Magic Sets Transforms of Functional Logic Programs

Julio Mariño-Carballo * and Juan José Moreno-Navarro

Universidad Politécnica de Madrid, School of Computer Science,
Campus de Montegancedo s/n, Boadilla del Monte E28660-SPAIN,
e-mail: jmarino(jjmoreno)@fi.upm.es

Abstract. This paper introduces transformation techniques for the efficient **bottom-up** execution of functional logic programs (FLP). We first propose a notion of bottom-up evaluation of FLP with possibly higher-order and non-strict semantics. Then we make this execution **goal-directed** by adapting the **magic-sets** idea to FLP. Coping with non-strict semantics is far from trivial and several solutions for this issue are proposed.

Functional logic languages use a subset of equational logic to generalize both Horn clause logic programming (Prolog, etc) and functional programming (ML, Haskell,etc). The resulting language gets polymorphic typing, lazy evaluation and higher order from FP, and logical variables, unification and deductive inference from LP. Although we will use Babel [10] throughout the paper, all the results apply to higher order equational logic with constructor-based rules and first order queries.

Narrowing has been used as the operational semantics of FL programs. Informally, it can be seen as a generalisation of term rewriting where unification replaces matching. Here two narrowing strategies implemented in Babel will be considered: innermost (**eager**) and **lazy** narrowing. The possibility of free variables in queries makes the execution non-deterministic in general. Implementations usually deal with this by means of backtracking. Unfortunately, backtracking based implementations suffer from various problems, like infinite loops – or, more generally, too **big search spaces** – and the risk of **reevaluation**. These problems may occur in Prolog as well, but laziness usually aggravates things. Regarding the former, consider this (contrived but) representative example:

```
f _ A := A=2 → loop.          g 1 := 1.
f _ 1 := 1.                   loop := loop.
```

eval f (g X) X.

A lazy narrowing strategy would select the outermost redex for narrowing:

$$\underline{\text{f(g X) X}} \leadsto_{X\mapsto 2} \text{loop},$$

thus entering the loop. An eager version would avoid that, by selecting the other redex: f$\overline{\text{(g X)}}$ X $\leadsto_{X\mapsto 1}$ f 1 1 $\leadsto$ 1. The problem, in general, is that delaying the binding of variables can lead to a bigger search space.

The reevaluation problem due to laziness appears when a needed redex is delayed and it is eventually evaluated more than once on backtracking. Yet another toy example:

g 0 X := 2·X.
g 1 X := X^2.
eval g X (fib 20).

If delayed, (fib 20) will be evaluated twice. Some techniques, like **demandness analysis**[9], have been proposed to anticipate the narrowing of some needed redexes, but it is not free of problems and the method, of course, can never be complete.

In some applications, avoidance of loops and reevaluation may be critical. In this work we make a first step towards complete methods via (goal directed) bottom up execution. Goal direction is needed to avoid indiscriminate memoing —to compute and store parts of the semantics which are not needed at all to evaluate the queries. Then we can obtain a no-slowdown

* Work partially done on leave at Cambridge University Computer Lab, funded by ESPRIT HCM grant "ABILE".

result[2] —although the waste of memory is still there. So we thought of adapting the idea of the *magic-sets transform* [16, 4, 1, 2, 7, 12, 13, 14, 15], born in the Datalog community, to languages based on equations. In abstract terms, the question is:

Question 1. Given program P and goal G, construct a new program P' such that its semantics contains all the answers to G while being as minimal as possible.

We will see that bringing these ideas to functional logic programs is not straightforward. On one hand, we need new techniques to define the bottom up evaluation of higher order programs. On the other hand, regarding the issue of the transformation itself, while we have a close correspondence of the classic method (up to some syntactic equivalence) for programs executed with an *innermost* selection rule, to cope with *lazy* programs we need to provide the method with some semantic information regarding which expressions demand reduction and which not. This is similar to the idea of minimal function graphs in FP, where the related question of finding the minimal semantics needed to compute G is considered. However, this is done regardless of its possible implementation as a rewritten program [5, 11, 6].

In this abstract we just present the main issues via self explanatory examples. The full paper will describe the transformation in detail, sketch some of the proofs and present the experimental results we are getting with a prototype implementation.

Bottom-up Evaluation

By bottom-up evaluation we mean a syntactic, constructive way to obtain the fixpoint model of a program. Following the s-semantics approach of [3] we define non-ground interpretations that arise as the limit of the unfolding of the rules of a program. These interpretations can then be more easily obtained from interpretation transformers.

