.

2.2.1 **Theorem** *A sequent $(M|_1^h;\ \vdash\ ;\ M)$ has a proof in* FORUM$^{\parr \multimap \forall}$ *iff $(!M_1 \multimap \cdots \multimap !M_h \multimap M)$ has a proof in linear logic.*

3 Syntax and Operational Semantics of SMR

The first subsection deals with the syntax of goals, the second with their "precedence relation" semantics. In the third subsection SMR and its operational semantics are introduced.

.1 Goals, Contexts and Goal Graphs

We build up the language of goals starting from the empty goal $\circ$ and the set of atoms $\mathbb{A}$, and freely composing with the two connectives $\diamond$ and $\triangleleft$. The connectives have to be thought of as associative and non-idempotent operators; moreover, $\diamond$ is commutative and $\triangleleft$ is not. The empty goal $\circ$ behaves as a unity for $\diamond$ and $\triangleleft$, like *true* does for the classical logic connective *and*. In the translation from SMR into linear logic it shall be mapped to an atom of a special class.

Suppose, from now on, that a special 0-arity predicate $\circ$ is in $\mathbb{A}$. We shall call $\circ$ the *empty goal*. Given σ, define $\circ\sigma = \circ$.

The set of *goals* $\mathbb{G}$ is the least set such that: 1) $\mathbb{A} \subset \mathbb{G}$. 2) If $h \in \mathsf{N}_1$ and $G_1, \ldots, G_h \in \mathbb{G}$ then $\diamond(G|_1^h) \in \mathbb{G}$ and $\triangleleft(G|_1^h) \in \mathbb{G}$. $\diamond$ and $\triangleleft$ are, respectively, the *parallel* and *sequential connective*; goals of the form $\diamond(G|_1^h)$ and $\triangleleft(G|_1^h)$ are, respectively, *parallel* and *sequential goals*. A generic element of $\{\diamond, \triangleleft\}$ shall be denoted by α. Given σ, define $\sigma\colon \mathbb{G} \setminus \mathbb{A} \to \mathbb{G}$ as $\alpha(G|_1^h)\sigma = \alpha(\sigma(G|_1^h))$.

Let us extend the syntax of goals by allowing one or more *holes* $\_$ to appear in place of atoms and of $\circ$. Then we have the set $\mathbb{K}$ of *contexts*. An alternative notation for $\alpha(K|_1^h)$ is $(K_1\ \alpha \cdots \alpha\ K_h)$.

Coordinates uniquely identify occurrences of atoms, empty goals and holes in a context.

Let $\aleph = \mathsf{N}_1^\star$ be the set of *coordinates*. Let $\mathbb{o} = \mathbb{A} \times \aleph$ be the set of *agents* and let $\square = \{\_\} \times \aleph$ be the set of *places*. Instead of $(\_, \kappa)$ we shall write $\__\kappa$. A generic element of $\mathbb{o} \cup \square$ shall be denoted by a.

As defined below, to every context is associated a hypergraph whose nodes are agents or places.

A *directed hypergraph* is a couple (N, H), where N is a finite set of *nodes* and $H \subseteq (\mathsf{P}(N) \setminus \{\varnothing\})^2$ is a set of *hyperarcs*.

A *context graph* is a directed hypergraph (N, H), where $N \subset \mathbb{o} \cup \square$. Let $\mathbb{Y}$ be the set of context graphs. A context graph (N, H) such that $N \subset \mathbb{o}$ is a *goal graph*.

The "top" and "bottom" of a context graph are, respectively, the sets of agents and places which have no incoming and no outgoing hyperarc.

Define $\mathrm{top}, \mathrm{bot}\colon \mathbb{Y} \to \mathsf{P}(\mathbb{o} \cup \square)$ as $\mathrm{top}(N, H) = \{\, a \in N \mid \forall (N_1, N_2) \in H : a \notin N_2 \,\}$ and $\mathrm{bot}(N, H) = \{\, a \in N \mid \forall (N_1, N_2) \in H : a \notin N_1 \,\}$.

We now want to associate to every context a context graph which represents it. Contexts are objects recursively made up from inner contexts in two possible ways: as a parallel or as a sequential composition. In the same way a context graph representing a context is

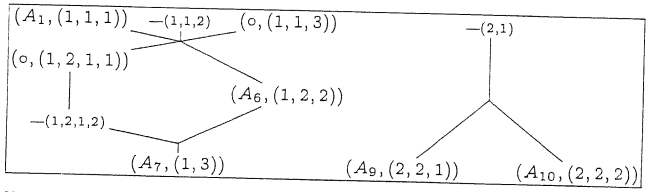
made up from the context graphs representing inner contexts. Parallel composition leads to the simple union of the context graphs; sequential composition introduces a hyperarc for every binary sequential composition. The coordinate mechanism, which is embedded in the following tricky definition, provides for a constructive way of generating distinct nodes from distinct occurrences of the same atom, empty goal or hole, in a context. Coordinates are assigned to atoms (or empty goals or holes) in the following way: the coordinate of the atom in the context is a string which records, from left to right, the positions of the contexts which contain the atom, from the outer to the inner.

For every κ define $[\cdot]_\kappa : \mathbb{K} \to \mathbb{Y}$ as

$$[K]_\kappa = \begin{cases} (\{(K,\kappa)\}, \varnothing) & \text{if } K \in \mathbb{A} \cup \{\_\} \\ (\bigcup_{i\in\mathsf{N}_1^h} N_i, \bigcup_{i\in\mathsf{N}_1^h} H_i) & \text{if } K = \diamond(K|_1^h) \text{ and } [K_i]_{\kappa\|i} = (N_i, H_i) \\ (\bigcup_{i\in\mathsf{N}_1^h} N_i, \bigcup_{i\in\mathsf{N}_1^h} H_i \cup \bigcup_{i\in\mathsf{N}_2^h} \{(\mathrm{bot}[K_{i-1}]_{\kappa\|(i-1)}, \mathrm{top}[K_i]_{\kappa\|i})\}) & \\ \qquad\qquad \text{if } K = \triangleleft(K|_1^h) \text{ and } [K_i]_{\kappa\|i} = (N_i, H_i) & \end{cases}.$$

We shall write $[K]$ instead of $[K]_\epsilon$. If $[K]_\kappa = (N, H)$, let $[K]_\kappa^{\mathsf{N}} = N$ and $[K]_\kappa^{\mathsf{H}} = H$.

The context $K = ((A_1 \diamond \_ \diamond \circ) \triangleleft ((\circ \triangleleft \_) \diamond A_6) \triangleleft A_7) \diamond (\_ \triangleleft (A_9 \diamond A_{10}))$, for example, yields

$[K] =$ 

Sometimes coordinates shall not be shown.

We write $a \lessdot K$ or $\__\kappa \lessdot K$ to say that agent a or place $\__\kappa$ appears in the context graph $[K]$.

