# Implementation of a Deterministic Partial E–Unification Algorithm for Macro Tree Transducers

Heinz Faßbender* Heiko Vogler Andrea Wedel*

Abt. Theoretische Informatik, Universität Ulm
89069 Ulm, Germany
e-mail: {fassbend,vogler,wedel}@informatik.uni-ulm.de

April 12, 1994

**Abstract**

During the execution of functional logic programs, particular $E$-unification problems have to be solved quite frequently. In this paper we contribute to the efficient solution of such problems in the case where $E$ is induced by particular term rewriting systems called macro tree transducers. We formalize the implementation of a deterministic partial $E$-unification algorithm on a deterministic abstract machine, called twin unification machine. The unification algorithm is based on a particular narrowing strategy which combines leftmost outermost narrowing with a local constructor consistency check and a particular occur check. The twin unification machine uses two runtime stacks; it is an extension of an efficient leftmost outermost reduction machine for macro tree transducers. The feasibility of the presented implementation technique is proved by an implementation which has been developed on a SPARCstation SLC.

*The work of the authors are supported by the Deutsche Forschungsgemeinschaft (DFG).

## 1 Introduction

The investigation of our paper shows an implementation technique which is expected to contribute to an efficient implementation of functional logic programming languages.

Consider, e.g., the functional logic programming language *BABEL* [MR92]. In Figure 1 we show a BABEL-program which defines the predicate *sublist* and the function *append*; *sublist* checks whether its first argument is a sublist of its second argument, and *append* concatenates two lists in the usual way.

$$\begin{array}{lcl}
sublist(x, y) & = & \textbf{if } y = append(z_1, append(x, z_2)) \textbf{ then } \mathrm{TRUE} \ \ \textbf{else } \mathrm{FALSE} \\
append(CONS(x_1, x_2), y_1) & = & CONS(x_1, append(x_2, y_1)) \\
append(NIL, y_1) & = & y_1
\end{array}$$

Figure 1: A functional logic program.

In computations of the predicate *sublist*, an equation like

$$t_y = append(z_1, append(t_x, z_2)) \qquad (*)$$

has to be solved, where $t_y$ and $t_x$ are the current values of the variables $y$ and $x$, respectively. More precisely, the computation machinery tries to find a substitution $\varphi$ such that the $\varphi$-instance of (*) is true in the equational theory $=_{E_{append}}$ which is induced by the set $E_{append}$; $E_{append}$ consists of the two equations for *append*. Clearly, this is nothing else but the $E_{append}$-unification problem for the terms $t_y$ and $append(z_1, append(t_x, z_2))$; and the computation machinery tries to compute an $E_{append}$-unifier $\varphi$, i.e., it tries to answer the question whether $t_y$ and $append(z_1, append(t_x, z_2))$ are $E_{append}$-unifiable, yes or no.

It is well known that the decidability of an $E$-unification problem depends on the set $E$ of equations. If $E$ is the empty set, then the $E$-unification problem coincides with the usual unification problem of terms which is decidable [Rob65]. If $E$ is the set of Peano's axioms, then the $E$-unification problem coincides with Hilbert's tenth problem which was shown to be undecidable [Mat70].

Clearly, for an unconditional equational specification of a function as, e.g., *append* in Figure 1, the basic computation model is a term rewriting system; Figure 2 shows the rewrite rules of the term rewriting system $\mathcal{R}_{append}$ which is appropriate to compute values of the function *append*.

$$\begin{array}{lcl}
append(CONS(x_1, x_2), y_1) & \rightarrow & CONS(x_1, append(x_2, y_1)) \\
append(NIL, y_1) & \rightarrow & y_1
\end{array}$$

Figure 2: Rewrite rules of the term rewriting system $\mathcal{R}_{append}$.

In the scope of this paper, we focus our attention to such $E$-unification problems which arise in functional logic programming languages and where the set $E$ is induced by a term rewriting system $\mathcal{R}$ in the sense that $E$ can be considered as the symmetric closure of $\mathcal{R}$. In this case, we denote $E$ by $E_{\mathcal{R}}$, and we will talk about the $E_{\mathcal{R}}$-unification problem.

Most of the approaches for trying to solve an $E_{\mathcal{R}}$-unification problem are based on the concept of narrowing [Lan75]. Every approach refers to particular term rewriting systems and to a particular narrowing relation, e.g.,

- canonical term rewriting systems and narrowing [Fay79, Hul80]
- canonical term rewriting systems and basic narrowing [Hul80, MH92]
- left-linear, non-overlapping term rewriting systems and D-narrowing [You88]
- canonical, uniform term rewriting systems and the leftmost outermost narrowing strategy [Pad87]
- totally-defined term rewriting systems and any innermost narrowing strategy [Fri85]
- canonical, totally-defined, not strictly subunifiable term rewriting systems and any narrowing strategy [Ech88]
- canonical, totally-defined, not strictly subunifiable term rewriting systems and unification-driven leftmost outermost narrowing [FV92b].

All these approaches have in common that they try to compute an $E_{\mathcal{R}}$-unifier of two terms $t$ and $s$ by starting from the term $equ(t, s)$, where $equ$ is some new binary symbol (in [Hul80] $equ$ is denoted by $H$). Usually, the computation is followed by or interleaved with unification steps.

Unfortunately, even for very simple term rewriting systems $\mathcal{R}$, the $E_{\mathcal{R}}$-unification problem is undecidable: Post's Correspondence Problems can be coded into term rewriting systems which have the form of tree homomorphisms. Thus, even for very simple term rewriting systems $\mathcal{R}$, any deterministic algorithm $\mathcal{A}$ which tries to compute an $E_{\mathcal{R}}$-unifier, can only be partial in the sense that, for every two terms $t$ and $s$ as input, $\mathcal{A}$ behaves in one of the following three ways:

1. $\mathcal{A}$ terminates and computes an $E_{\mathcal{R}}$-unifier of $t$ and $s$.
2. $\mathcal{A}$ terminates and answers that $t$ and $s$ are not $E_{\mathcal{R}}$-unifiable.
3. $\mathcal{A}$ does not terminate.

We will call such an algorithm a *deterministic partial $E_{\mathcal{R}}$-unification algorithm.*

In this paper, we formalize a special deterministic partial $E_{\mathcal{R}}$-unification algorithm which is appropriate for computing $E_{\mathcal{R}}$-unifiers, where $\mathcal{R}$ is taken from a class of particular term rewriting systems, called macro tree transducers. This special deterministic partial $E_{\mathcal{R}}$-unification algorithm is called *deterministic unification algorithm for macro tree transducers.* Moreover, we formalize an efficient implementation of the deterministic unification algorithm for macro tree transducers. By using this efficient implementation technique, an implementation of a complete functional logic programming language might hopefully also benefit (cf. the discussion in Section 6).

Our deterministic unification algorithm for macro tree transducers is based on a depth-first left-to-right traversal over the computation trees which are induced by the unification-driven leftmost outermost (for short: ulo) narrowing relation [FV92b]. As usual, a computation tree collects all possible computations which are induced by the underlying computation relation (here: the ulo narrowing relation) and which start from a particular sentential form (here: $equ(t, s)$). We note that, since our deterministic unification algorithm for macro tree transducers is based on a depth-first left-to-right traversal, we cannot obtain a better behaviour with respect to termination: it is possible that, in a computation tree, an infinite branch occurs left from the first solution; then our algorithm cannot terminate. We also note that a breadth-first left-to-right traversal behaves better; it is even a semi decision procedure. However, it is unacceptably inefficient.

In the rest of the introduction we explain the concept of macro tree transducer, the ulo narrowing relation, and the implementation of the deterministic unification algorithm for macro tree transducers.

In functional logic programming languages, it can be observed that recursion often occurs in the form of primitive recursion over some inductively defined data types like lists or tree-structured objects (cf. [Pét57] for primitive recursive functions over natural numbers; cf. [Hup78, Kla84, EV91] for primitive recursive functions over trees). In this paper we consider a subclass of the class $PREC$ of primitive recursive functions over trees; this subclass is computed by macro tree transducers [Eng80, CF82, EV85, EV86]. From the program schematic point of view, a macro tree transducer can be considered as a primitive recursive program scheme with parameters; it allows for simultaneous function definitions and for nesting of function calls in parameter positions in right hand sides of function definitions. Since it does not allow function calls in the recursion argument positions, the expressive power of macro tree transducers is rather restricted: the composition closure of macro tree transducers is tightly related to the second level of the LOOP-hierarchy (cf. Lemma 6.4 of [EV91]).

From the term rewriting system point of view, a macro tree transducer is constructor-based [You89], canonical (i.e., confluent and noetherian), left-linear, totally defined [Fri85], and not strictly subunifiable [Ech88]; moreover, for every function symbol $f$ and every constructor symbol $\sigma$, there exists exactly one rule the left hand side of which has the form $f(\sigma(x_1, \ldots, x_k), y_1, \ldots, y_n)$; the right hand side is a term over constructors, variables $y_1, \ldots, y_n$, and recursive function calls; in such a function call, the first argument is a variable $x_1, \ldots, x_k$. The latter restriction implies a recursive descent over the first function argument and thus, it guarantees termination. Figure 2 shows an example of a macro tree transducer with two rewrite rules; it contains the function symbol *append* and constructors $CONS$ and $NIL$.

Now we discuss the ulo narrowing relation introduced in [FV92b]. This relation combines leftmost outermost narrowing with a particular occur check and a local consistency check between head constructor symbols. As usual, all possible derivations induced by the ulo narrowing relation, can be collected in an ordered computation tree, called ulo narrowing tree. As an immediate consequence of Theorem 7.8 of [FV92b] we will prove that the depth-first left-to-right traversal over such narrowing trees is a deterministic partial $E_{\mathcal{R}}$-unification algorithm. The ulo narrowing relation has the advantage that, often, infinite branches left to the leftmost $E_{\mathcal{R}}$-unifier are cut off. This pruning is caused by the occur check, the local consistency check, and the fact that we use outermost narrowing; as usual, an outermost strategy avoids possibly infinite computations of deleted parameters of a function call (in opposite to innermost strategies).

Finally, we turn to the discussion of the implementation. We implement the deterministic unification algorithm for macro tree transducers which is induced by depth-first left-to-right traversals over ulo narrowing trees, on an abstract machine which is called the *twin unification machine*. This machine is an extension of the sdrs machine in [GFV91]; the latter machine implements the leftmost outermost reduction relation of macro tree transducers. The main component of the sdrs machine is a runtime stack which manages the environments during the evaluation of a term $t$; $t$ may contain function symbols and constructors of the macro tree transducer.

The implementation of the deterministic unification algorithm for macro tree transducers simulates the ulo narrowing relation. For this purpose, the sdrs machine is enriched by a second runtime stack. Then each of the two terms $t$ and $s$ which should be $E_{\mathcal{R}}$-unified, is evaluated on one of the two runtime stacks. More precisely, $t$ is evaluated on the left runtime stack to a term $hnf(t)$ in head normal form, i.e., the root symbol of $hnf(t)$ is either a variable or a constructor. Then the control switches to the right runtime stack which evaluates $s$ into head normal form $hnf(s)$, too. Then, one of the following cases occurs:

- If the two roots are labeled by the same constructor, then the control switches back to the left runtime stack and the computation continues with the evaluation of the first subterm of $hnf(t)$ into head normal form.
- If the two roots are labeled by different constructors, then backtracking is initiated.
- If one of the two terms $hnf(t)$ and $hnf(s)$ is a variable, say, $hnf(t)$ is a variable, then the occur check is applied to $hnf(s)$. If it fails, then $hnf(s)$ is evaluated to normal form on the right runtime stack and $hnf(t)$ is bound to this normal form of $hnf(s)$. If the occur check succeeds, then backtracking is initiated.

In order to handle backtracking, choice points are pushed to the runtime stacks; for the management of binding of variables, the twin unification machine uses a graph which results from the tree of the sdrs machine by sharing variables, and a trail with pointers to graph nodes (cf. the implementation of PROLOG on the WAM in [War83]).

An overview over the main ingredients of the paper and their connections is illustrated in Figure 3. It shall give the reader an orientation through the paper.

