Towards a Mathematical Specification of a Graph-Narrowing Machine*

Egon Börger
Dip. di Informatica
Universita di Pisa
Cso Italia 40
I-56100 PISA
boerger@di.unipi.it

Francisco J. López-Fraguas
Dep. Informática y Automática
Universidad Complutense
Av. Complutense s/n
28040 Madrid,Spain
fraguas@dia.ucm.es

Mario Rodríguez-Artalejo
Dep. Informática y Automática
Universidad Complutense
Av. Complutense s/n
28040 Madrid,Spain
mario@dia.ucm.es

Abstract

The paper provides a mathematical model for the innermost version of the functional logic programming language BABEL [MR89, MR92] and refines it stepwise towards a mathematical specification of its implementation by a graph-narrowing machine [KLMR90]. Our description directly reflects the basic intuitions underlying the language and can thus be used as a primary mathematical definition of innermost BABEL. For each refinement step a mathematical correctness proof is given, thus paving the way for a correctness proof of the graph-narrowing machine implementation (a full correctness proof could be achieved by providing some further refinement steps, leading to the machine's abstraction level). The specification uses evolving algebras, thus allowing the descriptions to be procedural and nevertheless abstract, readable as 'pseudocode over abstract data'.

Keywords: functional logic languages, mathematical specification, evolving algebras.

1 Introduction

Many investigations during the last years have been devoted to the combination of the functional and logic programming paradigms (for surveys on different approaches and proposals, see e.g. [DL86], [BL86], [AN89]). The interest of such an integration has been well motivated by several researchers and is presently quite widely accepted.

One of the possible approaches to logic + functional programming builds so called *functional logic languages* [Red85, Red87], which are syntactically very similar to functional languages, but have *narrowing* as operational semantics. From the point of view of functional logic programming, narrowing is a natural extension of the *reduction* mechanism used for functional programs. It reduces a computation expression by applying *rewrite rules*, but using *unification* instead of *matching*.

This paper deals with the mathematical specification of the *functional logic language* BABEL [MR89, MR92], ultimately aiming at a full correctnes proof for its implementation through the *innermost graph-narrowing machine* IBAM [KLMR90]. IBAM belongs to a series of abstract machines which were designed during the last years for compiler implementations of logic + functional languages (see e.g [Loo93] for information on such approaches). This kind of machines usually combine features from *reduction machines* for functional languages [PJ87, FH88] and *Warren's Prolog Engine* WAM [War83], [AK91].

*Technical Report DIA 94/5, March 1994, Dep.de Informatica y Automatica, Universidad Complutense, Madrid. An abridged version of this paper has appeared under the title *A model for mathematical analysis of functional logic programs and their implementations* in: B. Pehrson and I. Simon (Eds.) IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations, Elsevier, Amsterdam.

It is well known that providing mathematically precise but nevertheless understandable specifications of complex systems is a difficult task. Recently, Gurevich's evolving algebra approach [Gur88, Gur91] has proved very adequate for providing mathematical descriptions of various logic programming languages and implementation methods [BÖ0a, BB92, BR91, BR92a, BR92b, BR93, BS91]. The definitions of Prolog and the WAM [BR92b, BR92a], yielding correctness conditions for Prolog compilers, are particularly related to the present paper, which can be understood as a first step towards extending this specification methodology to functional logic languages and machines. *Evolving Algebras* provide procedural and nevertheless abstract descriptions, easy to adapt to different abstraction levels, and easy to read (without special mathematical training) as 'pseudocode over abstract data'. We present a series of successively more refined descriptions of BABEL by means of evolving algebras, starting at the abstract level of the observable computational behaviour, and coming close to the level of machine implementation. We provide detailed hints of *correctness proofs* for each of the descriptions with respect to the preceding one.

The organization of this paper is as follows. Section 1 is this Introduction. In Section 2 we provide the information on Evolving Algebras and the BABEL language which is needed for understanding the rest of the paper. Section 3 develops the description of innermost BABEL at the user level of observable behaviour, through a series of three algebras which specify essentially the search tree of BABEL goal expressions and the search for solutions via leftmost innermost narrowing and backtracking. Section 4 moves closer to the level of machine implementation by replacing the tree structure of BABEL's search space by a stack structure. In Section 5, we refine the description by taking care of features more related to the functional dimension of the language, namely the creation of computation tasks responsible for evaluating some subexpression of the main goal expression and returning the result to the activator task. This organization, and the way it combines with backtracking, do correspond to the actual IBAM implementation [KLMR90], which is not approached closer in this paper (next refinement steps should deal with machine code generation). Section 6 presents, on an abstract level, some optimizations of the IBAM implementation. Section 7 summarizes our conclusions and points to possible lines of future research. In an appendix whole sets of rules for all the refinements of the algebra can be found.

2 Notation and prerequisites

We expect from the reader only rudimentary knowledge of the language of first order logic, logic programming and term rewriting. We list nevertheless in this section some definitions which might help to read our description—which uses Gurevich's notion of *evolving algebra* [1] without presupposing more than what is listed here—as 'pseudocode over abstract data'. We briefly indicate also how our correctness proofs can be carried out in this framework.

The abstract data come as elements of (not further analysed) sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*. We shall allow the setup to *evolve* in time, by executing *function updates* of form

$$f(t_1, \ldots, t_n) := t$$

whose execution is to be understood as *changing* (or defining, if there was none) the value of function f at given arguments.We shall also allow some of the universes to *grow* in time, by executing *universe extensions*

extend A **by** $t_1, \ldots, t_n$ **with** *updates* **endextend**

where *updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A.

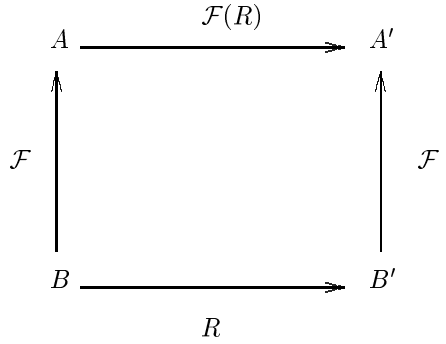
The precise way our 'abstract machines' (evolving algebras) may evolve in time will be determined by a finite set of *rules* of form

if *condition* **then** *updates*

where *condition* or guard is a boolean, the truth of which triggers *simultaneous* execution of all updates listed in *updates*. Simultaneous execution helps us avoid coding to, say, interchange two values.

Every *evolving algebra*—formally pair $(\mathcal{A}, \mathcal{R})$ where $\mathcal{A}$ is a first-order heterogeneous algebra with partial functions and possibly empty domains, and $\mathcal{R}$ is a finite system of *transition rules* as explained above—determines a class of structures that we shall call *algebras* or *states* of $(\mathcal{A}, \mathcal{R})$. Within such classes we will have a notion of *initial* and *terminal* algebras, expressing initial resp. final states of the target system. We are essentially interested only in those states which are reachable from inital states by $\mathcal{R}$. In our refinement steps we typically construct a "more concrete" evolving algebra $(\mathcal{B}, \mathcal{S})$ out of a given "more abstract" evolving algebra $(\mathcal{A}, \mathcal{R})$ and relate them by a (partial) *proof map* $\mathcal{F}$ mapping states B of $(\mathcal{B}, \mathcal{S})$ to states $\mathcal{F}(B)$ of $(\mathcal{A}, \mathcal{R})$, and rule sequences R of $\mathcal{R}$ to rule sequences $\mathcal{F}(R)$ of $\mathcal{S}$, so that the following diagram commutes:

[1] For motivation and a precise definition of evolving algebras see [Gur91].



We shall consider such a proof map to establish *correctness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if $\mathcal{F}$ preserves initiality, success and failure (value of *stop*) of states, since in that case we may view successful (failing) concrete computations as implementing successful (failing) abstract computations.

We shall consider such a proof map to establish *completeness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if every terminating computation in $(\mathcal{A}, \mathcal{R})$ is image under $\mathcal{F}$ of a terminating computation in $(\mathcal{B}, \mathcal{S})$, since in that case we may view every successful (failing) abstract computation as implemented by a successful (failing) concrete computation.

In case we establish both correctness and completeness in the above sense, as we do on every of our refinement steps, we may speak of operational equivalence of evolving algebras. Since this last notion is symmetric, it does not matter any more which way $\mathcal{F}$ goes. The attentive reader will notice that we indeed have, in a few places, found it more convenient to reverse the direction of $\mathcal{F}$, mapping the 'abstract' algebra into the 'concrete' one.

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **let** and **if then else**. We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever, trees etc (as well as the standard operations on them) at our disposal without further mention. We use usual notations, in particular Prolog notation for lists.

In the rest of this paper, the name "BABEL" will always refer to the innermost version of the BABEL language, as described in [KLMR90]. We do not presupose any specific knowledge of BABEL or its implementation. But obviously, some elementary knowledge of functional + logic programming languages can only be helpful, and some very basic facts about BABEL are reported here to help understanding our presentation of BABEL algebras.

BABEL's syntax is based on well-typed expressions built from variables, free constructors and function symbols. Types are statically checked at compile time. Expressions can be first-order terms using only constructors and variables, or possibly higher-order expressions involving also application, function symbols and predefined operations. These include boolean operations, a predefined equality, and operators used for building two kinds of *conditional expressions*:

$$\begin{array}{rll} (b \to e) & \text{meaning} & \textbf{if } b \textbf{ then } e \textbf{ else } \textit{undefined} \\ (b \to e_1 \# e_2) & \text{meaning} & \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 \end{array}$$

Conditional expressions of the form $(b \to e)$ are also called *guarded expressions*. They allow to express the effect of conditional rewrite rules, as well as Prolog clauses (taking the boolean constant **true** as e). In fact, BABEL programs consist of *defining rules* for function symbols, having the form

$$f(t_1, ..., t_n) := r$$

where r may be a guarded expression $(b \to e)$. In spite of the fact that a rule right hand side may be also not guarded, we shall use $f(t_1, \ldots, t_n) := b \to e$ as our default notation for BABEL rules. Number n, noted as $arity(f)$, is called the *program arity* of f.

The following are simple examples of defining rules which could be part of a BABEL program, using lists with constructors [] and [. | .] and natural numbers with constructors 0 and *suc*. They define functions called *map*, *plus* and *dominates*:

$$\begin{array}{l} map(F, [\,]) := [\,]. \\ map(F, [X|Xs]) := [@(F, X)|map(F, Xs)]. \end{array}$$

$$\begin{array}{l} plus(0, Y) := Y. \\ plus(suc(X), Y) := suc(plus(X, Y)). \end{array}$$

$$dominates(Xs, Ys) := map(plus(N), Ys) = Xs \to true.$$

This shows some typical uses of higher-order programming features in BABEL. Functions can be partially applied, as $plus(N)$ above, and an application operator @ is also available. BABEL's syntax does not use @ explicitely, but making it explicit is harmless and helps to simplify our descriptions. Note that higher-order variables (as F in the rules for *map*) can be used only in defining rules which introduce them in the left hand side. This, together with the absence of λ-terms in the syntax, ensures that the language can be executed without higher-order unification. Solving BABEL goals yields in general multiple solutions, which are computed by *narrowing* and *backtracking*. For instance, the rules above allow to compute two solutions for a particular goal, as indicated below:

$$\begin{array}{l}\textbf{solve}\ \ dominates([suc(0), X], [Y, 0]) \\ \quad \textbf{true}\ \ \{X/0, Y/suc(0)\} \\ \quad \textbf{true}\ \ \{X/suc(0), Y/0\}\end{array}$$

Before giving the formal specification of the operational semantics by means of evolving algebras, we still present an even simpler example which will serve as running example all along the paper, to help understanding our specification.

In the following program we define the function *append* returning the concatenation of two lists, and the predicate – defined as a *true*-valued function – *prefix* which is true for two lists when the first one is a prefix of the second one. In addition we have a function g simply switching the constants a and b.

$$\begin{array}{ll}(A_1) & append([], Ys) := Ys. \\ (A_2) & append([X|Xs], Ys) := [X|append(Xs, Ys)]. \\ (P_1) & prefix(Xs, Ys) := append(Xs, Zs) = Ys \rightarrow true. \\ (G_1) & g(a) := b. \\ (G_2) & g(b) := a.\end{array}$$

To solve, for instance, the goal $prefix([g(X), g(Y)], [a, X, b])$, means to find values (a most general substitution) for the variables X and Y such that the given expression is reducible to normal (irreducible) form by using the rules of the program and the predefined functions of the language. The reduction of expressions and the bindings for the variables is done simultaneously by *innermost narrowing*. Innermost means that redexes occupying a deeper position in the expression must be reduced before trying to reduce redexes in outer positions.

In our goal example there are two innermost redexes, namely $g(X)$ and $g(Y)$. One of them must be selected for narrowing it. A *selection function* could be considered for that purpose, as in the case of logic programming for selecting a literal to resolve upon. In BABEL the leftmost innermost redex is always selected.

In our example the leftmost innermos redex is $g(X)$, for which there are two rules – G_1 and G_2 – for narrowing it. These two alternatives would narrow the entire expression to $prefix([b, g(Y)], [a, X, b])$ and $prefix([a, g(Y)], [a, X, b])$ respectively, with respective substitutions $\{X/a\}$ and $\{X/b\}$. In both cases the new leftmost innermost redex is $g(Y)$.

It is clear that solving a goal generates a search space – similar to SLD-trees in logic programming – due to the different *don't know* alternative rules for the selected redex. The search tree for our example is depicted in Fig.1. At each node of the tree the current expression to be reduced is written (*append*, *prefix* and *true* are abbreviated) , and the position of the leftmost innermost redex is underscored. The branches for a node correspond to the alternative program rules which can be applied to the selected redex, and we annotate them with the substitution coming from unification with the head of the rule (only substitutions for the variables in the goal expression are written). The non failed leaves contain the normal forms obtained by different narrowing derivations, and for each one the answer substitution is built by composing all the substitutions in the path to the root, projected into the variables in the initial goal. In this particular tree the only answer consists of the normal form *true* together with the answer substitution $\{X/b, Y/a\}$.

