A Memoizing Semantics for Functional Logic Languages

Salvador España and Vicent Estruch

Departamento de Sistemas Informáticos y Computación-DSIC
Technical University of Valencia, C. de Vera s/n, 46022 Valencia, Spain.
{sespana,vestruch}@dsic.upv.es

Abstract. Declarative multi-paradigm languages combine the main features of functional and logic programming, like laziness, logic variables and non-determinism. The operational semantics of these languages is based on a combination of narrowing and residuation. In this article, we introduce a non-standard *memoizing* semantics for multi-paradigm declarative programs and prove its equivalence with the standard operational semantics. Both pure functional and pure logic programming have for long time taken advantage of tabling or memoizing schemes [15,19,7], which motivates the interest in the adapation of this technique to the integrated paradigm.

Keywords: Programming languages, formal semantics, memoization.

1 Introduction

Declarative multi-paradigm languages [11] (like Curry [10]) combine the main features of functional and logic programming. In comparison with functional languages, such integrated languages are more expressive thanks to the ability to perform function inversion and to implement partial data structures by means of logical variables. With respect to logic languages, multi-paradigm languages have a more efficient operational behaviour since functions allow more deterministic evaluations than predicates.

The operational semantics of these languages is usually based on a combination of two different operational principles: narrowing and residuation [12]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation (by rewriting). On the other hand, the *narrowing* mechanism allows the instantiation of variables in input expressions and, then, applies reduction steps to the function calls of the instantiated expression. Each function specifies a concrete *evaluation annotation* (see [12,10]) in order to indicate whether it should be evaluated by residuation (for functions annotated as *rigid*) or by narrowing (for *flexible* functions). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [5] is currently the best narrowing strategy for multi-paradigm functional logic programs. The formulation of needed narrowing is based on the use of *definitional trees* [3], which define a strategy to evaluate functions by applying narrowing steps. Recently, [13] introduced a *flat* representation for functional

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 109–123, 2004.

logic programs in which definitional trees are embedded in the rewrite rules by means of case expressions. The interest in using the flat representation arises because it provides more explicit control (hence the associated calculus is simpler than needed narrowing), while source programs can be still automatically translated to the new representation.

Several techniques from pure functional and pure logic languages have been extended to implement multi-paradigm languages. In particular, both pure paradigms have exploited memoization[1] to memoize sub-computations and reuse their results later. In some situations [15] a reduction of the asymptotic cost can be obtained[2]. The advantages of memoization have long been known to the functional programming community [15,8]. Tamaki and Sato [19] proposed an interpretation for logic programming based on memoization; this seminal paper has stimulated a large body of work [7,18,16,6]. In addition to reuse previous computed sub-goals, infinite paths in the search space can be detected, enabling better termination properties [7].

These properties motivate the adaptation of the memoization technique to the integrated functional logic paradigm. This adaptation is not straightforward due to some differences between functional evaluation and narrowing. Firstly, in addition to the computed normal form, functional logic programs also produce a computed answer. Secondly, non-determinism of narrowing computations leads to a large, possibly infinite, set of results arising from different sequences of narrowing steps. Previous work attempting to introduce memoization in narrowing [4] focused in finding a finite representation of a (possibly infinite) narrowing space by means of a graph representing the narrowing steps of a goal.

In this article, we define a new *memoizing* semantics for flat programs, the MLNT calculus, and prove its equivalence with the standard operational semantics.

The rest of the paper is organized as follows. In the next section we briefly introduce a *flat* representation for multi-paradigm functional logic programs and its operational semantics based on the LNT (Lazy Narrowing with definitional Trees) calculus [13]. Section 3 presents the proposed memoizing semantics and proves the completeness w.r.t. LNT calculus. Finally, Section 4 offers some conclusions.

2 The Flat Language

This section briefly introduces a *flat* representation for multi-paradigm functional logic programs and its standard operational semantics. Similar representations are considered in [13,14,17]. Unlike them, here we distinguish two kinds of case expressions in order to make also explicit the *flexible/rigid* evaluation annotations of source programs. The syntax for programs in the flat representation is as follows:

[1] Other terms (tabling, caching, etc.) are used to refer to the same concept too.

[2] Memoization can be viewed as an automatic technique for applying dynamic programming optimization [9].

$$
\begin{array}{lll}
\mathcal{R} ::= D_1 \ldots D_m & & \text{(program)} \\
\mathcal{D} ::= f(\overline{x_n}) = e & & \text{(rule)} \\
e \;::= x & & \text{(variable)} \\
\quad\quad | \; c(\overline{e_n}) & & \text{(constructor)} \\
\quad\quad | \; f(\overline{e_n}) & & \text{(function call)} \\
\quad\quad | \; \mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & & \text{(rigid case)} \\
\quad\quad | \; \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & & \text{(flexible case)} \\
\quad\quad | \; e_1\ \mathit{or}\ e_2 & & \text{(disjunction)} \\
p \;::= c(\overline{x_n}) & & \text{(flat pattern)}
\end{array}
$$

Here, we write $\overline{o_n}$ for the sequence of objects $o_1, \ldots, o_n$. Thus, a program $\mathcal{R}$ consists of a sequence of function definitions D such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is an expression e composed of variables ($\mathcal{X}$), constructors ($\mathcal{C}$), function calls ($\mathcal{F}$), and case expressions for pattern-matching. Variables are denoted by $x, y, z \ldots$, constructors by $a, b, c \ldots$, and defined functions by $f, g, h \ldots$ The general form of a case expression is:

$$(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots ; c_k(\overline{x_{n_k}}) \to e_k\}$$

where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions (possibly containing case structures). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding expression e_i. The difference between *case* and *fcase* only shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing). Functions defined only by *fcase* (resp. *case*) expressions are called *flexible* (resp. *rigid*).