This paper is organized in seven sections, where the second section contains preliminaries. In Section 3 we recall the definitions of macro tree transducer and the ulo narrowing relation. Furthermore, we present the deterministic unification algorithm for macro tree transducers. In Section 4 we present a slight modification of the implementation of the leftmost outermost reduction relation for macro tree transducers on the sdrs machine in

[GFV91]. We have decided to deserve a complete section for the repetition of this reduction machine, because it gives a good preparation for the implementation of the deterministic unification algorithm for macro tree transducers in Section 5. In Section 6 we compare the implementation of our machine on a SPARCstation SLC with the implementation of the BABEL system [Win94]. Finally, Section 7 contains some concluding remarks, comparisons with related work, and it indicates further research topics.

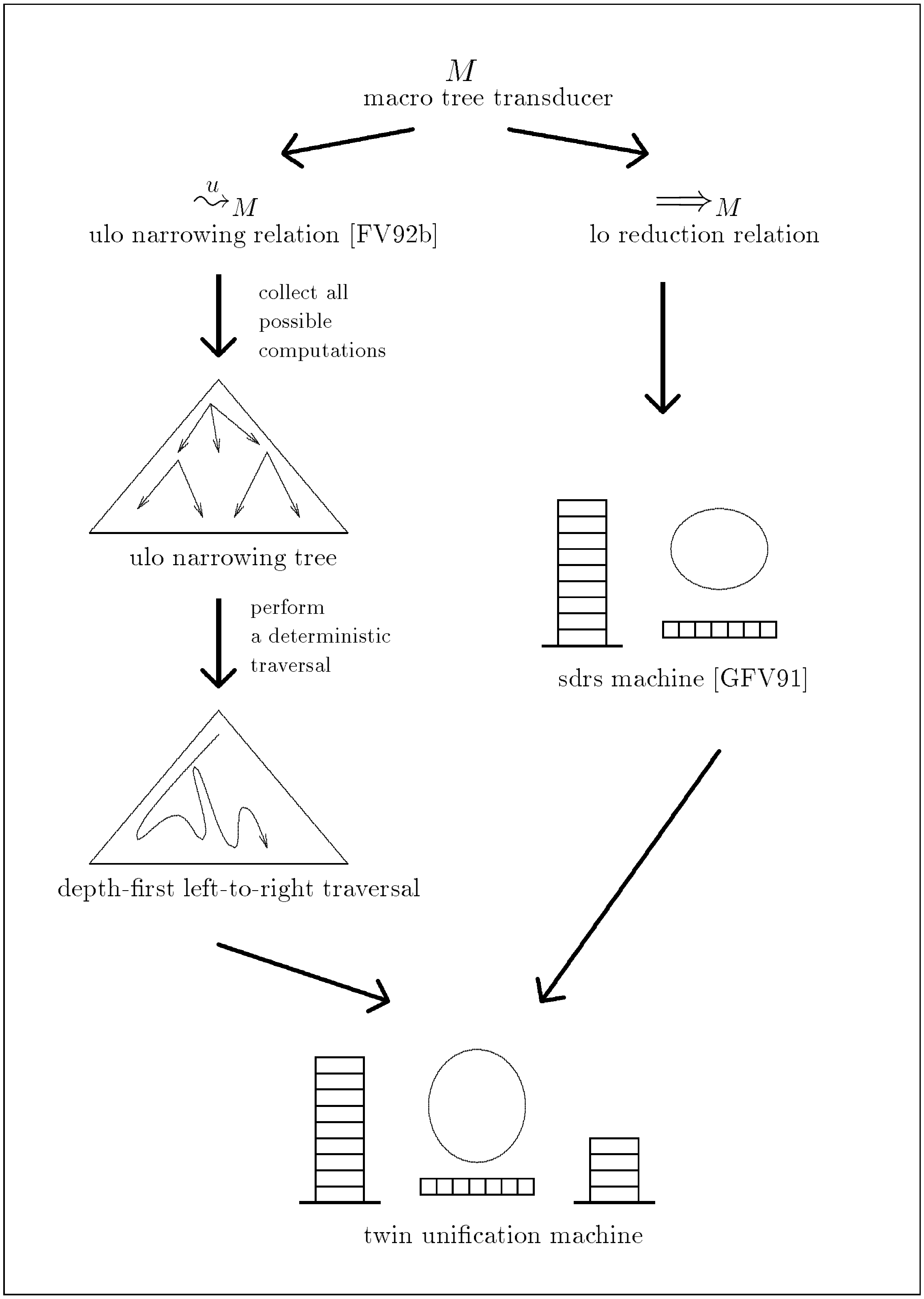

$M$
macro tree transducer
$\overset{u}{\rightsquigarrow}_M$
ulo narrowing relation [FV92b]
$\Longrightarrow_M$
lo reduction relation
collect all possible computations
ulo narrowing tree
perform a deterministic traversal
depth-first left-to-right traversal
sdrs machine [GFV91]
twin unification machine

Figure 3: Illustration of the main ingredients of our approach.

# 2 Preliminaries

We recall and collect some notations, basic definitions, and terminology which will be used in the rest of the paper. We have tried to be in accordance with the notations in [Hue80] and [DJ91] as much as possible.

## 2.1 General Notations

We denote the set of nonnegative integers by $\mathbb{N}$. The empty set is denoted by $\emptyset$. For $i, j \in \mathbb{N}$, $[i, j]$ denotes the set $\{i, i+1, \ldots, j\}$; thus $[i, j] = \emptyset$ if $i > j$. If $i = 1$, then we write $[j]$ instead of $[1, j]$; thus $[0] = \emptyset$. For a finite set $A$, $\mathcal{P}(A)$ is the *set of subsets* of $A$ and $card(A)$ denotes the *cardinality* of $A$. As usual for a set $A$, $A^*$ denotes the set $\bigcup_{n \in \mathbb{N}} \{a_1 a_2 \ldots a_n \mid \text{for every } i \in [n] : a_i \in A\}$ that is called the set of *words over A;* $\Lambda$ denotes the empty word. The $i$-th symbol of a word $w$ is denoted by $w[i]$.

## 2.2 Ranked Alphabets, Variables, and Terms

A pair $(\Omega, \underline{rank}_\Omega)$ is called *ranked alphabet,* if $\Omega$ is an alphabet and $\underline{rank}_\Omega : \Omega \longrightarrow \mathbb{N}$ is a total function. For $f \in \Omega$, $\underline{rank}_\Omega(f)$ is called *rank of f.* The subset $\Omega^{(m)}$ of $\Omega$ consists of all symbols of rank $m$ $(m \geq 0)$. Note that, for $i \neq j$, $\Omega^{(i)}$ and $\Omega^{(j)}$ are disjoint. If $\underline{rank}_\Omega(a) = n$, then we write $a^{(n)}$. If the ranks of the symbols are clear from the context, then we drop the function $\underline{rank}_\Omega$ from the denotation of the ranked alphabet $(\Omega, \underline{rank}_\Omega)$ and simply write $\Omega$.

Let $\mathcal{V}$ denote a fixed enumerable set of *variables* which is divided into three disjoint sets $X = \{x_1, x_2, \ldots\}$, $Y = \{y_1, y_2, \ldots\}$, and $FV = \{z_1, z_2, \ldots\}$ of *recursion variables, parameter variables,* and *free variables,* respectively.

Let $\Omega$ be a ranked alphabet and let $S$ be an arbitrary set. Then the set of *terms over* $\Omega$ *indexed by S,* denoted by $T\langle\Omega\rangle(S)$, is defined inductively as follows: (i) $S \cup \Omega^{(0)} \subseteq T\langle\Omega\rangle(S)$ and (ii) for every $f \in \Omega^{(k)}$ with $k \geq 1$ and $t_1, \ldots, t_k \in T\langle\Omega\rangle(S)$ : $f(t_1, \ldots, t_k) \in T\langle\Omega\rangle(S)$. The set $T\langle\Omega\rangle(\emptyset)$, denoted by $T\langle\Omega\rangle$, is called the set of *ground terms over* $\Omega$.

For a term $t \in T\langle\Omega\rangle(\mathcal{V})$, the set of *occurrences of t,* denoted by $O(t)$, is written in Dewey's notation. It is defined inductively on the structure of $t$ as follows:

(i) If $t \in \mathcal{V} \cup \Omega^{(0)}$, then $O(t) = \{\Lambda\}$, and

(ii) if $t = f(t_1, \ldots, t_n)$ where $f \in \Omega^{(n)}$ and $n > 0$, and for every $i \in [n] : t_i \in T\langle\Omega\rangle(\mathcal{V})$, then $O(t) = \{\Lambda\} \cup \bigcup_{i \in [n]} \{iu \mid u \in O(t_i)\}$.

The prefix order on $O(t)$ is denoted by $<$ and the lexicographical order on $O(t)$ is denoted by $<_{lex}$. The reflexive closures of $<$ and $<_{lex}$ are denoted by $\leq$ and $\leq_{lex}$, respectively. Clearly, $\leq \subseteq \leq_{lex}$. The minimal element with respect to $\leq_{lex}$ in a subset $S$ of $O(t)$ is denoted by $min_{lex} S$. For a term $t \in T\langle\Omega\rangle(\mathcal{V})$ and an occurrence $u$ of $t$, $t/u$ denotes the *subterm of t at occurrence u,* and $t[u]$ denotes the *label of t at occurrence u.* We use $\mathcal{V}(t)$ to denote the set of variables occurring in $t$. Finally, we define $t[u \leftarrow s]$ as the term $t$ in which we have replaced the subterm at occurrence $u$ by the term $s$.

## 2.3 Substitutions, Functions, and Congruences

A $(\mathcal{V},\Omega)$-*substitution* is an assignment $\varphi : \mathcal{V} \to T\langle\Omega\rangle(\mathcal{V})$, where the set $\{x \mid \varphi(x) \neq x, x \in \mathcal{V}\}$ is finite. The set $\{x \mid \varphi(x) \neq x\}$ is denoted by $\mathcal{D}(\varphi)$ and it is called the *domain of* $\varphi$. If $\mathcal{D}(\varphi) = \{x_1,\ldots,x_n\}$, then $\varphi$ is represented by $[x_1/\varphi(x_1),\ldots,x_n/\varphi(x_n)]$. If $\mathcal{D}(\varphi) = \emptyset$, then $\varphi$ is denoted by $\varphi_\emptyset$. We say that $\varphi$ is *ground,* if for every $x \in \mathcal{D}(\varphi) : \mathcal{V}(\varphi(x)) = \emptyset$. The set $\bigcup_{x\in\mathcal{D}(\varphi)} \mathcal{V}(\varphi(x))$ is denoted by $\mathcal{I}(\varphi)$ and it is called the set of *variables introduced by* $\varphi$. The set of $(\mathcal{V},\Omega)$-substitutions and the set of ground $(\mathcal{V},\Omega)$-substitutions are denoted by $Sub(\mathcal{V},\Omega)$ and $gSub(\mathcal{V},\Omega)$, respectively. The *composition* of two $(\mathcal{V},\Omega)$-substitutions $\varphi$ and $\psi$ is the $(\mathcal{V},\Omega)$-substitution which is defined by $\psi(\varphi(x))$ for every $x \in \mathcal{V}$. It is denoted by $\varphi \circ \psi$.

If two functions $f$ and $g$ from $A$ into $B$ are different only for a finite number of elements $a_1,\ldots,a_n \in A$ and if for every $j \in [n] : g(a_j) = b_j$, then we denote $g$ by $f[a_1/b_1,\ldots,a_n/b_n]$. The set of all functions from $A$ into $B$ is denoted by $[A \to B]$. A function $f : A \to B$ is denoted by $f_\emptyset$, if for every $a \in A : f(a)$ is undefined.

An equivalence relation $\sim$ on $T\langle\Omega\rangle(\mathcal{V})$ is called a *congruence relation over* $T\langle\Omega\rangle(\mathcal{V})$, if for every $f \in \Omega^{(n)}$ with $n > 0$ and, for every $t_1, s_1,\ldots,t_n,s_n \in T\langle\Omega\rangle(\mathcal{V})$ with $t_1 \sim s_1,\ldots,t_n \sim s_n$, the relation $f(t_1,\ldots,t_n) \sim f(s_1,\ldots,s_n)$ holds.