How well do context graphs represent contexts? The following proposition can be easily proved.

3.1.1 **Proposition** *For every κ the function $[\cdot]_\kappa : \mathbb{K} \to [\mathbb{K}]_\kappa$ is bijective.*

Given K and K', $K[\underline{K'}_\kappa]$ stands for K if $\__\kappa \not\lessdot K$ and for the context obtained by K replacing $\__\kappa$ with K' if $\__\kappa \lessdot K$. We shall write $K[\underline{K_1}_{\kappa_1}, \ldots, \underline{K_h}_{\kappa_h}]$ instead of $K[\underline{K_1}_{\kappa_1}] \ldots [\underline{K_h}_{\kappa_h}]$.

For example $((A_1 \diamond \_ \diamond \circ) \triangleleft ((\circ \triangleleft \_) \diamond A_6) \triangleleft A_7)[\underline{A_2}_{(1,2)}, \underline{\circ}_{(2,1,2)}] = (A_1 \diamond A_2 \diamond \circ) \triangleleft ((\circ \triangleleft \circ) \diamond A_6) \triangleleft A_7$.

3.2 Normalization of Goals

Somewhat orthogonal to the expansion of agents is a notion of normalization (for the terminology refer, for example, to [6]). We introduce a reduction system for goals; intuitively (semantically) the reduction conserves the precedence relations among agents represented by their underlying directed hyperarcs. Since empty goals do not yield further expansions, they are discarded.

Actually there are two conceptually distinct subreductions we consider:

1) Empty goals are discarded while conserving precedence relations among other agents, as in $(\circ \diamond A_1) \triangleleft ((\circ \triangleleft A_2 \triangleleft A_3) \diamond \circ) \triangleleft A_4 \gg (A_1) \triangleleft ((A_2 \triangleleft A_3)) \triangleleft A_4$. This corresponds to $\circ$ being a unity for $\diamond$ and $\triangleleft$. A reduction of this kind shall be written as $G \succ_\circ G'$.

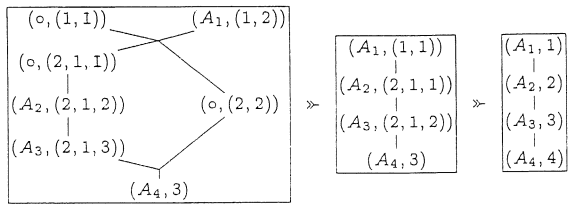


Fig. 2—Elimination of empty goals and of redundant syntax.

2) Redundant syntax is eliminated, as in $(A_1) \triangleleft ((A_2 \triangleleft A_3)) \triangleleft A_4 \gg A_1 \triangleleft A_2 \triangleleft A_3 \triangleleft A_4$. It is simply a statement of the associativity of $\diamond$ and $\triangleleft$. A reduction of this kind shall be written as $G \succ_\varsigma G'$.

The $\succ_\varsigma$ reduction prevents syntax to go too far away from our semantic requirements, which are better expressed by goal graphs. Notice that the second reduction conserves the shape of the hypergraphs, as could be shown formally. Fig. 2 represents the examples given above.

Define $\succ_G = \succ_\circ \cup \succ_\varsigma$ and let $\gg$ be the transitive reflexive closure of $\succ_G$. It can be shown with standard techniques that $\gg$ is terminating and confluent. So the normal form of a goal G under $\gg$ is unique, and shall be indicated with $\mathrm{nf}\, G$.

3.3 Clauses and Operational Semantics of SMR

SMR consists of three components: a set of programs, the set of goals we already defined and a transition relation which models the nondeterministic transformation of goals into goals. A program is a finite set of clauses. Each clause specifies the synchronous rewriting of some atoms in the top of a goal into the same number of goals. Rewriting takes place in the context of a larger goal, in which the rewritten atoms, considered as a multiset, are unifiable with the head of the clause, again considered as a multiset. The clause specifies also which goal takes the place of which atom (matching one of the atoms in its head), and the usual logic programming mechanism of instantiation with the unifier takes place. We do not insist on the unifiers being mgu's, though this special case can easily be accommodated in our setting.

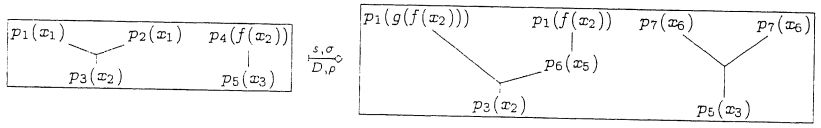
Let $\mathbb{D} = \{ (A|_1^h \leftarrow G|_1^h) \mid h \in \mathsf{N}_1,\ A_1, \ldots, A_h \in \mathbb{A} \setminus \{\circ\} \}$ be the set of (*distributed*) *clauses*. Given σ, define $\sigma : \mathbb{D} \to \mathbb{D}$ as $\sigma(A|_1^h \leftarrow G|_1^h) = (\sigma(A|_1^h) \leftarrow \sigma(G|_1^h))$.

Let $\mathbb{P} = \mathsf{P}_\mathsf{f}(\mathbb{D})$ be the set of *programs*.

The following definition needs some explanation. We want to define the set $\mathbb{s}$ of "selections." Remember that the top of a context goal, and, by extension, of a goal, is the set of agents in that goal which are preceded by no other agent. Every selection associates a unique index in N_1^h to a subset of cardinality h of the top. This is in order to associate to every atom in the head of a clause $A|_1^h \leftarrow G|_1^h$ a corresponding, selected agent in the goal to be rewritten.

Given G and $h \in \mathsf{N}_1^{|\mathrm{top}\,[G]|}$, let $\mathbb{s}_{G,h} = \{ s \mid s : T \to \mathsf{N}_1^h,\ T \subseteq \mathrm{top}\,[G],\ s \text{ is bijective} \}$ and let $\mathbb{s} = \bigcup_{G\in\mathbb{G}} \bigcup_{h\in\mathsf{N}_1^{|\mathrm{top}\,[G]|}} \mathbb{s}_{G,h}$ be the set of *selections*.

We now define the mechanism by which goals evolve by the action of clauses, in resolutions. It can be informally explained this way, given a goal G and a clause D:



$s = \{(p_2(x_1), (1,1,2)) \mapsto 2,\ (p_4(f(x_2)), (2,1)) \mapsto 1\}$
$D = (p_4(x_4), p_2(g(x_4)) \multimap p_7(x_3) \diamond p_7(x_3), p_1(x_4) \lhd p_6(x_5))$
$\sigma = [g(f(x_2))/x_1, f(x_2)/x_4]$
$\rho = [x_6/x_3]$

Fig. 3—Example of resolution (coordinates are not shown).

1) Let s be a selection of h agents in the top of G.
2) Rename apart the variables in D.
3) Let σ be a unifier between selected atoms and atoms in the head of D (corresponding to selected atoms through s).
4) Substitute, in G, the selected atoms with the goals in the body of D which correspond to them through s and the correspondence implied by their order in D. Then apply σ to the goal obtained, and have G'.

