

Multiple Evaluation Strategies for the Multiparadigm Programming Language Brooks

Petra Hofstedt and André Metzner

Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin, Germany
{ame,ph}@cs.tu-berlin.de

Abstract. This paper presents the functional logic programming language Brooks which inherits from the languages Curry and BABEL, but allows the integration of different narrowing strategies. We describe briefly the abstract machine BEBAM as target for the compilation of Brooks programs together with key points of the translation process. Furthermore differences to other multiparadigm languages are discussed.

1 Introduction

While many programming languages implement only a single paradigm like functional, logic, or object-oriented programming, in recent years the interest in languages with a broader scope, integrating several paradigms simultaneously, has grown. An important subclass of those multiparadigm languages is constituted by functional logic languages which combine the two main declarative paradigms functional and logic programming. They bring together deterministic computations from the functional world and nondeterministic search operations from the logic world.

Extensive research in this area yielded several languages utilizing different evaluation mechanisms, e.g. Escher [Llo99] uses rewriting, BABEL [MNRA92] uses lazy narrowing, and Curry [HAK⁺02] uses residuation and needed narrowing. However, combinations of different narrowing strategies in a single language appear to be rare. That is understandable, because from the viewpoint of a stand-alone language the gain in the combination of, say, lazy and needed narrowing is not obvious; the latter is usually regarded as superior to the former due to certain optimality properties [AEH00].

But declarative programs together with language evaluation mechanisms can also be considered as constraint solvers [Hof02], which allows for a smooth integration of such host languages into a bigger framework of cooperating solvers. In this context programs are not evaluated as a whole, but in a stepwise fashion under the strategic control of a meta-solver [FHM03]. It is not a priori clear how (meta-)solving strategies and evaluation mechanisms interact. We are currently investigating this topic and expect advantages by avoiding the commitment to a single evaluation mechanism.

Along these lines the need for an experimental host language platform arises. Despite certain advantages an interpreter approach has to offer, performance

requirements of the overall system led us to implement an abstract machine with integrated support for innermost, lazy, and needed narrowing. Our work is conceptually based on the machine described by [Loo95] which already supports innermost and lazy narrowing. Therefore one aim of our work was to examine the requirements of needed narrowing and to figure out necessary extensions. As high-level companion for the resulting low-level machine BEBAM we developed the language Brooks that can be seen as a Haskell descendant with logic features or – at least for the features presented in this paper – as a subset of Curry.¹

The paper is structured as follows: Section 2 gives a brief overview of Brooks and the usage of the language, and it shortly sketches the different evaluation mechanisms. The structure of the narrowing machine BEBAM is treated in Sect. 3. Section 4 is dedicated to the compilation of Brooks programs into BEBAM code. We consider functions to be evaluated using innermost and lazy narrowing in Sect. 4.1 and needed narrowing in Sect. 4.2, resp. Finally we discuss related work and conclude our paper in Sect. 5.

2 Elements of Brooks

This section gives a brief overview of Brooks as it is currently implemented by a prototypical compiler with the abstract machine BEBAM as target. In Sect. 2.1 we consider the elements of a Brooks program and sketch on different evaluation strategies in Sect. 2.2.

2.1 Programs, Data Types, and Function Definitions

A *Brooks program* consists of data type definitions and function definitions with a Haskell-like syntax.

A *data type definition* introduces a new type and a set of data constructors. Types and constructors start with capital letters. For example,

```
data Tree = Leaf Nat | Node Tree Tree
```

defines a data type `Tree` with two variants: a `Tree` is either a `Leaf` with a value of type `Nat` or a `Node` with two children of type `Tree` itself. As usual a constructor is an uninterpreted function.

Brooks has some built-in data types like `Nat` representing natural numbers and `Bool` for boolean values. Inspired by Curry there is furthermore the data type `Constraint` with the single constructor `Success` which comes in handy whenever only positive answers are of interest. This happens especially in the context of conditional rules where guard expressions are checked for satisfiability.

At present, Brooks is a monomorphic language, i.e. neither in data type definitions nor function type declarations universally quantified type variables are allowed. Due to time restrictions we left this aspect out initially, because it

¹ Brooks is the ‘B.’ in the name of the mathematician Haskell B. Curry and thus stands somewhere between Haskell and Curry.

was not directly relevant to our work and can be added later on in the usual way. Similar reasoning led to the omission of local definitions and lambda abstractions.