## 2.4 $E$-Unification

An *equation over* $\Omega$ *and* $\mathcal{V}$ is a pair $(t,s)$, where $t,s \in T\langle\Omega\rangle(\mathcal{V})$. As usual we denote an equation $(t,s)$ by $t = s$. In the rest of the paper, we let $E$ denote a finite set of equations over $\Omega$ and $\mathcal{V}$. The *E-equality,* denoted by $=_E$, is the finest congruence relation over $T\langle\Omega\rangle(\mathcal{V})$ containing every pair $(\psi(t),\psi(s))$, where $(t = s) \in E$ and $\psi$ is an arbitrary $(\mathcal{V},\Omega)$-substitution. If $t =_E s$, then $t$ and $s$ are called *E-equal* (cf. [HO80]). Two terms $t,s \in T\langle\Omega\rangle(\mathcal{V})$ are called *E-unifiable,* if there exists a $(\mathcal{V},\Omega)$-substitution $\varphi$ such that $\varphi(t) =_E \varphi(s)$.

A *deterministic partial E-unification algorithm* is a deterministic algorithm which takes as input a set $E$ of equations and two terms $t$ and $s$, and which behaves in one of the following three ways:

- It terminates and yields an $E$-unifier of $t$ and $s$.
- It terminates and answers that $t$ and $s$ are not $E$-unifiable.
- It does not terminate.

In Figure 4 we illustrate the behaviours of a deterministic partial $E$-unification algorithm; every pair $(t,s)$ of terms occurs in exactly one of the three illustrated groups. Roughly speaking, for two deterministic partial $E$-unification algorithms $\mathcal{A}$ and $\mathcal{B}$, we say that $\mathcal{A}$ is *better* than $\mathcal{B}$ if the group in the middle, i.e., the group of pairs for which the algorithm does not terminate, is smaller for $\mathcal{A}$ than for $\mathcal{B}$. Clearly, because of the undecidability of the general $E$-unification problem, there is no deterministic partial $E$-unification algorithm for which the group in the middle is empty for every set $E$ of equations.

|  |  |  |
|---|---|---|
| $t$ and $s$ are $E$-unifiable and algorithm terminates | $t$ and $s$ are $E$-unifiable or not $E$-unifiable and algorithm does not terminate | $t$ and $s$ are not $E$-unifiable and algorithm terminates |

Figure 4: Possible behaviours of a deterministic partial $E$-unification algorithm.

The set $\{\varphi \mid \varphi(t) =_E \varphi(s)\}$ is called the *set of E-unifiers of t and s,* and it is denoted by $\mathcal{U}_E(t,s)$ (cf. [Sie89]). Let $V$ be a finite subset of $\mathcal{V}$. We define the preorder $\preceq_E (V)$ on $(\mathcal{V},\Omega)$-substitutions by $\varphi \preceq_E \varphi' \ (V)$, if there exists a $(\mathcal{V},\Omega)$-substitution $\psi$ such that for every $x \in V : \psi(\varphi(x)) =_E \varphi'(x)$ (cf. [Sie89]).

Let $\Omega$ be divided into two disjoint sets $F$ and $\Delta$, let $t, s \in T\langle\Omega\rangle(\mathcal{V})$ and $V = \mathcal{V}(t) \cup \mathcal{V}(s)$. A $(\mathcal{V},\Delta)$-substitution which is an $E$-unifier of $t$ and $s$, is called an *$(E,\Delta)$-unifier of $t$ and $s$.* A set $S$ of $(\mathcal{V},\Delta)$-substitutions is a *ground complete set of $(E,\Delta)$-unifiers of $t$ and $s$ away from $V$* [Ech88] if the following three conditions hold:

1. For every $\varphi \in S$: $\mathcal{D}(\varphi) \subseteq V$ and $\mathcal{I}(\varphi) \cap V = \emptyset$.
2. For every $\varphi \in S$: $\varphi$ is an $(E,\Delta)$-unifier of $t$ and $s$.
3. For every ground $(E,\Delta)$-unifier $\varphi$ of $t$ and $s$, there is a $\psi \in S$ such that $\psi \preceq_E \varphi \ (V)$.

## 2.5 Term Rewriting Systems

A *term rewriting system*, denoted by $\mathcal{R}$, is a pair $(\Omega, R)$, where $\Omega$ is a ranked alphabet and $R$ is a finite set of rewrite rules of the form $l \to r$ such that $l, r \in T\langle\Omega\rangle(\mathcal{V})$ and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ (cf. [Hue80]). For every term rewriting system $\mathcal{R} = (\Omega, R)$, the *related set of equations,* denoted by $E_{\mathcal{R}}$, is the set $\{l = r \mid l \to r \in R\}$ (cf. [MH92]).

Let $\mathcal{R} = (\Omega, R)$ be a term rewriting system and let $t \in T\langle\Omega\rangle$.

- The *set of redex interfaces for $\mathcal{R}$ and $t$,* denoted by $redI(\mathcal{R},t)$, is the set
$$\{(u, \varphi, l \to r) \mid u \in O(t) \text{ with } t/u \notin \mathcal{V}, \varphi \in Sub(\mathcal{V},\Omega), l \to r \in R \text{ with } \varphi(l) = t/u\}.$$
- The *set of redex occurrences for $\mathcal{R}$ and $t$,* denoted by $redO(\mathcal{R},t)$, is the set
$$\{u \mid (u, \varphi, l \to r) \in redI(\mathcal{R},t)\}.$$

- The *reduction relation associated with $\mathcal{R}$,* denoted by $\Longrightarrow_{\mathcal{R}}$, is defined as follows: For every $t, s \in T\langle\Omega\rangle : t \Longrightarrow_{\mathcal{R}} s$, if the following two conditions hold:
  1. There is a redex interface $(u, \varphi, l \to r) \in redI(\mathcal{R}, t)$.
  2. $s = t[u \leftarrow \varphi(r)]$. □

If $\mathcal{R}$ is clear from the context, then we write $\Longrightarrow$ instead of $\Longrightarrow_{\mathcal{R}}$. We use the standard notation $\Longrightarrow^*$ to denote the transitive-reflexive closure of $\Longrightarrow$. A term rewriting system is *canonical,* if it is confluent and noetherian (cf. [HO80]). A term $t$ is a *normal form of a term* $s$, if $s \Longrightarrow^*_{\mathcal{R}} t$ and $t$ is irreducible, i.e., there does not exist any term $t'$ such that $t \Longrightarrow_{\mathcal{R}} t'$. A $(\mathcal{V}, \Omega)$-substitution $\varphi$ is *in normal form* if for every $x \in \mathcal{D}(\varphi)$, the term $\varphi(x)$ is irreducible.

Finally, we recall the definition of the leftmost outermost narrowing relation which will be used in the definition of the unification-driven leftmost outermost narrowing relation in Subsection 3.2.

Let $\mathcal{R} = (\Omega, R)$ be a term rewriting system and let $t \in T\langle\Omega\rangle(\mathcal{V})$.

- The *set of narrowing interfaces for $\mathcal{R}$ and $t$,* denoted by $narI(\mathcal{R}, t)$, is the set

  $\{(u, \varphi, l \to r, \rho) \mid u \in O(t)$ with $t/u \notin \mathcal{V}, l \to r \in R, \rho$ is a renaming of variables in $l$ such that $\mathcal{V}(\rho(l)) \cap \mathcal{V}(t) = \emptyset$, and $\varphi \in Sub(\mathcal{V}, \Omega)$ is the most general unifier of $\rho(l)$ and $t/u\}$.

- The *set of narrowing occurrences for $\mathcal{R}$ and $t$,* denoted by $narO(\mathcal{R}, t)$, is the set

$$\{u \mid (u, \varphi, l \to r, \rho) \in narI(\mathcal{R}, t)\}.$$

- The *leftmost outermost narrowing occurrence for $\mathcal{R}$ and $t$,* denoted by $lo\text{-}narO(\mathcal{R}, t)$, is the narrowing occurrence $min_{lex} narO(\mathcal{R}, t)$.

- The *set of leftmost outermost narrowing interfaces for $\mathcal{R}$ and $t$,* denoted by $lo\text{-}narI(\mathcal{R}, t)$, is the set

$$\{(u, \varphi, l \to r, \rho) \mid (u, \varphi, l \to r, \rho) \in narI(\mathcal{R}, t) \text{ and } u = lo\text{-}narO(\mathcal{R}, t)\}.$$

- The *leftmost outermost narrowing relation associated with $\mathcal{R}$,* denoted by $\overset{lo}{\rightsquigarrow}_{\mathcal{R}}$, is defined as follows: for every $t, s \in T\langle\Omega\rangle(\mathcal{V})$ and $\psi, \psi' \in Sub(\mathcal{V}, \Omega)$: $(t, \psi)$ *derives to* $(s, \psi')$ *by* $\overset{lo}{\rightsquigarrow}_{\mathcal{R}}$, denoted by $(t, \psi) \overset{lo}{\rightsquigarrow}_{\mathcal{R}} (s, \psi')$, if the following three conditions hold:
  1. there is a leftmost outermost narrowing interface $(u, \varphi, l \to r, \rho) \in lo\text{-}narI(\mathcal{R}, t)$
  2. $s = \varphi(t[u \leftarrow \rho(r)])$
  3. $\psi' = \psi \circ (\varphi|_{\mathcal{V}(t)})$

# 3 The Deterministic Unification Algorithm for Macro Tree Transducers

In this section we define the deterministic unification algorithm for macro tree transducers. For this purpose, we recall the definition of macro tree transducers from [Eng80, CF82] and introduce its leftmost outermost (for short: lo) reduction relation. After that, we recall the unification-driven leftmost outermost narrowing relation (for short: ulo narrowing relation) and the ulo narrowing trees. Finally, we define the deterministic unification algorithm for macro tree transducers which is shown to be a deterministic partial $E$-unification algorithm.

## 3.1 Macro Tree Transducer and LO Reduction Relation

We start this subsection by recalling the notion of macro tree transducer. For the sake of better readability, we first define the set of right hand sides of rewrite rules of a macro tree transducer.

**Definition 3.1** Let $F$ and $\Delta$ be ranked alphabets. For every $f \in F^{(n+1)}$ with $n \geq 0$ and $\sigma \in \Delta^{(m)}$ with $m \geq 0$, the set of $(f,\sigma)$-*right hand sides,* denoted by $RHS(f,\sigma)$, is the smallest set $RHS$ which is defined inductively as follows:

(i) For every $i \in [n]$: $y_i \in RHS$.

(ii) For every $\delta \in \Delta^{(k)}$ with $k \geq 0$ and for every $r_i \in RHS$ with $i \in [k]$ :
$$\delta(r_1, \ldots, r_k) \in RHS.$$

(iii) For every $g \in F^{(k+1)}$ with $k \geq 0$, $i \in [m]$, and for every $r_j \in RHS$ with $j \in [k]$ :
$$g(x_i, r_1, \ldots, r_k) \in RHS.$$

The set of the *right hand sides for* $F$ *and* $\Delta$*,* denoted by $RHS(F,\Delta)$, is the set

$$\bigcup_{f \in F, \sigma \in \Delta} RHS(f,\sigma).$$

□

**Definition 3.2** A *macro tree transducer* is a term rewriting system $(\Omega, R)$, where

- $\Omega$ is partitioned into two disjoint sets $F$ and $\Delta$, where $F$ and $\Delta$ are the sets of *function symbols* and *constructor symbols*, respectively; moreover, $F^{(0)} = \emptyset$.
- If $l \to r$ is in $R$, then $l = f(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n)$ and $r \in RHS(f,\sigma)$ for some $n \geq 0$, $f \in F^{(n+1)}$, $m \geq 0$, and $\sigma \in \Delta^{(m)}$. In this case, $l \to r$ is called $(f,\sigma) - rule$. Moreover, for every $f \in F$ and $\sigma \in \Delta$, $R$ contains exactly one $(f,\sigma) - rule$. □

In the sequel, we will denote a macro tree transducer $(\Omega, R)$ with $\Omega = F \cup \Delta$ by $(F, \Delta, R)$.