As in the case of Prolog, the search over this tree in BABEL is done in a leftmost, depth first manner, with chronological backtracking in case of failure or asking for more solutions. We do not give here a precise definition of the construction of the tree and the search on it; instead, we leave our specification to do the work.

3 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 Babel program (a set of defining rules for the user defined functions,

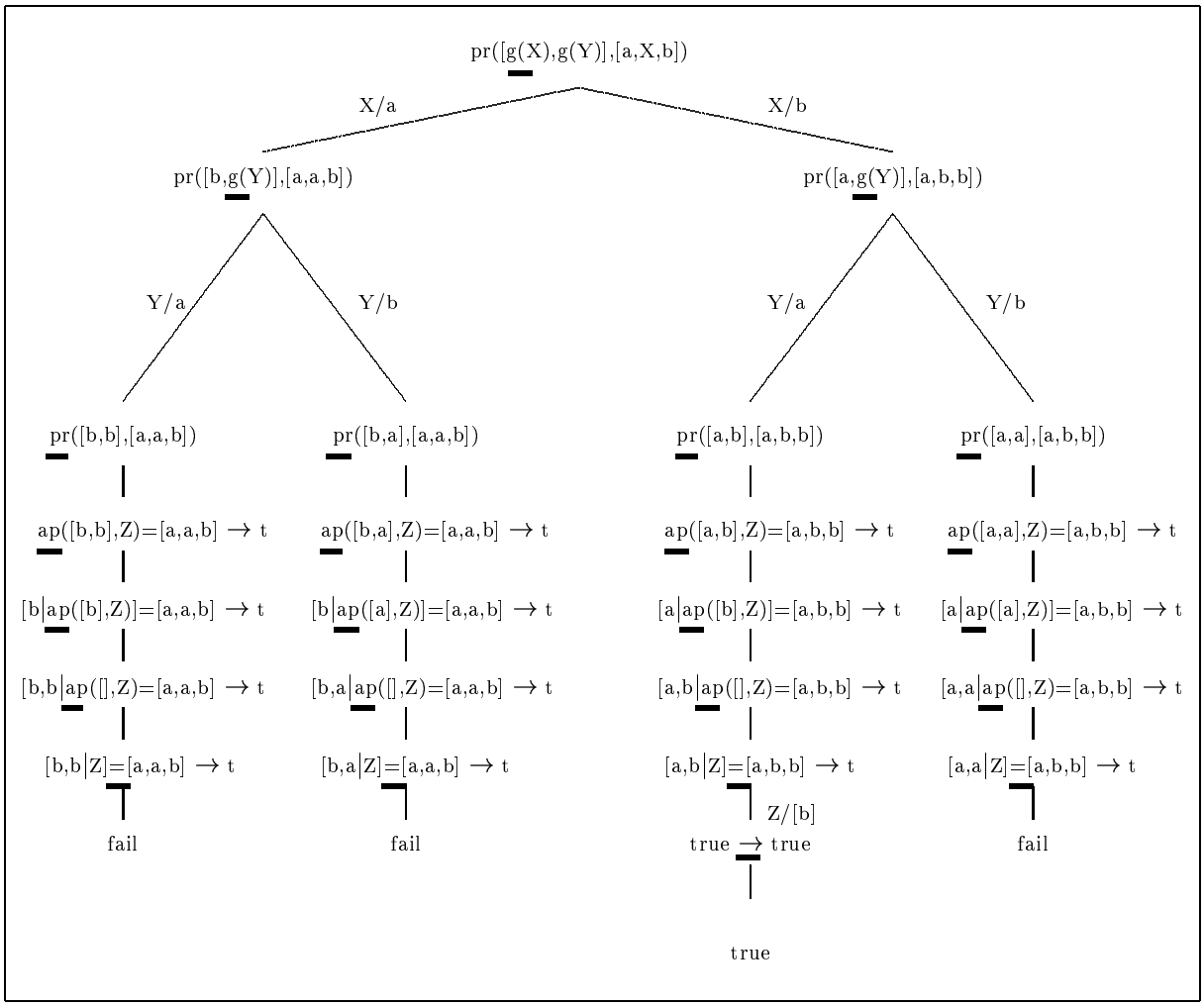


Figure 1: A narrowing tree

as explained in Section 2). Thus a description of Babel's operational semantics has to incorporate, besides reduction to normal form, the concepts of unification and backtracking which are well known from logic programming languages.

3.1 The narrowing–backtracking core

Our top level description of Babel reflects the backtracking structure as dynamically created *tree* structure with *father* relation representing backtracking to alternative (single) narrowing steps — corresponding to different rules for a certain function —, following an idea used already in [B̈90b] (in hybrid stack oriented form) and in [BR91, BR92a] as basis to define an abstract model for Prolog. Thus we represent *Babel computation states* in a set $NODE$ with two distinguished elements *root* and *currnode*—the latter representing the (dynamically) current state—and a total *backtracking father* function

$$bfather : NODE - \{root\} \rightarrow NODE$$

such that from each state (except *root*) there is a unique *bfather* path towards *root*. We create the *tree algebra*

$$(NODE; root, currnode; bfather)$$

dynamically as the computation proceeds, out of the initial state (determined by given program and expression) as the value of *currnode*, with the (empty) *root* as the value of *bfather(currnode)*.

Each element n of $NODE$ has to carry all information relevant—at the desired abstraction level—for the computation state it represents. At the top level, this information—we will call it *environment* of n—consists of the *expression* still to be reduced, the *substitution* computed so far, and the sequence of *alternative states* still to be tried, as we will explain below. When in *currnode* a *redex* $f(t_1, \ldots, t_n)$ with user-defined function f is encountered for reduction, we will create a son for each rule in the program which might be applied to the redex, to control the alternative computation threads. All such *candidate sons* are attached to n as a list $cands(n)$, in the order reflecting the ordering in which corresponding rules in the program should be applied. Therefore the signature of *cands* is

$$cands : NODE - \{root\} \rightarrow NODE^*$$

We require of course the *cands*-lists to be consistent with *bfather*, i.e. whenever *Son* is among $cands(Father)$, then $bfather(Son) = Father$.

This action of branching the tree with $cands(n)$ takes place at most once, when n gets first visited (in *Apply mode*) [2].

The mode then turns to *Select* for handling the alternatives. In *Select* mode a narrowing step is attempted, whereby the first son ¿from $cands(n)$ with applicable rule gets visited (becomes the value of *currnode*) and switches to mode *Eval* (for further evaluation of the resulting expression, see below). The selected son is simultaneously deleted from the $cands(n)$ list. If control ever returns to n, (by *backtracking*, cf. below), it will be in *Select* mode, and the next candidate son will be selected, if any.

If in *Select* mode $cands(n) = [\,]$, all attempts at narrowing from the state represented by n will have failed, and n will be *abandoned* by returning control to its backtracking father *bfather*. This action is usually called *backtracking*. The *bfather* function then may be seen as representing the structure of Babel's *backtracking behaviour*.

The *narrowing step*, applied at a son from the *cands* list with associated candidate rule (guarded equation) $f(t_1, \ldots, t_n) := b \rightarrow e$, is executed in terms of the *expression* attached to each state: if the redex $f(e_1, \ldots, e_n)$ in *expression* is unifiable with the left hand side $f(t_1, \ldots, t_n)$ of the defining rule associated to the son, the redex is replaced by the right hand side $b \rightarrow e$ of that defining rule, and the whole expression gets updated, for further evaluation (see below), by the (most general) unifier of the redex and the rule's left hand side. As we shall see, it is convenient to represent and parse also expressions in the form of *trees*, in order to describe the *search for redices* within an expression.

We complete now the definition of signature (statics) and transition rules (dynamics) which will make the above description precise. The environment will be associated to states by appropriate (in general partial) functions on the universe $NODE$. For each state we have to know the expression still to be reduced, provided by a function

$$exp \quad : \quad NODE \rightarrow EXPRESSION$$

where *EXPRESSION* is the domain of Babel expressions. Since reduction steps might take place in subexpressions of the given expression, we also introduce a function

[2] The different *modes* and their respective roles will become clear from the specifications we present below.

$$pos \quad : \quad NODE \ \rightarrow \ POSITION$$

providing for given state the *position* in the associated expression at which the next reduction step will take place. As usual, we assume that positions u are coded as finite sequences of positive integers; see e.g. [DJ90]. 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).

For accessing and manipulating subexpressions of given expressions (read: subtrees of trees), in connection with *POSITION* we will use four standard tree functions[3]: 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 [4] .

The substitution current at a state, accumulated from the unifications in preceding narrowing steps, is represented by a function

$$s \quad : \quad NODE \ \rightarrow \ SUBST$$

where $SUBST$ is a universe of abstract *substitutions* with a function

$$mgu \quad : \quad EXPRESSION \times EXPRESSION \ \rightarrow \ SUBST \cup \{nil\}$$

which at this level of description remains abstract, associating to two expressions either a substitution (their most general unifier), or the answer that there is none. For technical convenience we assume that *mgu* returns an idempotent substitution. Application of substitution θ to an expression e, and its composition with another substitution ρ will be denoted by the usual postfix notation, $e\theta$, $\theta\rho$.

The above mentioned switching of *modes* will be represented by a distinguished element $mode \in \{Apply, Select, Eval, \ldots\}$ indicating the action to be taken at *currnode*: creating the alternative states for narrowing by rule application, selecting among them, evaluating the expression obtained by narrowing, etc. To be able to speak about *termination* we will use a distinguished element $stop \in \{0, 1, -1\}$, to indicate respectively running of the system, halting with success and final failure.

We shall keep the above mentioned notion of *candidate defining rule* (for applying a narrowing step to an expression) abstract (regarding it as implementation defined), assuming only the following integrity constraints: every candidate (guarded) defining rule for a given expression

- has the proper function (symbol), i.e. the same function as the expression (*correctness*); and
- every defining rule whose left hand side unifies with the given expression is candidate defining rule for this expression (*completeness*).

Access to candidate equations is thus handled by a function

$$fundef \quad : \quad EXPRESSION \times PROGRAM \ \rightarrow \ RULES^{+},$$

of which we assume only to yield the (properly ordered) list of the candidate defining rules for the given expression in the given program. The program is represented by a distinguished element *pg* of *PROGRAM* (the *program*). Note that existence of *fundef* is all that we assume of the abstract universe *PROGRAM*.

This concludes the definition of the signature of Babel tree algebras. Minor additions, pertaining only to some special constructs, will be presented in corresponding sections.

Notationally, since we deal mainly with values of functions at *currnode*, we usually suppress this parameter writing

$$\begin{array}{ccc} bfather \equiv bfather(currnode) & cands \equiv cands(currnode) & fst\_cand \equiv fst(cands(currnode)) \\ exp \equiv exp(currnode) & pos \equiv pos(currnode) & s \equiv s(currnode), \end{array}$$

exp, pos and *s* constitute the *environment* of *currnode*. The selected *exp*-subexpression —the *current expression*—is accessed from the environment by

$$currexp \quad \equiv \quad exp[pos].$$

Now to dynamics. We assume the following **initialization of Babel tree algebras**: The child of *root* is supposed to be the initial value of *currnode*; it has the *initial expression* as associated expression, *pos* is the empty position and *s* the empty substitution; the mode is *Eval*, *stop* has value 0; *pg* has the given program as value. The list *cands* of alternative states is not (yet) defined at *currnode*. This initial algebra is graphically shown in Fig.2 for our running example. The current node *currnode* is boldfaced and the current position

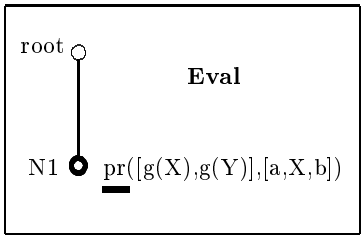


Figure 2: Initial Algebra

pos is underscored; the edges in the tree represent the *bfather* relation. The *mode* can be found in the right upper part of the tree.

We now define the rules by which the system attempts, given a Babel program, to reach a state with $stop = 1$ (due to final success of the computation) or with $stop = -1$ (due to final failure by backtracking all the way to *root*). No transition rule will be applicable in such a case, which is a natural notion of 'terminating state'. All transition rules will thus be tacitly assumed to stand under the guard $stop = 0$. We introduce the following abbreviation for backtracking:

$$\begin{array}{lll} backtrack & \equiv & \textbf{if } bfather = root \textbf{ then } stop := -1 \\ & & \qquad\qquad \textbf{else } currnode := bfather \\ & & \qquad\qquad\quad mode := Select \end{array}$$

A reduction step of a redex $f(t_1, \ldots, t_n)$ with user-defined f and arguments t_i in normal form is split into *applying* the function call (to create new candidate nodes for alternative states of *currnode*), to be followed by *selecting* one of them. We will correspondingly have two rules. The following **call rule** (*Call*), invoked by having a user–defined function call in *Apply* mode, will create as many sons of *currnode* as there are candidate defining rules in the procedure definition of its function symbol, to each of which the corresponding defining rule will be associated.

$$\begin{array}{l} \textbf{if } is\_user\_defined(currexp) \,\&\, mode = Apply \\ \quad \&\, currexp = f(e_1, \ldots, e_n) \,\&\, 0 \leq n = arity(f) \\ \textbf{then let } [\, dr_1, \ldots, dr_m \,] = fundef(currexp, pg) \\ \qquad \textbf{extend } NODE \textbf{ by } n_1, \ldots, n_m \textbf{ with} \\ \qquad\quad bfather(n_i) := currnode \\ \qquad\quad defrule(n_i) := dr_i \\ \qquad\quad cands := [\, n_1, \ldots, n_m \,] \\ \qquad \textbf{endextend} \\ \qquad mode := Select \end{array}$$

where *is_user_defined* is a Boolean function recognizing those expressions whose function symbols are user defined (as opposed to free constructors and predefined operations), *defrule* is an auxiliary labeling function, *arity(f)* indicates the (program) arity of f. Note that expressions and substitutions, attached to candidate sons, are at this point undefined, and that the value of *currnode* does not change.