Without loss of generality, we will assume some restrictions in flat programs in the following: on the one hand, all (f)*case* arguments are variables; on the other hand, *or* and (f)*case* expressions appear at the outermost positions i.e., they do not appear inside function and constructor calls. Every flat program can be automatically transformed to one which satisfies these restrictions.

Example 1. The following flat program will be used to illustrate some examples throughout the paper. It defines some functions on natural numbers which are represented by terms built from `Zero` and `Succ`. `double` is defined by means of the addition, `coin` nondeterministically computes `Zero` or `Succ(Zero)`. `add` is the arithmetic addition and `leq` defines the relation "less than or equal to".

```
double(x) = add(x, x)
coin      = Zero or Succ(Zero)
add(x, y) = fcase x of {Zero → y; Succ(m) → Succ(add(m, y))}
leq(x, y) = fcase x of {Zero → True;
                        Succ(m) → fcase y of {Zero → False;
                                              Succ(n) → leq(m, n)}}
```

Case Select

$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \Rightarrow_{id} [\![\sigma(e'_i)]\!]$ if $p_i = c(\overline{x_n})$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$

Case Guess

$[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow_{\sigma} [\![\sigma(e_i)]\!]$ if $\sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$

Case Eval

$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow_{\sigma} [\![\sigma((f)case\ e'\ of\ \{\overline{p_k \to e_k}\})]\!]$
if $[\![e]\!] \Rightarrow_{\sigma} [\![e']\!]$, $e \notin \mathcal{X}$, and $root(e) \notin \mathcal{C}$

Function Eval

$[\![f(\overline{e_n})]\!] \Rightarrow_{id} [\![\sigma(e')]\!]$ if $f(\overline{x_n}) = e' \in \mathcal{R}$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$

Or

$[\![e_1\ or\ e_2]\!] \Rightarrow_{id} [\![e_i]\!], i = 1, 2$

Fig. 1. LNT calculus

The operational semantics of flat programs is based on the LNT (Lazy Narrowing with definitional Trees) calculus [13]. In Figure 1, we present a slight extension of this calculus in order to cope with case expressions including evaluation annotations and disjunction; nevertheless, we still use the name "LNT calculus" for simplicity. First, let us note that the symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]$ are purely syntactical (i.e., they do not denote "the value of e"). Indeed, they are only used to *guide* the inference rules. LNT steps are labelled with the substitution computed in the step. The empty substitution is denoted by id. Let us briefly describe the LNT rules:

Case Select. It is used to select the appropriate branch of the current case expression.

Case Guess. It non-deterministically selects a branch of a flexible case expression and instantiates the variable at the case argument to the appropriate constructor pattern. The step is labelled with the computed substitution σ. Rigid case expressions with a variable argument *suspend*, giving rise to an abnormal termination.

Case Eval. It is used to evaluate case expressions with a function call or another case expression in the argument position. Here, $root(e)$ denotes the outermost symbol of e. This rule initiates the evaluation of the case argument by creating a (recursive) call for this subexpression.

Function Eval. This rule performs the unfolding of a function call. As in logic programming, we assume that rules are renamed so that they only contain fresh variables.

Or. It non-deterministically selects a choice of a disjunction expression.

Arbitrary LNT *derivations* are denoted by $e \Rightarrow^*_{\sigma} e'$ which is a shorthand for the sequence of steps $e \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} e'$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). We say that a LNT derivation $e \Rightarrow^*_{\sigma} e'$ is *successful* when e' is in *head normal form* (i.e., it is rooted by a constructor symbol) or it is a variable; in this case, we say that e evaluates to e' with *answer* σ. This calculus can be easily extended to evaluate expressions to normal form, but we keep the above presentation in both calculi for simplicity. In the rest of the paper, a *value* will denote a term in head normal form or a variable.

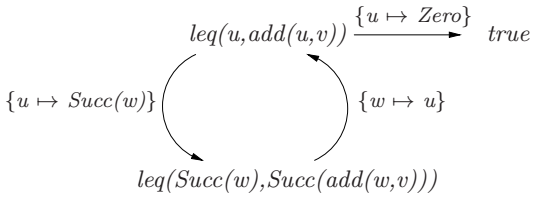


Fig. 2. Graph representation of the narrowing space of the goal $leq(u, add(u, v))$

3 MLNT Calculus

Previous work aiming at introducing memoization in narrowing [4] focused in finding a finite representation of a (possibly infinite) narrowing space by means of a graph representing the narrowing steps of a goal. Let us consider the example of Figure 2 extracted from [4]. This figure shows the graph representation[3] of the narrowing space of the goal $leq(u, add(u, v))$.

The vertices of this graph are goals i.e., terms being narrowed, and the edges are narrowing steps between these goals. Terms are considered the same vertex if they differ only by a renaming of variables. Solutions can be obtained by composing the set of substitutions which appear in any path in the graph whose origin is the goal vertex and whose destination vertex is a value.

Our proposal is based in a graph representation too, but in contrast to [4], every subgoal is explicitly introduced in the graph in order to take advantage of memoization. Therefore, the graph represents no longer the search space of the original goal, but also contains the search space of *all derived subgoals*. The sets of results are pairs composed by a value and a computed answer.