**Remark 3.3** With every macro tree transducer $M = (F, \Delta, R)$, a bijection $\pi : R \to [card(R)]$ is associated that describes an enumeration of $R$. The function $\pi$ is defined as follows: We suppose that there exist total orderings on $F$ and $\Delta$, i.e., $(f_1, \ldots, f_n)$ for the elements of $F$ and $(\sigma_1, \ldots, \sigma_\rho)$ for the elements of $\Delta$. In this case, $\pi$ maps the $(f_i, \sigma_j)$-rule to $((i-1) \cdot card(\Delta)) + j$. We write $R$ as follows:

$$\{f_i(\sigma_j(x_1, \ldots, x_{rank(\sigma_j)}), y_1, \ldots, y_{rank(f_i)-1}) \to r_{ji} \mid j \in [\rho], i \in [n]\}.$$

□

To give an example, the set $R_1$ of rewrite rules of the macro tree transducer $M_1 = (F_1, \Delta_1, R_1)$ is shown in Figure 5, where we assume to have a ranked alphabet $F_1 = \{sh^{(2)}, mi^{(1)}\}$ of function symbols and a ranked alphabet $\Delta_1 = \{\sigma^{(2)}, \alpha^{(0)}\}$ of constructors. Intuitively, $M_1$ defines two functions *shovel* and *mirror* with arity 2 and 1, respectively; *mirror* reflects terms over $\Delta$ at the vertical center line, and *shovel* accumulates in its second argument the *mirror*-image of the second subterm of its first argument. If we consider, e.g., the term $t_1 = \sigma(\sigma(\alpha, s_1), s_2)$ for some subterms $s_1$ and $s_2$, then for an arbitrary term $t_2$, $sh(t_1, t_2)$ derives to the term $\sigma(mi(s_1), \sigma(mi(s_2), t_2))$.

$$\begin{array}{lcll}
sh(\alpha, y_1) & \to & y_1 & (1) \\
sh(\sigma(x_1, x_2), y_1) & \to & sh(x_1, \sigma(mi(x_2), y_1)) & (2) \\
mi(\alpha) & \to & \alpha & (3) \\
mi(\sigma(x_1, x_2)) & \to & \sigma(mi(x_2), mi(x_1)) & (4)
\end{array}$$

Figure 5: Set of rewrite rules of the macro tree transducer $M_1$.

**Remark 3.4** Every macro tree transducer $M$ is a ctn-trs, i.e., it is canonical (i.e., confluent and noetherian) [FHVV93], constructor-based [You89], totally defined (i.e., every normal form does not contain any function symbol) [EV85], and it is not strictly sub-unifiable [Ech88]. A term rewriting system is strictly sub-unifiable if there exist two rewrite rules $l \to r$ and $l' \to r'$ such that (i) there exists an occurrence $u \in O(l) \cap O(l')$ where $l/u$ and $l'/u$ are unifiable and their most general unifier is neither a variable renaming nor the empty substitution, and (ii) for every $v \in O(l) \cap O(l')$ with $v < u$ we have $l[v] = l'[v]$. These conditions cannot be fulfilled by a macro tree transducer, because of the structure of the rewrite rules' left hand sides. □

In the rest of the paper, $M$ denotes an arbitrary, but fixed macro tree transducer $(F, \Delta, R)$.

The leftmost outermost reduction relation is a subset of the reduction relation (cf. Section 2) which only allows reduction at the leftmost outermost redex occurrence.

**Definition 3.5** Let $M = (F, \Delta, R)$ be a macro tree transducer and let $t \in T\langle F \cup \Delta \rangle$.

- The *leftmost outermost redex occurrence for $M$ and $t$,* denoted by $lo\text{-}redO(M, t)$, is the redex occurrence $min_{lex} redO(M, t)$.

- The *set of leftmost outermost redex interfaces for $M$ and $t$,* denoted by $lo\text{-}redI(M,t)$, is the set

$$\{(u,\varphi,l \to r) \mid (u,\varphi,l \to r) \in redI(M,t) \text{ and } u = lo\text{-}redO(M,t)\}.$$

- The *leftmost outermost (for short: lo) reduction relation associated with $M$,* denoted by $\overset{lo}{\Longrightarrow}_M$, is defined as follows. For every $t,s \in T\langle F \cup \Delta\rangle$ define $t \overset{lo}{\Longrightarrow}_M s$, if the following two conditions hold:

  1. There is a leftmost outermost redex interface $(u,\varphi,l \to r) \in lo\text{-}redI(M,t)$.
  2. $s = t[u \leftarrow \varphi(r)]$ □

Note that $lo\text{-}redO(M,t) \in redO(M,t)$ and that $lo\text{-}redI(M,t) \subseteq redI(M,t)$. We sometimes use indices for $\overset{lo}{\Longrightarrow}_M$ to indicate the lo redex occurrence or the applied rule. For instance, $\overset{lo}{\Longrightarrow}_{M,u,l\to r}$ denotes the reduction step in Definition 3.5. Furthermore, we often replace the applied rule by its number. A derivation by $\overset{lo}{\Longrightarrow}_{M_1}$, where $M_1$ is the macro tree transducer in Figure 5, is illustrated in the following example.

**Example 3.6** Let $M_1$ be the macro tree transducer in Figure 5. Consider the term $t = sh(\sigma(\alpha,\alpha), mi(\alpha))$.

$$\begin{aligned} t &\overset{lo}{\Longrightarrow}_{M_1,\Lambda,(2)} sh(\alpha, \sigma(mi(\alpha), mi(\alpha))) \\ &\overset{lo}{\Longrightarrow}_{M_1,\Lambda,(1)} \sigma(mi(\alpha), mi(\alpha)) \\ &\overset{lo}{\Longrightarrow}_{M_1,1,(3)} \sigma(\alpha, mi(\alpha)) \\ &\overset{lo}{\Longrightarrow}_{M_1,2,(3)} \sigma(\alpha,\alpha) \end{aligned}$$

Since there is no function call anymore, the term $\sigma(\alpha,\alpha)$ is the normal form of $t$. □

Note that, the reduction machine only accepts particular expressions as input. They are defined at the beginning of Section 4.

## 3.2 ULO Narrowing Relation

In [FV92b] we have introduced the unification-driven leftmost outermost narrowing relation (for short: ulo narrowing relation). We have shown that, for ctn-trs's, the ulo narrowing relation constitutes a universal unification algorithm for the class of equational theories which are induced by such term rewriting systems.

Roughly speaking, the ulo narrowing relation combines leftmost outermost narrowing with a particular occur check and a local consistency check between head constructor symbols. The local consistency check is based on additional rules called decomposition rules; the original macro tree transducer $M$ together with the decomposition rules form the extension of $M$.

**Definition 3.7** The *extension of $M$,* denoted by $\widehat{M}$, is the triple $(\widehat{F}, \Delta, \widehat{R})$, where

- $\widehat{F} = F \cup \{equ\}$, where $equ$ is a new binary symbol.
- $\widehat{R}$ contains the rules of $R$ and additionally, for every $\sigma \in \Delta^{(k)}$ with $k \geq 0$, the decomposition rule

$$equ(\sigma(x_1, \ldots, x_k), \sigma(x_{k+1}, \ldots, x_{2k})) \to \sigma(equ(x_1, x_{k+1}), \ldots, equ(x_k, x_{2k})).$$

□

The enumeration of the rules in $\widehat{R}$ is given by the bijection $\widehat{\pi} : \widehat{R} \to [card(\widehat{R})]$ such that $\widehat{\pi}|_R = \pi$, where $\pi$ is the bijection that induces the enumeration of $R$ (cf. Remark 3.3), and the decomposition rules are enumerated in any arbitrary order (which is irrelevant in the sequel).

As an example, the set $\widehat{R}_1$ of the extension $\widehat{M}_1 = (\widehat{F}_1, \Delta_1, \widehat{R}_1)$ of the macro tree transducer $M_1$, where $\widehat{F}_1 = \{sh^{(2)}, mi^{(1)}, equ^{(2)}\}$ and $\Delta_1 = \{\sigma^{(2)}, \alpha^{(0)}\}$, includes the rules in Figure 5 and in Figure 6.

$$\begin{array}{lcll} equ(\alpha, \alpha) & \to & \alpha & (5) \\ equ(\sigma(x_1, x_2), \sigma(x_3, x_4)) & \to & \sigma(equ(x_1, x_3), equ(x_2, x_4)) & (6) \end{array}$$

Figure 6: Decomposition rules of $\widehat{M}_1$.

The derivation forms of the ulo narrowing relation are pairs $(e, \varphi)$ consisting of a term $e \in T\langle \widehat{F} \cup \Delta \rangle(FV)$ and an $(FV, \Delta)$-substitution $\varphi$. We allow only variables of the set $FV$ in the derivation forms for preventing conflicts with variables occuring in the rewrite rules.

Intuitively, the ulo narrowing relation is defined as follows. If $(e, \varphi)$ is the current derivation form, then the leftmost occurrence of $equ$ in $e$ is considered; we call this occurrence the important occurrence in $e$ and we denote it by $impO(e)$. Let $e/impO(e) = equ(t_1, t_2)$ for some terms $t_1$ and $t_2$, and let $l_1$ and $l_2$ be the labels of the roots of $t_1$ and $t_2$, respectively. Then we distinguish the following cases.

- If $l_1 = l_2 = \sigma \in \Delta$, then the decomposition rule for $\sigma$ is applied.
- If $l_1, l_2 \in \Delta$ and $l_1 \neq l_2$, then the derivation is stopped without success.
- If $l_1 \in FV$ and $l_2 = \sigma \in \Delta$, then the occur check for $l_1$ is applied to the $(\Delta \cup FV)$-skeleton of $t_2$. If it succeeds, then the derivation is stopped; otherwise, the decomposition rule for $\sigma$ is applied.

  The $(\Delta \cup FV)$*-skeleton of a term* $t$ is the set of all occurrences $u \in O(t)$ such that there does not exist any prefix $v$ of $u$ which is labeled by a function symbol.
- If $l_2 = \sigma \in \Delta$ and $l_2 \in FV$, then the ulo narrowing relation behaves similarly to the previous case.

- If $l_1, l_2 \in FV$ and $l_1 \neq l_2$, then $e/impO(e)$ and every occurrence of $l_1$ and $l_2$ in $e$ are replaced by a free variable $z_k$ which is not yet used.
- If $l_1 = l_2 = z_i \in FV$, then $e/impO(e)$ is replaced by $z_i$.
- If $l_1 \in F$, then a leftmost outermost narrowing step is applied to $t_1$.
- If $l_1 \notin F$ and $l_2 \in F$, then a leftmost outermost narrowing step is applied to $t_2$.

We refer the reader to [FV92b] for a detailled motivation of these requirements. Now we recall the formal definitions of the involved notions and the ulo narrowing relation.