The following **selection rule** (*Sel*) attempts to select a candidate state (selecting thereby the associated rule). If there is none, the system backtracks. If the left hand side of the renamed selected rule does not unify with the call pattern, the corresponding son is erased ¿from the list of candidates. Otherwise the selected rule of the first successful candidate is activated: the corresponding son becomes the value of *currnode* in *Eval* mode (and gets erased from its father's *cands* list), its environment is defined as the result of narrowing applied to the current environment—replacing in *currnode*'s expression the current subexpression by the right hand side of the rule and applying the unifying substitution to both s and (new) *exp* for which the evaluation has to proceed then. Renaming of variables in an expression e is realized abstractly by introducing the current *variable renaming index* $vi \in \mathcal{N}$, and denoting by *rename (e,vi)*—or shorter e'—the result of e

[3] We consider these functions here as static, i.e. functions which are known and not updated by rules. One might view them as procedures.See [Gur91]

[4] $u.(i+1)$ is (next to the right) brother position of $u.i$, $u.1$ is the leftmost child of position u and u is father of all positions $u.i$.

after renaming of all variables at level vi. The update $vi := vi+1$ ensures freshness of subsequent renamings.

if $mode = Select$
thenif $cands = [\,]$ **then** *backtrack*
else let $(Lhs = Rhs) = defrule(fst\_cand)$
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 *fst_cand* denoting the first element in *cands* and with abbreviations:

$$\begin{aligned} go\_fst\_cand\ in\ Eval \quad &\equiv \quad currnode := fst\_cand \\ &\qquad\ mode := Eval \\ narrow\ curr\_env(E, \theta) \quad &\equiv \quad exp(fst\_cand) := exp[\,pos \leftarrow E\,]\,\theta \\ &\qquad\ pos\,(fst\_cand) := pos \\ &\qquad\ s(fst\_cand) := s\,\theta \end{aligned}$$

The application of the *Call* and *Select* rules for the running example is shown in Fig.3. Some steps of the backtracking due to the failure of the left part of the tree (see Fig.1) are shown in Fig.4, where discontinuous lines are used for that part of the tree which will not be visited any more, that is, abandoned nodes.

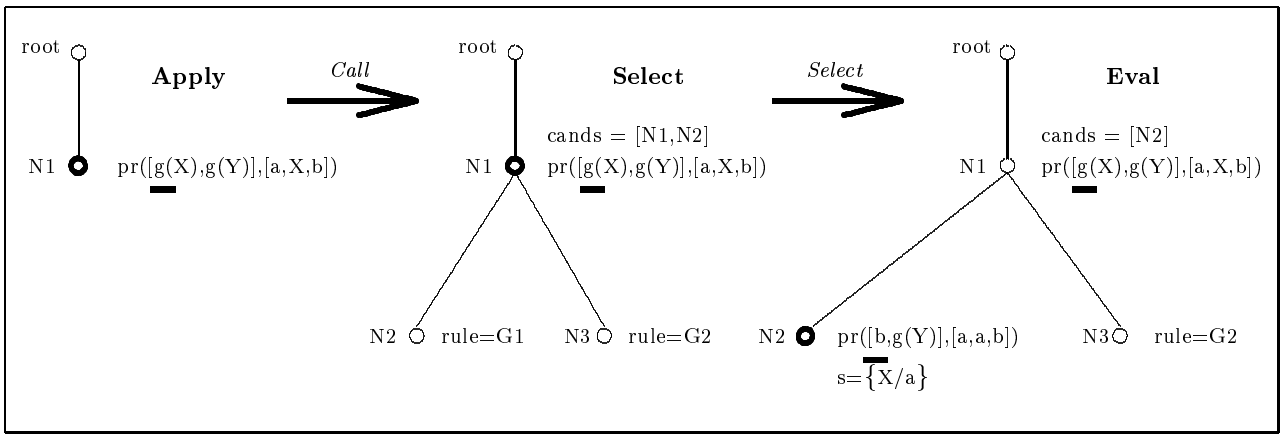


Figure 3: *Call* and *Select* rules

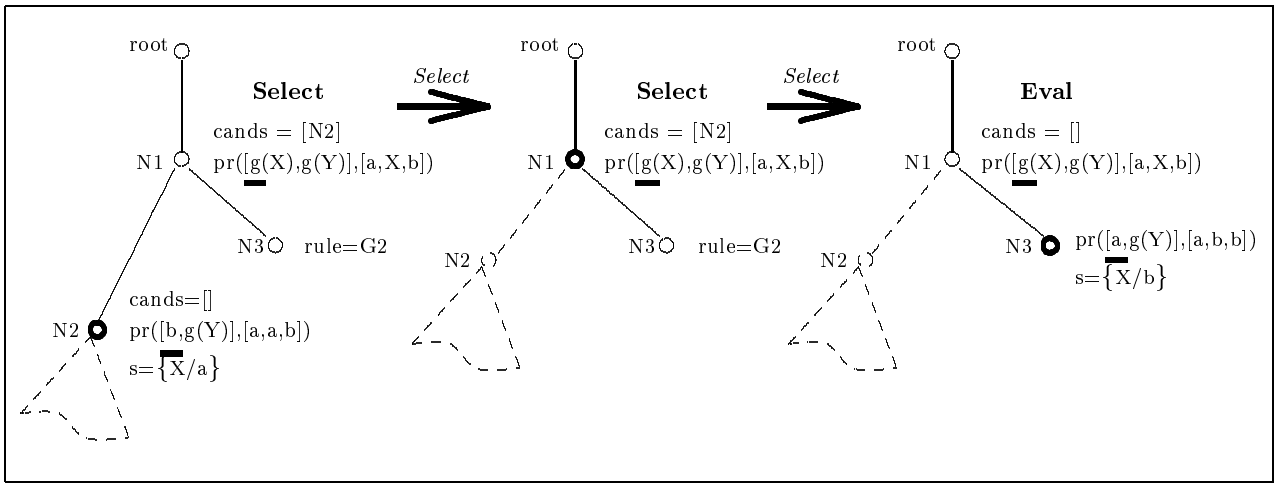


Figure 4: Bactracking

3.2 Rules for reduction to normal form

In this section we define the rules for evaluation of expressions 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 position *pos*.1, i.e. the extension of the current position by 1) 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 following **evaluation starting rule** (*EvStart*):

$$\begin{array}{lll} \textbf{if } mode = Eval \textbf{ thenif} & is\_not\_in\_normal\_form(currexp) \\ & \textbf{then } pos := pos.1 \\ & \textbf{else } mode := Continue \end{array}$$

where *is_not_in_normal_form* is an auxiliary Boolean–valued function (with the obvious meaning) and *Continue* a possible value of *mode* whose appearence here 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 (whose arguments have been already reduced to normal form). There is a special case: some predefined functions are considered as strict [5] only in the first argument. For these functions, once the first argument has been reduced to normal form, the evaluation must proceed at the father position in mode *Apply*. All this is formalized by the following **evaluation continuation rule** (*EvCont*) :

$$\begin{array}{lll} \textbf{if } mode = Continue \,\&\, pos = u.i \\ \textbf{thenif} & \neg occurs(u.(i+1), currexp) \textit{ or } nonstrict(u) \\ & \textbf{then} & pos := u \\ & & mode := Apply \\ & \textbf{else} & pos := u.(i+1) \\ & & mode := Eval \end{array}$$

where *nonstrict(u)* tells whether the subexpression *exp*[*u*] is an application of one of the predefined functions which are strict only in the first argument, i.e. it takes one of the forms

$$and(b_1, b_2), \quad or(b_1, b_2), \quad b \to e, \quad b \to e_1 \# e_2.$$

These expressions need special rules, given below, to describe their evaluation.

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. The latter feature is formalized abstractly by using a Boolean valued 0-ary function *more*—external in the sense of [Gur91]—to take care of external request of more solutions after a solution has been found. Otherwise the computation terminates with final success. This is described by the following **stop rule** (*Stop*) :

$$\begin{array}{lll} \textbf{if } mode = Continue \,\&\, pos = \epsilon \\ \textbf{then} & solution\_list := [\langle exp, s \mid_{Goalvars} \rangle \mid solution\_list] \\ & \textbf{if } more = 1 & \textbf{then } backtrack \\ & & \textbf{else } stop = 1 \end{array}$$

By $s \mid_V$ we denote the restriction of the substitution 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.

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 a partial application of a user defined function, it may consist of an application of the "apply" operator @, and last but not least, it may be the application of some other predefined function. For each of these cases there is a corresponding rule.

If in *Apply* mode *currexp* is an expression formed by a constructor c with number of arguments which does not exceed the constructor's arity *arity(c)* or an expression formed by a user–defined function but with less arguments than the (program) arity *arity(f)* of this function *f*—we denote this by a Boolean valued function *is_construction* [6] —, then the evaluation of *currexp* has finished and must *continue* at another (brother– or father–) expression, as described by the following **construction rule** (*Construction*) :

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

[5] 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.

[6] Note that partial applications of user defined functions are viewed in BABEL as analogous to applications of free constructors.

If the current subexpression is an expression $@(e_0, e_1)$ then the arguments e_i have already been reduced to normal form, and due to well-typedness e_0 is a partial application[7]. The system simply adds the second expression as further argument to the first expression. Note that *mode* does not change. This is described by the following **partial application rule** (*PApp*) :

$$\begin{array}{l}\textbf{if} \; mode = Apply \;\&\; currexp = @(e_0, e) \;\&\; e_0 = op(e_1, \ldots, e_m) \\ \textbf{then} \; exp := exp[pos \leftarrow op(e_1, \ldots, e_m, e)]\end{array}$$

where *op* is a constructor or a function symbol.

For the sake of example we show here two cases for predefined functions, namely the self–explanatory **rules for conditionals**: [8]

$$\begin{array}{l}\textbf{if} \;\; currexp = b \rightarrow e \\ \quad \& \; mode = Apply \\ \textbf{thenif} \;\; b = true \\ \qquad \textbf{then} \;\; exp := exp[pos \leftarrow e] \\ \qquad\qquad mode := Eval \\ \qquad \textbf{else} \; backtrack\end{array} \qquad \begin{array}{l}\textbf{if} \;\; currexp = b \rightarrow e_1 \# e_2 \\ \quad \& \; mode = Apply \\ \textbf{then if} \;\; b = true \\ \qquad \textbf{then} \; exp := exp[pos \leftarrow e_1] \\ \qquad \textbf{else} \; exp := exp[pos \leftarrow e_2] \\ \qquad mode := Eval\end{array}$$

Similar rules for the other predefined Boolean expressions are skipped here. The **equality rule** (*Eq*) — asking for unifiability of the equated expressions—relies upon the abstract unification function *mgu* introduced above:

$$\begin{array}{l}\textbf{if} \; currexp = eq(e_1, e_2) \;\&\; mode = Apply \\ \textbf{then let} \; mgu = mgu(e_1, e_2) \\ \qquad \textbf{if} \; mgu = nil \; \textbf{then} \; exp := exp[pos \leftarrow false] \\ \qquad\qquad\qquad \textbf{else} \;\; exp := (exp[pos \leftarrow true])mgu \\ \qquad\qquad\qquad\qquad s := s \; mgu\end{array}$$

4 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. We indicate at each step how to prove the correctness of the refinement.

4.1 Structure sharing for expressions

Here we introduce structure sharing for expressions: when during narrowing the left hand side of an equation 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*),*exp* in the selection, equality rule respectively is changed to be without applying the unifying substitution, and b) to those occurrences of *currexp* or expressions e, e_1, e_2, b in the rules the substitutions must be applied, where the condition to be checked or the operation to be performed on those expressions really depend on the substitution. [9] The latter happens in the following cases:

- in *currexp* when *procdef* is used in the call rule for collecting the candidate equations for narrowing *currexps*, [10]
- in the definition of *mgu* in the selection and equality rule,
- in *exp* when *solution_list* is updated in the stop rule,
- in b for the test $b = true$ in the rules for the predefined conditionals, and

[7] i.e. its number of arguments is smaller than the arity of its functor.

[8] The reader will notice that the case where b is a variable, say X, is missing. The two alternatives for the computation to proceed correspond to the bindings $\{X/true\}$ and $\{X/false\}$. For the sake of simplicity, instead of describing the branching by new rules, we assume X to be replaced by an expression *bool_gen*(X), where *bool_gen* is a user-defined function assumed to be in every program, with *BABEL* defining rules *bool_gen*(*true*) = *true* and *bool_gen*(*false*) = *false*. The *Call* rule for *bool_gen* will then take care of the branching. This is indeed what is done in the real implementation.

[9] This refinement, in combination with the normal form test refinement of the next subsection, allows to avoid the traversal of complex terms which originated from substitutions. Note that s always substitutes normal forms for variables because the patterns in rule heads are normal forms and because arguments of a function application are reduced to normal form before the function rule is applied. Hence e is in normal form iff $e \; s$ is.

[10] To consider here *currexps* is not needed if *procdef* depends only on the outer function symbol f of *currexp* – i.e. if *procdef* returns all the program rules for f –. But it would be needed for a possible refinement including some indexing mechanism.

