A model for mathematical analysis of functional logic programs and their implementations*

Egon Börger[a], Francisco J. López-Fraguas and Mario Rodríguez-Artalejo [b] †

[a] Dip. di Informatica, Universitá di Pisa, Cso Italia 40, I-56100 PISA, boerger@di.unipi.it

[b]Dep. Informática y Automática, Universidad Complutense, Av. Complutense s/n, 28040 Madrid,Spain, {fraguas,mario}@dia.ucm.es

We extend the mathematical definition and analysis of Prolog program execution on the WAM, developed in [2,3], to functional logic languages and machines. As reference language we choose the functional logic programming language BABEL [8].

1. Introduction

We extend the core Prolog model of [2] to a model for the functional logic programming language BABEL [8] by adding, to Prolog's backtracking structure, rules for the reduction of functional expressions to normal form. Then we define six typical provably correct refinements which are directed towards implementation of functional logic programs: structure sharing for expressions, explicit computation of the normal form condition, embedding of the backtracking tree into a stack, localization of the normal form computation for expressions (introducing local environments for computation of subexpressions) together with some optimizations in IBAM [6], a (graph—) narrowing machine actually implementing innermost BABEL. Thus the machinery of [2,3] for mathematical description and analysis of logic programs, is linked to functional logic programs and their implementation on machines which typically combine the WAM [9] with features from reduction machines [4] for functional languages (see also [7]).

The fundamental concept which allows us to describe complex program and machine structure by simple abstract mathematical specifications is Gurevich's notion of *evolving algebras* which we will use throughout and for which we refer the reader to [5]. We expect from the reader also basic knowledge of logic programming and term rewriting, and assume familiarity with [2]. For BABEL we note here only that programs consist of *defining rules* of the form $f(t_1, ..., t_n) := r$ where r may be a guarded expression $(b \rightarrow e)$. We will consider here only the first order subset of the language, and will omit correctness proofs for our refinements. For details and extensions we refer to [1].

*In: B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam.

†This research has been partially supported by the Spanish National Project TIC92–0793–C02–01 "PDR"

2. From Prolog to BABEL tree algebras

A BABEL computation can be seen as systematic search of a space of possible solutions to an initially given goal. The goals are expressions which have to be reduced to normal form using narrowing (unification and reduction) with respect to the given set of defining rules for the user defined functions. Thus defining the semantics of BABEL programs has to incorporate, besides reduction to normal form, the concepts of unification and backtracking—which can be taken almost literally from Prolog.

2.1. The narrowing–backtracking core

For the top level description of BABEL it suffices to enrich the core model for Prolog defined in [2] as follows. The role of Prolog goals is taken by the expression still to be reduced, provided by a function $exp : NODE \rightarrow EXPRESSION$. Since reduction steps might take place in subexpressions of the given expression, a function $pos : NODE \rightarrow POSITION$ provides for given state the *position* in the associated expression at which the next reduction step will take place. We assume, in accordance with usual practice, that positions u are coded as finite sequences of positive integers. The specification of the function *pos* depends on the reduction strategy; in this model for BABEL we will define it for the leftmost innermost strategy (which corresponds to an eager implementation).

The distinguished element *mode*, indicating the action to be taken at *currnode*, besides values *Apply*, *Select* for creating the alternative narrowing states and selecting among them, can also assume the additional value *Eval* for evaluating the expression obtained by narrowing.

procdef now handles access to candidate defining rules. The role of the activator is taken by the selected *exp*-subexpression defined by $currexp \equiv exp[pos]$, where for accessing and manipulating subexpressions of given expressions (read: subtrees of trees) we use four standard tree functions yielding the subexpression $e[u]$ of e at position u, the result $e[u \leftarrow e']$ of replacing $e[u]$ by e' in e, the information *occurs(u,e)* whether u is a legal position in e, and concatenation of positions.

For the initialization it is assumed that *pos* is the empty position and *mode* is *Eval*.