**Definition 3.8** Let $e \in T\langle \widehat{F} \cup \Delta \rangle(FV)$.

- The *important occurrence in e,* denoted by $impO(e)$, is the occurrence

$$min_{lex}\{u \in O(e) \mid e[u] = equ\}.$$

- $e$ is in *binding mode,* if $e/impO(e) = equ(z_i, z_j)$ and $z_i, z_j \in FV$.
- The $(\Delta \cup FV)$*-skeleton of* $e$ is the set

$$\{u \in O(e) \mid \text{for every } v \in O(e) \text{ with } v \leq u : e[v] \notin F\}.$$

- The *occur check for e succeeds* if $e$ is not in binding mode and there is exactly one $i \in [2]$ such that $e[impO(e)i] = z_j \in FV$ and there exists an occurrence $u$ in the $(\Delta \cup FV)$-skeleton of $e/(impO(e)(3-i))$ such that $(e/(impO(e)(3-i)))[u] = z_j$. $\square$

**Definition 3.9** The *unification-driven leftmost outermost narrowing relation associated with* $\widehat{M}$, denoted by $\overset{u}{\rightsquigarrow}_{\widehat{M}}$, is defined as follows. For every $e_1, e_2 \in T\langle \widehat{F} \cup \Delta \rangle(FV)$ and $\psi, \psi' \in Sub(FV, \Delta)$ : $(e_1, \psi) \overset{u}{\rightsquigarrow}_{\widehat{M}} (e_2, \psi')$ if $e_1/impO(e_1) = equ(t_1, t_2)$, for some $t_1, t_2 \in T\langle F \cup \Delta \rangle(FV)$ and one of the following four conditions holds:

1. $((t_1[\Lambda], t_2[\Lambda] \in \Delta$ and $t_1[\Lambda] = t_2[\Lambda])$ or $(((t_1[\Lambda] \in \Delta$ and $t_2[\Lambda] \in FV)$ or $(t_1[\Lambda] \in FV$ and $t_2[\Lambda] \in \Delta))$ and the occur check fails for $e_1))$, $e_1/impO(e_1)$ is unifiable with the left hand side $l$ of a decomposition rule $l \rightarrow r$ with the most general unifier $\varphi$, $e_2 = \varphi(e_1[impO(e_1) \leftarrow r])$, and $\psi' = \psi \circ (\varphi|_{\mathcal{V}(e_1)})$.
2. $t_1[\Lambda], t_2[\Lambda] \in FV$ and
   (a) $t_1 \neq t_2$, $e_2 = \varphi(e_1[impO(e_1) \leftarrow z_k])$, and $\psi' = \psi \circ \varphi$,
   where $\varphi = [t_1/z_k, t_2/z_k]$ and $k = min\{i \mid z_i \in FV \backslash (\mathcal{V}(e_1) \cup \mathcal{D}(\psi) \cup \mathcal{I}(\psi))\}$ or
   (b) $t_1 = t_2$, $e_2 = e_1[impO(e_1) \leftarrow t_1]$, and $\psi' = \psi$.
3. $t_1[\Lambda] \in F$, $(t_1, \varphi_\emptyset) \overset{lo}{\rightsquigarrow}_M (t_1', \varphi)$, $e_2 = \varphi(e_1[impO(e_1)1 \leftarrow t_1'])$, and $\psi' = \psi \circ \varphi$.
4. $t_1[\Lambda] \notin F$, $t_2[\Lambda] \in F$, $(t_2, \varphi_\emptyset) \overset{lo}{\rightsquigarrow}_M (t_2', \varphi)$, $e_2 = \varphi(e_1[impO(e_1)2 \leftarrow t_2'])$, and $\psi' = \psi \circ \varphi$. $\square$

If $(e_1, \psi) \overset{u}{\leadsto}_{\widehat{M}} (e_2, \psi')$, then we say that $(e_1, \psi)$ derives to $(e_2, \psi')$ by $\overset{u}{\leadsto}_{\widehat{M}}$. We sometimes use indices for $\overset{u}{\leadsto}_{\widehat{M}}$ to indicate the important occurrence, the applied rule, or the substitution. Furthermore, we often replace the applied rule by its number. In case 2 in the previous definition we use $bm$ as index to indicate that the current term is in binding mode.

In fact, the ulo narrowing relation $\overset{u}{\leadsto}_{\widehat{M}}$ is correct with respect to $(E_M, \Delta)$-unification in the sense that, if $(equ(t,s), \varphi_\emptyset)$ derives to $(e, \varphi)$ by $\overset{u}{\leadsto}_{\widehat{M}}$ for some constructor term $e \in T\langle\Delta\rangle(FV)$ and some $(FV, \Delta)$-substitution $\varphi$, then $\varphi$ is an $(E_M, \Delta)$-unifier of $t$ and $s$. The next theorem shows this connection formally.

**Theorem 3.10** (cf. Theorem 7.8 of [FV92b])
Let $M = (F, \Delta, R)$ be a macro tree transducer, $t, s \in T\langle F \cup \Delta\rangle(FV)$, and let $V$ be the finite set $\mathcal{V}(t) \cup \mathcal{V}(s)$. If there exists a derivation:

$$(equ(t,s), \varphi_\emptyset) = (e_0, \varphi_0) \overset{u}{\leadsto}_{\widehat{M}} (e_1, \varphi_1) \overset{u}{\leadsto}_{\widehat{M}} (e_2, \varphi_2) \overset{u}{\leadsto}_{\widehat{M}} \cdots \overset{u}{\leadsto}_{\widehat{M}} (e_n, \varphi_n),$$

where $e_n \in T\langle\Delta\rangle(FV)$, then $\varphi_n|_V$ is an $(E_M, \Delta)$-unifier of $t$ and $s$. □

In the following example we show a derivation by the ulo narrowing relation for the extension $\widehat{M_1}$ (cf. Figure 5 and 6 for the set of rules).

**Example 3.11**

$$\begin{array}{ll}
 & (equ(\sigma(z_1, z_2), \sigma(mi(\sigma(\alpha, z_3)), z_2)), \varphi_\emptyset) \\
\overset{u}{\leadsto}_{\widehat{M}_1, \Lambda, (6), \varphi_\emptyset} & (\sigma(equ(z_1, mi(\sigma(\alpha, z_3))), equ(z_2, z_2)), \varphi_\emptyset) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 12, (4), \varphi_\emptyset} & (\sigma(equ(z_1, \sigma(mi(z_3), mi(\alpha))), equ(z_2, z_2)), \varphi_\emptyset) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 1, (6), [z_1/\sigma(z_4, z_5)]} & (\sigma(\sigma(equ(z_4, mi(z_3)), equ(z_5, mi(\alpha))), equ(z_2, z_2)), [z_1/\sigma(z_4, z_5)]) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 112, (3), [z_3/\alpha]} & (\sigma(\sigma(equ(z_4, \alpha), equ(z_5, mi(\alpha))), equ(z_2, z_2)), [z_1/\sigma(z_4, z_5), z_3/\alpha]) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 11, (5), [z_4/\alpha]} & (\sigma(\sigma(\alpha, equ(z_5, mi(\alpha))), equ(z_2, z_2)), [z_1/\sigma(\alpha, z_5), z_3/\alpha]) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 122, (3), \varphi_\emptyset} & (\sigma(\sigma(\alpha, equ(z_5, \alpha)), equ(z_2, z_2)), [z_1/\sigma(\alpha, z_5), z_3/\alpha]) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 12, (5), [z_5/\alpha]} & (\sigma(\sigma(\alpha, \alpha), equ(z_2, z_2)), [z_1/\sigma(\alpha, \alpha), z_3/\alpha]) \\
\overset{u}{\leadsto}_{\widehat{M}_1, 2, bm, \varphi_\emptyset} & (\sigma(\sigma(\alpha, \alpha), z_2), [z_1/\sigma(\alpha, \alpha), z_3/\alpha])
\end{array}$$

By Theorem 3.10 follows that, the substitution $[z_1/\sigma(\alpha, \alpha), z_3/\alpha]$ is an $(E_{M_1}, \Delta)$-unifier of the sd-expressions $\sigma(z_1, z_2)$ and $\sigma(mi(\sigma(\alpha, z_3)), z_2)$. □

Note that, the twin unification machine only accepts particular expressions as input. They are defined at the beginning of Section 5.

## 3.3 ULO Narrowing Trees

The ulo narrowing relation is nondeterministic, because at the important occurrence more than one rule may be applicable. Consider, e.g., the fourth derivation step in Example 3.11; here also rule (4) can be applied by using the substitution $[z_3/\sigma(z_6, z_7)]$. Thus usually, there exist more than one derivation by the ulo narrowing relation starting with the same derivation form.

As usual, all derivations by the ulo narrowing relation starting with the same derivation form $(e, \varphi_\emptyset)$ can be collected into one tree, called the *ulo narrowing tree for* $(e, \varphi_\emptyset)$. This is similar to the collection of all SLD-resolutions for one goal $g$ in the SLD-tree for $g$ (cf. [Llo87]).

**Definition 3.12** Let $M = (F, \Delta, R)$ be a macro tree transducer and let $t$ and $s$ be terms in $T\langle F \cup \Delta\rangle(FV)$. The *ulo narrowing tree for* $(equ(t, s), \varphi_\emptyset)$ is the tree $T$ which fulfills the following three conditions:

1. The root of $T$ is labeled by $(equ(t, s), \varphi_\emptyset)$.
2. Every node of $T$ is labeled by a derivation form of a derivation by $\overset{u}{\rightsquigarrow}_{\widehat{M}}$ starting from $(equ(t, s), \varphi_\emptyset)$.
3. If a node $nd$ is labeled by a derivation form $(e_0, \varphi)$ and for every $i \in [m, n]$, $(e_0, \varphi)$ derives to $(e_i, \varphi_i)$ by the $i$-th rule, then $nd$ has $n - m + 1$ sons labeled by $(e_i, \varphi_i)$ from left to right for $i \in [m, n]$. $\square$

In Figure 7 the ulo narrowing tree for the derivation form $(equ(sh(z_1, \alpha), mi(\sigma(z_2, \alpha))), \varphi_\emptyset)$ is shown, where the second components of the derivation forms (i.e., the substitutions) are omitted.

## 3.4 The Deterministic Unification Algorithm

Now we define an algorithm which is based on a depth-first left-to-right traversal over ulo narrowing trees; we will prove that this algorithm is a deterministic partial $E_M$-unification algorithm (as defined in Section 2.4).

**Definition 3.13** Let $M = (F, \Delta, R)$ be a macro tree transducer. The *deterministic unification algorithm for $M$* is the following algorithm:

**INPUT** terms $t$ and $s$ from $T\langle F \cup \Delta\rangle(FV)$.

**OUTPUT** either of the following two outputs:

- "$t$ and $s$ are $(E_M, \Delta)$-unifiable by $\varphi$"
- "$t$ and $s$ are not $(E_M, \Delta)$-unifiable"