- in e for the condition $e = op(e_1, \ldots, e_m)$ in the partial application rule; $e\,s$ is needed because e might be a higher order variable coming from a program rule. As an example, consider F in the rules for *map* in the example of section 2; when we reduce, f.i., $map(plus(0), [0])$ replacing it by the right handside of the second rule for *map*, the (sub)expression $@(F, X)$ is generated, together with the substitution $\{F/plus(0), X/0\}$. We need to 'dereference' F for the partial application rule, in order to reduce $@(F, X)$ to $plus(0, X)$.

The *proof mapping* $\mathcal{F}$: $(\mathcal{A}_1, \mathcal{R}_1) \rightarrow (\mathcal{A}_0, \mathcal{R}_0)$ is defined as follows: $\mathcal{F}$ preserves all the universes, all the rules (in the sense that $\mathcal{F}$ maps rules from $\mathcal{R}_1$ to homonymous rules from $\mathcal{R}_0$), and all the operations except *exp*. Given an instance A_1 of $(\mathcal{A}_1, \mathcal{R}_1)$, for all $n \in NODE$

$$exp_{\mathcal{F}(A_1)}(n) = [exp_{A_1}(n)]s(n)$$

4.2 Refining normal form test

In this section we refine 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—namely a variable, a constructor symbol or a user–defined non evaluable function symbol (i.e. of arity > 0)—is reached. [11] Thus the **new evaluation starting rule**:

$$\begin{array}{l} \textbf{if } mode = Eval \textbf{ thenif } \neg atomic(currexp) \textbf{ then } pos := pos.1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } mode := Continue \end{array}$$

where $atomic(e)$ recognizes if the expression e is a variable, a constructor symbol c or a function symbol f with $arity(f) > 0$.

The *proof mapping* $\mathcal{F} : (\mathcal{A}_1, \mathcal{R}_1) \rightarrow (\mathcal{A}_2, \mathcal{R}_2)$ preserves all the universes and operations and maps $\mathcal{R}_1$-rules to homonymous $\mathcal{R}_2$-rules except the evaluation starting rule. A single application of this rule in $\mathcal{R}_1$ may correspond to a sequence of rules in $\mathcal{R}_2$, if *currexp* is in normal form and of form $\varphi(t_1, \ldots, t_n)$, with $n > 0$ and φ a constructor symbol c or a user–defined function symbol f with $n < arity(f)$ [12]. In this case $(\mathcal{A}_2, \mathcal{R}_2)$ requires a sequence of rules among evaluation starting/continuation/stop rules in order to recognize that the subexpression is in normal form.

The sequence $traversal(currexp)$ of $\mathcal{R}_2$-rules which corresponds to an appearance of *EvStart* in a sequence of $\mathcal{R}_1$-rules is defined by

$$\begin{array}{l} traversal(e) \;=\; EvStart \quad \text{if } e = X \mid c \mid f \\ traversal(\varphi(t_1, \ldots, t_n)) \;=\; EvStart, traversal(t_1), EvCont, \ldots, traversal(t_n), EvCont, Construction \\ \text{if } \; e = \varphi(t_1, \ldots, t_n), \;\; \text{where } \; \varphi = c \;\text{ or }\; \varphi = f,\; n < arity(f) \end{array}$$

4.3 From backtracking tree to backtracking stack

Classification of Babel tree nodes suggests a straightforward *stack* representation: disregarding the abandoned nodes (since they, once abandoned, play no further role in the computation), we may view the path of active nodes as a stack—if *cands* lists are represented in some other way. It suffices to change the signature of *cands* to

$$cands \quad : \quad NODE - \{root\} \;\rightarrow\; RULES^*$$

and to update—when creating (and passing control to) *one* new state by action of the *Call* rule—the value of *cands* for this new state to the complete list of rules applicable to that call. Backtracking will then consist in looking for remaining candidate *rules*.

In the following **new call rule** we create therefore one new state which becomes current state, which upon failure will backtrack to *currnode*, and which through appropriate decoration of its *cands* value will control execution of the alternatives for the current call. To keep our correctness proof simple, we let here the new node fetch its environment for narrowing from its backtracking father *currnode* [13].

[11] 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.

[12] Note that $n = arity(f)$ would mean that *currexp* is not in normal form and that the call rule could be applied.

[13] In the later refinement to states controlling subcomputations, a copy of the narrowing environment will be stored at the new node, as it happens in the IBAM.

if $is\_user\_defined(currexp)$ & $mode = Apply$
& $currexp = f(e_1, \ldots, e_n)$ & $0 \leq n = arity(f)$
then extend $NODE$ **by** N **with**
$bfather(N) := currnode$
$currnode := N$
$cands(N) := fundef(currexp, pg)$
endextend
$mode := Select$

The **new selection rule** checks whether there are (still) rules to consider; in positive case it executes the narrowing step and restores at *currnode* the relevant information (expression, position, substitution) for the now starting evaluation of the narrowed expression. Remember that this information is kept by the backtracking father (which is responsible for the currently executed call).

if $mode = Select$ **thenif** $cands = [\,]$
then *backtrack*
else let $(Lhs = Rhs) = fst\_cands$
let $\theta = mgu(exp(bfather)[pos(bfather)]\ s(bfather), Lhs')$
$cands := rest(cands)$
if $\theta \neq nil$ **then** $mode := Eval$
$narrow\ bfather\_env(Rhs', \theta)$
$vi := vi + 1$

where

$$narrow\ bfather\_env(E, \theta) \equiv \begin{array}{l} exp := exp(bfather)[pos(bfather) \leftarrow E] \\ pos := pos(bfather) \\ s := s(bfather)\ \theta \end{array}$$

The effect of the new *Call* and *Select* rules for our running example is shown in Fig.5.

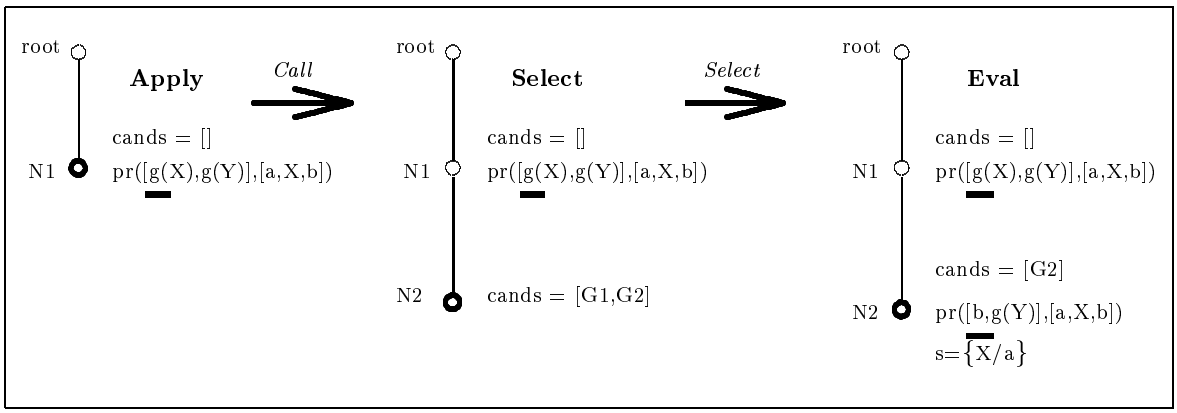


Figure 5: *Call* and *Select* rules in Stack Algebras

For the definition of *backtrack* to work properly, *cands* must be initialized in this algebra to be [ ] for the child of *root* (the initial *currnode*). Otherwise, since the new call rule attributes *cands* to the child created for *currnode*, *cands* would remain undefined for the child of *root*, and the condition $cands = [\,]$ in the selection rule would crash if backtracking takes place at the child of *root*.

We define the *proof mapping* by a function $\mathcal{F}$ which associates homonymous rules and maps stack elements to tree nodes using an auxiliary function $F : NODE_3 \rightarrow NODE_2$ such that:

$$\begin{array}{cc} exp(F(n)) = exp(bfather(n)) & pos(F(n)) = pos(bfather(n)) \\ s(F(n)) = s(bfather(n)) & cands(F(n)) = mk\_cands(F(n), cands(n)) \\ bfather(F(n)) = F(bfather(n)) & F(root) = root \end{array}$$

where $mk\_cands(n, [l_1, \ldots, l_m]) \equiv [\langle n, l_1\rangle, \ldots, \langle n, l_m\rangle]$ keeping in mind that candidate nodes in the tree can be thought of as $\langle node, defrule\rangle$ pairs, since these are their only decorations. Consequently when *fst_cands* is chosen by rule Sel_3, then $F(currnode_3) = currnode_2$ is a pair $\langle n, fst\_cands\rangle$.

5 Babel graph algebras

In this section the normal form computation for expressions will be 'localized'. In order to keep the correctness proof simple, this is done in two steps: introduction of states which control normalization of subexpressions, completed by subsequent replacement of 'global' by dedicated 'local' environments.

5.1 States controlling subcomputations

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 calls are performed, the created state remains current one until the end of the computation for the whole *exp*.

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. We will call this state the *activator* of the state created by the call and denote it 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 (global variable) *lastnode* is introduced to store the (chronologically) last created node. 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 *Sel* rules. Since in this first step towards 'local', subexpression-normalization, the global expressions are still kept (to simplify the proof map) 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*, using the abbreviation

$$\begin{array}{lll} store\ act\_env\ at\ N & \equiv & act\_exp(N) := exp \\ & & act\_pos(N) := pos \\ & & act\_s(N) := s \end{array}$$

if $is\_user\_defined(currexp)\ \&\ mode = Apply$
$\&\ currexp = f(e_1, \ldots, e_n)\ \&\ 0 \leq n = arity(f)$
then extend *NODE* **by** N **with**
$currnode := N$
$cands(N) := fundef(currexp\ s, pg)$
$bfather(N) := lastnode$
$act\_node(N) := currnode$
$store\ act\_env\ at\ N$
endextend
$mode := Select$

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*. Formally:

$$\begin{array}{l} \theta = mgu(act\_exp[act\_pos]\ act\_s, Lhs') \\ narrow\ act\_env(E, \theta) \equiv \begin{array}{l} exp := act\_exp[act\_pos \leftarrow E] \\ pos := act\_pos \\ s := act\_s\,\theta \end{array} \end{array}$$

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* $nonstrict(u)$
  ...

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, $pos = act\_pos = \epsilon$). As a consequence, the condition $mode = Continue \,\&\, pos = \epsilon$ in the *Stop rule* changes to $mode = Return \,\&\, bfather = root$ in the **modified stop rule**.

The new mode *Return* is governed by a **Return rule** (*Ret*) 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
  $mode := Continue$

with the abbreviation

$$\begin{array}{lll} return\ curr\_env\ to\ act\_env & \equiv & exp(act\_node) := exp \\ & & pos(act\_node) := pos \\ & & s(act\_node) := s \end{array}$$

In Fig.6 some relevant steps for our running example are displayed. We have graphically emphasized the *act_node* relation, which is represented by solid edges. The resulting tree can be understood as a *task tree* . The *bfather relation* is represented by discontinous arrows, while *lastnode* is no explicitly drawn, since it is always the deepmost rightmost active node. Following these arrows from *lastnode* up to *root*, the backtracking stack can be reconstructed. For the sake of clarity, the decoration of nodes is omitted when unchanged or irrelevant.

At step (a) the *Call rule* is going to be applied for narrowing $g(X)$. A new node N_2 is created and activated for this purpose, its backtracking father and activator node being the previously current one. The activator environment (which is framed in the picture) is copied into N_2 (step (b)). This safe copy will remain unchanged for all the life of N_2. After using the program rule G_1 for narrowing $g(X)$ into b, step (c) will be reached, where the *Return rule* gives back the current environment to the activator, as the result of the subcomputation (step (d)). The computation proceeds under the control of N_1, and after some steps the nodes N_3 and N_4 will be created for reducing $g(Y)$ and $prefix([b,b],[a,X,b])$ respectively; both N_3 and N_4 have N_1 as activator, but their backtracking fathers are N_2 and N_3 respectively, since the latter were the respective last nodes when N_3 and N_4 were introduced (steps (e) and (f)).

In (g) we show the situation encountered when N_3 is reactivated for trying the alternative rule G_2, after the failure of all the alternatives for N_4 and its created descendants (they are abandoned nodes now). N_3 is at this point as it was in (f). Neither the environment of its activator N_1 (which also remains as it was in (f)) or of its backtracking father N_2 (which remains as it was in (c)) can be used for restoring the the computation to the state as it was when N_3 was created. This clearly shows the role of the safe copy Env_2 stored together with N_3.

Finally, (i) illustrates how a succesful derivation is going to finish in mode *Return* with the child of *root* as *currnode*, having the empty position and the computed normal form and substitution as environment. The *Stop rule* would add the computed solution to the list of solutions, and would ask the user for more solutions to be computed by backtracking.