Example 1. **fun** plus: nat → nat → nat.
plus 0 N := N.
plus (s X) N := plus X (s N).

Repeatedly unfolding the rules gives the sequence of programs:

$\mathrm{unf}_{Plus}(Plus)$: plus 0 N := N.
plus (s X) N := plus X (s N).
plus (s 0) N := s N.

$\mathrm{unf}^2_{Plus}(Plus)$: plus 0 N := N.
plus (s X) N := plus X (s N).
plus (s 0) N := s N.
plus (s(s 0)) N := s(s N).

etc

Observe that the answers for a query *plus A B* correspond to the non-recursive rules obtained during the unfolding. A rule that cannot be further unfolded is called a **fact**. All the observable properties of programs can be obtained in terms of the facts of the unfolding sequence. Given that non-ground interpretations are sets of facts, we have two different ways of obtaining them as a fixpoint:

Definition 2 Strict transformer. Given interpretation I for program P

$$ST_P(I) = \{Head\theta = Val | Head = Body \in P \wedge [\![Body]\!]_I \vdash^{\theta} Val\}$$

The **valuation relation** $[\![\cdot]\!]_I \vdash^{\theta}$ narrows according to the facts in I.

Example 2. Defining *map* as
map F [] := [].
map F [H|Ts] := [F H|map F Ts].

[2] we will see that in some examples we even get a drastic speed-up when using bottom-up evaluation

we will obtain the interpretations:

$I^0 = \{\ \}$
$I^1 = ST_{map}(I^0) = \{\text{map F [ ] := [ ]}\}$.
$I^2 = I^1 \cup \{\text{map F } [X_1] = [\text{F } X_1]\}$.
$I^3 = I^2 \cup \{\text{map F } [X_1, X_2] = [\text{F } X_1, \text{F } X_2]\}$.
$I^4 = I^3 \cup \{\text{map F } [X_1, X_2, X_3] = [\text{F } X_1, \text{F } X_2, \text{F } X_3].\}$
...

The other way is to use a **non-strict** interpretation transformer. The difference is that the non-strict valuation relation gives a new constant $\perp$ as value for any expression undefined in the original interpretation.

Example 3. For the definition of map we would obtain the interpretations:

$I^0 = \{\ \}$
$I^1 = LT_{map}(I^0) = \{\text{map F [] = []}, \text{map F } [X_1|Ts] = [\text{F } X_1|\ \perp]\}$
$I^2 = I^1 \uplus \{\text{map F [] = []}, \text{map F } [X_1] = [\text{F } X_1], \text{map F } [X_1, X_2|Ts] = [\text{F } X_1, \text{F } X_2|\ \perp]\}. \ldots$

We get the following soundness/completeness results:

Proposition 3. *The limit of ST^i_P corresponds to the set of facts in the eager unfolding of P, while the limit of LT^i_P corresponds to the set of facts in the lazy unfolding of P*

Just a word about how these interpretations are obtained in practice. In its current form, these interpretation transformers re-obtain the previous interpretations at every new iteration. What is needed is just the difference between successive interpretations, that is

$$P^n_\Delta = P_n - P_{n-1}$$

so that $P_n = P_1 \cup (\bigcup_{i=2}^n P^i_\Delta)$. The immediate consequences lemma proves that P^n_Δ is the set of facts when the interpetation transformer is forced to use at least one fact from P^{n-1}_Δ in the valuation of every right hand side.

Eager Magic Sets Transform

Eager narrowing selects leftmost innermost positions to narrow. This makes very easy to adapt the magic sets idea. The reason is that arguments are always passed in normal form. More formally:

Lemma 4. *Given a narrowing step $e \rightsquigarrow_{\sigma_{\text{out}}} e[u \leftarrow r\sigma_{\text{in}}]\sigma_{\text{out}}$ in an innermost narrowing sequence, the terms substituted for in σ_{in} do not contain function symbols.*

What we obtain for eager Babel is essentially the same as in Prolog but changing left-to-right literal ordering with leftmost-innermost redex ordering. The idea is to make a "declarative reading" of the innermost execution rule:

- The first redex of a rule (the leftmost-innermost) is called when a call matches the lhs
- Other redexes inside a rule are called when the previous (to the left or the last argument inside) have been called and succeeded. The result of their evaluations are now available to this redex
- The goal gives the first redex invoked

We will add a new predicate to express the fact that some expression has been "called" in the eager semantics.

Example 4. Usual definition for list appending:

fun append: list T → list T → list T.
  append [ ] Xs := Xs.
  append [H|Ts] Xs := [H|append Ts Xs].

eval append L (append [3] [4]).