Fig. 3 shows an example. Here is the formal definition.

The relation $\mapsto\!\diamond \subset \mathbb{G}^2 \times \mathbb{s} \times \mathbb{D} \times \mathbb{p} \times \mathbb{o}$ is defined as follows: $(G, G', s, D, \rho, \sigma) \in \mapsto\!\diamond$, where $D = (A|_1^h \multimap G|_1^h)$, iff:

1) $s \in \mathbb{s}_{G,h}$.
2) $\lceil D\rho \rceil \cap \lceil G \rceil = \varnothing$.
3) $\forall (A, \kappa) \in \operatorname{dom} s : A\sigma = A_{s(A,\kappa)}\rho\sigma$.
4) Let K be such that $G = K\left[\underline{A'_1}_{\kappa_1}, \ldots, \underline{A'_h}_{\kappa_h}\right]$, where $\operatorname{dom} s = \{(A'_i, \kappa_i) \mid i \in \mathsf{N}_1^h\}$; then $G' = \sigma\left(K\left[\underline{G_{s(A'_1,\kappa_1)}\rho}_{\kappa_1}, \ldots, \underline{G_{s(A'_h,\kappa_h)}\rho}_{\kappa_h}\right]\right)$; goals $G_{s(A'_i,\kappa_i)}\rho$ are called *replacing* goals.

Given P define the relation $\overset{\diamond}{\underset{P}{\mapsto}} \subset \mathbb{G}^2 \times \mathbb{o}$ as $\overset{\diamond}{\underset{P}{\mapsto}} = \{(G, G', \sigma) \mid \exists s : \exists D \in P : \exists \rho : (G, G', s, D, \rho, \sigma) \in \mapsto\!\diamond\}$. Instead of $(G, G', s, D, \rho, \sigma) \in \mapsto\!\diamond$ and $(G, G', \sigma) \in \overset{\diamond}{\underset{P}{\mapsto}}$ we shall write $G \overset{s,\sigma}{\underset{D,\rho}{\mapsto}}\diamond G'$ and $G \overset{\sigma}{\underset{P}{\mapsto}}\diamond G'$.

A goal may evolve either because of a resolution or because of a reduction. The relation $\succ_{\mathsf{G}}$ is not suitable to be translated into FORUM, in particular the problem is with the subreduction $\succ_{\mathsf{o}}$. Then, in place of $\succ_{\mathsf{G}}$, we introduce the less declarative reduction relation $\succ_{\mathsf{T}}$, where $\succ_{\mathsf{o}}$ is replaced by $\succ_{\mathsf{H}}$. The $\succ_{\mathsf{H}}$ reduction allows collapsing empty goals appearing in the top only. This mechanism can be faithfully represented in FORUM$^{\otimes\multimap\forall}$, whereas with $\succ_{\mathsf{o}}$ this is not possible. With "successful" computations (*i.e.* computations ending in an empty goal) this only has the effect of delaying reduction of non-top empty goals until they eventually reach the top.

The relation $\succ_{\mathsf{H}} \subset \mathbb{G}^2$ is the least set such that:

1) If $G = K[\underline{\diamond(\overbrace{\circ, \ldots, \circ}^{h})}_{\kappa}]$, $h \in \mathsf{N}_1$, $\__\kappa \lessdot K$ and $\{(\circ, \kappa \parallel i) \mid i \in \mathsf{N}_1^h\} \subseteq \operatorname{top}[G]$ then $G \succ_{\mathsf{H}} K[\underline{\circ}_\kappa]$.
2) If $G = K[\underline{\lhd(\circ, G|_1^h)}_{\kappa}]$, $h \in \mathsf{N}_1$, $\__\kappa \lessdot K$ and $(\circ, \kappa \parallel 1) \in \operatorname{top}[G]$ then $G \succ_{\mathsf{H}} K[\underline{\lhd(G|_1^h)}_{\kappa}]$.

Let $\succ_{\mathsf{T}} = \succ_{\mathsf{H}} \cup \succ_{\mathsf{S}}$. For $\beta \in \{\circ, \mathsf{H}, \mathsf{S}, \mathsf{G}, \mathsf{T}\}$ let the relation $\succ_\beta \subset \mathbb{G}^2 \times \mathbb{o}$ be defined as $\{(G, G', [\,]) \mid G \succ_\beta G'\}$. Clearly $\succ_{\mathsf{T}} \subset \succ_{\mathsf{G}}$.

For every P and for $\beta \in \{\mathsf{G}, \mathsf{T}\}$, define the relation $\overset{\diamond}{\underset{P}{\succ}}_\beta \subset \mathbb{G}^2 \times \mathbb{o}$ as $\overset{\diamond}{\underset{P}{\succ}}_\beta = \succ_\beta \cup \overset{\diamond}{\underset{P}{\mapsto}}$. Instead of $(G, G', \sigma) \in \overset{\diamond}{\underset{P}{\succ}}_\beta$ we shall write $G \overset{\sigma}{\underset{P}{\succ}}_\beta G'$. Clearly $\overset{\diamond}{\underset{P}{\succ}}_{\mathsf{T}} \subset \overset{\diamond}{\underset{P}{\succ}}_{\mathsf{G}}$.

Let SMR be the triple $(\mathbb{P}, \mathbb{G}, \{\overset{\diamond}{\underset{P}{\succ}}_{\mathsf{G}} \mid P \in \mathbb{P}\})$.

Let $\beta \in \{\mathsf{G}, \mathsf{T}\}$ and $h \in \mathsf{N}$: an object $G_0 \overset{\sigma_1}{\underset{P}{\succ}}_\beta \cdots \overset{\sigma_h}{\underset{P}{\succ}}_\beta G_h$ is a (β-)*computation (by P)*, if $G_h = \circ$ it is a *successful* β-computation *of* G_0 *yielding* $\sigma_1 \cdots \sigma_h$; let $\mathbb{C}_P^\beta$ be the set of β-computations by P. A generic element of $\mathbb{C}_P^\beta$ shall be denoted by C. Let $h \in \mathsf{N}_1$ and $C = (G_0 \overset{\sigma_1}{\underset{P}{\succ}}_\beta \cdots \overset{\sigma_h}{\underset{P}{\succ}}_\beta G_h)$: for $k \in \mathsf{N}_1^h$ every object $G_{k-1} \overset{\sigma_k}{\underset{P}{\succ}}_\beta G_k$ is the kth *step* in C; if $G_{k-1} \succ_\beta G_k$ it is a *reduction* step ($\succ_\beta$-*step*), if $G_{k-1} \overset{\sigma}{\underset{P}{\mapsto}}\diamond G_k$ it is a *resolution* step ($\mapsto\!\diamond$-*step*). Define $|\cdot|_{\mathsf{R}} : \mathbb{C}_P^\beta \to \mathsf{N}$ so that $|C|_{\mathsf{R}}$ is the number of $\mapsto\!\diamond$-steps in C. Define the relation $\overset{\diamond}{\underset{P}{\succ}}_\beta \subset \mathbb{G}^2 \times \mathbb{o}$ as $\overset{\diamond}{\underset{P}{\succ}}_\beta = \{(G_0, G_h, \sigma_1 \cdots \sigma_h) \mid (G_0 \overset{\sigma_1}{\underset{P}{\succ}}_\beta \cdots \overset{\sigma_h}{\underset{P}{\succ}}_\beta G_h) \in \mathbb{C}_P^\beta\}$. Instead of $(G, G', \sigma) \in \overset{\diamond}{\underset{P}{\succ}}_\beta$ we shall write $G \overset{\sigma}{\underset{P}{\succ}}_\beta G'$.