A *function* in Brooks consists of a function type declaration and a function definition. There are two different forms of function definitions related to the different evaluation strategies. We discuss the traditional rule-based form below and the other tree-based form in Sect. 2.2. A rule-based function is defined by a sequence of conditional rules, i.e. rules with left-linear constructor patterns and optional Bool- or Constraint-typed guard expressions. Rules with identical pattern parts but different guards can be bundled together to a rule block which affects code translation and gains efficiency (cf. Sect. 4.1), e.g.

$$\begin{array}{l}
 f \text{ pat}_1 \dots \text{pat}_n \mid \text{guard}_1 = \text{body}_1 \\
 f \text{ pat}_1 \dots \text{pat}_n \mid \text{guard}_2 = \text{body}_2 \\
 f \text{ pat}'_1 \dots \text{pat}'_n \mid \text{guard}'_1 = \text{body}'_1 \\
 f \text{ pat}'_1 \dots \text{pat}'_n \mid \text{guard}'_2 = \text{body}'_2
 \end{array}
 \implies
 \begin{array}{l}
 f \text{ pat}_1 \dots \text{pat}_n \mid \text{guard}_1 = \text{body}_1 \\
 \phantom{f \text{ pat}_1 \dots \text{pat}_n} \mid \text{guard}_2 = \text{body}_2 \\
 f \text{ pat}'_1 \dots \text{pat}'_n \mid \text{guard}'_1 = \text{body}'_1 \\
 \phantom{f \text{ pat}'_1 \dots \text{pat}'_n} \mid \text{guard}'_2 = \text{body}'_2 .
 \end{array}$$

For reasons of confluence we require according to [Loo95] that if the left hand sides of variable disjoint variants of two rules for the same function are unifiable with a most general unifier σ then either the right-hand sides must be syntactically identical under σ or the guards must be incompatible, i.e. not simultaneously satisfiable. However, since in general we allow free variables in right-hand sides of rules as well as in guards, keeping confluence is, after all, left in the user's responsibility (cf. [Han95]). Types of free variables must be given explicitly using the **where** construct. Further features of Brooks are the *off-side rule*, *infix operators*, and *currying* along with *higher order functions*. The example program in Fig. 1 can't show every detail but may give a general impression.

```

data Seq = Nil | Nat : Seq          -- Sequences of natural numbers

(==:=) :: Seq -> Seq -> Constraint -- Equality constraint for sequences
Nil    ::= Nil                    = Success
(x : xs) ::= (y : ys) | x ::= y = xs ::= ys

(++ ) :: Seq -> Seq -> Seq         -- Concatenation of sequences
Nil    ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

last :: Seq -> Nat                -- Get "logically" the last element
last xs | (ys ++ (x : Nil)) ::= xs = x where ys :: Seq
                                           x :: Nat

```

Fig. 1. Brooks code to (inefficiently) extract the last element from a sequence using a search-oriented technique. The idea is that x is the last element of xs if there exists a sequence ys so that x appended to ys is equal to xs . Note that the equality constraint ($==:=$) for **Nat** is predefined.

2.2 Functions and Evaluation Mechanisms

Our main interest was in the implementation of an abstract machine which supports evaluation by innermost, lazy, and needed narrowing (see [Han94] for an extensive survey of narrowing strategies). *Innermost narrowing* is the simplest of the three strategies and often more efficient than lazy narrowing but is incomplete even if the corresponding term rewriting system is confluent and terminating. It does not allow for infinite data structures and impedes certain elegant programming techniques. Therefore modern implementations of functional-logic programming languages favor *lazy narrowing* or *needed narrowing*.

Needed narrowing is optimal wrt. the length of derivations and the number of computed solutions but in its original form can only be applied to a restricted class of programs, called inductively sequential systems [AEH00]. However, there are extensions of needed narrowing like weakly needed narrowing or parallel needed narrowing which go beyond this limitation [AEH97] and are deployed by Curry. For time reasons they are currently missing from Brooks, but the integration should pose no fundamental problems.

The choice between different evaluation strategies is in principle made via a compile time switch except that for the time being needed narrowing gets a special treatment. It deviates from the other strategies in that the pattern unification process doesn't work in a rule-oriented way but is guided by *definitional trees*. A definitional tree is a particular hierarchical structure to describe the order in which arguments are to be inspected. Although such trees can be generated algorithmically from rule-based function definitions we decided that for the first version of Brooks a manual specification suffices.

Definitional trees are formally introduced in [Ant92] and have an intuitive structure, so instead of restating definitions we content ourselves with Example 1 which moreover demonstrates the Brooks syntax for definitional trees.

Example 1. The following rule-based definition specifies the boolean function “less than or equal” for natural numbers (of the non-built-in type `Nat'`) built upon the constructors `Z` (zero) and `S` (successor):

```
data Nat' = Z | S Nat'

leq :: Nat' -> Nat' -> Bool
leq Z    y    = True      -- R1
leq (S x) Z   = False     -- R2
leq (S x) (S y) = leq x y -- R3
```

An according definitional tree is given in Fig. 2 in graphical form and as Brooks code using the `case` construct. The needed narrowing evaluation of a call `leq t1 t2` demands at first the inspection of `t1` because the patterns of all rules have constructor terms at position 1 but not at position 2. Evaluation of `t2` is not needed if `t1` is evaluated to `Z` because `y` in rule `R1` matches any argument. Otherwise the decision between the rules `R2` and `R3` is based on the head constructor of the second argument, i.e. `t2` must be evaluated.