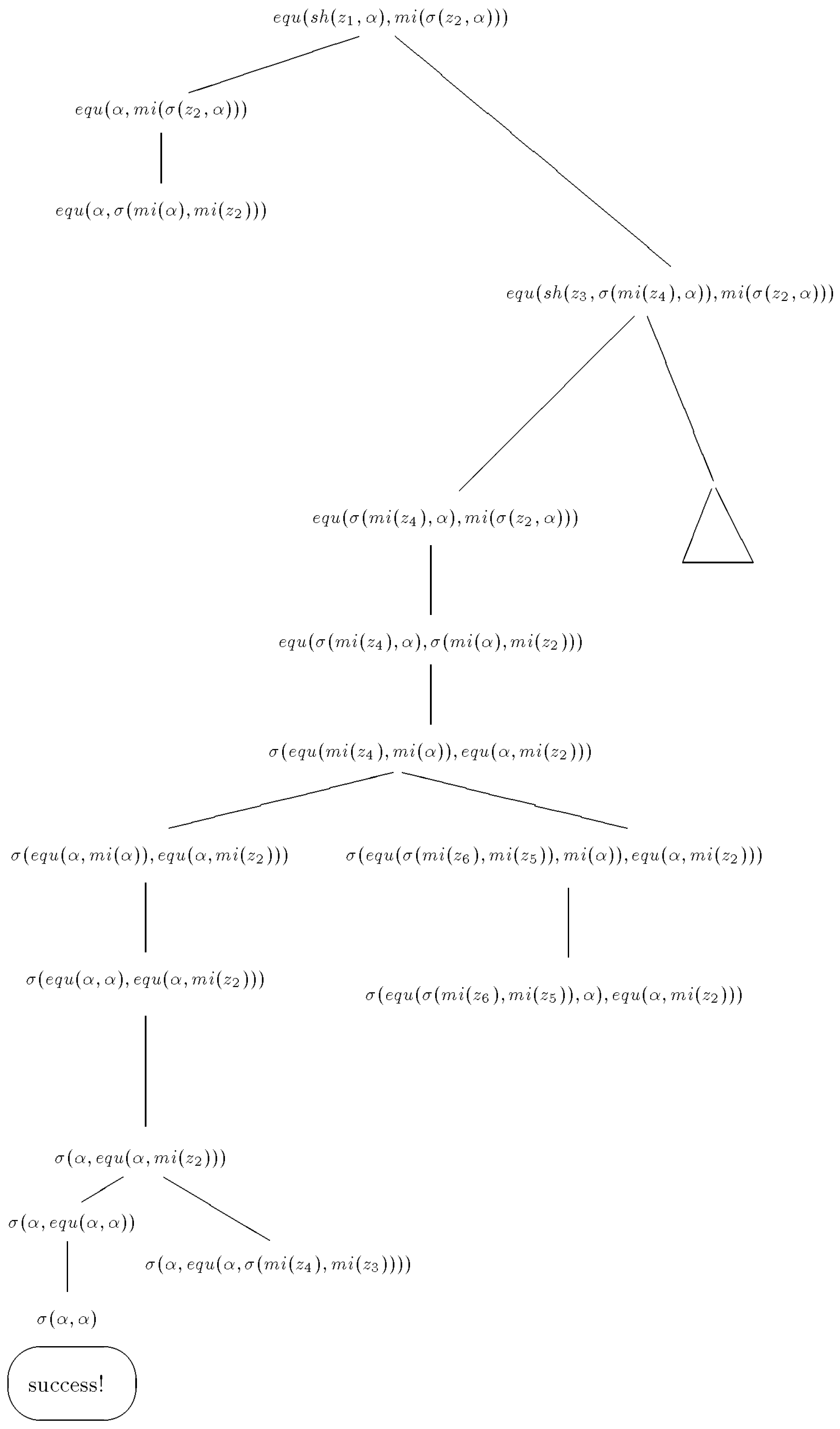

$equ(sh(z_1,\alpha),mi(\sigma(z_2,\alpha)))$
$equ(\alpha,mi(\sigma(z_2,\alpha)))$
$equ(\alpha,\sigma(mi(\alpha),mi(z_2)))$
$equ(sh(z_3,\sigma(mi(z_4),\alpha)),mi(\sigma(z_2,\alpha)))$
$equ(\sigma(mi(z_4),\alpha),mi(\sigma(z_2,\alpha)))$
$equ(\sigma(mi(z_4),\alpha),\sigma(mi(\alpha),mi(z_2)))$
$\sigma(equ(mi(z_4),mi(\alpha)),equ(\alpha,mi(z_2)))$
$\sigma(equ(\alpha,mi(\alpha)),equ(\alpha,mi(z_2)))$
$\sigma(equ(\sigma(mi(z_6),mi(z_5)),mi(\alpha)),equ(\alpha,mi(z_2)))$
$\sigma(equ(\alpha,\alpha),equ(\alpha,mi(z_2)))$
$\sigma(equ(\sigma(mi(z_6),mi(z_5)),\alpha),equ(\alpha,mi(z_2)))$
$\sigma(\alpha,equ(\alpha,mi(z_2)))$
$\sigma(\alpha,equ(\alpha,\alpha))$
$\sigma(\alpha,equ(\alpha,\sigma(mi(z_4),mi(z_3))))$
$\sigma(\alpha,\alpha)$
success!

Figure 7: An ulo narrowing tree.

**PROCEDURE**

1. Construct the ulo narrowing tree for $(equ(t,s), \varphi_\emptyset)$.
2. Perform a depth-first left-to-right traversal over this tree until one of the following two situations occurs:
   (a) A node is reached which is labeled by $(e, \varphi)$ for some $e \in T\langle\Delta\rangle(FV)$. Then stop the algorithm and output: "$t$ and $s$ are $(E_M, \Delta)$-unifiable by $\varphi|_V$" where $V = \mathcal{V}(t) \cup \mathcal{V}(s)$.
   (b) The root is reached from its rightmost son. Then stop the algorithm and output "$t$ and $s$ are not $(E_M, \Delta)$-unifiable".

**END.** □

To show that the deterministic unification algorithm for $M$ is really a deterministic partial $E_M$-unification algorithm, we have to recall the completeness result for $\overset{u}{\leadsto}_{\widehat{M}}$, i.e., the fact that $\overset{u}{\leadsto}_{\widehat{M}}$ generates a ground complete set of $(E_M, \Delta)$-unifiers.

**Theorem 3.14** (cf. Theorem 7.8 of [FV92b])
Let $M = (F, \Delta, R)$ be a macro tree transducer, $t, s \in T\langle F \cup \Delta\rangle(FV)$, and let $V$ be the finite set $\mathcal{V}(t) \cup \mathcal{V}(s)$. Let $S$ be the set of all $(FV, \Delta)$-substitutions $\varphi$ such that there exists a derivation:

$$(equ(t,s), \varphi_\emptyset) = (e_0, \varphi_0) \overset{u}{\leadsto}_{\widehat{M}} (e_1, \varphi_1) \overset{u}{\leadsto}_{\widehat{M}} (e_2, \varphi_2) \overset{u}{\leadsto}_{\widehat{M}} \cdots \overset{u}{\leadsto}_{\widehat{M}} (e_n, \varphi_n),$$

where $e_n \in T\langle\Delta\rangle(FV)$ and $\varphi = \varphi_n|_V$. Then $S$ is a ground complete set of $(E_M, \Delta)$-unifiers of $t$ and $s$ away from $V$. □

The next lemma closes the gap between $E_M$-unifiers of $t$ and $s$ and $(E_M, \Delta)$-unifiers of $t$ and $s$.

**Lemma 3.15** Let $M = (F, \Delta, R)$ be a macro tree transducer and let $t, s \in T\langle F \cup \Delta\rangle(FV)$.

1. If $\varphi$ is an $E_M$-unifier of $t$ and $s$, then there exists an $(E_M, \Delta)$-unifier of $t$ and $s$.
2. Every $(E_M, \Delta)$-unifier of $t$ and $s$ is also an $E_M$-unifier of $t$ and $s$.

*Proof:* Statement 2 follows immediately from the definitions of $E_M$-unifiers and $(E_M, \Delta)$-unifiers in Subsection 2.4.

Statement 1 can be shown as follows: Let $\varphi$ be an $E_M$-unifier of $t$ and $s$ such that for some $z \in FV$, the image $\varphi(z)$ contains a function symbol. Since every macro tree transducer is totally defined, the normal form of any ground instance of $\varphi(z)$ is an element of $T\langle\Delta\rangle$, i.e., it does not contain any function symbol. Now, let $\psi$ be an arbitrary $(FV, \Delta)$-substitution such that for every $z \in FV$, the term $\psi(\varphi(z))$ is ground. Then the $(FV, \Delta)$-substitution $\varphi'$ is an $(E_M, \Delta)$-unifier of $t$ and $s$ where $\varphi'(z)$ is the normal form of $\psi(\varphi(z))$. □

We finish this section by showing that the deterministic unification algorithm for $M$ is really a deterministic partial $E_M$-unification algorithm.

**Theorem 3.16** Let $M = (F, \Delta, R)$ be a macro tree transducer. The deterministic unification algorithm for $M$ is a deterministic partial $E_M$-unification algorithm.

*Proof:* Let $t, s \in T\langle F \cup \Delta\rangle(FV)$.

- Assume that the deterministic unification algorithm for $M$ terminates on input $t$ and $s$ and that it outputs "$t$ and $s$ are $(E_M, \Delta)$-unifiable by $\varphi$". Then, since every path of the ulo narrowing tree corresponds to a derivation by $\overset{u}{\leadsto}_{\widehat{M}}$ and since this derivation ends up with a term $e$ in $T\langle\Delta\rangle(FV)$, it follows from Theorem 3.10 that $\varphi$ is an $(E_M, \Delta)$-unifier of $t$ and $s$. Thus, by Lemma 3.15 (2), $\varphi$ is also an $E_M$-unifier of $t$ and $s$.
- Now assume that the deterministic unification algorithm for $M$ terminates on input $t$ and $s$ and that it outputs "$t$ and $s$ are not $(E_M, \Delta)$-unifiable". Then the depth-first left-to-right traversal has been finished without finding an $(E_M, \Delta)$-unifier of $t$ and $s$. Since the ulo narrowing tree contains all possible derivations by $\overset{u}{\leadsto}_{\widehat{M}}$, it follows from Theorem 3.14 that there does not exist any $(E_M, \Delta)$-unifier of $t$ and $s$. Thus, by Lemma 3.15 (1), $t$ and $s$ are not $E_M$-unifiable.

There is nothing to show in the case where the deterministic unification algorithm for $M$ does not terminate. $\square$

# 4 Reduction Machine

The implementation of the deterministic unification algorithm for macro tree transducers (as defined in Definition 3.13) is an extension of the implementation of the lo reduction relation for macro tree transducers on a reduction machine in [GFV91]. For a better understanding of the former implementation which will be presented in Section 5, we first recall the latter implementation in a slightly modified way.

The implementation of the lo reduction relation in [GFV91] only accepts *ground syntax directed expressions* as input; these are terms over $F$ and $\Delta$ in which the first argument of a function call only contains constructors, i.e., it does not contain any other function call.

**Definition 4.1** Let $F$ and $\Delta$ be ranked alphabets. The set of *ground syntax-directed expressions (for short: gsd-expressions) over $F$ and $\Delta$,* denoted by $gsdExp(F, \Delta)$, is the smallest set $gsdExp$ which is defined inductively as follows:

(i) For every $\delta \in \Delta^{(k)}$ with $k \geq 0$ and for every $t_i \in gsdExp$ with $i \in [k] : \delta(t_1, \ldots, t_k) \in gsdExp$.

(ii) For every $f \in F^{(n+1)}$ with $n \geq 0$, $t \in T\langle\Delta\rangle$, and for every $t_i \in gsdExp$ with $i \in [n]$ : $f(t, t_1, \ldots, t_n) \in gsdExp$. $\square$

For example, the term $sh(\sigma(\alpha, \alpha), mi(\alpha))$ is a gsd-expression over $F_1$ and $\Delta_1$, whereas the term $sh(mi(\alpha), \alpha)$ is not a gsd-expression, because the first argument of the function call of $sh$ includes the function call of $mi$.

**Observation 4.2** The set $gsdExp(F, \Delta)$ is closed under $\overset{lo}{\Longrightarrow}_M$, i.e., if $t \in gsdExp(F, \Delta)$ and $t \overset{lo}{\Longrightarrow}_M s$, then $s \in gsdExp(F, \Delta)$. $\square$

We start with the definition of the reduction machine by presenting its instantaneous descriptions and its machine instructions. Then we present the compilation of rewrite rules and gsd-expressions into code of the reduction machine. We always suppose that a macro tree transducer $M = (F, \Delta, R)$ is given.

## 4.1 Instantaneous Descriptions of the Reduction Machine

In this subsection we introduce the instantaneous descriptions of the reduction machine. An instantaneous description contains the following components, where $PA$ and $Adr$ are sets of program addresses and graph addresses, respectively:

- *program store:* It is a function $ps : PA \to Instr_R$, where $PA = \mathbb{N}$ and the set $Instr_R$ of the reduction machine's instructions will be explained later.

  The program store contains the translation of both, the rewrite rules and the gsd-expression, into a program of the reduction machine. This component remains unchanged during the evaluation of programs.

- *instruction pointer:* It is an element $ip \in PA$.

  The instruction pointer points to the program address of the instruction that has to be executed next.

- *tree:* It is the function $T : Adr \to TNodes$, where $Adr = \mathbb{N}$ and the set $TNodes$ of tree nodes only contains *constructor nodes* of the form

  $\langle CON, \delta, a_1, \ldots, a_k \rangle$ where $\delta \in (\Delta \cup \{\#\})^{(k)}$, $k \geq 0$, and for every $i \in [k] : a_i \in Adr$.

  A tree is necessary for the representation of the recursion arguments of the functions occurring in the gsd-expression $e$. This representation is organized as follows. The root is labeled by a new constructor # the rank of which is equal to the number of the recursion arguments occurring in $e$. The subtrees represent the recursion arguments from left to right. Every recursion argument is a tree over $\Delta$.