We get the magic program:

fun append: list T $\to$ list T $\to$ list T.
  append [ ] Xs := callappend [] Xs $\to$ Xs.
  append [H|Ts] Xs := callappend [H|Ts] Xs $\to$ [H| append Ts Xs].
fun callappend: list T $\to$ list T $\to$ bool.
  callappend [3] [4] := true.
  callappend L R := R = (append [3] [4]) $\to$ true.
  callappend Ts Xs := callappend [H|Ts] Xs $\to$ true.

now the sequence of interpretations is:
P^0 = both functions undefined
P^1_Δ = append undefined, callappend [3] [4] := true
P^2_Δ = callappend [] [4] := true
P^3_Δ = append [] [4] := [4]
P^4_Δ = append [3] [4] := [3, 4]
P^5_Δ = callappend X [3, 4] := true
P^6_Δ = append [] [3, 4] := true
P^7_Δ = append [H] [3, 4] := [H, 3, 4]
fixpoint

Proposition 5. *If* $e \rightsquigarrow^*_\sigma e'$ *then there exists an environment* ρ *such that* $[\![e\sigma]\!]_{P^\omega}(\rho) = [\![e'\sigma]\!]_{P^\omega}(\rho)$

Coping with Laziness

Lazy narrowing tries to apply narrowing in outermost positions. That means that it is not direct to know whether a given redex has to be narrowed in a context independent way. More specifically, (4) no longer holds, i.e. some arguments are passed needing further evaluation. In order to give a recursive definition of the lazy narrowing strategy we will work with Babel programs in a canonical form:

Definition 6 Uniform programs. A Babel program is uniform when for every set of rules defining a function symbol

$$\begin{array}{l} f\ t_{11} \dots t_{n1}\ := e_1 \\ \dots \\ f\ t_{1m} \dots t_{nm}\ := e_m \end{array}$$

every t_{ij} is a variable, a constant or a constructor applied to variables, and $\forall ijk.t_{ij} \in \mathbf{Var} \to t_{kj} \in \mathbf{Var}$

In [8] it is shown how a given Babel program can be translated into an "equivalent" uniform one. This transformation preserves the declarative semantics although has terrible consequences for reevaluation. This was a serious problem for backtracking-based implementations of lazy narrowing but it just does not matter for bottom-up execution. Uniform programs suggest the following strategy

- Reduce the goal to normal form
- To reduce a expression $f\ e_1 \dots e_n$, obtain head normal forms for its demanded arguments, then reduce right hand side after matching

So now we can adapt the ideas in the previous section only that now we have calls to get normal form (NF), calls to get head normal form (HNF), and results in NF and (quasi)results in HNF. An informal translation scheme is in figure 1.

Example 5. We get the following program for the example in Fig. 2:

callnf (tonth (s(s(s 0)) (netc 0)) := true.
callnf (tonth N Ts) := callnf (tonth (s N) [H|Ts]) $\to$ true.
callhnf Exp := callhnf (tonth (s N) Exp) $\to$ true.
callhnf (tonth (s N) [H|Ts]) := callhnf (tonth (s N) Exp), hnf Exp = [H|Ts] $\to$ true.
nf (tonth 0 _) := [ ].
nf (tonth (s N) [H|Ts]) := [H|nf (tonth N Ts)].
hnf (netc N) := [N|netc (s N)].

Goal:

- **eval** f $e_1 \ldots e_n$. $\Rightarrow$ callnf (f $e_1 \ldots e_n$) = true.

Magic predicates:

- Given a rule
$$f\ t_1 \ldots t_n = g\ e_1 \ldots e_n.$$
we have:
$$\text{callnf}(g\ e_1 \ldots e_n) = \text{callnf}(f\ t_1 \ldots t_n) \rightarrow \text{true}.$$
$$\text{callhnf}(g\ e_1 \ldots e_n) = \text{callhnf}(f\ t_1 \ldots t_n) \rightarrow \text{true}.$$
- Given a rule
$$f\ t_1 \ldots t_n = c\ e_1 \ldots e_n.,\ \ c \in DC_\Sigma$$
we have:
$$\text{callnf}\ e_i = \text{callnf}(f\ t_1 \ldots t_n),\ \text{nf}\ e_1 = R_1, \ldots, \text{nf}\ e_{i-1} = R_{i-1} \rightarrow \text{true}.$$
- Demanded arguments: given a rule:
$$f\ t_1 \ldots t_n = \ldots$$
we have:
$$\text{callhnf}(\text{Exp}) = \text{callhnf}(f\ t_1 \ldots t_{i-1}\ \text{Exp}\ e_{i+1} \ldots e_n)$$
$$\text{callhnf}(f\ t_1 \ldots t_i\ e_{i+1} \ldots e_n) = \text{callhnf}(f\ t_1 \ldots t_{i-1}\ \text{Exp}\ e_{i+1} \ldots e_n),$$
$$\text{hnf}(\text{Exp}) = t_i \rightarrow \text{true}.$$