The following two theorems are crucial to show, respectively, the correctness and the completeness of the translation from SMR into FORUM. The first is proved by transforming a successful G-computation into an equivalent T-computation, moving to the right (*i.e.* delaying) until possible all occurrences of $\succ_{\mathsf{o}}$-steps and, recursively, all $\succ_{\mathsf{S}}$-steps which depend on each other. The second amounts to an inductive construction of the desired computation.

3.3.1 **Theorem** *If C is a successful* G*-computation of G yielding σ there exists a successful* T*-computation C' of G yielding σ such that $|C|_{\mathsf{R}} = |C'|_{\mathsf{R}}$.*

3.3.2 **Theorem** *If $G \overset{\sigma}{\underset{P}{\succ}}_{\mathsf{G}} \circ$ then for every $G' \in \{G'' \mid \operatorname{nf} G'' = \operatorname{nf} G\}$ it holds $G' \overset{\sigma}{\underset{P}{\succ}}_{\mathsf{G}} \circ$.*

4 SMR and Linear Logic

We first present the translation of SMR into FORUM$^{\otimes\multimap\forall}$, then we prove that it is correct and complete wrt linear logic.

4.1 Translation of SMR into FORUM$^{\otimes\multimap\forall}$

Let us augment the set of variables by a denumerable set $\mathbb{w}$ of *process variables*, which are not allowed to appear in SMR atoms. Agents, *i.e.* atoms decorated by a coordinate, are translated into atoms. The terms inside are left untouched, and the relative position in the goal (coordinate) yields a process variable, which is appended to the resulting atom. Since atoms in SMR do not contain process variables, name clashes are avoided. The empty goal translates into a special atom of the kind $\circ(\pi)$.

Let $\circ$ be a distinguished predicate of arity 1. The function $[\![\cdot]\!] : \mathbb{p} \to \mathbb{p}$ is chosen such that it is one-one, it holds $\operatorname{ar}[\![p]\!] = \operatorname{ar} p + 1$ and $[\![\circ]\!] = \circ$.

While π stands for a generic process variable, object process variables are ψ_0, ψ_1, ... Given a coordinate κ, with $\underline{\kappa}$ we shall indicate the unique natural number associated to κ by some bijective function between coordinates and naturals. Given a denumerable set S, we shall indicate with $\langle S \rangle$ the sequence obtained by S by ordering its elements according to a total order of choice. This is alternative to using equivalence classes in the definitions to come, when order of elements is of no importance.

Define $[\![\cdot]\!] : \mathbb{o} \to \mathbb{A}$ as $[\![p(t|_1^h), \kappa]\!] = [\![p]\!](t|_1^h, \psi_{\underline{\kappa}})$. We shall write $A\psi_{\underline{\kappa}}$ instead of $[\![A, \kappa]\!]$.

We shall call atoms obtained by the translation *agents*, too. In particular, we shall refer to atoms $\circ\pi$ as *success agents*.

Let us call *elementary method* a method of the form $(A_1 \parr \cdots \parr A_h) \multimap (A'_1 \parr \cdots \parr A'_{h'})$. To every hyperarc in a goal graph corresponds an elementary method in the translation of the goal relative to the hypergraph. Two auxiliary functions are helpful.

Define $\mathrm{sa}\colon \mathbb{o} \to \mathbb{A}$ as $\mathrm{sa}(A,\kappa) = \circ\psi_{\underline{\kappa}}$ and $\mathrm{hm}\colon (\mathsf{P}_{\mathsf{F}}(\mathbb{o}) \setminus \{\varnothing\})^2 \to \mathbb{M}$ as $\mathrm{hm}(N_1,N_2) = (\bigparr\langle[\![N_2]\!]\rangle \multimap \bigparr\langle \mathrm{sa}\, N_1\rangle)$.

Given G and κ, the translation $[\![G]\!]_\kappa$ is a method $M_1 \multimap \cdots \multimap M_h \multimap M$, where $M_1, \ldots, M_h$ are the elementary methods obtained by the hyperarcs in $[G]_\kappa$, and M is the translation of $\mathrm{top}[G]_\kappa$. The structure of the hypergraph is kept by process variables (identity of agents), by the $\parr$ connective (parallelism among agents) and by the $\multimap$ connectives which appear in $M_1, \ldots, M_h$ (directionality of hyperarcs). The outer $\multimap$'s do not play a role wrt the structure of the hypergraph. They are used to "load" the left linear context with the structure. Notice that the order $M_1, \ldots, M_h$ is not important.

For every κ define $[\![\cdot]\!]_\kappa\colon \mathbb{G} \to \mathbb{M}$ as

$$[\![G]\!]_\kappa = (M_1 \multimap \cdots \multimap M_h \multimap \bigparr\langle[\![\mathrm{top}[G]_\kappa]\!]\rangle),$$

where $(M|_1^h) = \langle \mathrm{hm}[G]^{\mathsf{H}}_\kappa\rangle$.

Define $[\![\cdot]\!]\colon \mathbb{D} \to \mathbb{M}$ as

$$[\![A|_1^h \leftarrow G|_1^h]\!] = \bigwith\langle\lceil A|_1^h \leftarrow G|_1^h\rceil\rangle : \bigwith_{i\in \mathsf{N}_1^h} \psi_{\underline{i}} : (\bigparr_{i\in\mathsf{N}_1^h} [\![G_i]\!]'_i \multimap \bigparr_{i\in\mathsf{N}_1^h} A_i\psi_{\underline{i}}),$$

where, for every κ, $[\![\cdot]\!]'_\kappa\colon \mathbb{G} \to \mathbb{M}$ is defined as

$$[\![G]\!]'_\kappa = \begin{cases} G\psi_{\underline{\kappa}} & \text{if } G \in \mathbb{A} \\ \bigwith\langle\{\,\psi_{\underline{\kappa}} \mid (A,\kappa) \in [G]^{\mathsf{N}}_\kappa\,\}\rangle : ((\circ\psi_{\underline{\kappa}} \multimap \bigparr\langle \mathrm{sa}\,\mathrm{bot}[G]_\kappa\rangle) \multimap [\![G]\!]_\kappa) & \text{if } G \notin \mathbb{A} \end{cases}.$$