  The tree in Figure 8 is the representation of the recursion arguments of the gsd-expression $\sigma(sh(\alpha, mi(\sigma(\alpha, \alpha))), mi(\alpha))$; note that, three recursion arguments occur in this gsd-expression.

- *data stack:* It is an element $ds \in DS$, where $DS = Adr^*$.

  The data stack is only used for the bottom-up creation of the tree.

- *runtime stack:* It is an element $rs \in RS$, where $RS = SYMB^*$ and $SYMB = \{F, Y\} \cup PA \cup \mathbb{N} \cup Adr$.

  The runtime stack is the central component of the machine. It is used to store the environments of function calls and to manage the evaluation of parameters in the correct environment. An environment of a function call is represented by an *F-block* which has the following structure

  $$F : ra : dl : recarg : a_1 : \ldots : a_n$$

  where

  - $F$ is a tag.
  - $ra \in PA$ is the return address of the function call which has caused the existence of this $F$-block.
  - $dl \in \mathbb{N}$ is the dynamic link to the top of the block beneath it.
  - $recarg \in Adr$ is the pointer to the root of the recursion argument of the function call.
  - $a_1, \ldots, a_n \in PA$ are the program addresses of the parameters of the function call.

  For the evaluation of parameters, the runtime stack uses Y-blocks. A *Y-block* has the following structure

  $$Y : ra : sl$$

  where

- $Y$ is a tag.
- $ra \in PA$ is the return address at which the evaluation must be resumed after the evaluation of the parameter which has caused the existence of this $Y$-block.
- $sl \in \mathbb{N}$ is the static link to the top of the $F$-block containing the environment of the current environment.

- *output tape:* It is an element $ot \in OT$, where $OT = \Delta^*$.

  The result of the computation is written to the output tape which is a write only tape.

Since the program store remains unchanged during the evaluation, we will denote an instantaneous description always by a tuple

$$(ip, T, ds, rs, ot) \in ID_R$$

where

$$ID_R = PA \times [Adr \rightarrow TNodes] \times DS \times RS \times OT.$$

Both stacks, the runtime stack and the data stack, are assumed to grow to the left. Such a stack $st$ with $m$ elements is written in the form $st.1 : \ldots : st.m$. If we want to exhibit a finite number $k$ of top elements from the rest $st'$ of the stack $st$, then we use the notation $st.1 : \ldots : st.k : st'$ for $st$.

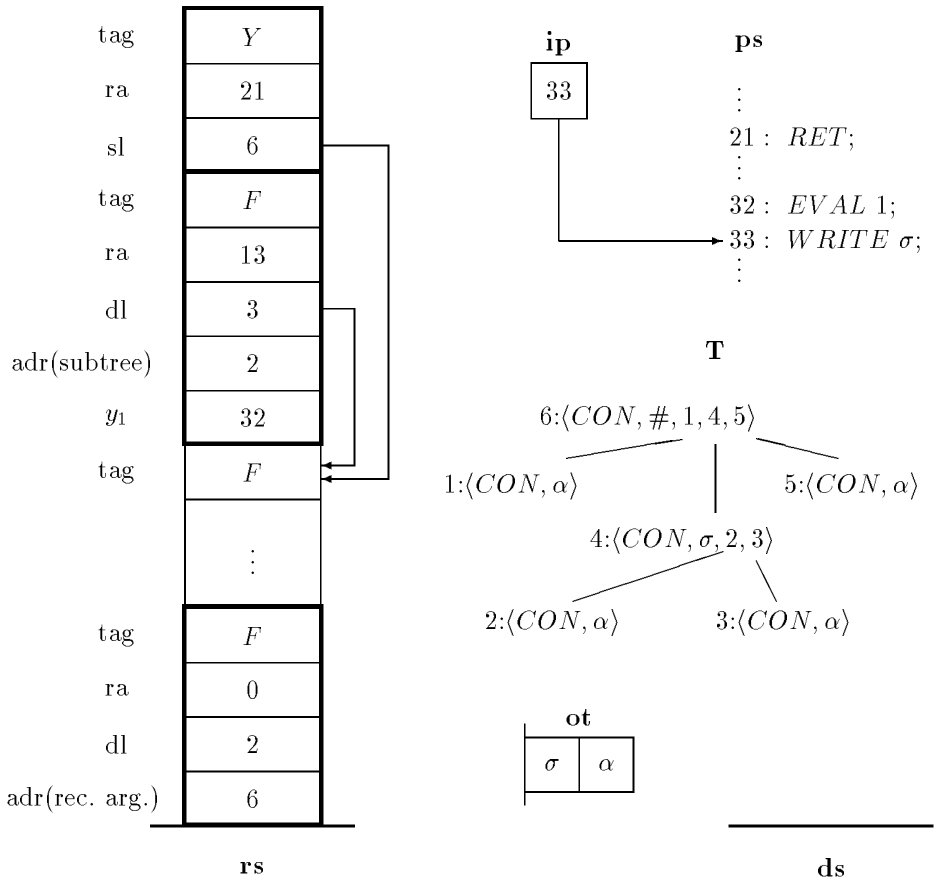
An example of an instantaneous description of a reduction machine is shown in Figure 8 in which also the program store is illustrated.

## 4.2 Machine Instructions of the Reduction Machine

In this subsection we introduce the instructions of the reduction machine and their semantics (cf. Figures 9 and 11). The semantics of an instruction *inst* is a function $\mathcal{C}_R[\![ \; inst \; ]\!] : ID_R \rightarrow ID_R$. We distinguish between control instructions which are responsible for the evaluation of the gsd-expression, and tree instructions which build up the tree at the beginning of the computation.

### Control Instructions

- $JMP\ n$ sets the instruction pointer to $n$.
- The $JMR$-instruction realizes the indexing in the reduction machine. Using the instruction $JMR(\sigma_1 : m_1, \ldots, \sigma_p : m_p)$, the instruction pointer is set to the program address which corresponds to the label of the root of the current recursion argument. The pointer in the fourth square of the topmost block points to the root of the recursion argument in the tree. Whenever $JMR$ is applied, the topmost block is an $F$-block, because this instruction is only executed after a sequence which consists of a $CREATE$-instruction followed by a $JMP$-instruction.



tag
ra
sl
tag
ra
dl
adr(subtree)
$y_1$
tag
tag
ra
dl
adr(rec. arg.)
$Y$
21
6
$F$
13
3
2
32
$F$
$F$
0
2
6
rs
ip
33
ps
21 : $RET$;
32 : $EVAL$ 1;
33 : $WRITE$ $\sigma$;
T
6:$\langle CON, \#, 1, 4, 5\rangle$
1:$\langle CON, \alpha\rangle$
5:$\langle CON, \alpha\rangle$
4:$\langle CON, \sigma, 2, 3\rangle$
2:$\langle CON, \alpha\rangle$
3:$\langle CON, \alpha\rangle$
ot
$\sigma$
$\alpha$
ds

Figure 8: A reduction machine.

- $CREATE(ra, j, m_1, \ldots, m_n)$ increments the instruction pointer and pushes a new $F$-block on top of the runtime stack with the following components: tag $F$, return address $ra$, $n+2$ as dynamic link, a pointer to the $j$-th subterm of the current recursion argument, and the addresses $m_1, \ldots, m_n$ as code addresses for the evaluation of the parameters of the function call. Since the topmost block can be a $Y$-block, we need the auxiliary function *env* to calculate the current environment and then to retrieve the current recursion argument. The function $env : RS \to \mathbb{N}$ yields the position of the top in the $F$-block which contains the current environment (cf. Figure 10).

- The instruction $EVAL\ i$ serves for the evaluation of the $i$-th parameter value of a function call. A $Y$-block is pushed on top of the stack, and the instruction pointer is set to the program address for the computation of the parameter variable $y_i$. This address is found out by means of the function *env*. By means of the static link, the $Y$-block is connected to the $F$-block which contains the environment of the current environment; the static link is evaluated by means of the auxiliary function $next : RS \to \mathbb{N}$ (cf. Figure 10).

$\mathcal{C}_R$ $[\![$ $JMP\ n$ $]\!]$ $(ip, T, ds, rs, ot) =$
  $(n, T, ds, rs, ot)$
$\mathcal{C}_R$ $[\![$ $JMR(\sigma_1 : m_1, \ldots, \sigma_\rho : m_\rho)$ $]\!]$ $(ip, T, ds, rs, ot) =$
  **if** $rs.4 = adr$ and $T(adr) = \langle CON, \sigma_j, adr_1, \ldots, adr_k\rangle$ where $\sigma_j \in \Delta^{(k)}$
  **then** $(m_j, T, ds, rs, ot)$
$\mathcal{C}_R$ $[\![$ $CREATE(ra, j, m_1, \ldots, m_n)$ $]\!]$ $(ip, T, ds, rs, ot) =$
  **if** $rs.(env(rs) + 3) = adr$, $T(adr) = \langle CON, \sigma, adr_1, \ldots, adr_k\rangle$, and $j \in [k]$
  **then** $(ip + 1, T, ds, F : ra : n + 2 : adr_j : m_1 : \cdots : m_n : rs, ot)$
$\mathcal{C}_R$ $[\![$ $EVAL\ i$ $]\!]$ $(ip, T, ds, rs, ot) =$
  $(rs.(env(rs) + 3 + i), T, ds, Y : ip + 1 : next(rs) : rs, ot)$
$\mathcal{C}_R$ $[\![$ $RET$ $]\!]$ $(ip, T, ds, rs, ot) =$
  **if** $rs = F : ra : n + 2 : adr : m_1 : \ldots : m_n : rs'$ or $rs = Y : ra : sl : rs'$
  **then** $(ra, T, ds, rs', ot)$
$\mathcal{C}_R$ $[\![$ $WRITE\ \sigma$ $]\!]$ $(ip, T, ds, rs, ot) =$
  $(ip + 1, T, ds, rs, ot\sigma)$

Figure 9: Control instructions of the reduction machine.

- The $RET$-instruction deletes the topmost block on the runtime stack and it sets the instruction pointer to the return address of this block.
- $WRITE\ \sigma$ appends the constructor $\sigma$ to the end of the output tape and increments the instruction pointer.

$env : RS \to \mathbb{N}$
$env(rs) =$ **if** $rs.1 = F$ **then** $1$
  **if** $rs.1 = Y$ **then** $3 + rs.3$
$next : RS \to \mathbb{N}$
$next(rs) =$ **if** $rs.1 = F$ and $rs.(3 + rs.3) = F$ **then** $3 + rs.3$
  **if** $rs.1 = F$ and $rs.(3 + rs.3) = Y$ **then** $5 + rs.3 + rs.(5 + rs.3)$
  **if** $rs.1 = Y$ and $rs.(3 + rs.3) = F$ **then** $3 + rs.3 + next(rs.(3 + rs.3) : \ldots) - 1$
$new : [Adr \to TNodes] \times \mathbb{N} \to Adr^*$
$new(T, n) =$ **if** $n = 0$ **then** $\Lambda$
  **if** $n > 0$ and $adr_1, \ldots, adr_n$ are the minimal addresses such that
    $T(adr_i)$ is not defined **then** $adr_1 : \ldots : adr_n$

Figure 10: Auxiliary functions *env*, *next*, and *new*.

## Tree Instructions

- $NODE(\sigma, n)$ creates a new node $tadr$ in the tree with the label $\langle CON, \sigma, a_1, \ldots, a_n\rangle$, where $a_1, \ldots, a_n$ are tree addresses of the direct descendants of $tadr$. These $n$ tree addresses are taken from the data stack and they are replaced by the tree address of the new node. For a tree $T$ and a number $n$, the auxiliary function *new* yields $n$ graph addresses which are not yet used (cf. Figure 10).

```
$\mathcal{C}_R$ $[\![ NODE(\sigma, n) ]\!]$ $(ip, T, ds, rs, ot) =$
    **if** $ds = a_n : \cdots : a_1 : ds'$
    **then** $(ip + 1, T[tadr/\langle CON, \sigma, a_1, \ldots, a_n\rangle], tadr : ds', rs, ot)$
    where $tadr = new(T, 1)$
$\mathcal{C}_R$ $[\![ NODE\# ]\!]$ $(ip, T, ds, rs, ot) =$
    **if** $ds = a_n : \cdots : a_1$
    **then** $(ip + 1, T[tadr/\langle CON, \#, a_1, \ldots, a_n\rangle], \Lambda, F : 0 : 2 : tadr : rs, ot)$
    where $tadr = new(T, 1)$
```

Figure 11: Tree instructions of the reduction machine.