The backtracking behaviour of BABEL programs is defined by the corresponding two Prolog rules in [2], handling expressions and their positions instead of goals: in the *Call* rule, $currexp = f(e_1, \ldots, e_n)$ is added as condition; the *Selection* rule, when switching to mode *Eval*, uses unification of *currexp* with the left hand side of the candidate defining rule and narrowing of the current environment:

if $is\_user\_defined(currexp)$ & $mode = Apply$
  & $currexp = f(e_1, \ldots, e_n)$
then let $[dr_1, \ldots, dr_m] = procdef(currexp, db)$
  extend *NODE* **by** $n_1, \ldots, n_m$ **with**
    $bfather(n_i) := currnode$
    $rule(n_i) := dr_i$
    $cands := [n_1, \ldots, n_m]$
  endextend
  $mode := Select$

if $mode = Select$
thenif $cands = [\,]$ **then** *backtrack*
else let $(Lhs := Rhs) = rename(rule(fst\_cand), vi)$
  let $\theta = mgu(currexp, Lhs)$
  $cands := rest(cands)$
  if $\theta \neq nil$
  then *go_fst_cand in Eval*
    narrow curr_env(Rhs, θ)
    $vi := vi + 1$

with *go_fst_cand in Eval* and *narrow curr_env*(E, θ) abbreviating respectively

$currnode := fst\_cand, mode := Eval$
$exp(fst\_cand) := exp[\, pos \leftarrow E\,]\,\theta,\ pos(fst\_cand) := pos,\ s(fst\_cand) := s\,\theta$

2.2. Reduction to normal form

Since evaluation follows the leftmost innermost strategy, it is started by searching the position of the leftmost innermost subexpression of the current expression which is not in normal form. If *currexp* is not in normal form, we check whether its leftmost subexpression (at the extension *pos*.1 of the current position) is responsible for this. Otherwise we have to *continue* the evaluation for the next relevant subexpression (brother- or father-expression of *currexp*, see below). This is described by the **evaluation starting rule**:

if $mode = Eval$ **thenif** $is\_not\_in\_normal\_form(currexp)$ **then** $pos := pos.1$
else $mode := Continue$

where *is_not_in_normal_form* is an auxiliary Boolean–valued function (with the obvious meaning) and *Continue* as value of *mode* signals that *currexp* has been reduced to normal form.

Once *currexp* appears to be in normal form, the computation goes on to *evaluate* the next brother expression if there is one; otherwise the computation passes to *apply* the outermost symbol of the father expression (all of which arguments have been reduced then to normal form). There is a special case: some predefined functions are considered as strict [3] only in the first argument. For these functions, once the first argument has been reduced to normal form, the evaluation must proceed in the father position in mode *Apply*. All this is formalized by the **evaluation continuation rule**:

if $mode = Continue\ \&\ pos = u.i$ **thenif** $\neg occurs(u.(i+1), currexp)\ or\ nonstrict(u)$
then $pos := u,\ \ mode := Apply$
else $pos := u.(i+1),\ \ mode := Eval$

where *nonstrict(u)* tells whether the subexpression $exp[u]$ is an application of one of the predefined functions $and(b_1, b_2)$, $or(b_1, b_2)$, $b \rightarrow e$ (read *if b then e*), $b \rightarrow e_1 \# e_2$ (read *if b then* e_1 *else* e_2) which are strict only in the first argument and need special evaluation rules.

If there is no father expression, reflected by *pos* coming back to its initial empty value (say ϵ), the whole initial expression has been reduced to normal form, and hence the computation of one solution has been completed. Then the computed solution is added to the *solution_list*, and the user is asked interactively whether more solutions are wanted, using a Boolean valued 0-ary function *more*—external in the sense of [5]—to take care of external request of more solutions. Otherwise the computation terminates with final success. This is described by the **stop rule**:

if $mode = Continue\ \&\ pos = \epsilon$ **then** $solution\_list := [\langle exp, s\,|_{Goalvars}\rangle \mid solution\_list]$
if $more = 1$ **then** *backtrack*
else $stop = 1$

where $s\,|_V$ denotes the restriction of s to the set of variables V; *Goalvars* is a (static) 0-ary operation ranging over a universe *VAR* of variables. *Goalvars* is initialized to be the set of variables of the initial expression to be reduced, and is not changed during the computation.

[3] f is *strict* in its i-th argument iff $f(a_1, \ldots, a_i, \ldots, a_n)$ is undefined whenever a_i is undefined.