4.2 Correctness and Completeness of the Translation

Let $\mathsf{R}\,\dfrac{\Psi;\Gamma \vdash \Lambda; M}{\Psi;\Gamma \mid M \multimap \bigparr(M|_1^h) \vdash \Lambda \curlyvee M_1 \curlyvee \cdots \curlyvee M_h;}$ be (the scheme of) an inference rule, called *resolution*, defined as a shorthand of the following (scheme of) derivation:

$$\multimap_{\mathsf{L}}\dfrac{\Psi;\Gamma \vdash \Lambda; M \qquad \parr_{\mathsf{L}}\dfrac{\mid\dfrac{}{\Psi;\mid M_1 \vdash M_1;} \quad \vdots \quad \mid\dfrac{}{\Psi;\mid M_h \vdash M_h;}}{\Psi;\mid \bigparr(M|_1^h) \vdash M_1 \curlyvee \cdots \curlyvee M_h;}}{\Psi;\Gamma \mid M \multimap \bigparr(M|_1^h) \vdash \Lambda \curlyvee M_1 \curlyvee \cdots \curlyvee M_h;}.$$

Given P, let $(\!|G|\!)_P = \{\,([\![P]\!]; \bigparr\langle \mathrm{sa}\,\mathrm{bot}[G]\rangle, \mathrm{hm}[G]^{\mathsf{H}} \vdash A|_1^h;)\rho \mid \rho \in \mathbb{p}_{\mathbb{m}},\ \{A|_1^h\} = [\![\mathrm{top}[G]]\!]\,\}$, where $\mathbb{p}_{\mathbb{m}}$ is the set of renaming substitutions on the set of process variables. An element of $(\!|G|\!)_P$ is a *representation* of G.

The following correctness theorem establishes a first connection between SMR and FORUM.

4.2.1 **Theorem** *If $G \overset{\sigma}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$ then for every $\Sigma \in (\!|G\sigma|\!)_P$ there exists a proof Π of* $\text{FORUM}^{\parr\multimap\forall}$ *with conclusion Σ.*

Sketch of proof Let C be a successful G-computation of G yielding σ. By theorem 3.3.1 there exists a successful T-computation C' of G yielding σ. The proof is by induction on $|C'|_{\mathsf{R}} = |C|_{\mathsf{R}}$. From C' we shall build Π from bottom to top.

.1 If $|C'|_{\mathsf{R}} = 0$ then $C' = (G \succ_{\mathsf{T}} \cdots \succ_{\mathsf{T}} \circ)$. To every $\succ_{\mathsf{S}}$-step corresponds a renaming of some process variables. To every $\succ_{\mathsf{H}}$-step corresponds a sequence, from bottom to top, of a D_{L} rule (the focused method is one of the methods corresponding to a "top" hyperarc in $[G]^{\mathsf{H}}$) and of a resolution rule followed by some applications of $\parr_{\mathsf{R}}$ and L. This reduces the problem to finding a proof for $\Sigma' \in (\!|G'|\!)_P$, if $G \succ_{\mathsf{H}} G'$. Proceed inductively. The final step consists in finding a proof of $([\![P]\!]; \bigparr\langle \mathrm{sa}\,\mathrm{bot}[G'']\rangle \vdash \langle \mathrm{sa}\,\mathrm{bot}[G'']\rangle;)\rho$, which is made up of $\parr_{\mathsf{L}}$ and I rules, as the right branch in R is; here ρ is a renaming substitution on process variables.

.2 If $|C'|_{\mathsf{R}} > 0$ then $C' = (G \succ_{\mathsf{T}} \cdots \succ_{\mathsf{T}} G' \overset{\sigma'}{\underset{P}{\Longrightarrow}} G'' \overline{P}^{\circ}_{\mathsf{T}} \cdots \overline{P}^{\circ}_{\mathsf{T}} \circ)$. The first $\succ_{\mathsf{T}}$-steps are dealt with as in point 1, without the final step. We have to prove that for every $\Sigma' \in (\!|G'\sigma|\!)_P$ there exists a proof Π' of $\text{FORUM}^{\parr\multimap\forall}$ such that its conclusion is Σ'. A derivation Δ such that its conclusion is $\Sigma' \in (\!|G'\sigma|\!)_P$, relative to the step $G' \overset{s,\sigma'}{\underset{D,\rho}{\Longrightarrow}}\circ G''$, can be built bottom-up as a sequence of the following rules: D_{C} ($[\![D]\!]$ is focused), a sequence of $\forall_{\mathsf{L}}$ (D is instantiated by $\rho\sigma'$), R (the right linear context contains a representation of the replacing goals), then, by means of $\parr_{\mathsf{R}}$, $\forall_{\mathsf{R}}$, $\multimap_{\mathsf{R}}$ and L rules, the right linear context is unloaded and the left linear context is loaded with the representations of the replacing goals. In these representations process variables are either unified with previous "top" process variables (replacing goals are joined to the goal) or created unique by $\forall_{\mathsf{R}}$ (they are relative to inner coordinates in the replacing goals). This guarantees the correspondence between the representation of the new goal and its goal graph.

We have that the only premise of Δ is $\Sigma'' \in (\!|G'''|\!)_P$ and $G''' \succcurlyeq G''$. Since $G''' \succcurlyeq G'' \overset{\sigma''}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$, there is, by the induction hypothesis, a proof Π'' such that its conclusion is $\Sigma''\sigma''$, and the theorem is proved.

The other direction of the connection between SMR and FORUM is stated by the following completeness theorem.

.2.2 **Theorem** *If for $\Sigma \in (\!|G\sigma|\!)_P$ there is a proof Π of* $\text{FORUM}^{\parr\multimap\forall}$ *such that its conclusion is Σ then $G \overset{\sigma}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$.*

ketch of proof Observe that the application of rules L, $\parr_{\mathsf{R}}$, $\multimap_{\mathsf{R}}$ and $\forall_{\mathsf{R}}$ is deterministic, in the sense that a bottom-up construction of a proof every step is uniquely determined. The only choice left is that a new variable in $\forall_{\mathsf{R}}$: since we build up the proof modulo renaming of process variables, this is not important. We shall show that every possible choice of rules D_{L}, D_{C}, $\parr_{\mathsf{L}}$, $\multimap_{\mathsf{L}}$ and $\forall_{\mathsf{L}}$ leads to a proof only if they yield applications of the R rule.