Furthermore, search spaces from different goals may share common parts in order to further improve memoization. For instance, consider the narrowing space of the goal $double(coin)$ shown in Figure 3. The term $add(coin, coin)$ denoted by `v2` requires subgoal *coin* at position 1 to be reduced, thus a new vertex `v3` associated to *coin* is introduced in the graph. Later, `v2` is reduced to `v3`. Therefore, `v3` has played a double role.

The proposed calculus consists of a state transition system MLNT where a MLNT-State is a 4-tuple from

$$Eval \times Ready \times Suspended \times Graph$$

Eval may be a special symbol *void* or an *EvaluationState*. An *EvaluationState* is a 3-tuple composed by a goal to be solved, a flat expression and a substitution.

Non-determinism (for instance, variable instantiation in a $(f)case$) is addressed by considering every possible choice and creating a different *EvaluationState* associated to it. Note that despite non-determinism, *only one* graph is

[3] For simplicity, we omitted some intermediate steps which basically correspond to $(f)case$ reductions.

considered which memoizes the information of all these branches as soon as they are computed.

Since the computation process must pursue only a single branch at a given moment, we maintain a set of ready *EvaluationState*s which are stored to be selected later. This set is the second component of a MLNT-State.

Every goal evaluation that needs a subgoal to be solved is suspended and stored in a "suspended table". This table maps every suspended *EvaluationState* to the set of subgoal's solutions already used to continue this *EvaluationState*. These sets of solutions are needed to avoid resuming the suspended *EvaluationState* several times with the same subgoal's solution.

Given an initial goal e, the initial MLNT-state associated to it is the tuple: $\langle void, \{\langle e, e, id\rangle\}, \emptyset, \emptyset\rangle$. A sequence of steps is applied and the calculus proceeds until *Eval* is *void*, the set *Ready* is empty and no rules can be applied.

The aim of this calculus is to obtain a graph which contains the original goal e as a vertex and also value vertices connected to it by a path. The edges of the graph are labelled with substitutions. The composition of substitutions in a path, in reverse order, gives the computed answer. This scheme can be easily adapted to evaluate expressions to normal form.

Let us briefly describe the MLNT rules shown in Figure 4. (sel) and (sol) are the only ones which overlap and the only non-deterministic too. They may be applied to a MLNT-state with *void* in the *Eval* field:

(sel) This rule takes an *EvaluationState* element of the set *Ready* and puts it in the field *Eval* to be used later by other rules.

(sol) A solution may be generated whenever a path appears in the graph from a goal to a value vertex. A new solution is searched in the graph for some vertex v corresponding to a goal appeared in the $(f)case$ argument of some suspended *EvaluationState*. A new *EvaluationState* is created from the suspended one by replacing its goal argument by the solution previously found, and is stored in the set *Ready* to be selected later. The mapping *Suspended* is updated to reflect the fact that this solution has already been used.

The rest of the rules are applied when *Eval* is not *void*, they are non-overlapping and deterministic:

(or) The outermost position of the second component of *Eval* is an *or* expression. This rule breaks this expression into two who are introduced in the set *Ready* to be selected later.

(val) The second component of *Eval* must be a value (head normal form or a variable) in order to apply this rule. A new edge in the graph is inserted connecting the goal being solved to the value.

(goal) The outermost position of the second component of *Eval* is a function call $f(\overline{t_n})$. This expression is always a term because of the restrictions imposed to flat programs. An edge is added from the goal being solved to this

function call when they are different. If the goal was not in the graph, a new *EvaluationState* $Unfold(f(\overline{t_n}))$ is introduced in the set *Ready* to be selected later.

The three following rules (casec), (casef) and (casev) correspond to the case where an expression of the form $(f)case\ e\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$ is at the outermost position of the second component of *Eval*.

(casec) This rule is used when e in the above expression is constructor rooted (with root c_i). The corresponding pattern $c_i(\overline{x_{n_i}})$ is selected and matched against e. The associated expression e_i replaces the current expression of the field *Eval*. Other fields remain unchanged.

(casef) In this case, e is a function call $f(\overline{t_n})$ which must be computed. The current *EvaluationState* of field *Eval* is stored in the mapping *Suspended*. The evaluation $\langle f(\overline{t_n}), f(\overline{t_n}), id\rangle$ is also introduced in the set *Ready*.

(casev) This rule is applied to a *fcase* expresion whenever e is an unbound variable. This variable may be bound to every $c_i(\overline{x_{n_i}})$ pattern. Thus, a new *EvaluationState* is inserted in the set *Ready* for every possible variable instantiation. This set of *EvaluationState*s is finite and equal to the arity of the *fcase* expression. Note that this rule does not consider a rigid *case*, the evaluation is suspended whenever this situation is produced.

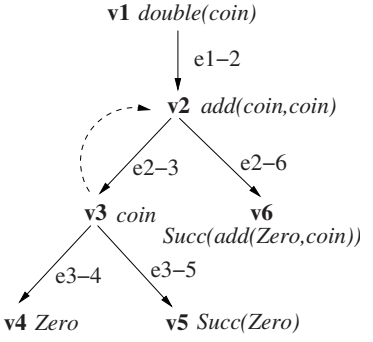
To illustrate this calculus, a trace of *double*(*coin*) is shown in Figure 3. The following domains and auxiliary functions are used to simplify the MLNT rules shown in Figure 4:

Eval: is $(EvaluationState \cup \{void\})$
EvaluationState: $Term \times Expr \times Substitution$.
Ready: $2^{EvaluationState}$.
Suspended: set of partial mappings from *EvaluationState* to $2^{Substitution \times Term}$.
Graph: set of partial mappings from *Term* to $2^{Substitution \times Term}$ a set of pairs ⟨edge label, destination vertex⟩. $Vertices(G)$ will denote $Dom(G)$

The partial mappings Suspended and Graph described above consider terms modulo variable renaming.