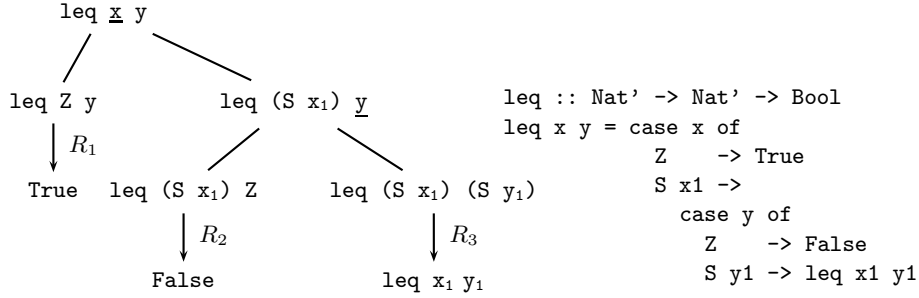


Fig. 2. Definitional tree for the function `leq` in graphical form and in Brooks syntax.

3 Structure of the Narrowing Machine

The target for the compilation of Brooks code is an abstract machine which can be regarded as a suitably modified version of the machine described in [Loo95] as execution environment of the language BABEL. Because both languages utilize narrowing techniques as evaluation mechanisms the choice of this machine as a starting point appeared sensible. However, BABEL is restricted to innermost and lazy narrowing and not concerned with needed narrowing, so one task was to identify necessary extensions. The resulting **Brooks Extended BABEL Abstract Machine** is named BEBAM in the following while BAM designates the original machine.

The (BE)BAM descends from a typical reduction machine commonly used for functional languages augmented with facilities of the Warren Abstract Machine [War83] to support features of logic languages like unification and backtracking. It combines environment-based reduction with graph-based representation of data structures. Furthermore graph nodes are used to store administrative information about suspended (lazy) calculations, partial applications of higher-order functions, variable bindings, etc. However, no graph reduction of expressions takes place.

The machine state is given by the static program store and the seven dynamic elements instruction pointer (ip), graph store (G), graph pointer (gp), data stack (ds), (environment) stack (st), environment pointer (ep), backtrack pointer (bp), and trail (tr): $\langle ip, G, gp, ds, st, ep, bp, tr \rangle \in State$. At any given time the graph store can be seen as a (partial) function which maps graph addresses to graph nodes. With $Graph$ as the set of such functions, $State = \mathbb{N} \times Graph \times \mathbb{N} \times \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^*$ holds.

Allocation of graph nodes currently happens in the most straightforward way: the graph pointer gp points to the next free graph node and gets continuously adjusted, i.e. is increased by allocations and set back by backtracking which in general causes the graph to shrink. While a choicepoint exists, graph addresses of variables which get bound and of suspensions which are evaluated, are noted in the trail. These bindings and values may depend on information which is only valid in the current evaluation branch and will thus be invalidated by backtrack-

ing; the trail allows to undo affected bindings. The data stack is mainly used for purposes of parameter passing. A complete and detailed description of the BEBAM can be found in [Met02]. In the context of this paper the environment stack is of most interest.

On the one hand this stack contains argument blocks for function calls and activation records (environments) which consist of local variables, a pointer to the previously active environment, and a return address for the control flow.² On the other hand choicepoints are placed on the stack in order to keep track of the current machine state whenever a nondeterministic branching decision has to be made. That happens when function definitions with multiple rules and/or multiple guards are encountered (see Sect. 4). The backtrack pointer bp in the state always points to the currently active choicepoint. With Env and CP as the sets of all environments and choicepoints, resp., the set of possible stacks can be described as $Stack_{BEBAM} = (Env \cup (CP^+ \times Env))^+$.

In the BAM the stack looks essentially the same with the small but important difference that the translation of BABEL code never leads to multiple choicepoints stacked on top of each other with no intervening environment, i.e. CP^+ is only CP in the above formula. In Brooks stacked choicepoints may be caused in two ways: by multiple rule blocks even in the case of innermost/lazy narrowing (see Sect. 4.1) and via nested evaluations in the context of anticipated substitutions in the case of needed narrowing (see Sect. 4.2).

Because in the BAM the environment which was active when the choicepoint was created is always located directly beneath the choicepoint and thus easily found, there is no need to store the environment pointer in the choicepoint. However, in the BEBAM it is generally unknown how many choicepoints have been placed on top of this environment, thus it is wise to store the environment pointer in the choicepoint to facilitate the correct reconstruction of the old state. The other choicepoint contents are the same as in the BAM and include e.g. information about the trail length, the graph pointer, the backtrack pointer, and the backtrack address at creation time. The backtrack mechanisms in the BAM and BEBAM are very similar, so we skip the details here.