In *Apply* mode there are several cases to consider, depending on the value of *currexp*: it can be an expression formed by a user–defined function applied to arguments which are all in normal form (in which case narrowing takes place as described above by *Call* and *Selection* rules), it may be a constructor expression, or the application of some predefined function. For each of these cases there is a corresponding rule.

If in *Apply* mode *currexp* has a constructor c as topmost symbol, then the evaluation has to stop and to *continue* at another (brother– or father–) expression, as described by the **construction rule**:

if $mode = Apply \,\&\, is\_construction(currexp)$ **then** $mode := Continue$

We skip the rules for predefined functions which can be defined like Prolog built-in predicates in [2].

3. BABEL stack algebras

In order to come closer to a realistic implementation, in this section we refine BABEL tree algebras by structure sharing for expressions, explicit computation of the *normal_form* condition and embedding of the backtracking tree structure into a stack. For exact formulation and proof of the correctness statements see [1].

Structure sharing for expressions can be obtained in a provably correct way as follows. When during narrowing the left hand side of a defining rule is replaced by its right hand side, we put only the source code expression involved, without applying the unifying substitution which is kept apart. This means that *a)* the update for $exp(fst\_cand)$ in the *Selection* rule is changed to be without applying the unifying substitution, and *b)* the substitution must be applied to those occurrences of *currexp* or expressions e in the rules , where the condition to be checked or the operation to be performed on those expressions really depend on the substitution ; see [1].

Refining the normal form test means to define the *is_normal_form* function used in the evaluation starting rule. The idea is to traverse an expression in mode *Eval* until a non decomposable normal form is reached. [4] Thus the **new evaluation starting rule**, where $atomic(e)$ recognizes if the expression e is a variable or a constant symbol c:

if $mode = Eval$ **thenif** $\neg atomic(currexp)$ **then** $pos := pos.1$ **else** $mode := Continue$

The provably corret refinement to **stack representation** of BABEL trees is similar to the corresponding stack implementation in [3] and therefore skipped here.

4. Localizing the environment and Optimizations

The normal form computation for expressions is localized in two steps: introduction of states which control normalization of subexpressions, completed by subsequent replacement of 'global' by dedicated 'local' environments.

Up to now, states (nodes of *NODE*) reflect only backtracking. They are created when a function call is made for reducing *currexp*, but their role does not correspond exactly to computing just that subexpression. In fact, if no more call is performed, the created state remains the current one until the end of the computation for the whole *exp*.

[4] By this we come closer to implementations, where in the compilation process all the expressions in the source program are traversed and code instructions are generated for them.

We now make states responsible only for the computation of the corresponding subexpression. Once this computation is finished, control will be returned (by a *Return rule*) to the state which was current in the moment of the call: the *activator* of the state created by the call and denoted by a function *act_node* from *NODE* to *NODE*.

This *call*ing structure induces a new tree structure in *NODE* (which is actually the core of the IBAM) and imposes two changes in connection with backtracking:

- *currnode* may now be updated also by the new *Return rule*. Consequently, when a new state is created by a call, *currnode* may not represent the last created state to which to backtrack from the new state. Therefore a 0-ary function *lastnode* is introduced to store that information. Initially *lastnode* is set to be the child of *root*.

- All the alternatives for a given call have to be tried with the same environment, namely *exp, pos, s* as they were in the moment of the call. In the previous algebras, these values were accessed from the environment of the backtracking father. This will not be safe any more, since *exp*,*pos* and *s* for the backtracking father could have changed in the meantime, had control come back to it by the return rule. Therefore in the moment of a call, 'safe copies' of *exp*,*pos* and *s* are stored in the new state. We denote these values by functions *act_exp, act_pos, act_s* defined on *NODE*.

act_node, act_env, lastnode are handled by additional updates to *Call* and *Selection* rules. Since in this first step towards 'local', subexpression-normalization, the global expressions are still kept as decorations of states, we obtain the **modified call rule** by changing the update of *bfather* to *bfather*(N) :=*lastnode* and by adding the updates *act_node(N)* :=*currnode* and *store act_env at N*, defined by

store act_env at $N \equiv$ *act_exp*$(N) :=$ *exp*, *act_pos*$(N) :=$ *pos*, *act_s*$(N) := s$

The **modified select rule** is literally the same as before, replacing *bfather_env* by *act_env*, both in the definition of the mgu θ and in the abbreviation *narrow bfather_env*.
In addition *lastnode*, to which *bfather* will be set by the following execution of *Call* rule, is updated to *currnode*. The **modified evaluation continuation rule** obtains the additional test whether the current position is root of a subcomputation which has been activated by *currnode*; in this case it switches to a new mode *Return*. Formally it is sufficient to replace the guard in the evaluation continuation rule by the following one:

if *mode* = *Continue* **thenif** *pos* = *act_pos* **then** *mode* := *Return*
else let $u.i$ = *pos*
if $\neg$*occurs*$(u.(i+1),$ *currexp*$)$ *or* ...