ObtainSolution: Searches a path in the graph from a given vertex to a value vertex. Returns a pair ⟨computed answer,value ⟩, where the computed answer is obtained by composing the path's sequence of substitutions (taken in reverse orden). This function is non-deterministic, but it must guarantee that every solution will be found eventually.

UnionMap: This function performs the union of two partial mappings and is used to update *Suspended* and *Graph* mappings. $Dom(UnionMap(G,H)) = Dom(G) \cup Dom(H)$. $UnionMap(G,H)(x) = G(x)$ if $x \notin Dom(H)$ and $UnionMap(G,H)(x) = H(x)$ if $x \notin Dom(G)$. $UnionMap(G,H)(x) = G(x) \cup H(x)$ whenever both exists. For instance, $UnionMap(G, \{t \to \emptyset\})$ adds vertex t to G.



ev1 = ⟨double(coin), double(coin), {}⟩
ev2 = ⟨double(coin), add(coin, coin), {}⟩
ev3 = ⟨add(coin, coin), *fcase* coin *of* {Zero → coin; Succ(m) → Succ(add(m, coin))}, {}⟩
ev4 = ⟨coin, coin, {}⟩
ev5 = ⟨coin, Zero *or* Succ(Zero), {}⟩
ev6 = ⟨coin, Zero, {}⟩
ev7 = ⟨coin, Succ(Zero), {}⟩
ev8 = ⟨add(coin, coin), *fcase* Zero *of* {Zero → coin; Succ(m) → Succ(add(m, coin))}, {}⟩
ev9 = ⟨add(coin, coin), coin, {}⟩
ev10 = ⟨add(coin, coin), *fcase* Succ(Zero) *of* {Zero → coin; Succ(m) → Succ(add(m, coin))}, {}⟩
ev11 = ⟨add(coin, coin), Succ(add(Zero, coin)), {}⟩

Step	Rule	Eval	Ready	Suspended	Graph
		void	{ev1}	∅	∅
1	sel	ev1	∅	∅	∅
2	goal	*void*	{ev2}	∅	⟨{v1}, {}⟩
3	sel	ev2	∅	∅	⟨{v1}, {}⟩
4	goal	*void*	{ev3}	∅	⟨{v1,v2}, {e1-2}⟩
5	sel	ev3	∅	∅	⟨{v1,v2}, {e1-2}⟩
6	casef	*void*	{ev4}	{ev3 → {}}	⟨{v1,v2}, {e1-2}⟩
7	sel	ev4	∅	{ev3 → {}}	⟨{v1,v2}, {e1-2}⟩
8	goal	*void*	{ev5}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
9	sel	ev5	∅	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
10	or	*void*	{ev6, ev7}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
11	sel	ev6	{ev7}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
12	val	*void*	{ev7}	{ev3 → {}}	⟨{v1,v2,v3,v4}, {e1-2,e3-4}⟩
13	sel	ev7	∅	{ev3 → {}}	⟨{v1,v2,v3,v4}, {e1-2,e3-4}⟩
14	val	*void*	∅	{ev3 → {}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
15	sol	*void*	{ev8}	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
16	sel	ev8	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
17	casec	ev9	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
18	goal	*void*	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
19	sol	*void*	{ev10}	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
20	sel	ev10	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
21	casec	ev11	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
22	val	*void*	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5,v6}, {e1-2,e3-4,e3-5,e2-3,e2-6}⟩

Fig. 3. Trace of the goal *double*(*coin*). Bottom table corresponds to the sequence of MLNT-states. The graphs in this sequence are represented as pairs of sets: The first component is the set of vertices. The second component is the set of edges. Both sets use the notation of the graph in the top-left part of the figure. Top-left is the search space graph; dashed edges represent dependence relations between goals and suspended evaluations in these vertices; edges are no labeled since this example has no variables. Top-right table corresponds to *EvaluationState*s used in the trace.