However, there is a subtle point concerning local variables which is worth to mention. Backtracking in the BAM always abandons a whole rule and starts afresh with the next one, i.e. bindings of local variables made before backtracking occurs can be safely set back because no local variable can have been in a bound state at choicepoint creation time. But backtracking in the BEBAM is not limited to the rule level if needed narrowing is used, so it is quite likely that some local variables are already in a bound state at choicepoint creation time and others are not but become bound later on. Therefore at first glance it may appear to be necessary to memorize the binding situation in the choicepoint.

² An argument block is always associated with an environment, but strictly speaking not part of it. If no choicepoint protects the block, it is removed as soon as the function body is entered in order to avoid space wastage by (now) superfluous data. However, to avoid distraction by minor details we treat these blocks as environment parts in the following discussion.

If the state had to be reconstructed *exactly*, that would in fact be needed, but fortunately it turns out that it is not required at all to reconstruct the binding state of local variables. The structure of definitional trees ensures that when an alternative branch is taken, local variables either are automatically rebound, or are already correctly bound, or are not used anymore.

4 Compilation of Brooks Programs

A Brooks program consists of collections of data type definitions and function definitions (cf. Sect. 2). The latter can be of two different types: functions to be evaluated by innermost or lazy narrowing are defined by rules, and functions to be evaluated by needed narrowing are defined by definitional trees: $Function = FunctionByRules \cup FunctionByTree$. Section 4.1 is concerned with the translation of rule-based functions and introduces various aspects shared by both approaches. Afterwards, Sect. 4.2 focuses on the translation of tree-based functions.

4.1 Innermost and Lazy Narrowing

As long as only innermost and lazy narrowing are considered, Brooks code translation for the BEBAM is similar in spirit to BABEL code translation for the BAM. The most noticeable difference is an additional layer of branching due to support for rule blocks, which is illustrated in Fig. 3 with a simple example.

Nesting of choicepoints is especially advantageous when lazy narrowing is deployed. Backtracking to the start of the next rule involves a potentially expensive reevaluation of just evaluated suspensions if they depend on variables which get bound during unification of arguments and patterns (cf. Sect. 3). This is deplorable, but necessary if the pattern parts differ from rule to rule.

However, inside a rule block the patterns are invariant by definition, i.e. reevaluations would lead to the same results and should therefore be avoided. This is accomplished by a nested choicepoint which is created after successful pattern unification and ensures that backtracking affects only the guard parts until all components of the rule block have been handled.

To facilitate a more precise description of the translation process, we define formally the essence of a rule-based function definition. A function is given by its name and a sequence of rule blocks: $FunctionByRules = \mathcal{D} \times Ruleblock^+$, where \mathcal{D} denotes the set of defined function symbols. Each rule block consists of a number of patterns according to the function arity and a sequence of guard-body pairs: $Ruleblock = Pattern^* \times (Guard \times Expr)^+$. The patterns are terms $\mathcal{T}(\mathcal{C}, \mathcal{X})$ over the sets of constructor symbols \mathcal{C} and the variables \mathcal{X} . A guard is a (possibly empty) conjunction of boolean or constraint expressions to be checked for satisfiability: $Guard = Expr^*$.

The translation of a Brooks function is initiated by *trans_func* (see Fig. 4) which constructs a typical chain of evaluation branches for the different rule blocks. TRY-ME-ELSE creates a new choicepoint with the given argument as

```

data Nat' = Z
          | S Nat'

f :: Nat' -> Nat'
f (S x) | g x = S Z
        | h x = S (S Z)
f Z      = Z

g, h :: Nat' -> Bool
g x = ...
h x = ...

f : TRY-ME-ELSE f.2;
    -- pattern unification for (S x)
    TRY-ME-ELSE f.1.2;
    -- check (g x) for satisfiability
    -- body: (S Z)
    RETURN;
f.1.2 : TRUST-ME-ELSE-FAIL;
        -- check (h x) for satisfiability
        -- body: (S (S Z))
        RETURN;
f.2 : TRUST-ME-ELSE-FAIL;
     -- pattern unification for (Z)
     -- body: (Z)
     RETURN;

```

Fig. 3. Brooks example code on the left which leads to nested choicepoint creation in the machine code on the right. Jump targets are designated with hierarchically structured symbolic labels generated during the translation process.

backtrack address corresponding to an alternative branch where evaluation continues after backtracking. Occurrences of `RETRY-ME-ELSE` constitute the middle part of the chain and simply modify the backtrack address of the already existing choicepoint. When no more alternatives are available at this level, i.e. at the end of the chain, `TRUST-ME-ELSE-FAIL` removes the choicepoint.

trans_func hands over the work to *trans_ruleblock* that matches actual arguments against formal parameter patterns by calling *trans_pat* (explained below), removes the arguments from the data stack, and calls *trans_guard_body_pairs* to translate the sequence of guards and associated rule bodies. For this purpose *trans_guard_body_pairs* builds a `TRY-ME-ELSE` chain quite similar to *trans_func*, leading to the choicepoint nesting discussed above. The details of expression translation are not of specific interest here.