For the *proof mapping* $\mathcal{F}$ from $(\mathcal{A}_4, \mathcal{R}_4)$ to $(\mathcal{A}_3, \mathcal{R}_3)$, let A_4 be an instance of $(\mathcal{A}_4, \mathcal{R}_4)$. The instance $A_3 \equiv \mathcal{F}(A_4)$ of $(\mathcal{A}_3, \mathcal{R}_3)$ is defined as follows: let $S_0, \ldots, S_n \in NODE_{A_4}$ be a *bfather*-chain from $root_{A_4}$ to $lastnode_{A_4}$, i.e.

$$S_0 = root_{A_4} \quad S_n = lastnode_{A_4} \quad S_{i-1} = bfather_{A_4}(S_i), \text{for} 1 \leq i \leq n$$

Then we define

$$NODE_{A_3} = \{S_0, \ldots, S_n\} \quad currnode_{A_3} = S_n \quad root_{A_3} = S_0$$

The A_3-environment for S_n is that of $currnode_{A_4}$

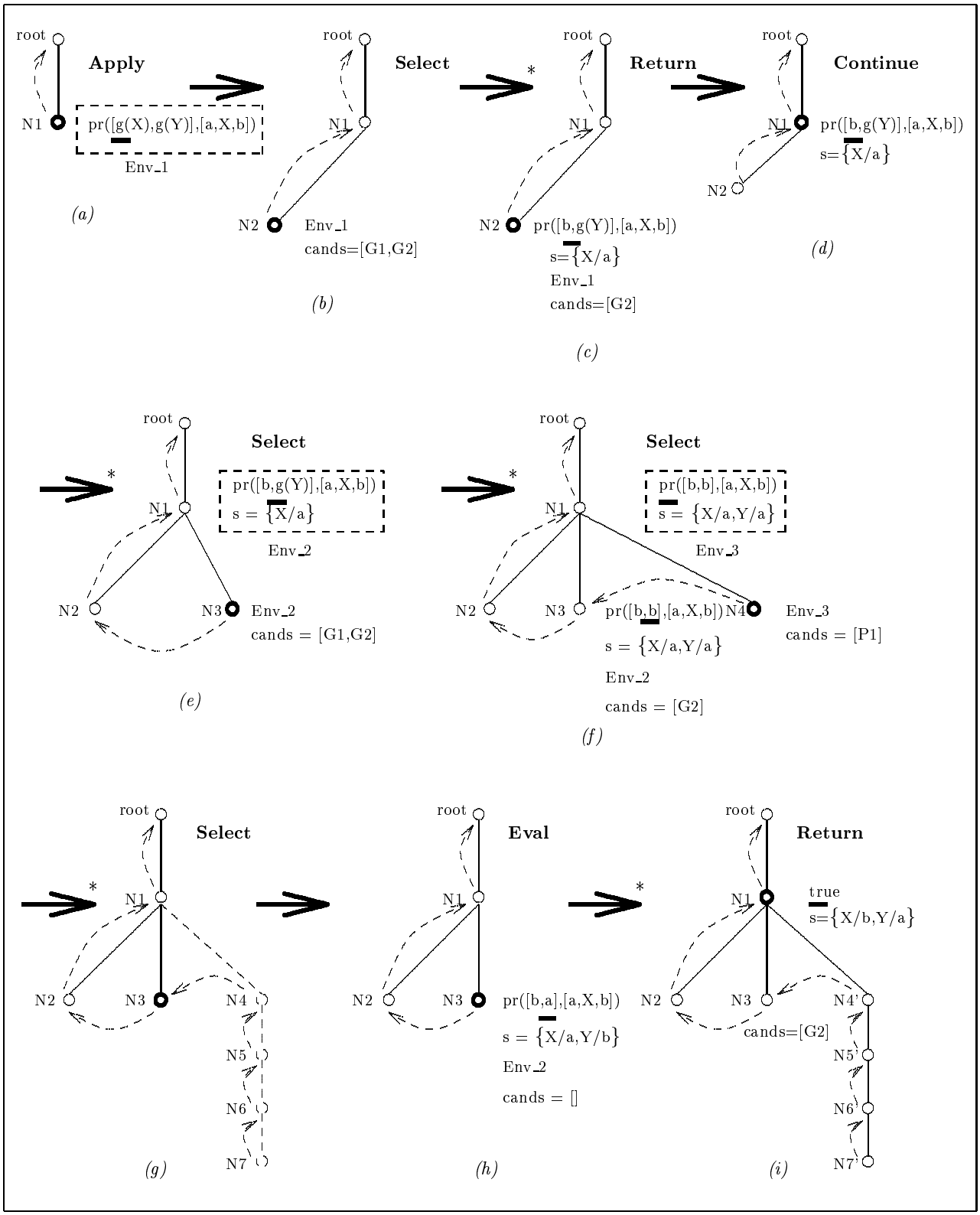


Figure 6: Graph Algebras

$$\begin{aligned} exp_{A_3}(S_n) &= exp_{A_4}(currnode_{A_4}) \\ pos_{A_3}(S_n) &= pos_{A_4}(currnode_{A_4}) \\ s_{A_3}(S_n) &= s_{A_4}(currnode_{A_4}) \end{aligned}$$

The A_3-environment for the remaining nodes in $NODE_{A_3}$ is reconstructed from the *act_env* in the backtracking chain: therefore, for all $1 \leq i \leq n-1$

$$\begin{aligned} exp_{A_3}(S_i) &= act\_exp_{A_4}(S_{i+1}) \\ pos_{A_3}(S_i) &= act\_pos_{A_4}(S_{i+1}) \\ s_{A_3}(S_i) &= act\_s_{A_4}(S_{i+1}) \end{aligned}$$

$mode_{A_3}$ is the same as $mode_{A_4}$, if it is not *Return*. Otherwise $mode_{A_3}$ is *Continue*. For the mapping of a sequence R_4 of rules in $\mathcal{A}_4$ to a sequence R_3 in $\mathcal{A}_3$, let us notice that a single occurrence of *EvCont* in $\mathcal{A}_3$ is followed in $\mathcal{A}_4$ by *Ret*, *EvCont* if it correspond to a just finished subcomputation (more of such pairs of rules will appear if there is a chain of returns). At the end of the computation, *Stop* applies in $\mathcal{A}_3$, while *EvCont*, *Stop* are needed in $\mathcal{A}_4$. Therefore, given R_4, R_3 is the result of deleting in R_4 the pairs *EvCont*, *Ret*, and replacing *EvCont*, *Stop* by *Stop*.

5.2 Subcomputations with local expressions

In this refinement step the 'global' environments are replaced 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)*. Notice that the use of the safe copies *act_exp, act_pos* of the activator environment has moved from the *Select* rule to the *Return* rule. In addition, due to the redefinition of *pos* in the selection rule, the condition $pos = act\_pos$ in *EvCont* rule for switching to mode *Return* must be replaced by $pos = \epsilon$.

The *proof mapping* $\mathcal{F}$ from $(\mathcal{A}_4, \mathcal{R}_4)$ to $(\mathcal{A}_5, \mathcal{R}_5)$ preserves all universes, rules and operations, except *exp, pos, act_exp* and *act_pos*, which are 'local' in $\mathcal{A}_5$. For a given instance A_4 of $\mathcal{A}_4$ we define $A_5 \equiv \mathcal{F}(A_4)$ as follows: for each $S \in NODE - \{root\}$

$$\begin{aligned} pos_{A_5}(S) &= \textbf{if } bfather(S) = root \textbf{ then } pos_{A_4}(S) \\ &\quad \textbf{else } u \textbf{ where } act\_pos_{A_4}(S) \cdot u = pos_{A_4}(S) \\ exp_{A_5}(S) &= \textbf{if } bfather(S) = root \textbf{ then } exp_{A_4}(S) \\ &\quad \textbf{else } exp_{A_4}(S)[act\_pos_{A_4}(S)] \\ act\_pos_{A_5}(S) &= \textbf{if } bfather(S) = root \textbf{ then } act\_pos_{A_4}(S) \\ &\quad \textbf{else } u \textbf{ where } act\_pos_{A_4}(act\_node_{A_4}(S)) \cdot u = act\_pos_{A_4}(S) \\ act\_exp_{A_5}(S) &= \textbf{if } bfather(S) = root \textbf{ then } act\_exp_{A_4}(S) \\ &\quad \textbf{else } act\_exp_{A_4}(S)[act\_pos_{A_4}(S)] \end{aligned}$$

6 Optimizations

In this section we introduce abstract versions of some time and space optimizations in the IBAM. Their role and justification are much better understood in this more abstract setting than at the implementation level [14].

6.1 Return without restoring

When a task returns a value , it is not always needed to use the safe copies (*act_exp, act_pos*) of its activator for returning to it the computed value. In some situations the current values *exp(act_node), pos(act_node)*

[14] There is another optimization in the IBAM which is not covered by this section, namely *detection of deterministic computations*, also called *dynamic cut*. It consists in the following: when applying a program rule for narrowing an expression, if no variable becomes bound during unification with the head of the rule and solving the conditions in the guard, then there is no need of trying alternative rules, since they cannot produce more general solutions. This should be not considered as an optimization of the implementation, but of BABEL operational semantics itself, hence the natural place for considering it is at the most abstract description level of the operational semantics. It would not be difficult to do so and to proof the correctness of the optimization, relying on the theorems [LW91] stating the safeness of dynamic cut.

can be used instead for this purpose. This will save the time of restoring the activator with these safe values. The obvious conditions for this to be correct are

$$\begin{array}{ll} (i) & act\_exp[act\_pos \leftarrow exp] = exp(act\_node)[pos(act\_node) \leftarrow exp] \\ (ii) & pos(act\_node) = act\_pos \end{array}$$

But (ii) can only hold (in mode *Return*) if $act\_pos = \epsilon$ or if it is the first time the current task returns a value to its activator, and in both cases (i) is always verified (trivially for the first case, and because nothing has been done under the control of *act_node* since *currnode* was activated).

The **optimized return rule** simply needs to transform the updates of *exp(act_node)* and *pos(act_node)* into conditional updates, distinguishing the optimizable cases from the rest. This means to refine once more the abbreviation *return curr_env to act_env* to

if $pos(act\_node) = act\_pos$
 then $exp(act\_node) := exp(act\_node)[pos(act\_node) \leftarrow exp]$
 else $exp(act\_node) := act\_exp[act\_pos \leftarrow exp]$
 $pos(act\_node) := act\_pos$
$s(act\_node) := s$

This optimization covers the *first return optimization* of the IBAM, which corresponds to the second case discussed above.

6.2 Optimized last return

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 succesful computation. If by backtracking it is reactivated later on, it will fail immediately and backtracking will be done to its backtracking father. We 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 adding a conditional update for *lastnode*:

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

Note that the effect of this optimization and the previous one could be combined into a single **optimized return rule**.

6.3 Optimized call

This optimization complements the last return optimization. It consists in the following: if the current node N_0 is the last created one, the list of its alternatives is empty, and furthermore a call, creating a child node N_1, is going to be done as the *last* thing to do for N_0, then the value returned by N_1 will be returned also as the value for N_0. This transmission of the values returned by N_1 — maybe several times because of backtracking on N_1 — will be the only role of N_0 in the future (because N_0 has no alternatives and there is no task between N_0, N_1). This fact suggests that we can reuse the node corresponding to N_0 for N_1.

The **optimized call rule** only needs an additional test for deciding if the node can be reused.

if $is\_user\_defined(currexp) \;\&\; mode = Apply$
 $\&\; currexp = f(e_1, \ldots, e_n) \;\&\; 0 \leq n = arity(f)$
thenif $lastnode = currnode \;\&\; cands = [\,] \;\&\; pos = \epsilon$
 then $cands := fundef(currexp, pg)$ *% The node is reused*
 else ...

7 Conclusions and future work

Starting from related work for Prolog and the WAM, we have used Evolving Algebras to give a description of the innermost narrowing semantics for the functional logic language BABEL, and we have specified a series of provably correct refinement steps towards a definition of BABEL's implementation by the innermost graph-narrowing abstract machine IBAM [KLMR90].

Thus, we have paved the way for a full correctness proof for IBAM, which we believe will arise quite naturally by further refinement of the current description. This will be subject of future work. Moreover, we plan to develop a description of *lazy* semantics and implementation techniques for BABEL, which involves a more difficult kind of control [MKLR90, Loo93] and will require substantial modifications of the present description.

References

[AK91] H. Aït Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[AN89] H. Aït Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[Bö0a] E. Börger. A logical operational semantics for full Prolog. Part i: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 1990.

[Bö0b] E. Börger. A logical operational semantics for full Prolog. Part ii: Built-in predicates for database manipulations. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1990.

[BB92] C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic*, volume 626 of *Lecture Notes in Computer Science*, pages 15–34. Springer, 1992.

[BL86] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[BR91] E. Börger and D. Rosenzweig. A formal specification of Prolog by tree algebras. In V. Čerić, V. Dobrić, V. Lužar, and R. Paul, editors, *Information Technology Interfaces*, pages 513–518, Zagreb, 1991. University Computing Centre.

[BR92a] 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 *Sciene of Computer Programming*, 1994.

[BR92b] 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.

[BR93] E. Börger and E. Riccobene. A formal specification of Parlog. In M. Droste and Y. Gurevich, editors, *Semantics of Programming Languages and Model Theory.* Gordon and Breach, 1993, pages 1-42.

[BS91] E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 67–79. Springer, 1991.

[DJ90] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier Noth-Holland, 1990.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, Equations.* Prentice Hall, 1986.

[FH88] A.J. Field and P.G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[KLMR90] 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.

[Loo93] R. Loogen. Relating the implementation techniques of functional and functional logic languages. to appear in New Generation Computing, 1993.

[LW91] R. Loogen and S. Winkler. Dinamyc detection of determinism in functional logic languages. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 528 of *Lecture Notes in Computer Science*, pages 335–346. Springer, 1991.

[MKLR90] J.J Moreno Navarro, H. Kuchen, R. Loogen, and M. Rodríguez Artalejo. Lazy narrowing in a graph machine. In *2nd. International Conference on Algebraic and Logic Programming (ALP)*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1990.

[MR89] J.J Moreno Navarro and M. Rodríguez Artalejo. BABEL: A functional and logic language based on constructor discipline and narrowing. In *1st. International Conference on Algebraic and Logic Programming (ALP)*, volume 343 of *Lecture Notes in Computer Science*, pages 223–232. Springer, 1989.

[MR92] 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.

[PJ87] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Red85] U.S. Reddy. Narrowing as the operational semantics of functional logic languages. In *International Symposium on Logic Programming*, pages 138–151. IEEE Comp. Soc. Press, 1985.

[Red87] U.S. Reddy. Functional logic languages, part I. In *Workshop on Graph reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 401–425. Springer, 1987.