Rule	*Eval*	*Ready*	*Suspended*	*Graph*
(sel)	void	R	S	G
	$\Rightarrow e$	$R - \{e\}$	S	G
	where $e \in R$			
(sol)	void	R	S	G
	$\Rightarrow \mathit{void}$	$R \cup \{e\}$	S'	G
	where $s = \langle t, (f)\mathit{case}\ f(\overline{t_n})\ \mathit{of}\ \{\overline{p_n \to e_n}\}, \sigma\rangle \in \mathit{Dom}(S)$			
	$f(\overline{t_n}) \in \mathit{Vertices}(G)$			
	$\langle \varphi, r\rangle = \mathit{ObtainSolution}(G, f(\overline{t_n})) \notin S(s)$			
	$e = \langle t, (f)\mathit{case}\ r\ \mathit{of}\ \{\overline{p_n \to \varphi(e_n)}\}, \varphi \circ \sigma\rangle$			
	$S' = \mathit{UnionMap}(S, \{s \to \{\langle \varphi, r\rangle\}\})$			
(or)	$\langle t, e_1\ \mathit{or}\ e_2, \sigma\rangle$	R	S	G
	$\Rightarrow \mathit{void}$	R'	S	G
	where $R' = R \cup \{\langle t, e_1, \sigma\rangle, \langle t, e_2, \sigma\rangle\}$			
(val)	$\langle t, \mathit{val}, \sigma\rangle$	R	S	G
	$\Rightarrow \mathit{void}$	R	S	G'
	where val in head normal form, $G' = \mathit{UnionMap}(G, \{t \to \{\langle \sigma, \mathit{val}\rangle\}\})$			
(goal)	$\langle t, f(\overline{t_n}), \sigma\rangle$	R	S	G
	$\Rightarrow \mathit{void}$	R'	S	G'
	where			
	$G' = \begin{cases} \mathit{UnionMap}(G, \{t \to \{\langle \sigma, f(\overline{t_n})\rangle\}\}) & \text{if } t \neq f(\overline{t_n}) \vee \sigma \neq id \\ \mathit{UnionMap}(G, \{t \to \emptyset\}) & \text{otherwise} \end{cases}$			
	$R' = \begin{cases} R \cup \{\mathit{Unfold}(f(\overline{t_n}))\} & \text{if } f(\overline{t_n}) \notin \mathit{vertices}(G) \\ R & \text{otherwise} \end{cases}$			
(casec)	e	R	S	G
	$\Rightarrow e'$	R	S	G
	where $e = \langle t, (f)\mathit{case}\ c(\overline{b_k})\ \mathit{of}\ \{\overline{p_n \to e_n}\}, \sigma\rangle$			
	$p_i = c(\overline{x_k})$ is the pattern that matches $c(\overline{b_k})$			
	$e' = \langle t, \varphi(e_i), \varphi \circ \sigma\rangle,\ \varphi = \{\overline{x_k \mapsto b_k}\}$			
(casef)	e	R	S	G
	$\Rightarrow \mathit{void}$	R'	S'	G
	where $e = \langle t, (f)\mathit{case}\ f(\overline{t_n})\ \mathit{of}\ \{\overline{p_n \to e_n}\}, \sigma\rangle$			
	$S' = \mathit{UnionMap}(S, \{e \to \emptyset\}),\ R' = R \cup \{\langle f(\overline{t_n}), f(\overline{t_n}), id\rangle\}$			
(casev)	e	R	S	G
	$\Rightarrow \mathit{void}$	R'	S	G
	where $e = \langle t, \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_n \to e_n}\}, \sigma\rangle$			
	$\varphi_i = \{x \mapsto p_i\},\ \forall i = 1, \ldots, n$			
	$R' = R \cup \{\langle t, \varphi_i(e_i), \varphi_i \circ \sigma\rangle : i \in \{1, \ldots, n\}\}$			

Fig. 4. MLNT Calculus

Unfold: Given a function call $f(\overline{t_n})$, creates a new *EvaluationState* $\langle f(\overline{t_n}), e, \sigma\rangle$ where e is obtained by unfolding $f(\overline{t_n})$ and σ is the associated substitution. Example: *Unfold*(`add(coin, coin)`) is:

$$\langle \texttt{add(coin, coin)},\ \textit{fcase}\ \texttt{coin}\ \textit{of}\ \{\ \texttt{Zero} \to \texttt{coin};\ \texttt{Succ(m)} \to \texttt{Succ(add(m, coin))}\},\ \texttt{id}\rangle$$

Theorem 1 (Completeness). *Let e be an expression, e' a value, and $\mathcal{R}$ a flat program. For each LNT derivation $e \Rightarrow^*_\sigma e'$ in $\mathcal{R}$, there exists a sequence of MLNT transition steps $\langle void, \{\langle e, e, id\rangle\}, \emptyset, \emptyset, \emptyset\rangle = st_1 \Rightarrow st_2 \Rightarrow \ldots \Rightarrow st_k = \langle void, R, S, G\rangle$ such that G contains e and e' as vertices connected by a path $e = e_1, \ldots, e_r = e'$, and the composition of edge labels of this path (taken in reverse order) $label(e_{r-1}, e_r) \circ \cdots \circ label(e_1, e_2)$ is the computed answer σ. Where $label(edge)$ denotes the substitution that labels an edge.*

Informally speaking, we would like to reason by induction over the length of the LNT derivation relating the application of a rule in LNT calculus to the application of one or more steps in MLNT calculus. This is not easy because the graph in the MLNT derivation is not updated every step. Only the MLNT rules (val) and (goal) update the graph (a composition of substitutions is stored in every *EvaluationState* and is used later to label the edges created by these rules). This is the reason why the proof is decomposed in three stages. First, we demonstrate for arbitrary sequences of LNT rules Case Select, Case Guess and Or. In a second stage, the LNT rule Function Eval is also considered. Finally, the LNT rule Case Eval is included.

Note also that memoization should be taken into account in this proof. For instance, whenever the MLNT rule (goal) is applied, the unfolding of the corresponding function call expression is inserted into the set *Ready* only the first time this function is to be evaluated (the graph allows the MLNT state transition system to check it). For simplicity, the proof of Lemma 2 considers only the case when every function call appears in the graph for the first time. The general case should take into account *EvaluationState*s that remain in the set *Ready* but which are not taken into account in the induction steps of the proof.

Lemma 1. *Let e and e' be two expressions and $\mathcal{R}$ a flat program. For each (not necessarily successful) LNT derivation $e \Rightarrow^*_\sigma e'$ in $\mathcal{R}$ which only uses the rules* Case Select, Case Guess *and* Or, *there exists a sequence of MLNT transition steps $\langle\langle h, e, \varphi\rangle, R, S, G\rangle = st_1 \Rightarrow \ldots \Rightarrow st_k = \langle\langle h, e', \sigma \circ \varphi\rangle, R', S, G\rangle$.*

Proof. The proof proceeds by induction on the length n of this LNT derivation.

Base case $(n = 0)$. Trivial.