The pattern translation via *trans_pat* (see Fig. 5) is built around the three machine instructions `MATCH-VAR`, `MATCH-CONSTR-ELSE`, and `INITIATE`. The generated unification code follows closely the structure of the pattern in question. `MATCH-VAR` unconditionally binds a local variable to an argument, and `MATCH-CONSTR-ELSE` tries to match the specified constructor. Such a match can only be performed against a constructor or an unbound variable, but not a suspension. Therefore, when a lazy strategy is used, `INITIATE` examines the situation right before the call to `MATCH-CONSTR-ELSE` and, if it proves necessary, temporarily sets up an environment with the local variables of the suspension and enforces an evaluation to weak head normal form (WHNF).

To give an impression of the machine's inner workings and to emphasize a useful extension of the BEBAM over the BAM, details about the instruction `MATCH-CONSTR-ELSE` are shown in Fig. 6. Four different cases have to be distinguished depending on the term found on top of the data stack. It can be

$$\begin{aligned}
& \text{trans_func} : \text{FunctionByRules} \rightarrow \text{SymbolicInstruction}^+ \\
& \text{trans_func}(\langle f, \langle \text{ruleblock}_i \rangle_{i \in \{1, \dots, \text{no_of_ruleblocks}\}} \rangle) = \\
& \quad f : \quad \text{TRY-ME-ELSE}(f.2) \\
& \quad \quad \text{trans_ruleblock}(f, f.1)(\text{ruleblock}_1) \\
& \quad f.2 : \quad \text{RETRY-ME-ELSE}(f.3) \\
& \quad \quad \text{trans_ruleblock}(f, f.2)(\text{ruleblock}_2) \\
& \quad \vdots \\
& \quad f.\text{no_of_ruleblocks} : \text{TRUST-ME-ELSE-FAIL} \\
& \quad \quad \text{trans_ruleblock}(f, f.\text{no_of_ruleblocks})(\text{ruleblock}_{\text{no_of_ruleblocks}})
\end{aligned}$$

Fig. 4. Translation of a function with multiple rule blocks. The type of *trans_ruleblock* is $\mathcal{D} \times \text{Label} \rightarrow \text{Ruleblock} \rightarrow \text{SymbolicInstruction}^+$, where the first two arguments serve technical purposes further down the call chain, e.g. jump label generation and function specific lookup of variable index positions (cf. *trans_pat* in Fig. 5).

$$\begin{aligned}
& \text{trans_pat} : \mathcal{D} \times \mathbb{N} \rightarrow \mathcal{T}(\mathcal{C}, \mathcal{X}) \rightarrow \text{SymbolicInstruction}^+ \\
& \text{trans_pat}(f, \text{else_addr})(\text{pat}) = \\
& \quad \text{case pat of} \\
& \quad \quad x \text{ with } x \in \mathcal{X} \rightarrow \\
& \quad \quad \quad \text{MATCH-VAR}(\text{variable_index}^{(f)}(x)) \\
& \quad \quad c \ t_1 \ \dots \ t_n \text{ with } c \in \mathcal{C}, \ n \geq 0 \rightarrow \\
& \quad \quad \quad \text{case compilation_mode of} \\
& \quad \quad \quad \quad \text{Innermost/Needed Narrowing} \rightarrow \\
& \quad \quad \quad \quad \quad \epsilon \\
& \quad \quad \quad \quad \text{Lazy Narrowing} \rightarrow \\
& \quad \quad \quad \quad \quad \text{INITIATE} \\
& \quad \quad \quad \quad \text{MATCH-CONSTR-ELSE}(c, n, \text{else_addr}) \\
& \quad \quad \quad \quad \text{trans_pat}(f, 0)(t_1) \\
& \quad \quad \quad \quad \vdots \\
& \quad \quad \quad \quad \text{trans_pat}(f, 0)(t_n)
\end{aligned}$$

Fig. 5. Translation of a pattern. The helper function *variable_index* maps local variable symbols to index positions in the environment. The parameter *else_addr* is always zero for innermost and lazy, but not for needed narrowing. Note that the reason why **INITIATE** seems to be unused for needed narrowing is simply that it has been already generated when *trans_pat* is entered (cf. *trans_tree* in Sect. 4.2).

rooted with the correct constructor which means to inspect the components; it can be an unbound logical variable which means to build a graph representation of the pattern term in a multistep process and to bind the variable; it can be a $\langle \text{HOLE} \rangle$ node which is an intermediate artefact of building a graph representation; or finally it can be a constructor-rooted term which does not match the pattern.