If the D_{C} rule is chosen, the focused method becomes the translation $[\![D]\!]$ of a clause D. Then some applications of $\forall_{\mathsf{L}}$ are compulsory. After that, the R scheme is the only possible: it can only be part of a proof if the variables chosen in the $\forall_{\mathsf{L}}$ inferences correspond to a $\mapsto\!\circ$-step, *i.e.* to a resolution in SMR. Moreover, the clause must be applicable to the top of the goal, represented in the atomic context, then the D_{L} inference should have been wise. After R, all inferences are deterministic again. In this way we have a derivation Δ such that its conclusion is $\Sigma' \in (\!|G\sigma'|\!)_P$ and its only premise is $\Sigma'' \in (\!|G'|\!)_P$. Now it is easy to show that $G' \succcurlyeq G''$ and $G \overset{\sigma'}{\underset{P}{\Longrightarrow}}\circ G''$. Notice that, by theorem 3.3.2, if $G' \overset{\sigma''}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$ then $G'' \overset{\sigma''}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$.

If the D_{L} rule is chosen, the focused method becomes an elementary method relative to a hyperarc. The R scheme must immediately follow: it can only lead to a proof if the elementary method is applicable to the top, *i.e.* the atomic context. This is only possible if, in the top, one or more empty goals appear suitable for a $\succ_{\mathsf{H}}$-step. The exact matching of process variables, *i.e.* coordinates in the goal graph, is ensured both by the translation and the $\forall_{\mathsf{R}}$ rule. Then we obtain a derivation Δ such that its conclusion is $\Sigma' \in (\!|G|\!)_P$ and its only premise is Σ'', where $\Sigma'' \in (\!|G'|\!)_P$ and $G \succ_{\mathsf{T}} G'$.

By considering proofs modulo renaming of process variables, the proof of the theorem is easily obtained by induction on the number of the R rule applications in Π.

Then we can prove the result which tightly links SMR and linear logic:

.2.3 **Theorem** *$G \overset{\sigma}{\underset{P}{\Longrightarrow}}_{\mathsf{G}} \circ$ iff there is a proof for $(!M_1 \multimap \cdots \multimap !M_h \multimap \bigparr\langle \mathrm{sa}\,\mathrm{bot}[G]\rangle \multimap M'_1 \multimap \cdots \multimap M'_k \multimap \bigparr\langle[\![\mathrm{top}[G]]\!]\rangle)$ in linear logic, where $(M|_1^h) = \langle[\![P]\!]\rangle$ and $(M'|_1^k) = \langle \mathrm{hm}[G]^{\mathsf{H}}\rangle$.*

5 Conclusions

We obtained both a declarative and operational understanding of sequencing by associating to every task a couple of statements: 1) that the task i has to be performed by an agent

(say A_i) and 2) that when the task is accomplished a signal ($\circ_i$) is issued. The above treatment of sequentiality clearly encompasses paradigms more general than SMR. SMR by itself is a powerful language, as many examples show [10, 5]. We think also that SMR and its methodology are worthy as specification tools, and we are currently investigating their use for the specification of GAMMA [3] and other formalisms.

The translation makes use of the full $\otimes\multimap\forall$ fragment of linear logic, thus making full *logical* use of these connectives. This is opposed to, for example, classical logic programming, in which $\Rightarrow$ and $\forall$ are only used in left rules. An important point is that all structural information in SMR goes into the logic, with no need to resort to trickeries with terms. We are also pleased by the correspondence between parts in the sequences of FORUM and our framework: the program in the classical context, the structure of the goal in the left linear context and the top of the goal in the atomic context. The translation is very conservative wrt computational complexity, and FORUM guarantees good operational properties.

If we are satisfied with the translation of SMR, we certainly are not with its *logic*. We think that to fully bring sequentiality to the rank of logic some new logic with a non-commutative connective, together with commutative ones, has to be studied. At least one attempt in this direction exists, pomset logic [12], but until now this logic lacks either a cut-elimination theorem or, equivalently, a sequentialization theorem for its proof nets. Our future work shall go in the direction of investigating that logic with the aim to bring the concept of abstract logic programming [9] in a non-commutative setting, too.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[3] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, Nov. 1990.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] A. Guglielmi. Concurrency and plan generation in a logic programming language with a sequential operator. In P. Van Hentenryck, editor, *Logic Programming, 11th International Conference, S. Margherita Ligure, Italy*, pages 240–254. The MIT Press, 1994.

[6] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[7] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *1992 Workshop on Extensions to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, 1993.

[8] D. Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994.

[9] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[10] L. Monteiro. Distributed logic: A logical system for specifying concurrency. Technical Report CIUNL-5/81, Departamento de Informática, Universidade Nova de Lisboa, 1981.

[11] L. Monteiro. Distributed logic: A theory of distributed programming in logic. Technical report, Departamento de Informática, Universidade Nova de Lisboa, 1986.

[12] C. Retoré. Pomset logic. Available by anonymous ftp from cma.cma.fr, Dec. 1993.

A Structural (Meta-Logical) Semantics for Linear Objects

Giuseppe Manco

CNUCE–CNR

Via S. Maria 36, 56125 Pisa, Italy

Ph: +39 (0) 50 593348

e-mail manco@orione.cnuce.cnr.it

Franco Turini

Dipartimento di Informatica

Universitá di Pisa

Corso Italia 40, 56125 Pisa, Italy

Ph: +39 (0) 50 887253

e-mail turini@di.unipi.it

Abstract

We propose a meta-logical reconstruction of the linear logic programming language *Linear Objects*. The meta-logic is based on a CLP schema, that can handle multisets of formulas. The meta-logic provides a useful semantics for studying the structure of LO programs, and for comparing LO with other proposals in the field of computational logic.

Keywords: Constraint Logic Programming, Linear Logic, Metaprogramming.

1 Introduction

1.1 Motivations

In recent years a great amount of research has been devoted to model state-change in logic programming. In particular, the problem has been addressed within the research activities on concurrent logic programming. J. M. Andreoli and R. Pareschi [AP91b, AP91a] have defined a new language, *Linear Objects* (*LO*), that is well-suited for expressing state-change, object-identity and object-to-class inheritance,

proper of the object-oriented programming languages. *LO* finds its theoretical basis in Linear Logic, a new constructive logic investigated by Girard [Gir87, Gal92, Sce94] and mainly used in modeling typed lambda-calculus and concurrency. Furthermore, linear logic has proved useful to address state-change related problems, like planning problems, by allowing the use of consumable-reusable resources [BG94, KY93, MTV93, ACP92]. Such an approach, however, at a first sight, seems to be hardly usable to express state-change and concurrency in a logic programming framework, because of the gap between the semantics proposed for *LO* (and related languages) and the traditional semantics of logic programming, as originally proposed [VK76, AV82]. As an example, *LO* lacks a fixpoint semantics based on the definition of an immediate consequence operator [CC94]. The concurrent semantics, on the other hand, has been fully investigated [ACP92, ALPT93], because this is the computational aspect mostly stressed by the authors: "computation is performed by concurrent agents that are themselves characterized by multiple internal threads of computation" [ALPT93].