Answer functions:

- f $t_1 \ldots t_n$ = Body $\Rightarrow$
  nf(f $t_1 \ldots t_n$) = callnf(f $t_1 \ldots t_n$) $\rightarrow$ nf(Body)
  hnf(f $t_1 \ldots t_n$) = callhnf(f $t_1 \ldots t_n$) $\rightarrow$ hnf(Body)

Fig. 1. A translation scheme for a lazy interpreter

Δ evaluation:

$\mathbf{P'}^0_\Delta$ = { }
$\mathbf{P'}^1_\Delta$ = {callnf (tonth (s(s(s 0))) (netc 0)) := true}
$\mathbf{P'}^2_\Delta$ = {callhnf (netc 0) := true}
$\mathbf{P'}^3_\Delta$ = {hnf (netc 0) := [0|netc (s 0)]}
$\mathbf{P'}^4_\Delta$ = {callnf (tonth (s(s(s 0))) [0|netc (s 0)]) := true}
$\mathbf{P'}^5_\Delta$ = {callnf (tonth (s(s 0)) (netc (s 0))) := true,
callhnf [0|netc (s 0)] := true}
$\mathbf{P'}^6_\Delta$ = {callhnf (netc (s 0)) := true}
$\mathbf{P'}^7_\Delta$ = {hnf (netc (s 0)) := [s 0|netc (s(s 0))]}
$\mathbf{P'}^8_\Delta$ = {callnf (tonth (s(s 0)) [s 0|netc (s(s 0))]) := true}
$\mathbf{P'}^9_\Delta$ = {callnf (tonth (s 0)) (netc(s(s 0))) := true,
callhnf [s 0|netc (s(s 0))] := true}
$\mathbf{P'}^{10}_\Delta$ = {callhnf (netc (s(s 0))) := true}
$\mathbf{P'}^{11}_\Delta$ = {hnf (netc (s(s 0))) := [s(s 0)|netc (s(s(s 0)))]}
$\mathbf{P'}^{12}_\Delta$ = {callnf (tonth (s 0) [s(s 0)|netc (s(s(s 0)))]) := true}
$\mathbf{P'}^{13}_\Delta$ = {callnf (tonth 0 (netc (s(s(s 0))))) := true,
callhnf [s(s 0)|netc (s(s(s 0)))] := true}
$\mathbf{P'}^{14}_\Delta$ = {nf(tonth 0 _) := []}
$\mathbf{P'}^{15}_\Delta$ = {nf(tonth (s 0) [s(s 0)|netc(s(s(s 0)))]) := [s(s 0)]}
$\mathbf{P'}^{16}_\Delta$ = {nf(tonth (s 0) (netc(s(s 0))) := [s(s 0)]}
$\mathbf{P'}^{17}_\Delta$ = {nf(tonth (s(s 0)) [s 0|netc (s(s 0))]) := [s 0, s(s 0)]}
$\mathbf{P'}^{18}_\Delta$ = {nf(tonth (s(s 0)) (netc(s 0))) = [s 0, s(s 0)]}
$\mathbf{P'}^{19}_\Delta$ = {nf(tonth (s(s(s 0))) [0|netc(s 0)]) := [0, s 0, s(s 0)]}
$\mathbf{P'}^{20}_\Delta$ = {nf(tonth (s(s(s 0))) (netc 0)) := [0, s 0, s(s 0)]}

The completeness result is analogous to the eager case.

```
data nat = 0 | s nat.
data list T = [] | [T|list T].

fun netc: nat → list nat.
    netc N = [N|netc (s N)].

fun tonth: nat → list T → list T.
    tonth 0 _ = [].
    tonth (s N) [H|Ts] = [H|tonth N Ts].

eval tonth (s(s(s 0))) (netc 0).

⇒ [0, s 0, s(s 0)]
```

Fig. 2. A Babel program to compute a finite portion of an infinite list

Improving the Lazy Transform

The previous approach is rather disappointing: on one hand, there is too much metainterpretation, in fact the resulting program is a strict metainterpreter. More striking is the fact that the resulting program is no longer typeable!