For innermost and lazy narrowing the last situation causes backtracking because the argument $else_addr$ is always set to zero in calls to $trans_pat$. If an $else_addr$ is given, no backtracking is performed but a direct jump to this address. This behavior is exploited by $trans_tree$ in Sect. 4.2 and is the main difference to the otherwise similar instruction MATCH-CONSTR in the BAM which does not know about an $else_addr$ and always performs backtracking in case of a mismatch.

$$\begin{aligned}
& \mathcal{E}xec \llbracket \text{MATCH-CONSTR-ELSE}(c, n, else_addr) \rrbracket (ip, G, gp, ds, st, ep, bp, tr) = \\
& \quad \text{let } ds \rightarrow d_0 \cdot ds' \\
& \quad \text{in case } G(d_0) \text{ of} \\
& \quad \quad \langle \text{CONSTR}, c, b_1 : \dots : b_n \rangle \rightarrow \\
& \quad \quad \quad \text{let } ds^* = \text{dereference}(G, b_1) : \dots : \text{dereference}(G, b_n) \cdot ds' \\
& \quad \quad \quad \text{in } (ip + 1, G, gp, ds^*, st, ep, bp, tr) \\
& \quad \quad \quad \text{where} \\
& \quad \quad \quad \quad \text{dereference} : \text{Graph} \times \mathbb{N} \rightarrow \mathbb{N} \\
& \quad \quad \quad \quad \text{dereference}(G, addr) = \dots \\
& \quad \quad \quad \quad \{- \text{follow indirections through variable and suspension nodes -}\} \\
& \quad \quad \langle \text{VAR}, ? \rangle \rightarrow \\
& \quad \quad \quad \text{let } G^* = G[d_0 / \langle \text{VAR}, gp \rangle, gp / \langle \text{CONSTR}, c, (gp + 1) : \dots : (gp + n) \rangle, \\
& \quad \quad \quad \quad (gp + 1) / \langle \text{HOLE} \rangle, \dots, (gp + n) / \langle \text{HOLE} \rangle] \\
& \quad \quad \quad \quad ds^* = (gp + 1) : \dots : (gp + n) \cdot ds' \\
& \quad \quad \quad \text{in } (ip + 1, G^*, gp + n + 1, ds^*, st, ep, bp, tr \cdot d_0) \\
& \quad \quad \langle \text{HOLE} \rangle \rightarrow \\
& \quad \quad \quad \text{let } G^* = G[d_0 / \langle \text{CONSTR}, c, gp : \dots : (gp + n - 1) \rangle, \\
& \quad \quad \quad \quad gp / \langle \text{HOLE} \rangle, \dots, (gp + n - 1) / \langle \text{HOLE} \rangle] \\
& \quad \quad \quad \quad ds^* = gp : \dots : (gp + n - 1) \cdot ds' \\
& \quad \quad \quad \text{in } (ip + 1, G^*, gp + n, ds^*, st, ep, bp, tr) \\
& \quad \quad \text{otherwise } \rightarrow \\
& \quad \quad \quad \text{if } else_addr = 0 \text{ then } \text{backtrack}(ip, G, gp, ds, st, ep, bp, tr) \\
& \quad \quad \quad \quad \text{else } (else_addr, G, gp, ds, st, ep, bp, tr)
\end{aligned}$$

Fig. 6. The central operation for pattern unification (some technical details concerning choicepoint handling are omitted). $G[addr/node]$ denotes the graph G modified at address $addr$ to contain node $node$. Stacks grow to the left, the trail to the right.

4.2 Needed Narrowing

Independent of the way definitional trees for needed narrowing are obtained, given explicitly or generated algorithmically, function definitions eventually reflect the underlying tree structure: $FunctionByTree = \mathcal{D} \times \mathcal{X}^* \times Tree$, where the set of trees can be described inductively:

$$\langle x, \langle \langle pat_i, tree_i \rangle \rangle_{i \in \{1, \dots, n\}} \rangle \in Tree \quad \text{with } x \in \mathcal{X}, \quad pat_1, \dots, pat_n \in \mathcal{C} \cdot (\mathcal{X})^*, \\ tree_1, \dots, tree_n \in Tree \text{ for } n \in \mathbb{N}$$

corresponding to the Brooks construct:

$$\text{case } x \text{ of } \{ pat_1 \rightarrow tree_1 ; \dots ; pat_n \rightarrow tree_n \}$$

and

$$\langle \langle guard_i, body_i \rangle \rangle_{i \in \{1, \dots, n\}} \in Tree \quad \text{with } guard_1, \dots, guard_n \in Guard, \\ body_1, \dots, body_n \in Expr \text{ for } n \in \mathbb{N}$$

corresponding to the Brooks fragment:

$$\dots \mid guard_1 \rightarrow body_1 \\ \vdots \\ \mid guard_n \rightarrow body_n$$