It has been argued that concurrency semantics helps in characterizing a parallel implementation of the language, which covers all its features, and that phase semantics allows a declarative formalization of the language. We find, however, that these semantics are incomplete for at least two reasons. First, concurrent semantics like CHAM [ALPT93] or IAM [ACP92] are too abstract to provide a real support for implementation, expecially in relation with the problem of selecting object properties via pattern matching (pattern-matching object selection). Secondly, phase semantics, though very elegant, is too far from the semantics of logic programming to provide a good way of expressing the "logic programming" features of the language. It has been showed [Bro93, BMPT94] that a fixpoint semantics can be a good basis for providing programming-in-the-large features to logic programming. In general, providing *LO* with a semantics related to the classical logic programming semantics can help in reusing in this new context a wealth of existing results. In particular the problem of structuring *LO* programs more in the spirit of object oriented languages could be solved in this way.

In this paper we attempt a meta-logical reconstruction of *LO*. The result is a meta-interpreter written in a meta-constraint-logic programming language, that allows us to handle *LO* programs as object programs, and to solve the pattern-matching object-selection problem by means of constraint solving. This kind of implementation allows us to provide a formal and conservative method for exploring the logic programming features of the language, and, at the same time, the use of an instance of the *CLP* scheme helps us to provide a formal framework for the object-oriented based features of the language, i.e. communication and inheritance via associativity.

The structure of the paper is as follows. Subsection 1.2 introduces the language *LO* and its formal semantics, according to [AP91b]. A formal (and brief) introduction to the *CLP* schema of programming is also provided. In sect. 2 we provide the definition of the metainterpreter. Section 3 contains the correctness result for the

proposed meta-logic. Finally, in sect. 4 we will discuss the main results of the paper.

1.2 Preliminary Notions

We will adopt the following conventions. u,v,w,x,y,z will denote object-term-variables, C,C', C_1 will denote multiset-variables, s,t will denote object-terms, p,q will denote predicate symbols, f,g will denote function symbols, $\mathcal{C}$will denote multisets. According to [JM94], where a general introduction to Constraint Logic Programming is given, a (possibly many-sorted) *signature* defines a set of function and predicate symbols and associate an arity with each symbol. If Σ is a signature, a Σ-*structure* $\mathcal{D}$ consists of a set D and an assignment of functions and relations on D to the symbols of Σ which respects the arietes of the symbols. A first order Σ-*formula* is defined as usually. A Σ-*theory* is a collection of closed Σ-formulas, and a *model* of a Σ-theory T is a Σ-structure $\mathcal{D}$ such that all formulas of T evaluate to *true* under the interpretation provided by $\mathcal{D}$. A *primitive constraint* has the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \Sigma$ is a predicate symbol. Every *constraint* is a formula built from primitive constraints.

As we noted previously, the *LO* language wiews the computation as performed by concurrent agents that are themselves characterized by multiple concurrent internal threads of computation. Agents can self-replicate, and their communications can be performed either via context-sharing or variable-sharing. *LO*'s operators correspond to Linear Logic connectives [Gir87]. Hence, *LO* can be wiewed as a "linear logic programming language". In Particular, LO is defined by the following abstract syntax:

Method	::=	**Head** $\circ\!\!-$ **Body.**
Head	::=	**A** | **A** $\parr$ **Head**
Body	::=	$\top$| **A** | **Body** $\&$ **Body** |
		Body $\parr$ **Body**

An *LO* program can be defined as a collection of method formulas (*program formulas*), and a goal can be defined as a body formula (*resource formulas*); a context is a finite multiset of resource formulas.

EXAMPLE 1 *In the following we show a three-clauses LO program and a context made up by resource-formulas.*

$$p \parr a \circ\!\!- r \,\&\, (q \parr a).$$
$$q \parr a \parr b \circ\!\!- \top.$$
$$r \parr b \circ\!\!- \top.$$

$$p \parr a, q \,\&\, t$$

□

An LO sequent is a pair written as $\mathcal{P} \vdash \mathcal{C}$, where $\mathcal{P}$ is a program and $\mathcal{C}$ is a context. The linear interpretation for an LO sequent is the formula $\&\,(\mathcal{P}) \multimap \parr\,(\mathcal{C})$. A proof is a tree structure whose nodes are labeled with LO sequents. We say that there exists a proof for a sequent $\mathcal{P} \vdash \mathcal{C}$ if there is a proof tree whose leaves are labeled by empty sequents, the root is labeled by the sequent itself and the branches are obtained as instances of the following inference figures.

- Decomposition

$$\frac{\mathcal{P} \vdash \mathcal{C}, R_1, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \parr R_2}\,[\parr] \qquad \frac{}{\mathcal{P} \vdash \mathcal{C}, \top}\,[\top] \qquad \frac{\mathcal{P} \vdash \mathcal{C}, R_1 \quad \mathcal{P} \vdash \mathcal{C}, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \,\&\, R_2}\,[\&]$$

- Progression

$$\frac{\mathcal{P} \vdash \mathcal{C}, R}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_n}\,[\multimap] \qquad \text{if } (A_1 \parr \cdots \parr A_n \multimap R) \in \mathcal{P}$$

Notice that, by definition, the elements of a multiset are not ordered. Therefore, the order of the atoms in the left-hand side of a program formula is not relevant.

The central point of our discussion concerns the way of modeling dynamics in Object-Oriented Logic Programming provided by LO. A computation can be identified with a proof-search, and a proof tree, when read bottom-up, can be seen as a trace of a computation. Each branch of a proof represents the evolution of an object: the nodes represent a snapshot of the object state, while the edges represent the object state-transition.

EXAMPLE 2 *The following proof-tree shows a computation for the sequent $\mathcal{P} \vdash p, a \parr b$, where $\mathcal{P}$ represents the program of example 1.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{P} \vdash \top}\,[\top]}{\mathcal{P} \vdash b, r}\,[\multimap] \quad \dfrac{\dfrac{\dfrac{}{\mathcal{P} \vdash \top}\,[\top]}{\mathcal{P} \vdash b, q, a}\,[\multimap]}{\mathcal{P} \vdash b, q \parr a}\,[\parr]}{\mathcal{P} \vdash b, r \,\&\, (q \parr a)}\,[\&]}{\mathcal{P} \vdash p, a, b}\,[\multimap]}{\mathcal{P} \vdash p, a \parr b}\,[\parr]$$

2 leaves of empty formulas

Computation begins within the context $p, a \parr b$. After having enriched the context via application of the $[\parr]$ inference figure, a state transition is performed and, subsequently, a fork operation splits the proof-tree in two branches, which represent now two independent concurrently evolving agents. □

A method can be triggered by the object if its state contains all the resources in the head of the method (inference figure $[\multimap]$). In this case, the object may perform a transition to a new state obtained by replacing the resources of the head in the old state. The sequentiality of the operation is guaranteed in this context by the application of the linear implication operator. The *with* operator allows object clonation and instance creation, while the *par* operator simply adds new components in the context of the object.