The reason why metainterpretation is hard to remove is because we need some kind of continuation-passing mechanism, because a redex may not be demanded by its parent redex but by a higher ancestor:

```
     f X := [X].
     g X := s X.
     h [s X] := X.
eval h(f(g Y)).
```

Here h demands that the result of $f(g\ Y)$ be (at least) as normalized as $[s\ \bot]$, thus demanding $g\ Y$ to be as normalized as $s\ \bot$.

We say that a redex in a expression e is demanded when it is going to be narrowed somewhere in the lazy narrowing of e. We propose an improved approach where this demandness information is computed during bottom-up execution. To make the presentation shorter, we just present the simplest case of flat data, so that we have NF = HNF. A possible recursive characterization of demandness is:

- Let $f\ e_1 \ldots e_n$ be an expression. Let $g\ z_1 \ldots z_m$ be a redex inner to the previous expression.
- z_i is demanded iff $g\ z_1 \ldots z_n$ is and (z_i is demanded by a lhs of a rule for g (after z_k, $k < i$)), or z_i has a needed residual after reducing $g\ z_1 \ldots z_m$, i.e. $g\ z_1 \ldots z_m \rightsquigarrow z_i$

Example 6. Consider this program:

```
    f X := j X.
    j X := X.
    g X := b.
    h a := b.
    h b := a.
eval h(f(g X)).
```

Applying the characterization:

- *Exp* is demanded when *j Exp* is
- *j Exp* is demanded when *f Exp* is, etc

This can be encoded in the following magic program:

```
f X := dem (f X) → j X.      dem (h(f(g X))) := true.
j X := dem (j X) → X.        dem (f(g X)) := dem (h(f(g X))) → true
g X := dem (g X) → b.        dem (f X) := dem (h(f X)).
h a := dem (h a) → b.        dem (j X) := dem (f X) → true.
h b := dem (h b) → a.        dem X := dem (j X) → true.
```

Notice that with this new transform results are not meta-interpreted! Besides, part of the metainterpretation in the calls can easily be removed. What we have obtained is a non-strict program and we need a non-strict operator to get the right results.

We are working in an extension of this idea to non-flat data by providing *dem* with an extra argument that expresses the degree of normalization expected for the result, and a special dem_{NF} predicate to express that some redex has to be evaluated to normal form.

Conclusion

Bottom-up evaluation can be a promising technique to avoid reevaluation and loops when implementing FL programs, even with non-strict and higher-order semantics. This opens the way for the design of abstract machines based on Δ-bottom-up evaluation. From a theoretical standpoint we hope this work serves to find analogues of concepts from logic and functional programming in each other, i.e. a denotational presentation of magic sets like minimal function graphs or the use of non-ground semantics as a tool in FP.

Some open questions are whether there is some way of disposing of unnecessary facts thus cutting the waste of memory and whether these transforms apply to other classes of equational systems. It seems that abandoning the constructor discipline leads to totally interpreted systems, for every simple operational semantics relies on the definition of normal forms.

References

1. C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.
2. Roland Bol and Lars Degerstedt. The underlying search for magic. pages 793–811.
3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20, 1994.
4. Y. Sagiv F. Bancilhon, D. Maier and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. ACM Symposium on Principles of Database Systems*, pages 1–15, Boston, Mass., March 1986.
5. Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th POPL*, St. Petersburg, Florida, January 1986.
6. Neil D. Jones and Mads Rosendahl. Higher-order minimal function graphs. Technical Report D-204, DIKU, 1994.
7. David Kemp, Divesh Srivastava, and Peter J. Stuckey. Magic sets and bottom-up evaluation of well-founded models. pages 337–351.
8. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In H. Kirchner and W. Wechler, editors, *2nd International Conference on Algebraic and Logic Programming, Nancy (France)*, pages 298–317. CRIN (Centre de Recherche en Informatique de Nancy), LNCS, Springer Verlag, October 1990.
9. J.J. Moreno-Navarro, H. Kuchen, J. Mariño Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 167–183. Springer LNCS 714, 1993.
10. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12:191–223, 1992.
11. Alan Mycroft and Mads Rosendahl. Minimal function graphs are not instrumented. In *WSA'92*, pages 60–67, Bordeaux.
12. R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *J. Logic Programming*, 11(3):189–216, 1991.
13. Raghu Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. pages 321–336.
14. Yehoshua Sagiv. Is there anything better than magic? pages 235–254.
15. S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. pages 557–574.
16. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. CS Press, 1992. Ch. 13.