The translation is then driven by this structure and realized by *trans_tree*, shown in Fig. 7. The interesting part is the recursive case, where the walk through the tree is still in progress. At first the (graph address of the) relevant argument is loaded onto the data stack with the **LOAD** instruction which automatically dereferences variable bindings, so that either a constructor-rooted term, or an unbound logical variable, or a suspended evaluation, gets on top of the data stack. Because of an upcoming unification operation with a constructor term, a potential suspension must be evaluated to WHNF which is done by **INITIATE** (cf. Sect. 4.1).

The instruction **SKIP-CONSTR** operates in conjunction with the immediately following **TRY-ME-ELSE** and the **MATCH-CONSTR-ELSE** which is hidden in *trans_pat* (cf. Fig. 5). The **TRY-ME-ELSE** chain represents a branching process which may be deterministic or nondeterministic depending on the loaded argument. If it is a constructor-rooted term only a single branch has to be chosen in a deterministic way because all other branches correspond to different constructors, i.e. cannot match. If it is an unbound variable this variable will be instantiated right now according to the first branch, but this is a nondeterministic decision and may be revised later via backtracking to choose a different branch.

If **SKIP-CONSTR** finds a constructor term on top of the data stack, it skips the next instruction, i.e. reflects the determinism of the situation by avoiding the creation of a choicepoint. By itself that is not enough because the first branch is not necessarily the right one and a regular **MATCH-CONSTR** would still cause backtracking, albeit affecting the wrong choicepoint. Only the extended

$$\begin{aligned}
& \text{trans_tree} : \mathcal{D} \times \text{Label} \rightarrow \text{Tree} \rightarrow \text{SymbolicInstruction}^+ \\
& \text{trans_tree}(f, l)(tree) = \\
& \text{case } tree \text{ of} \\
& \quad \langle \langle guard_i, body_i \rangle \rangle_{i \in \{1, \dots, n\}} \rightarrow \\
& \quad \quad \text{trans_guard_body_pairs}(f, l)(tree) \\
& \quad \langle x, \langle \langle pat_i, tree_i \rangle \rangle_{i \in \{1, \dots, n\}} \rangle \rightarrow \\
& \quad \quad \text{LOAD}(\text{variable_index}^{(f)}(x)) \\
& \quad \quad \text{INITIATE} \\
& \quad \quad \text{SKIP-CONSTR} \\
& \quad \quad \text{TRY-ME-ELSE}(l.2) \\
& \quad \quad \text{trans_pat}(f, l.2.1)(pat_1) \\
& \quad \quad \text{trans_tree}(f, l.1.1)(tree_1) \\
& \quad l.2 : \text{RETRY-ME-ELSE}(l.3) \\
& \quad l.2.1 : \text{trans_pat}(f, l.3.1)(pat_2) \\
& \quad \quad \text{trans_tree}(f, l.2.1)(tree_2) \\
& \quad \quad \vdots \\
& \quad \quad \vdots \\
& \quad l.n : \text{TRUST-ME-ELSE-FAIL} \\
& \quad l.n.1 : \text{trans_pat}(f, 0)(pat_n) \\
& \quad \quad \text{trans_tree}(f, l.n.1)(tree_n)
\end{aligned}$$

Fig. 7. Translation of a definitional tree (some minor technical details are omitted). The function *trans_pat* for pattern translation is shown in Fig. 5.

functionality of MATCH-CONSTR-ELSE makes the approach work (cf. Sect. 4.1 and Fig. 6). Backtracking is not performed in case of a mismatch, but instead the search for the matching alternative is continued by a direct jump to the next MATCH-CONSTR-ELSE.

The net effect can be understood as a kind of *if-then-elseif-...* chain which tries all possible constructors in sequence until the matching one is found. A slight optimization might be the introduction of a more direct *switch*-like construct but the idea would be the same.

Note that the mechanism also behaves as intended if the loaded argument was an unbound logical variable: SKIP-CONSTR does nothing, the choicepoint is created, and the *else_addr* of MATCH-CONSTR-ELSE won't be used because the unification of an unbound variable and a constructor always succeeds.

The presentation of translation schemes for innermost/lazy narrowing and needed narrowing side by side in the framework of a single machine points up an interesting difference in the utilization of the common choicepoint concept. Leaving guards out of account, one can say that needed narrowing uses choicepoints in a very fine-grained way, i.e. each choicepoint corresponds directly to a single variable which will get bound to different constructor terms in the course of computing all solutions. The decision to create a choicepoint is made dynamically at run time when the actual need arises to handle nondeterminism.