[War83] D.H.D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, 1983.

A Rules for the algebras

This appendix contains complete sets of rules for all the refinements of the algebra. A mark (♣) in a rule indicates that it has been changed with respect to the previous algebra.

A.1 BABEL tree algebras

Call rule

$$
\begin{array}{l}
\textbf{if } \mathit{is\_user\_defined}(\mathit{currexp}) \,\&\, \mathit{mode} = \mathit{Apply} \\
\quad \&\, \mathit{currexp} = f(e_1, \ldots, e_n) \,\&\, 0 \leq n = \mathit{arity}(f) \\
\textbf{then let } [\mathit{dr}_1, \ldots, \mathit{dr}_m] = \mathit{fundef}(\mathit{currexp}, \mathit{pg}) \\
\qquad \textbf{extend } \mathit{NODE} \textbf{ by } n_1, \ldots, n_m \textbf{ with} \\
\qquad\quad \mathit{bfather}(n_i) := \mathit{currnode} \\
\qquad\quad \mathit{cands} := [n_1, \ldots, n_m] \\
\qquad \textbf{endextend} \\
\qquad \mathit{defrule}(n_i) := \mathit{dr}_i \\
\qquad \mathit{mode} := \mathit{Select}
\end{array}
$$

Select rule

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \mathit{Select} \\
\textbf{thenif } \mathit{cands} = [\,] \textbf{ then } \mathit{backtrack} \\
\qquad \textbf{else let } (\mathit{Lhs} = \mathit{Rhs}) = \mathit{defrule}(\mathit{fst\_cand}) \\
\qquad\quad \textbf{let } \theta = \mathit{mgu}(\mathit{currexp}, \mathit{Lhs}') \\
\qquad\quad \mathit{cands} := \mathit{rest}(\mathit{cands}) \\
\qquad\quad \textbf{if } \theta \neq \mathit{nil} \textbf{ then } \mathit{go\_fst\_cand\ in\ Eval} \\
\qquad\qquad\qquad\qquad\quad \mathit{narrow\ curr\_env}(\mathit{Rhs}', \theta) \\
\qquad\qquad\qquad\qquad\quad \mathit{vi} := \mathit{vi} + 1
\end{array}
$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$
\begin{array}{lcl}
\mathit{go\_fst\_cand\ in\ Eval} & \equiv & \mathit{currnode} := \mathit{fst\_cand} \\
 & & \mathit{mode} := \mathit{Eval}
\end{array}
$$

$$
\begin{array}{lcl}
narrow\ curr\_env(E,\theta) & \equiv & exp(fst\_cand) := exp[\,pos \leftarrow E\,]\,\theta \\
 & & s(fst\_cand) := s\,\theta \\
 & & pos(fst\_cand) := pos
\end{array}
$$

$$
\begin{array}{lcl}
backtrack & \equiv & \textbf{if}\ bfather = root\ \textbf{then}\ stop := -1 \\
 & & \textbf{else}\ currnode := bfather \\
 & & \quad mode := Select
\end{array}
$$

Evaluation starting rule

$$
\begin{array}{l}
\textbf{if}\ mode = Eval\ \textbf{thenif}\ \ is\_not\_in\_normal\_form(currexp) \\
\quad \textbf{then}\ pos := pos.1 \\
\quad \textbf{else}\ mode := Continue
\end{array}
$$

Evaluation continuation rule

$$
\begin{array}{l}
\textbf{if}\ mode = Continue\ \&\ pos = u.i \\
\textbf{thenif}\ \ \neg occurs(u.(i+1), currexp)\ or\ nonstrict(u) \\
\quad \textbf{then}\ \ pos := u \\
\quad\quad\ mode := Apply \\
\quad \textbf{else}\ \ pos := u.(i+1) \\
\quad\quad\ mode := Eval
\end{array}
$$

Stop rule

$$
\begin{array}{l}
\textbf{if}\ mode = Continue\ \&\ pos = \epsilon \\
\textbf{then}\ \ solution\_list := [\langle exp, s\mid_{Goalvars}\rangle \mid solution\_list] \\
\quad \textbf{if}\ more = 1\ \ \textbf{then}\ backtrack \\
\quad\quad\ \ \textbf{else}\ stop = 1
\end{array}
$$

Construction rule

$$
\begin{array}{l}
\textbf{if}\ mode = Apply\ \&\ is\_construction(currexp) \\
\textbf{then}\ mode := Continue
\end{array}
$$

Partial application rule

$$
\begin{array}{l}
\textbf{if} mode = Apply\ \&\ currexp = @(e_0, e)\ \&\ e_0 = op(e_1, \ldots, e_m) \\
\textbf{then}\ exp := exp[pos \leftarrow op(e_1, \ldots, e_m e)]
\end{array}
$$

Rules for conditionals

$$
\begin{array}{l}
\textbf{if}\ \ currexp = b \rightarrow e \\
\quad \&\ mode = Apply \\
\textbf{thenif}\ \ b = true \\
\quad \textbf{then}\ \ exp := exp[pos \leftarrow e] \\
\quad\quad\ mode := Eval \\
\quad \textbf{else}\ backtrack
\end{array}
\qquad
\begin{array}{l}
\textbf{if}\ \ currexp = b \rightarrow e_1 \# e_2 \\
\quad \&\ mode = Apply \\
\textbf{then if}\ \ b = true \\
\quad\quad \textbf{then}\ exp := exp[pos \leftarrow e_1] \\
\quad\quad \textbf{else}\ exp := exp[pos \leftarrow e_2] \\
\quad\quad mode := Eval
\end{array}
$$

Equality rule

$$
\begin{array}{l}
\textbf{if}\ \ currexp = eq(e_1, e_2) \\
\quad \&\ mode = Apply \\
\textbf{then let}\ mgu = mgu(e_1, e_2) \\
\quad \textbf{if}\ mgu = nil\ \textbf{then}\ exp := exp[pos \leftarrow false] \\
\quad\quad\ \ \textbf{else}\ \ exp := (exp[pos \leftarrow true])mgu \\
\quad\quad\ \ s := s\,mgu
\end{array}
$$

A.2 Structure sharing for expressions

Call rule (♣)

$$
\begin{array}{l}
\textbf{if}\ \ is\_user\_defined(currexp)\ \&\ mode = Apply \\
\quad \&\ currexp = f(e_1, \ldots, e_n)\ \&\ 0 \leq n = arity(f) \\
\textbf{then let}\ [\,dr_1, \ldots, dr_m\,] = fundef(currexp\ s, pg) \\
\quad \textbf{extend}\ NODE\ \textbf{by}\ n_1, \ldots, n_m\ \textbf{with} \\
\quad\quad bfather(n_i) := currnode \\
\quad\quad cands := [\,n_1, \ldots, n_m\,] \\
\quad \textbf{endextend} \\
\quad defrule(n_i) := dr_i \\
\quad mode := Select
\end{array}
$$

Select rule (♣)

$$
\begin{array}{l}
\textbf{if } mode = Select \\
\textbf{thenif } cands = [\,] \textbf{ then } backtrack \\
\qquad \textbf{else let } (Lhs = Rhs) = defrule(fst\_cand) \\
\qquad\quad \textbf{let } \theta = mgu(currexp\ s, Lhs') \\
\qquad\quad cands := rest(cands) \\
\qquad\quad \textbf{if } \theta \neq nil \textbf{ then } \begin{array}{l} go\_fst\_cand\ in\ Eval \\ narrow\ curr\_env(Rhs', \theta) \\ vi := vi + 1 \end{array}
\end{array}
$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$
\begin{array}{lcl}
go\_fst\_cand\ in\ Eval & \equiv & \begin{array}{l} currnode := fst\_cand \\ mode := Eval \end{array} \\[2ex]
narrow\ curr\_env(E, \theta) & \equiv & \begin{array}{l} exp(fst\_cand) := exp[\,pos \leftarrow E] \\ s(fst\_cand) := s\,\theta \\ pos(fst\_cand) := pos \end{array} \\[2ex]
backtrack & \equiv & \textbf{if } bfather = root \textbf{ then } stop := -1 \\
 & & \qquad \textbf{else } currnode := bfather \\
 & & \qquad\quad mode := Select
\end{array}
$$

Evaluation starting rule

$$
\begin{array}{l}
\textbf{if } mode = Eval \textbf{ thenif } is\_not\_in\_normal\_form(currexp) \\
\qquad \textbf{then } pos := pos.1 \\
\qquad \textbf{else } mode := Continue
\end{array}
$$

Evaluation continuation rule

$$
\begin{array}{l}
\textbf{if } mode = Continue\ \&\ pos = u.i \\
\textbf{thenif } \neg occurs(u.(i+1), currexp)\ or\ nonstrict(u) \\
\qquad \textbf{then } \begin{array}{l} pos := u \\ mode := Apply \end{array} \\
\qquad \textbf{else } \begin{array}{l} pos := u.(i+1) \\ mode := Eval \end{array}
\end{array}
$$

Stop rule (♣)

$$
\begin{array}{l}
\textbf{if } mode = Continue\ \&\ pos = \epsilon \\
\textbf{then } solution\_list := [\langle exp\ s, s\,|_{Goalvars}\rangle \mid solution\_list] \\
\qquad \textbf{if } more = 1 \textbf{ then } backtrack \\
\qquad\quad \textbf{else } stop = 1
\end{array}
$$

Construction rule

$$
\begin{array}{l}
\textbf{if } mode = Apply\ \&\ is\_construction(currexp) \\
\textbf{then } mode := Continue
\end{array}
$$

Partial application rule (♣)

$$
\begin{array}{l}
\textbf{if } mode = Apply\ \&\ currexp = @(e_0, e)\ \&\ e_0\ s = op(e_1, \ldots, e_m) \\
\textbf{then } exp := exp[pos \leftarrow op(e_1, \ldots, e_m e)]
\end{array}
$$

Rules for conditionals (♣)

$$
\begin{array}{l}
\textbf{if } currexp = b \rightarrow e \\
\quad \&\ mode = Apply \\
\textbf{thenif } b\ s = true \\
\qquad \textbf{then } \begin{array}{l} exp := exp[pos \leftarrow e] \\ mode := Eval \end{array} \\
\qquad \textbf{else } backtrack
\end{array}
\qquad\qquad
\begin{array}{l}
\textbf{if } currexp = b \rightarrow e_1 \# e_2 \\
\quad \&\ mode = Apply \\
\textbf{then if } b\ s = true \\
\qquad \textbf{then } exp := exp[pos \leftarrow e_1] \\
\qquad \textbf{else } exp := exp[pos \leftarrow e_2] \\
\qquad mode := Eval
\end{array}
$$

Equality rule (♣)

$$
\begin{array}{l}
\textbf{if } \mathit{currexp} = \mathit{eq}(e_1, e_2) \\
\quad \& \ \mathit{mode} = \mathit{Apply} \\
\textbf{then let } \mathit{mgu} = \mathit{mgu}(e_1\ s, e_2\ s) \\
\qquad \textbf{if } \mathit{mgu} = \mathit{nil} \textbf{ then } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow \mathit{false}] \\
\qquad\qquad\qquad \textbf{else } \mathit{exp} := (\mathit{exp}[\mathit{pos} \leftarrow \mathit{true}]) \\
\qquad\qquad\qquad\quad s := s\ \mathit{mgu}
\end{array}
$$

A.3 Refining test for normal form

Call rule

$$
\begin{array}{l}
\textbf{if } \mathit{is\_user\_defined}(\mathit{currexp}) \ \& \ \mathit{mode} = \mathit{Apply} \\
\quad \& \ \mathit{currexp} = f(e_1, \ldots, e_n) \ \& \ 0 \leq n = \mathit{arity}(f) \\
\textbf{then let } [\, dr_1, \ldots, dr_m \,] = \mathit{fundef}(\mathit{currexp}\ s, pg) \\
\qquad \textbf{extend } \mathit{NODE} \textbf{ by } n_1, \ldots, n_m \textbf{ with} \\
\qquad\quad \mathit{bfather}(n_i) := \mathit{currnode} \\
\qquad\quad \mathit{cands} := [\, n_1, \ldots, n_m \,] \\
\qquad \textbf{endextend} \\
\qquad \mathit{defrule}(n_i) := dr_i \\
\qquad \mathit{mode} := \mathit{Select}
\end{array}
$$

Select rule

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \mathit{Select} \\
\textbf{thenif } \mathit{cands} = [\,] \textbf{ then } \mathit{backtrack} \\
\qquad\qquad \textbf{else let } (\mathit{Lhs} = \mathit{Rhs}) = \mathit{defrule}(\mathit{fst\_cand}) \\
\qquad\qquad\quad \textbf{let } \theta = \mathit{mgu}(\mathit{currexp}\ s, \mathit{Lhs}') \\
\qquad\qquad\quad \mathit{cands} := \mathit{rest}(\mathit{cands}) \\
\qquad\qquad\quad \textbf{if } \theta \neq \mathit{nil} \textbf{ then } \ \mathit{go\_fst\_cand\ in\ Eval} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{narrow\ curr\_env}(\mathit{Rhs}', \theta) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{vi} := \mathit{vi} + 1
\end{array}
$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$
\begin{array}{lcl}
\mathit{go\_fst\_cand\ in\ Eval} & \equiv & \mathit{currnode} := \mathit{fst\_cand} \\
 & & \mathit{mode} := \mathit{Eval}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{narrow\ curr\_env}(E, \theta) & \equiv & \mathit{exp}(\mathit{fst\_cand}) := \mathit{exp}[\,\mathit{pos} \leftarrow E] \\
 & & s(\mathit{fst\_cand}) := s\ \theta \\
 & & \mathit{pos}(\mathit{fst\_cand}) := \mathit{pos}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{backtrack} & \equiv & \textbf{if } \mathit{bfather} = \mathit{root} \textbf{ then } \mathit{stop} := -1 \\
 & & \qquad\qquad\quad \textbf{else } \mathit{currnode} := \mathit{bfather} \\
 & & \qquad\qquad\qquad\quad \mathit{mode} := \mathit{Select}
\end{array}
$$