Inductive case $(n > 0)$. Assume that the LNT derivation has the form

$$e \Rightarrow_\theta e^a \Rightarrow^*_\gamma e'$$

where $\sigma = \gamma \circ \theta$. Now, we distinguish several cases depending on the applied rule in the first step:

Case Select. Then, e has the form $[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!]$ and $e^a = [\![\theta(e'_i)]\!]$ if $p_i = c(\overline{x_n})$ and $\theta = \{\overline{x_n \mapsto e_n}\}$. A single MLNT step suffices: $\langle\langle h, (f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}, \varphi\rangle, R, S, G\rangle \Rightarrow_{casec} \langle\langle h, \theta(e'_i), \theta\circ\varphi\rangle, R, S, G\rangle$.

Case Guess. Then, e has the form $[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]$ and $e^a = [\![\theta(e_j)]\!]$, where $\theta = \{x \mapsto p_j\}$ for some $j \in \{1, \ldots, k\}$. In this case the corresponding MLNT steps are: $\langle\langle h, e, \varphi\rangle, R, S, G\rangle \Rightarrow_{casev} \langle void, R^I, S, G\rangle \Rightarrow_{sel} \langle\langle h, e^a, \theta \circ \varphi\rangle, R^{II}, S, G\rangle$.

Case Or. In this case, e has the form $[\![e_1\ or\ e_2]\!]$ and, in one LNT-step, non-deterministically we get $e^a = e_i$, $i \in \{1, 2\}$ with $\theta = id$. In this case the corresponding MLNT steps are: $\langle\langle h, e_1\ or\ e_2, \varphi\rangle, R, S, G\rangle \Rightarrow_{or} \langle void, R^I, S, G\rangle \Rightarrow_{sel} \langle\langle h, e^a, \varphi\rangle, R^{II}, S, G\rangle$, where $R^I = R \cup \{\langle e, e_1, \varphi\rangle, \langle e, e_2, \varphi\rangle\}$ and the *EvaluationState* with the adequate e_i is selected by rule (sel). □

Lemma 2. *Let e and e' be two expressions and $\mathcal{R}$ a flat program. For each (not necessarily successful) LNT derivation $e \Rightarrow^*_\sigma e'$ in $\mathcal{R}$ which does not use the rule* Case Eval*, there exists a sequence of MLNT transition steps $\langle\langle h, e, \varphi\rangle, R, S, G\rangle = st_1 \Rightarrow \ldots, \Rightarrow st_k = \langle\langle h', e', \varphi'\rangle, R', S, G'\rangle$ such that there exist a path in G' from h to h' and $\varphi' \circ \sigma' \circ \varphi = \sigma \circ \varphi$, where σ' is the composition of edge labels of this path (taken in reverse order).*

Proof. If the rule Function Eval has not been used in the LNT derivation, it suffices to apply Lemma 1. Now, let us suppose that Function Eval has been used at least once in the derivation. In this case, the derivation $e = e_1 \Rightarrow_{\sigma_1} e_2 \Rightarrow_{\sigma_2} \ldots \Rightarrow_{\sigma_{n-1}} e_n = e'$ may be expressed:

$$e_1 \Rightarrow^*_{\varphi_1} e_{i_1} \Rightarrow_{\theta_1} e_{i_1+1} \Rightarrow^*_{\varphi_2} e_{i_2} \Rightarrow_{\theta_2} e_{i_2+1} \ldots e_{i_k+1} \Rightarrow^*_{\varphi_{k+1}} e_{i_{k+1}} = e_n$$

where $e_{i_j} \Rightarrow_{\theta_j} e_{i_j+1}$, $j = 1, \ldots, k$ are the only steps where the rule Function Eval has been applied. Therefore, by Lemma 1, the derivations $e_{i_j+1} \Rightarrow^*_{\varphi_{j+1}} e_{i_{j+1}}$ are equivalent to MLNT-sequences:

$$\langle\langle h_j, e_{i_j+1}, \rho_j\rangle, R_j, S, G_j\rangle \Rightarrow^* \langle\langle h_j, e_{i_{j+1}}, \varphi_{j+1} \circ \rho_j\rangle, R'_j, S, G_j\rangle$$

Now, it suffices to join these sequences of MLNT transitions by applying the MLNT-rules (goal) and (sel) whenever LNT rule Function Eval has been applied, so that $e_{i_j} \Rightarrow_{\theta_j} e_{i_j+1}$ is associated to $\langle\langle h_j, e_{i_{j+1}}, \varphi_{j+1} \circ \rho_j\rangle, R^I_j, S, G_j\rangle \Rightarrow_{goal} \langle void, R^{II}_j, S, G_{j+1}\rangle \Rightarrow_{sel} \langle\langle h_{j+1}, e_{i_{j+1}+1}, \rho_{j+1}\rangle, R_{j+1}, S, G_{j+1}\rangle$. □

Lemma 3. *Let e be an expression, $\mathcal{R}$ a flat program and e' a value. For each LNT derivation $e \Rightarrow^*_\sigma e'$ in $\mathcal{R}$, there exists a sequence of MLNT transition steps $\langle\langle h, e, \varphi\rangle, R, S, G\rangle = st_1 \Rightarrow \ldots \Rightarrow st_k = \langle\langle h', e', \varphi'\rangle, R', S, G'\rangle$ such that there exist a path in G' from h to h' and $\varphi' \circ \sigma' \circ \varphi = \sigma \circ \varphi$, where σ' is the composition of edge labels of this path (taken in reverse order).*

Proof. The proof proceeds by induction on the number of times the rule Case Eval has been used.

Base case. Lemma 2.