For technical convenience, we assume that *act_pos* is initialized to be ϵ for the child of *root*, which means that the switching to mode *Return* will also happen when the evaluation of the whole initial expression finishes (in this case, *pos* = *act_pos* = ϵ). As a consequence, the condition *mode* = *Continue* & *pos* = ϵ in *Stop* changes to *mode* = *Return* & *bfather* = *root*.

The new mode *Return* is governed by a **Return rule** through which control is returned to the activating node, with expression (still globally) updated by the result of the just terminated subcomputation.

if *mode* = *Return* & *bfather* $\neq$ *root* **then** *currnode* := *act_node*
return curr_env to act_env
mode := *Continue*

where *return curr_env to act_env* abbreviates

$$exp(act\_node) := exp,\ pos(act\_node) := pos,\ s(act\_node) := s$$

Subcomputations with local expressions are obtained by replacing 'global' environments by 'local' ones, i.e. handling of *act_env* (narrowing of and returning to in *Selection* and *Return* rules) is done with the relevant 'local' expression. For the new **selection rule**, *narrow act_env(Rhs,θ)* is refined using the 'local' updates $exp := Rhs$, $pos := \epsilon$.

In the new **return rule** the subcomputation result is returned to *act_node* by placing it into the expression of *act_node* at its activation position (thus making it 'global' relative to *act_node*). Formally this means to refine *return curr_env to act_env* assigning $act\_exp[act\_pos \leftarrow exp]$ to *exp(act_node)* and *act_pos* to *pos(act_node)*. In addition, due to the redefinition of *pos* in the selection rule, the condition $pos = act\_pos$ in the *Evaluation continuation* rule for switching to mode *Return* must be replaced by $pos = \epsilon$.

Some **Optimizations** in the IBAM can be conveniently specified at this level of abstraction. As an example, we explain here the **optimized last return**, which means the following: When a a value is returned by a task which is the last (active) created task and has an empty list of alternatives, then this task will not perform any other successful computation. If by backtracking it is reactivated later on, it will fail immediately and backtracking will be done to its backtracking father. One can anticipate this situation by resetting *lastnode* to the backtracking father of the task (the task node itself could in fact be collected as garbage). The **optimized last return rule** expresses this by adding a conditional update for *lastnode*:

if $lastnode = currnode \ \&\ cands = [\,]$ **then** $lastnode := bfather$

For the specification of other optimizations, see again [1].

REFERENCES

1. E. Börger, F. J. López Fraguas, and M. Rodríguez Artalejo. Towards a mathematical specification of narrowing machines. Research report DIA 94/5, Dep. Informática y Automática, Universidad Complutense, Madrid, March 1994.
2. E. Börger and D. Rosenzweig. A simple mathematical model for full Prolog. Research report TR-33/92, Dipartimento di Informatica, Universitá di Pisa, Pisa, October 1992. to appear in *Science of Computer Programming*, 1994.
3. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. Research report TR-14/92, Dipartimento di Informatica, Universitá di Pisa, Pisa, 1992. to appear in: *Logic Programming: Formal Methods and Practical Applications* (C.Beierle, L.Plümer, Eds.), North-Holland, Series in Computer Science and Artificial Intelligence, 1994.
4. A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, 1988.
5. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.
6. H. Kuchen, R. Loogen, J.J Moreno Navarro, and M. Rodríguez Artalejo. Graph-based implementation of a functional logic language. In *ESOP*, volume 432 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 1990.
7. R. Loogen. Relating the implementation techniques of functional and functional logic languages. to appear in New Generation Computing.

8. J.J Moreno Navarro and M. Rodríguez Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:189–223, 1992.
9. D.H.D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, 1983.