Evaluation starting rule (♣)

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \mathit{Eval} \textbf{ thenif } \neg \mathit{atomic}(\mathit{currexp}) \textbf{ then } \mathit{pos} := \mathit{pos}.1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{mode} := \mathit{Continue}
\end{array}
$$

Evaluation continuation rule

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \mathit{Continue} \ \& \ \mathit{pos} = u.i \\
\textbf{thenif } \ \neg \mathit{occurs}(u.(i+1), \mathit{currexp}) \ \mathit{or}\ \mathit{nonstrict}(u) \\
\qquad \textbf{then } \ \mathit{pos} := u \\
\qquad\qquad\quad \mathit{mode} := \mathit{Apply} \\
\qquad \textbf{else } \ \mathit{pos} := u.(i+1) \\
\qquad\qquad\quad \mathit{mode} := \mathit{Eval}
\end{array}
$$

Stop rule

$$\begin{array}{l}\textbf{if } mode = Continue \,\&\, pos = \epsilon \\ \textbf{then } \; solution\_list := [\langle exp\ s, s\,|_{Goalvars}\rangle \mid solution\_list] \\ \qquad \textbf{if } more = 1 \;\; \textbf{then } backtrack \\ \qquad\qquad\qquad \textbf{else } stop = 1\end{array}$$

Construction rule

$$\begin{array}{l}\textbf{if } mode = Apply \,\&\, is\_construction(currexp) \\ \textbf{then } mode := Continue\end{array}$$

Partial application rule

$$\begin{array}{l}\textbf{if} mode = Apply \,\&\, currexp = @(e_0, e) \,\&\, e_0\ s = op(e_1, \ldots, e_m) \\ \textbf{then } exp := exp[pos \leftarrow op(e_1, \ldots, e_m e)]\end{array}$$

Rules for conditionals

$$\begin{array}{l}\textbf{if } \; currexp = b \rightarrow e \\ \quad \,\&\, mode = Apply \\ \textbf{thenif } \; b\ s = true \\ \qquad \textbf{then } \; exp := exp[pos \leftarrow e] \\ \qquad\qquad mode := Eval \\ \qquad \textbf{else } backtrack\end{array} \qquad \begin{array}{l}\textbf{if } \; currexp = b \rightarrow e_1 \# e_2 \\ \quad \,\&\, mode = Apply \\ \textbf{then if } \; b\ s = true \\ \qquad \textbf{then } exp := exp[pos \leftarrow e_1] \\ \qquad \textbf{else } exp := exp[pos \leftarrow e_2] \\ \qquad mode := Eval\end{array}$$

Equality rule

$$\begin{array}{l}\textbf{if } \; currexp = eq(e_1, e_2) \\ \quad \,\&\, mode = Apply \\ \textbf{then let } mgu = mgu(e_1\ s, e_2\ s) \\ \qquad \textbf{if } mgu = nil \;\; \textbf{then } exp := exp[pos \leftarrow false] \\ \qquad\qquad\qquad \textbf{else } \; exp := (exp[pos \leftarrow true]) \\ \qquad\qquad\qquad\quad s := s\ mgu\end{array}$$

A.4 BABEL stack algebras

Call rule (♣)

$$\begin{array}{l}\textbf{if } \; is\_user\_defined(currexp) \,\&\, mode = Apply \\ \quad \,\&\, currexp = f(e_1, \ldots, e_n) \,\&\, 0 \leq n = arity(f) \\ \textbf{then extend } NODE \textbf{ by } N \textbf{ with} \\ \qquad bfather(N) := currnode \\ \qquad currnode := N \\ \qquad cands(N) := fundef(currexp, pg) \\ \qquad \textbf{endextend} \\ \qquad mode := Select\end{array}$$

Select rule (♣)

$$\begin{array}{l}\textbf{if } mode = Select \textbf{ thenif } cands = [\,] \\ \textbf{then } backtrack \\ \textbf{else let } (Lhs = Rhs) = fst\_cands \\ \qquad \textbf{let } \theta = mgu(exp(bfather)[pos(bfather)]\ s(bfather), Lhs') \\ \qquad cands := rest(cands) \\ \qquad \textbf{if } \theta \neq nil \textbf{ then } \; mode := Eval \\ \qquad\qquad\qquad\qquad narrow\ bfather\_env(Rhs', \theta) \\ \qquad\qquad\qquad\qquad vi := vi + 1\end{array}$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$\begin{array}{lcl} narrow\ bfather\_env(E, \theta) & \equiv & exp := exp(bfather)[pos(bfather) \leftarrow E] \\ & & pos := pos(bfather) \\ & & s := s(bfather)\ \theta \end{array}$$

$$\begin{array}{lll} \mathit{backtrack} & \equiv & \textbf{if } \mathit{bfather} = \mathit{root} \textbf{ then } \mathit{stop} := -1 \\ & & \quad \textbf{else } \mathit{currnode} := \mathit{bfather} \\ & & \quad\quad \mathit{mode} := \mathit{Select} \end{array}$$

Since this point *cands* is assumed to be initialized to [] for the child of *root*.

Evaluation starting rule

$$\begin{array}{l} \textbf{if } \mathit{mode} = \mathit{Eval} \textbf{ thenif } \neg \mathit{atomic}(\mathit{currexp}) \textbf{ then } \mathit{pos} := \mathit{pos}.1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{mode} := \mathit{Continue} \end{array}$$

Evaluation continuation rule

$$\begin{array}{l} \textbf{if } \mathit{mode} = \mathit{Continue} \,\&\, \mathit{pos} = u.i \\ \textbf{thenif } \neg \mathit{occurs}(u.(i+1), \mathit{currexp}) \textit{ or } \mathit{nonstrict}(u) \\ \qquad \textbf{then } \mathit{pos} := u \\ \qquad\qquad\quad \mathit{mode} := \mathit{Apply} \\ \qquad \textbf{else } \mathit{pos} := u.(i+1) \\ \qquad\qquad\quad \mathit{mode} := \mathit{Eval} \end{array}$$

Stop rule

$$\begin{array}{l} \textbf{if } \mathit{mode} = \mathit{Continue} \,\&\, \mathit{pos} = \epsilon \\ \textbf{then } \mathit{solution\_list} := [\langle \mathit{exp}\ s, s\,|_{\mathit{Goalvars}}\rangle \mid \mathit{solution\_list}] \\ \qquad \textbf{if } \mathit{more} = 1 \textbf{ then } \mathit{backtrack} \\ \qquad\qquad\qquad\quad \textbf{else } \mathit{stop} = 1 \end{array}$$

Construction rule

$$\begin{array}{l} \textbf{if } \mathit{mode} = \mathit{Apply} \,\&\, \mathit{is\_construction}(\mathit{currexp}) \\ \textbf{then } \mathit{mode} := \mathit{Continue} \end{array}$$

Partial application rule

$$\begin{array}{l} \textbf{if } \mathit{mode} = \mathit{Apply} \,\&\, \mathit{currexp} = @(e_0, e) \,\&\, e_0\ s = \mathit{op}(e_1, \ldots, e_m) \\ \textbf{then } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow \mathit{op}(e_1, \ldots, e_m e)] \end{array}$$

Rules for conditionals

$$\begin{array}{l} \textbf{if } \mathit{currexp} = b \rightarrow e \\ \quad \,\&\, \mathit{mode} = \mathit{Apply} \\ \textbf{thenif } b\ s = \mathit{true} \\ \qquad \textbf{then } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e] \\ \qquad\qquad\ \mathit{mode} := \mathit{Eval} \\ \qquad \textbf{else } \mathit{backtrack} \end{array} \qquad \begin{array}{l} \textbf{if } \mathit{currexp} = b \rightarrow e_1 \# e_2 \\ \quad \,\&\, \mathit{mode} = \mathit{Apply} \\ \textbf{then if } b\ s = \mathit{true} \\ \qquad \textbf{then } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e_1] \\ \qquad \textbf{else } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e_2] \\ \qquad\ \mathit{mode} := \mathit{Eval} \end{array}$$

Equality rule

$$\begin{array}{l} \textbf{if } \mathit{currexp} = \mathit{eq}(e_1, e_2) \\ \quad \,\&\, \mathit{mode} = \mathit{Apply} \\ \textbf{then let } \mathit{mgu} = \mathit{mgu}(e_1\ s, e_2\ s) \\ \qquad \textbf{if } \mathit{mgu} = \mathit{nil} \textbf{ then } \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow \mathit{false}] \\ \qquad\qquad\qquad\quad \textbf{else } \mathit{exp} := (\mathit{exp}[\mathit{pos} \leftarrow \mathit{true}]) \\ \qquad\qquad\qquad\qquad\ s := s\ \mathit{mgu} \end{array}$$

A.5 BABEL graph algebras

Call rule (♣)

$$\begin{array}{l} \textbf{if } \mathit{is\_user\_defined}(\mathit{currexp}) \,\&\, \mathit{mode} = \mathit{Apply} \\ \quad \,\&\, \mathit{currexp} = f(e_1, \ldots, e_n) \,\&\, 0 \leq n = \mathit{arity}(f) \\ \textbf{then extend } \mathit{NODE} \textbf{ by } N \textbf{ with} \\ \qquad \mathit{currnode} := N \\ \qquad \mathit{cands}(N) := \mathit{fundef}(\mathit{currexp}, \mathit{pg}) \\ \qquad \mathit{bfather}(N) := \mathit{lastnode} \\ \qquad \mathit{act\_node}(N) := \mathit{currnode} \\ \qquad \mathit{store\ act\_env\ at\ } N \\ \quad \textbf{endextend} \\ \quad \mathit{mode} := \mathit{Select} \end{array}$$

with the abbreviation

$$\begin{array}{lll} \textit{store act\_env at } N & \equiv & \textit{act\_exp}(N) := \textit{exp} \\ & & \textit{act\_pos}(N) := \textit{pos} \\ & & \textit{act\_s}(N) := s \end{array}$$

Select rule (♣)

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Select } \textbf{thenif } \textit{cands} = [\,] \\ \textbf{then } \textit{backtrack} \\ \textbf{else let } (\textit{Lhs} = \textit{Rhs}) = \textit{fst\_cands} \\ \quad \textbf{let } \theta = \textit{mgu}(\textit{act\_exp}[\textit{act\_pos}]\ \textit{act\_s}, \textit{Lhs}') \\ \quad \textit{cands} := \textit{rest}(\textit{cands}) \\ \quad \textbf{if } \theta \neq \textit{nil } \textbf{then } \textit{mode} := \textit{Eval} \\ \qquad\qquad\qquad\qquad \textit{lastnode} := \textit{currnode} \\ \qquad\qquad\qquad\qquad \textit{narrow act\_env}(\textit{Rhs}', \theta) \\ \qquad\qquad\qquad\qquad \textit{vi} := \textit{vi} + 1 \end{array}$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$\begin{array}{lll} \textit{narrow act\_env}(E, \theta) & \equiv & \textit{exp} := \textit{act\_exp}[\textit{act\_pos} \leftarrow E] \\ & & \textit{pos} := \textit{act\_pos} \\ & & s := \textit{act\_s}\,\theta \end{array}$$

$$\begin{array}{lll} \textit{backtrack} & \equiv & \textbf{if } \textit{bfather} = \textit{root } \textbf{then } \textit{stop} := -1 \\ & & \qquad\qquad\qquad\quad \textbf{else } \textit{currnode} := \textit{bfather} \\ & & \qquad\qquad\qquad\qquad\ \ \textit{mode} := \textit{Select} \end{array}$$

Evaluation starting rule

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Eval } \textbf{thenif } \neg \textit{atomic}(\textit{currexp}) \textbf{ then } \textit{pos} := \textit{pos}.1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \textit{mode} := \textit{Continue} \end{array}$$

Evaluation continuation rule (♣)

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Continue} \\ \textbf{thenif } \textit{pos} = \textit{act\_pos } \textbf{then } \textit{mode} := \textit{Return} \\ \textbf{else let } u.i = \textit{pos} \\ \quad \textbf{if } \neg \textit{occurs}(u.(i+1), \textit{currexp})\ \textit{or}\ \textit{nonstrict}(u) \\ \quad \textbf{then } \textit{pos} := u \\ \qquad\quad\ \ \textit{mode} := \textit{Apply} \\ \quad \textbf{else } \textit{pos} := u.(i+1) \\ \qquad\quad\ \ \textit{mode} := \textit{Eval} \end{array}$$

We assume here that *act_pos* is initialized to be ϵ for the child of *root*.