Inductive case. Assume that the LNT derivation has the form

$$e = e_1 \Rightarrow_{\sigma_1} e_2 \Rightarrow_{\sigma_2} \ldots \Rightarrow_{\sigma_{n-1}} e_n = e'$$

and the rule Case Eval is used in the derivation. There is a maximal subsequence of k contiguous Case Eval derivations:

$$e_{i+1} \Rightarrow_{\sigma_i+1} \ldots e_{i+k-1} \Rightarrow_{\sigma_{i+k-1}} e_{i+k}$$

where $e_{i+j} = [\![(f) case\ e^j\ of\ \{\overline{p_m \rightarrow e_m}\}]\!],\ j = 0, \ldots, k$
Note that $k < n$ since e_n is a value and, by the condition of maximality, e^k is also a value. Therefore, the sequence

$$e^1 \Rightarrow_{\sigma_{i+1}} e^2 \Rightarrow_{\sigma_{i+2}} \ldots \Rightarrow_{\sigma_{i+k}} e^k$$

is also a *successful derivation* of length less than n, therefore the induction hypothesis can be applied: $\langle\langle h, e_1, \varphi\rangle, R, S, G\rangle \Rightarrow^* \langle\langle h', e_{i+1}, \varphi'\rangle, R^I, S^I, G^I\rangle$ $\Rightarrow_{case f}\ \langle void, R^{II}, S^{II}, G^I\rangle\ \Rightarrow_{sel}\ \langle\langle e^1, e^1, id\rangle, R^{III}, S^{II}, G^I\rangle\ \Rightarrow_{Ind.\ Hypothesis}$ $\langle\langle h'', e^k, \varphi''\rangle, R^{IV}, S^{III}, G^{II}\rangle \Rightarrow_{val} \langle void, R^{IV}, S^{III}, G^{III}\rangle\ \Rightarrow_{sol}\ \langle\langle h''', e_{i+k}\varphi'''\rangle,$ $R^V, S^{IV}, G^{III}\rangle \Rightarrow^* \langle void, R^{VI}, S^V, G^{IV}\rangle$
such that G^{IV} contains the path $e_1, \ldots, e_n$. □

Proof (of Theorem 1). $\langle void, \{\langle e, e, id\rangle\}, \emptyset, \emptyset\rangle \Rightarrow_{sel} \langle\langle e, e, id\rangle, \emptyset, \emptyset, \emptyset\rangle$
$\Rightarrow_{Lemma3} \langle\langle h, e', \varphi\rangle, R, S, G'\rangle \Rightarrow_{val} \langle void, R, S, G\rangle$ □

Note that a successful LNT derivation obtains a single solution whereas MLNT can obtain a set of solutions and is complete. Despite these differences, soundness may be formulated as follows:

Theorem 2 (Soundness). *Let e be a goal, $\mathcal{R}$ be a flat program and let $st_1 = \langle void, \{\langle e, e, id\rangle\}, \emptyset, \emptyset, \emptyset\rangle \Rightarrow \ldots \Rightarrow \langle void, R, S, G\rangle = st_k$ be a sequence of MLNT transition steps. For every path $e = e_1, \ldots, e_r = e'$ in G such that e' is a value and $label(e_{r-1}, e_r) \circ \cdots \circ label(e_1, e_2) = \sigma$, there exists a successful LNT derivation $e \Rightarrow^*_\sigma e'$*

Proof. The proof is decomposed in two parts. First, we introduce the notion of *restricted* subsequence of MLNT transition steps associated to an edge and show how this sequence can be obtained. This sequence allow us to relate MLNT and LNT steps in a more direct manner than the original MLNT sequence[4].

In a second part, we obtain a sequence of LNT steps from the restricted sequence of MLNT transition steps. This part proceeds by induction when MLNT

[4] The sequence of MLNT and LNT steps cannot be related in a direct way for several reasons. On the one hand, some MLNT steps are used to construct parts of the graph which are not needed to obtain the solution e' (they may be used to construct other solutions). On the other hand, MLNT steps may appear in many different arrangements. For instance, steps associated to different solutions or even to different subgoals might be interleaved.

rules (casef) and (sol) does not appear in the sequence. This special case is a base case for a recursive, constructive, procedure that covers the general case.

Given an edge of G, a *restricted* subsequence[5] of MLNT transition steps $m_1, \ldots, m_p$ associated to this edge is composed by those steps from the original sequence $st_1, \ldots, st_k$ satisfying the following conditions:

- The step is needed to construct the edge assuming that the origin vertex already exists and the *EvaluationState* associated to the unfolding of this edge is in the set *Ready* in the first MLNT state of the restricted sequence.
- The steps needed to construct subgoals are not included.

In order to obtain a restricted sequence of an edge, note that only MLNT rules (val) and (goal) update the graph and both create an edge (and the destination vertex, if necessary). This step can be traced back obtaining previous *EvaluationState*s until a (val) or (goal) rule has been applied. Whenever a (sel) rule is used to obtain an *EvaluationState* e, we need to search for the previous step that put e in the set *Ready*. If the rule that put e was (sol), we consider the (casef) step that inserted the *EvaluationState* in *Suspended* that later was used to create e by (sol).

In the second part of the proof, we will relate a *restricted* sequence of MLNT steps to the corresponding LNT steps. In a first stage, we prove by induction for sequences where (casef) and (sol) rules have not been applied. We distinguish several cases depending on the applied rule in the first step. We show here only two rules, since they are similar Lemma 1:

(or) Since the sequence is *restricted*, this step is always followed by a (sel) step: $\langle\langle t, e_1 \; or \; e_2, \sigma\rangle, R, S, G\rangle \Rightarrow_{or} \langle void, R \cup \{\langle t, e_1, \sigma\rangle, \langle t, e_2, \sigma\rangle\}, S, G\rangle \Rightarrow_{sel} \langle\langle t, e_i, \sigma\rangle, R \cup \{\langle t, e_j, \sigma\rangle\}, S, G\rangle$ where $i, j \in \{1, 2\}$, $i \neq j$. The corresponding sequence of LNT steps is a single Case or step: $[\![e_1 \; or \; e_2]\!] \Rightarrow_{id} [\![e_i]\!]$.