In contrast to that, with innermost/lazy narrowing the decision to create a choicepoint is statically made at compile time whenever a function is defined by more than a single rule. It is very well possible that a choicepoint is created although the situation turns out later to be deterministic, rendering the creation pointless with hindsight. The reason for that inefficiency is the coarse-grained usage: in a function call a single choicepoint potentially covers all unbound variables in all arguments simultaneously. In general, it is too costly to determine beforehand if variables will get bound or not, so only after the whole unification process is completed, it becomes clear if the choicepoint was indeed needed.

5 Conclusion and Related Work

With the long-term goal in mind to embed a host language with flexible evaluation strategies into a framework of cooperating constraint solvers, we attempted to explore the possibilities for an implementational integration of different narrowing strategies. As a result we developed the abstract machine BEBAM which currently supports innermost, lazy, and needed narrowing; other narrowing strategies as well as residuation may be added at a later stage.

Most functional logic languages are rather tightly associated with specific evaluation strategies, e.g. Escher is based on rewriting, BABEL is based on lazy narrowing, and Curry is based on residuation and needed narrowing. Consequently implementations of these languages tend to focus on the respective strategies or emphasize additional features.

For example, in [LK99] a stack based graph reduction machine is presented that is designed for lazy narrowing and appears to be structurally roughly comparable to the BEBAM. It additionally handles the concurrent aspects of residuation and has special support for encapsulated search operations, which are the main concern of the paper. However, the given machine description is not detailed enough to evaluate with certainty if innermost and needed narrowing are already supported or would require some modifications of the machine semantics.

The BEBAM can be regarded as an extension of a narrowing machine originally designed for innermost and lazy narrowing in the context of BABEL [Loo95]. The main contribution of the BEBAM as described in this paper is therefore the additional support for needed narrowing while avoiding a complete redesign; the idea was to extend the machine, not to start from scratch. Apart from minor issues three relatively small extensions turned out to suffice for that purpose: choicepoints need a pointer to their associated environment, choicepoint construction must be handled conditionally (SKIP-CONSTR), and pattern unification must not necessarily cause backtracking in case of a mismatch but allow for a direct jump to an alternative branch (MATCH-CONSTR-ELSE).

As companion to the BEBAM we have designed the functional logic language Brooks which has a syntax resembling a restricted form of Haskell but enables the use of logic features. We showed how compilation of Brooks code to BEBAM code can take place utilizing the specific machine facilities. In doing so we focused on the construction of TRY-ME-ELSE chains together with some aspects of pattern

unification because in this regard needed narrowing differs most from innermost and lazy narrowing.

The BEBAM and a compiler for Brooks have been prototypically implemented in Haskell as a testbed for the developed ideas and to prove their validity in practice [Met02]. Of course, for serious usage the language needs to be augmented with some convenience features like polymorphism, lambda abstractions, built-in sequences, etc., which were deliberately omitted due to time constraints.

References

- [AEH97] S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. *Proc. of the 14th Int. Conf. on Logic Programming (ICLP'97)*, pages 138–152, July 1997.
- [AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
- [Ant92] S. Antoy. Definitional Trees. In *Proc. of the 3rd Int. Conf. on Algebraic and Logic Programming (ALP'92)*, Springer LNCS 632, pages 143–157, 1992.
- [FHM03] St. Frank, P. Hofstedt, and P.R. Mai. Meta-S: A strategy-oriented Meta-Solver Framework. To appear in *Proc. of the 16th Int. FLAIRS Conf.*, AAAI, 2003.
- [HAK⁺02] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, W. Lux, J.J. Moreno-Navarro, and F. Steiner. Curry – An Integrated Functional Logic Language. Language Report, Version 0.7.2 of September 16, 2002. <http://www.informatik.uni-kiel.de/~mh/curry/papers/report.pdf>.
- [Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
- [Han95] M. Hanus. On extra Variables in (Equational) Logic Programming. In *Proc. of the 12th Int. Conf. on Logic Programming (ICLP'95)*, pages 665–679. MIT Press, 1995.
- [Hof02] P. Hofstedt. A General Approach for Building Constraint Languages. In *AI 2002: Advances in Artificial Intelligence*, Springer LNCS 2557, 2002.
- [LK99] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int. Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.
- [Llo99] J.W. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3), 1999.
- [Loo95] R. Loogen. *Integration funktionaler und logischer Programmiersprachen*. R. Oldenbourg Verlag, 1995.
- [Met02] A. Metzner. *Eine abstrakte Maschine für die (schrittweise) Abarbeitung funktional-logischer Programme*. Diploma thesis, Technische Universität Berlin, July 2002.
- [MNRA92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.
- [War83] D.H.D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI International, Menlo Park, California, October 1983.