Stop rule (♣)

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Return } \& \textit{ bfather} = \textit{root} \\ \textbf{then } \textit{solution\_list} := [\langle \textit{exp } s, s\,|_{\textit{Goalvars}}\rangle \mid \textit{solution\_list}] \\ \quad \textbf{if } \textit{more} = 1 \ \textbf{ then } \textit{backtrack} \\ \qquad\qquad\quad \textbf{else } \textit{stop} = 1 \end{array}$$

Construction rule

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Apply } \& \textit{ is\_construction}(\textit{currexp}) \\ \textbf{then } \textit{mode} := \textit{Continue} \end{array}$$

Return rule (♣)

$$\begin{array}{l} \textbf{if } \textit{mode} = \textit{Return } \& \textit{ bfather} \neq \textit{root } \textbf{then } \textit{currnode} := \textit{act\_node} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{return curr\_env to act\_env} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{mode} := \textit{Continue} \end{array}$$

with the abbreviation

$$\begin{array}{lcl} \mathit{return\ curr\_env\ to\ act\_env} & \equiv & \mathit{exp}(\mathit{act\_node}) := \mathit{exp} \\ & & \mathit{pos}(\mathit{act\_node}) := \mathit{pos} \\ & & \mathit{s}(\mathit{act\_node}) := \mathit{s} \end{array}$$

Partial application rule

$$\begin{array}{l} \textbf{if}\ \mathit{mode} = \mathit{Apply}\ \&\ \mathit{currexp} = @(e_0, e)\ \&\ e_0\ s = \mathit{op}(e_1, \ldots, e_m) \\ \textbf{then}\ \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow \mathit{op}(e_1, \ldots, e_m e)] \end{array}$$

Rules for conditionals

$$\begin{array}{l} \textbf{if}\ \ \mathit{currexp} = b \rightarrow e \\ \quad \&\ \mathit{mode} = \mathit{Apply} \\ \textbf{thenif}\ \ b\ s = \mathit{true} \\ \qquad \textbf{then}\ \ \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e] \\ \qquad\quad \mathit{mode} := \mathit{Eval} \\ \qquad \textbf{else}\ \mathit{backtrack} \end{array} \qquad \begin{array}{l} \textbf{if}\ \ \mathit{currexp} = b \rightarrow e_1 \# e_2 \\ \quad \&\ \mathit{mode} = \mathit{Apply} \\ \textbf{then if}\ \ b\ s = \mathit{true} \\ \qquad \textbf{then}\ \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e_1] \\ \qquad \textbf{else}\ \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow e_2] \\ \qquad\quad \mathit{mode} := \mathit{Eval} \end{array}$$

Equality rule

$$\begin{array}{l} \textbf{if}\ \ \mathit{currexp} = \mathit{eq}(e_1, e_2) \\ \quad \&\ \mathit{mode} = \mathit{Apply} \\ \textbf{then}\ \textbf{let}\ \mathit{mgu} = \mathit{mgu}(e_1\ s, e_2\ s) \\ \qquad \textbf{if}\ \mathit{mgu} = \mathit{nil}\ \textbf{then}\ \mathit{exp} := \mathit{exp}[\mathit{pos} \leftarrow \mathit{false}] \\ \qquad\qquad\qquad \textbf{else}\ \ \mathit{exp} := (\mathit{exp}[\mathit{pos} \leftarrow \mathit{true}]) \\ \qquad\qquad\qquad\quad s := s\ \mathit{mgu} \end{array}$$

A.6 Subcomputations with local expressions

Call rule

$$\begin{array}{l} \textbf{if}\ \ \mathit{is\_user\_defined}(\mathit{currexp})\ \&\ \mathit{mode} = \mathit{Apply} \\ \quad \&\ \mathit{currexp} = f(e_1, \ldots, e_n)\ \&\ 0 \leq n = \mathit{arity}(f) \\ \textbf{then}\ \ \textbf{extend}\ \mathit{NODE}\ \textbf{by}\ N\ \textbf{with} \\ \qquad \mathit{currnode} := N \\ \qquad \mathit{cands}(N) := \mathit{fundef}(\mathit{currexp}, \mathit{pg}) \\ \qquad \mathit{bfather}(N) := \mathit{lastnode} \\ \qquad \mathit{act\_node}(N) := \mathit{currnode} \\ \qquad \mathit{store\ act\_env\ at}\ N \\ \qquad \textbf{endextend} \\ \qquad \mathit{mode} := \mathit{Select} \end{array}$$

with the abbreviation

$$\begin{array}{lcl} \mathit{store\ act\_env\ at}\ N & \equiv & \mathit{act\_exp}(N) := \mathit{exp} \\ & & \mathit{act\_pos}(N) := \mathit{pos} \\ & & \mathit{act\_s}(N) := \mathit{s} \end{array}$$

Select rule (♣)

$$\begin{array}{l} \textbf{if}\ \mathit{mode} = \mathit{Select}\ \textbf{thenif}\ \mathit{cands} = [\,] \\ \textbf{then}\ \mathit{backtrack} \\ \textbf{else}\ \ \textbf{let}\ (\mathit{Lhs} = \mathit{Rhs}) = \mathit{fst\_cands} \\ \qquad \textbf{let}\ \theta = \mathit{mgu}(\mathit{act\_exp}[\mathit{act\_pos}]\ \mathit{act\_s}, \mathit{Lhs}\,') \\ \qquad \mathit{cands} := \mathit{rest}(\mathit{cands}) \\ \qquad \textbf{if}\ \theta \neq \mathit{nil}\ \textbf{then}\ \ \mathit{mode} := \mathit{Eval} \\ \qquad\qquad\qquad\qquad\quad \mathit{lastnode} := \mathit{currnode} \\ \qquad\qquad\qquad\qquad\quad \mathit{narrow\ act\_env}(\mathit{Rhs}', \theta) \\ \qquad\qquad\qquad\qquad\quad \mathit{vi} := \mathit{vi} + 1 \end{array}$$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$\begin{array}{lll} narrow\ act\_env(E,\theta) & \equiv & exp := E \\ & & pos := \epsilon \\ & & s := act\_s\,\theta \end{array}$$

$$\begin{array}{lll} backtrack & \equiv & \textbf{if}\ bfather = root\ \textbf{then}\ stop := -1 \\ & & \textbf{else}\ currnode := bfather \\ & & \quad mode := Select \end{array}$$

Evaluation starting rule

$$\begin{array}{l} \textbf{if}\ mode = Eval\ \textbf{thenif}\ \neg atomic(currexp)\ \textbf{then}\ pos := pos.1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ mode := Continue \end{array}$$

Evaluation continuation rule (♣)

$$\begin{array}{l} \textbf{if}\ mode = Continue \\ \textbf{thenif}\ pos = \epsilon\ \textbf{then}\ mode := Return \\ \textbf{else let}\ u.i = pos \\ \quad \textbf{if}\ \neg occurs(u.(i+1), currexp)\ or\ nonstrict(u) \\ \quad \textbf{then}\ \ pos := u \\ \qquad\qquad mode := Apply \\ \quad \textbf{else}\ \ pos := u.(i+1) \\ \qquad\qquad mode := Eval \end{array}$$

Stop rule

$$\begin{array}{l} \textbf{if}\ mode = Return\ \&\ bfather = root \\ \textbf{then}\ \ solution\_list := [\langle exp\ s, s\,|_{Goalvars}\rangle \mid solution\_list] \\ \quad \textbf{if}\ more = 1\ \ \textbf{then}\ backtrack \\ \qquad\qquad\qquad\ \ \textbf{else}\ stop = 1 \end{array}$$

Construction rule

$$\begin{array}{l} \textbf{if}\ mode = Apply\ \&\ is\_construction(currexp) \\ \textbf{then}\ mode := Continue \end{array}$$

Return rule (♣)

$$\begin{array}{l} \textbf{if}\ mode = Return\ \&\ bfather \neq root\ \textbf{then}\ \ currnode := act\_node \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ curr\_env\ to\ act\_env \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mode := Continue \end{array}$$

with the abbreviation

$$\begin{array}{lll} return\ curr\_env\ to\ act\_env & \equiv & exp(act\_node) := act\_exp[act\_pos \leftarrow exp] \\ & & pos(act\_node) := act\_pos \\ & & s(act\_node) := s \end{array}$$

Partial application rule

$$\begin{array}{l} \textbf{if} mode = Apply\ \&\ currexp = @(e_0, e)\ \&\ e_0\ s = op(e_1, \ldots, e_m) \\ \textbf{then}\ exp := exp[pos \leftarrow op(e_1, \ldots, e_m e)] \end{array}$$

Rules for conditionals

$$\begin{array}{l} \textbf{if}\ \ currexp = b \rightarrow e \\ \quad \&\ mode = Apply \\ \textbf{thenif}\ \ b\ s = true \\ \qquad \textbf{then}\ \ exp := exp[pos \leftarrow e] \\ \qquad\qquad mode := Eval \\ \qquad \textbf{else}\ backtrack \end{array} \qquad \begin{array}{l} \textbf{if}\ \ currexp = b \rightarrow e_1 \# e_2 \\ \quad \&\ mode = Apply \\ \textbf{then if}\ \ b\ s = true \\ \qquad \textbf{then}\ exp := exp[pos \leftarrow e_1] \\ \qquad \textbf{else}\ exp := exp[pos \leftarrow e_2] \\ \quad mode := Eval \end{array}$$

Equality rule

$$\begin{array}{l} \textbf{if}\ \ currexp = eq(e_1, e_2) \\ \quad \&\ mode = Apply \\ \textbf{then let}\ mgu = mgu(e_1\ s, e_2\ s) \\ \quad \textbf{if}\ mgu = nil\ \textbf{then}\ exp := exp[pos \leftarrow false] \\ \qquad\qquad\qquad\ \ \textbf{else}\ \ exp := (exp[pos \leftarrow true]) \\ \qquad\qquad\qquad\qquad\ \ s := s\ mgu \end{array}$$

A.7 Optimizations

Optimized Call rule (♣)

if $is\_user\_defined(currexp)$ & $mode = Apply$
& $currexp = f(e_1, \ldots, e_n)$ & $0 \leq n = arity(f)$
thenif $lastnode = currnode$ & $cands = [\,]$ & $pos = \epsilon$
then $cands(t) := fundef(currexp, pg)$ % *The node is reused*
else extend $NODE$ **by** N **with**
$currnode := N$
$cands(N) := fundef(currexp, pg)$
$bfather(N) := lastnode$
$act\_node(N) := currnode$
store act_env at N
endextend
$mode := Select$

with the abbreviation

$$\begin{array}{lll} \textit{store act\_env at } N & \equiv & act\_exp(N) := exp \\ & & act\_pos(N) := pos \\ & & act\_s(N) := s \end{array}$$

Select rule

if $mode = Select$ **thenif** $cands = [\,]$
then *backtrack*
else let $(Lhs = Rhs) = fst\_cands$
let $\theta = mgu(act\_exp[act\_pos]\ act\_s, Lhs')$
$cands := rest(cands)$
if $\theta \neq nil$ **then** $mode := Eval$
$lastnode := currnode$
narrow act_env(Rhs', θ)
$vi := vi + 1$

with *fst_cand* denoting the first element in *cands* and with abbreviations:

$$\begin{array}{lll} narrow\ act\_env(E, \theta) & \equiv & exp := E \\ & & pos := \epsilon \\ & & s := act\_s\,\theta \end{array}$$

$$\begin{array}{lll} backtrack & \equiv & \textbf{if } bfather = root \textbf{ then } stop := -1 \\ & & \textbf{else } currnode := bfather \\ & & \quad mode := Select \end{array}$$

Evaluation starting rule

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

Evaluation continuation rule

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

Stop rule

$$\begin{array}{l}\textbf{if } mode = Return \;\&\; bfather = root\\ \textbf{then } \; solution\_list := [\langle exp\ s, s\ |_{Goalvars}\rangle \mid solution\_list]\\ \qquad \textbf{if } more = 1 \;\; \textbf{then } backtrack\\ \qquad\qquad\qquad \textbf{else } stop = 1\end{array}$$

Construction rule

$$\begin{array}{l}\textbf{if } mode = Apply \;\&\; is\_construction(currexp)\\ \textbf{then } mode := Continue\end{array}$$

Optimized Return rule (♣)

$$\begin{array}{l}\textbf{if } mode = Return \;\&\; bfather \neq root\\ \textbf{then } \; currnode := act\_node\\ \qquad return\ curr\_env\ to\ act\_env\\ \qquad mode := Continue\\ \qquad \textbf{if } lastnode = currnode \;\&\; cands = [\,] \textbf{then } lastnode := bfather\end{array}$$

with the abbreviation

$$\begin{array}{lll} return\ curr\_env\ to\ act\_env & \equiv & \textbf{if } pos(act\_node) = act\_pos\\ & & \quad \textbf{then } \; exp(act\_node) := exp(act\_node)[pos(act\_node) \leftarrow exp]\\ & & \quad \textbf{else } \; exp(act\_node) := act\_exp[act\_pos \leftarrow exp]\\ & & \qquad\quad pos(act\_node) := act\_pos\\ & & \quad s(act\_node) := s\end{array}$$

Partial application rule

$$\begin{array}{l}\textbf{if } mode = Apply \;\&\; currexp = @(e_0, e) \;\&\; e_0\ s = op(e_1, \ldots, e_m)\\ \textbf{then } exp := exp[pos \leftarrow op(e_1, \ldots, e_m e)]\end{array}$$

Rules for conditionals

$$\begin{array}{l}\textbf{if } \; currexp = b \rightarrow e\\ \quad \;\&\; mode = Apply\\ \textbf{thenif } \; b\ s = true\\ \qquad \textbf{then } \; exp := exp[pos \leftarrow e]\\ \qquad\quad mode := Eval\\ \qquad \textbf{else } backtrack\end{array} \qquad \begin{array}{l}\textbf{if } \; currexp = b \rightarrow e_1 \# e_2\\ \quad \;\&\; mode = Apply\\ \textbf{then if } \; b\ s = true\\ \qquad \textbf{then } exp := exp[pos \leftarrow e_1]\\ \qquad \textbf{else } exp := exp[pos \leftarrow e_2]\\ \quad mode := Eval\end{array}$$

Equality rule

$$\begin{array}{l}\textbf{if } \; currexp = eq(e_1, e_2)\\ \quad \;\&\; mode = Apply\\ \textbf{then let } mgu = mgu(e_1\ s, e_2\ s)\\ \qquad \textbf{if } mgu = nil \;\; \textbf{then } exp := exp[pos \leftarrow false]\\ \qquad\qquad\qquad \textbf{else } \; exp := (exp[pos \leftarrow true])\\ \qquad\qquad\qquad\quad s := s\ mgu\end{array}$$