(casev) This step is also followed by a (sel) step: $\langle\langle t, fcase \; x \; of \; \{\overline{p_k \rightarrow e_k}\}, \sigma\rangle, \sigma\rangle, R, S, G\rangle \Rightarrow_{casev} \langle void, R', S, G\rangle \Rightarrow_{sel} \langle\langle t, \varphi_i(e_i), \varphi_i \circ \sigma\rangle, R'', S, G\rangle$ where $i \in \{1, \ldots, k\}$. The corresponding sequence of LNT steps is a single Case Guess step: $[\![fcase \; x \; of \; \{\overline{p_k \rightarrow e_k}\}]\!] \Rightarrow_{\varphi_i} [\![\varphi_i(e_i)]\!]$.

By simple concatenation of the associated LNT steps, we can prove the result for any path in the graph which does not need pairs of (casef) and (sol) rules.

In a second stage, the rules (casef) and (sol) are considered. This part is proved in a constructive manner using recursion. The base case (there are no pairs of (casef) and (sol) rules) is guaranteed because there is a partial dependence relation between goals and subgoals. This case corresponds to the second stage.

Let us consider the transition steps $\langle\langle h, (f)case \; r_1 \; of \; \{\overline{p_k \rightarrow e_k}\}, \theta\rangle, R, S, G\rangle \Rightarrow_{casef} \langle void, R', S', G\rangle \Rightarrow_{sol} \langle void, R'', S'', G'\rangle \Rightarrow_{sel} \langle\langle h, (f)case \; r_j \; of \; \{\overline{p_k \rightarrow e_k}\}, \varphi \circ \theta\rangle, R''', S'', G'\rangle$ where $r_1 = f(\overline{t_n})$ is a goal. The rule (sol) uses a tuple $\langle\varphi, r\rangle$ given by *ObtainSolution* which has traversed a path $r_1, \ldots, r_j$ in G. By recursion, we obtain a sequence of LNT steps $s_1 \Rightarrow^* s_p$ associated to

[5] By subsequence we do not mean contiguous.

$r_1, \ldots, r_j$. Now, it suffices to create the following LNT steps to obtain the LNT sequence associated to the MLNT transition steps:

$$\begin{array}{r} (f)case\ s_1\ of\ \{\overline{p_k \to e_k}\} \\ \Rightarrow (f)case\ s_2\ of\ \{\overline{p_k \to e_k}\} \\ \vdots \\ \Rightarrow (f)case\ s_p\ of\ \{\overline{p_k \to e_k}\} \end{array}$$

and the claim follows. □

4 Conclusions

This work presents a non-standard memoizing semantics for functional logic programs (for a class of flat programs, with no loss of generality) and demonstrates the equivalence w.r.t. a standard operational semantics for such programs. This coud provide the theoretical basis for a complete sequential implementation of a functional logic language with memoization.

This work considers a pure memoization approach where every goal is memoized. In some situations, memoizing all subgoals is not feasible from a practical point of view. Therefore, an obvious extension is a mixed approach which allows memoizing only some subgoals, as is done in most logic programming systems [18].

Other extensions to this work could consider the computed graph which may be used for other purposes such as partial evaluation [1] and debugging [2].

Acknowledgements. We gratefully acknowledge Germán Vidal for many useful questions, suggestions and helpful discussions.

References

1. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
2. M. Alpuente, F. J. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
3. S. Antoy. Definitional Trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, volume Springer LNCS 632, pages 143–157, 1992.
4. S. Antoy and Z. M. Ariola. Narrowing the narrowing space. In *9th Int'l Symp. on Prog. Lang., Implementations, Logics, and Programs (PLILP'97)*, volume Springer LNCS 1292, pages 1–15, 1997.
5. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Journal of the ACM (JACM)*, volume 47, pages 776–822. ACM Press New York, NY, USA, 2000.

6. J. Barklund. Tabulation of Functions in Definite Clause Programs. In Manuel Hermenegildo and Jaan Penjam, editors, *Proc. of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, pages 465–466. Springer Verlag, 1994.
7. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.
8. B. Cook and J. Launchbury. Disposable Memo Functions. *ACM SIGPLAN Notices*, 32(8):310–318, 1997.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.
10. M. Hanus. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry` (2000).
11. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
12. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
13. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
14. T. Hortalá-González and E. Ullán. An abstract machine based system for a lazy narrowing calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS'2001)*, volume Springer LNCS 2024, pages 216–232, 2001.
15. J. Hughes. Lazy memo-functions. Proc. of the 2nd Conference on Functional Programming Languages and Computer Architecture (FPCA). *Lecture Notes in Computer Science*, 201:129–146, 1985.
16. M. Leuschel, B. Mertens, and K. Sagonas. Preserving termination of tabled logic programs while unfolding. in proc. of lopstr'97: Logic program synthesis and transformation, n. fuchs,ed. lncs 1463:189–205, 1997.
17. W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.
18. I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, 1999.
19. H. Tamaki and T. Sato. Old Resolution with Tabulation. In *3rd International Conference on Logic Programming*, volume Springer LNCS 225, pages 84–98, 1986.