EXAMPLE 3 *We give some explanation of ex. 1. The first clause lets us perform a state-transition from a context containing formulas p, a to a new context where the formulas are repalced by the formula $r \,\&\, (q \parr a)$. The formula brings to the clonation of the new contex for the $\&$ connective and by way of inference figure $[\parr]$); once cloned, an instance of the context is enriched with the subcontext q, a.*

The second and third clauses allow process-termination for agents containing the matching subcontexts.

The context shown allows process clonation and context enrichment. □

Notice that there are mainly two kinds of nondeterminism related to the system. The don't know nondeterminism, which is due to the search-rule, and the nondeterminism due to the context selection: given two (or more) methods and a context which contains information available for both, either one method can be applied (*competitive nondeterminism*).

Describing operational semantics of the language by means of inference figures can help to define a meta-program which explains the behaviour of such a language. In fact, inference figures can be shown to have a correspondance with the kernel clauses of the metainterpreter. So we can use the meta-interpreter to semantically characterize the object-language. In a more precise way, once established a "proof-theoretic" semantics for the object language $\mathcal{L}$, we can express the provability relation of such language with a metaprogram written in the metalanguage $\mathcal{M}$. The model of such a program can be used as a semantics for $\mathcal{L}$:

- given a function ρ mapping a program $\mathcal{P}$ in $\mathcal{L}$ to is meta-representation $\mathcal{V}_\mathcal{P}$ in $\mathcal{M}$;
- given a set of clauses $\mathcal{K}$, kernel of the metainterpreter, defining the predicate $demo(g)$ representing the provability relation of g in $\mathcal{L}$

we reach a correct and complete formalization of the object language $\mathcal{L}$ if we can prove that, given a goal g and its metarepresentation G,

$$\mathcal{P} \vdash_\mathcal{L} g \Leftrightarrow demo(G) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow \omega$$

Such a result proposes $\mathcal{K}$ as a real formal semantics for $\mathcal{L}$, because in $\mathcal{K}$ we can establish the properties of $\mathcal{L}$. Moreover, metalogical axioms show that the logic language $\mathcal{L}$ can be expressed within logic programming itself, and that Logic Programming is a formalism with an expressive power at least similar to that of $\mathcal{L}$.

To conclude this brief introduction of the background material, we recall the semantics definitions for $CLP(\mathcal{X})$ ($\mathcal{X}$ stands for a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$, where Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas representing the class of constraints, and $\mathcal{T}$ is a first-order Σ-theory representing an axiomatization of the properties of $\mathcal{D}$).

Let the pair $\langle \mathcal{D}, \mathcal{L} \rangle$ be the constraint domain. Assume that the binary predicate symbol $\doteq$ is contained in Σ and is interpreted as identity in $\mathcal{D}$. A *valuation* is a mapping from terms to D and formulas to closed $\mathcal{L}$-formulas. A $\mathcal{D}$-interpretation of a formula is an interpretation of the formula with the same domain as $\mathcal{D}$ and the same interpretation for the symbols in Σ as $\mathcal{D}$.

We present a fixed-point semantics by defining the usual consequence operator over $\mathcal{D}$-interpretations:

$$
\begin{array}{lll}
p(t) \in T_P^{\mathcal{D}}(I) & \text{iff} & \exists p(s) \leftarrow p_1(s_1), \ldots, p_n(s_n) \in P, \\
& & \{p_1(t_1) \ldots, p_n(t_n)\} \subseteq I \wedge \\
& & \exists \text{ a valuation } v \text{ in } \langle \mathcal{D}, \mathcal{L} \rangle \text{ such that} \\
& & \mathcal{D} \models v(s) \doteq t \wedge v(s_1) \doteq t_1 \wedge \ldots \wedge v(s_n) \doteq t_n
\end{array}
$$

2 The Meta Definition

The CLP scheme defines a class of logic programming languages, each of which is obtained by specifying the constraint system. In general, a CLP scheme is obtained by replacing the unification algorithm, which solves constraints over finite trees in $T_\Sigma(V)$, with a more general (or more specialistic) method of constraint resolution. In our case, the problem is to deal with object selection via pattern-matching, i.e. multiple selection in a multiset of formulas. It is the case of the $[\circ\!-]$ inference rule. A CLP instance can help us in such a problem. By defining a new unification algorithm, which can deal with multiset-union and complex-context unification, we can define a metainterpreter with the main properties of LO. The new constraint solver must now deal with multisets of finite trees over $T_\Sigma(V)$.

EXAMPLE 4 *Consider the following constraints over multisets of formulas:*

$$
\begin{array}{l}
C = \{a, p(t)\} \uplus \{q(x), a\} \\
\{a, b, c\} = \mathcal{C} \uplus \{c\} \\
\{p(x), a(x, y)\} \uplus C = C_1 \uplus \{a(c, z), q(s)\}
\end{array}
$$

The instance of CLP we want to consider is defined over a constraint solver which can solve such constraints. the solver has to produce the following answers to these constraints:

$$
\begin{array}{l}
C = \{a, p(t), q(x), a\} \\
C = \{a, b\} \\
C = \{q(s)\}, C_1 = \{p(c)\}, x = c
\end{array}
$$

□

Let Σ contain the function symbols and constants $\emptyset$ (*empty multiset*), $\{\}$ (*multiset constructor*) and $\uplus$ (*multiset union*). Let D be the set of multisets over finite trees, where each node of each tree is labelled by a constant or a function symbol, and the number of children of each node is the arity of the label of the node. Let $\mathcal{D}$ interpret the function symbols $\emptyset$,$\{\}$,$\uplus$ as their usual meaning over finite trees, and the other function symbols of Σ as tree constructors, where each $f \in \Sigma$ of arity n maps n trees to a tree whose root is labelled by f and whose subtrees are the arguments of the mapping. The primitive constraints are equations between terms of the same sort, and let $\mathcal{L}$ be the constraints generated by these primitive constraints. So our constraint domain is $\langle \mathcal{D}, \mathcal{L} \rangle$.

We now define the kernel of the metainterpreter and the function ρ of interpretation for the object programs.

Definition 2.1 *The following rules define the kernel $\mathcal{K}$ of the metainterpreter of LO.*

$$
agent(C \uplus \top). \tag{1}
$$

$$
agent(C \uplus \{x \,\&\, y\}) \leftarrow agent(C \uplus \{x\}), \; agent(C \uplus \{y\}). \tag{2}
$$

$$
agent(C \uplus \{x \parr y\}) \leftarrow agent(C \uplus \{x, y\}). \tag{3}
$$

$$
agent(C \uplus C') \leftarrow method(C' \circ\!- z), \; agent(C \uplus \{z\}). \tag{4}
$$

Moreover, given an LO program $\mathcal{P}$, we define the function ρ, as follows:

- $\rho(R \circ\!- B) = method(\|R\| \circ\!- B)$, *where*
  $\|A_1 \parr \cdots \parr A_n\| = \{A_1, \ldots A_n\}$

□