- By the $NODE\#$-instruction, the tree addresses $a_1, \ldots, a_n$ of all recursion arguments in the given gsd-expression are connected to a tree the root of which is labeled by $\langle CON, \#, a_1, \ldots, a_n\rangle$. Furthermore, the initial $F$-block with return address 0 and the pointer to the #-node is pushed on the runtime stack and the data stack becomes empty.

### State Transitions

The transitions of the machine are determined by the code $ps$ that is generated by translating the rewrite rules of the macro tree transducer $M$ and the gsd-expression $e$. The machine execution starts with the instantaneous description

$$(1, T_\emptyset, \Lambda, \Lambda, \Lambda)$$

where $T_\emptyset$ is assumed to be the empty tree.

The transition rule

$$(ip, T, ds, rs, ot) \vdash \mathcal{C}_R [\![ \; ps(ip) \; ]\!] \; (ip, T, ds, rs, ot)$$

is applied until one of the following conditions is true:

- $ip = 0$ and $rs = \Lambda$ (successful computation):

  This indicates that the evaluation has been successful. The output tape contains the result.

- ($ps$ is not defined for $ip$) or ($ip = 0$ and $rs \neq \Lambda$) (failure):

  In this case a failure has occurred. This case is not possible for programs which result from the translation of the rewrite rules of a macro tree transducer together with a gsd-expression.

## 4.3 Compilation of Rewrite Rules and GSD-Expressions

In this subsection we present the compilation of the rewrite rules of the macro tree transducer $M = (F, \Delta, R)$ together with a gsd-expression $e$ into code of the reduction machine. We always suppose that $F = \{f_1, \ldots, f_r\}$ and $\Delta = \{\sigma_1, \ldots, \sigma_\rho\}$. Note that the rewrite rules of $M$ are ordered by the enumeration $\pi$ (cf. Remark 3.3), i.e., there is the following ordering of the rules in $R$:

$$R = \{f_i(\sigma_j(x_1,\ldots,x_{rank(\sigma_j)}),y_1,\ldots,y_{rank(f_i)-1}) \to r_{ji} \mid j \in [\rho], i \in [r]\}.$$

For the sake of simplicity, we always use tree-structured addresses in the generated programs. Such an address is a string of nonnegative integers separated by dots, and it is possible that an instruction is labeled by several (possibly none) tree-structured addresses. Clearly, such addresses may now also appear as parameters of instructions. We assume that a load program exists, which transforms a program with tree-structured addresses into code with usual addresses as it is defined in the previous subsection.

In the description of the compilation schemes we use, for every $i \in \mathbb{N}$, the following metavariables: $r_i \in RHS(F,\Delta)$; $e, e_i \in gsdExp(F,\Delta)$; $t, t_i \in T\langle\Delta\rangle$; and $\alpha, \alpha.i$ are tree-structured addresses.

For the set $R$ of rewrite rules and any gsd-expression $e$, the function *trans* (cf. Figure 12) produces the code of the reduction machine which starts with a JMP $r+1$-instruction; at program address $r+1$ the code starts which is generated by the translation of $e$. Furthermore, for every $i \in [r]$, the rewrite rules of a function symbol $f_i$ are translated by the function *functrans* where the code starts at address $i$. The translation of the rules of $f_i$ is followed by a *RET*-instruction which deletes the topmost block on the runtime stack. This block is an $F$-block which was pushed on the runtime stack at the beginning of the function call's evaluation.

```
$trans(R,e) =$
            $JMP\ r+1;$
   $1:$     $functrans(\{f_1(\sigma_j(x_1,\ldots,x_{rank(\sigma_j)}),y_1,\ldots,y_{rank(f_1)-1}) \to r_{j1} \mid j \in [\rho]\}, 1)\ RET;$
            $\vdots$
   $r:$     $functrans(\{f_r(\sigma_j(x_1,\ldots,x_{rank(\sigma_j)}),y_1,\ldots,y_{rank(f_r)-1}) \to r_{jr} \mid j \in [\rho]\}, r)\ RET;$
   $r+1:$   $goaltrans(e, r+1)$
```

Figure 12: Compilation scheme *trans*.

The code for $f_i$ which is produced by *functrans* (cf. Figure 13), starts with a JMR-instruction. By this instruction, indexing is realized in the reduction machine, i.e., if the root of the current recursion argument is labeled by $\sigma_j$, then the machine jumps to the code for the right hand side of the $(f_i, \sigma_j)$-rule. This code is produced by the function *rhstrans*.

The function *rhstrans* (cf. Figure 14) is defined inductively on the structure of the right hand sides (cf. Definition 3.1).

- If the right hand side is a parameter variable $y_i$, then an *EVAL* $i$-instruction is produced.
- A right hand side the root of which is labeled by a constructor $\sigma_j$, is translated into a *WRITE* $\sigma_j$-instruction which, recursively, is followed by the translations of its subtrees.

```
$functrans(\{f_i(\sigma_j(x_1, \ldots, x_{rank(\sigma_j)}), y_1, \ldots, y_{rank(f_i)-1}) \rightarrow r_{ji} \mid j \in [\rho]\}, i) =$
                    $JMR(\sigma_1 : i.1, \ldots, \sigma_\rho : i.\rho);$
    $i.1 :$           $rhstrans(r_{1i}, i.1)\ \ JMP\ i.(\rho+1);$
                    $\vdots$
    $i.(\rho-1) :$    $rhstrans(r_{(\rho-1)i}, i.(\rho-1))\ \ JMP\ i.(\rho+1);$
    $i.\rho :$        $rhstrans(r_{\rho i}, i.\rho)$
    $i.(\rho+1) :$
```

Figure 13: Compilation scheme *functrans*.

- The code which is produced by the translation of a function call $f_i(x_j, r_1, \ldots, r_n)$, starts with a $CREATE$-instruction which creates an $F$-block on the runtime stack. This $F$-block contains a pointer to the $j$-th subtree of the current recursion argument, and the addresses for the evaluation of the parameters $r_1, \ldots, r_n$. The $CREATE$-instruction is followed by a $JMP$ $i$-instruction which jumps to the code for $f_i$. Furthermore, the code for $r_1, \ldots, r_n$ is produced by recursive calls of *rhstrans* which are followed by a $RET$-instruction. This instruction deletes the $Y$-block which has been pushed on the runtime stack at the beginning of the parameter's evaluation.

```
$rhstrans(y_i, \alpha)$                    $=$  $EVAL\ i;$

$rhstrans(\sigma_j(r_1, \ldots, r_n), \alpha)$       $=$  $WRITE\ \sigma_j;$
                                        $rhstrans(r_1, \alpha.1) \ldots rhstrans(r_n, \alpha.n)$

$rhstrans(f_i(x_j, r_1, \ldots, r_n), \alpha)$    $=$              $CREATE(\alpha.(n+1), j, \alpha.1, \ldots, \alpha.n);$
                                                     $JMP\ i;$
                                        $\alpha.1 :$         $rhstrans(r_1, \alpha.1)\ RET;$
                                                     $\vdots$
                                        $\alpha.n :$         $rhstrans(r_n, \alpha.n)\ RET;$
                                        $\alpha.(n+1) :$
```

Figure 14: Compilation scheme *rhstrans*.

For a gsd-expression $e$, the function *goaltrans* (cf. Figure 15) constructs the tree of recursion arguments in $e$. This is realized by the function *maketree* and the $NODE\#$-instruction. Furthermore, *goaltrans* translates $e$ into code of the reduction machine by the function *exptrans*.

```
$goaltrans(e, \alpha) =$           $maketree(e)$
                             $NODE\#;$
                      $\alpha.1 :$  $exptrans(e, 1, \alpha.1)$
                             $RET;$
```

Figure 15: Compilation scheme *goaltrans*.

The code for the creation of the tree is produced by the functions *maketree* and *makenodes* (cf. Figure 16). If the root of the argument $e$ of *maketree* is labeled by a constructor, then *maketree* is applied to the subtrees of $e$. If the root of $e$ is labeled by a function symbol, then *makenodes* is applied to the recursion argument of the function call, i.e., the first subtree of $e$, and *maketree* is applied to the other subtrees of $e$.

The function *makenodes* constructs the code for the bottom up-construction of the recursion arguments.

$$\begin{array}{ll}
maketree(\sigma_j(e_1,\ldots,e_n)) & = maketree(e_1)\ldots maketree(e_n) \\
maketree(f_i(t,e_1,\ldots,e_n)) & = makenodes(t)\ maketree(e_1)\ldots maketree(e_n) \\
\hline
makenodes(\sigma_j(t_1,\ldots,t_n)) & = makenodes(t_1)\ldots makenodes(t_n)\ NODE(\sigma,n);
\end{array}$$

Figure 16: Tree construction schemes *maketree* and *makenodes.*

The function *exptrans* (cf. Figure 18) is defined very similar to the function *rhstrans*. It has one more parameter, the second one, which is responsible for choosing the correct recursion argument. In this parameter, the function *count* is applied which counts the number of function calls which are left to the current position in the gsd-expression (cf. Figure 17).

$$\begin{array}{l}
count : gsdExp(F,\Delta) \to \mathbb{N} \\
count(e) = \textbf{if } e = \sigma_j(e_1,\ldots,e_n) \textbf{ then } \sum_{i=1}^{n} count(e_i) \\
\qquad\qquad\ \ \textbf{if } e = f_i(t,e_1\ldots,e_n) \textbf{ then } 1 + \sum_{i=1}^{n} count(e_i)
\end{array}$$

Figure 17: Auxiliary function *count.*

$$\begin{array}{llll}
exptrans(\sigma_i(e_1,\ldots,e_n),j,\alpha) & = & & WRITE\ \sigma_i; \\
 & & & exptrans(e_1,j,\alpha.1) \\
 & & & \vdots \\
 & & & exptrans(e_n, j + \sum_{k=1}^{n-1} count(e_k), \alpha.n) \\
exptrans(f_i(t,e_1,\ldots,e_n),j,\alpha) & = & & CREATE(\alpha.(n+1), j, \alpha.1,\ldots,\alpha.n); \\
 & & & JMP\ i; \\
 & & \alpha.1 : & exptrans(e_1, j+1, \alpha.1)\ RET; \\
 & & & \vdots \\
 & & \alpha.n : & exptrans(e_n, j+1+\sum_{k=1}^{n-1} count(e_k), \alpha.n)\ RET; \\
 & & \alpha.(n+1) : &
\end{array}$$

Figure 18: Compilation scheme *exptrans.*

In Figure 19 the translation of the rewrite rules in $R_1$ (cf. Figure 5) and the gsd-expression $e = sh(\sigma(\alpha,\alpha), mi(\alpha))$ is shown. The left column includes the *JMP* 22-instruction to the translation of $e$ and the translation of the rewrite rules for $sh$. The column in the middle includes the translation of the rules for $mi$ and finally, the right column includes the translation of $e$. The computation of this program is the implementation of the derivation by $\overset{l_O}{\Longrightarrow}_{M_1}$ starting with $e$ as it is shown in Example 3.6. The stop instantaneous description is the tuple $(0, T, \Lambda, \Lambda, \sigma\alpha\alpha)$ where

$$T = T_\emptyset[1/\langle CON, \alpha\rangle, 2/\langle CON, \alpha\rangle, 3/\langle CON, \sigma, 1, 2\rangle, 4/\langle CON, \alpha\rangle, 5/\langle CON, \#, 3, 4\rangle].$$

1 : $JMP\ 22;$
2 : $JMR(\alpha : 3, \sigma : 5);$
3 : $EVAL\ 1;$
4 : $JMP\ 12;$
5 : $CREATE(12, 1, 7);$
6 : $JMP\ 2;$
7 : $WRITE\ \sigma;$
8 : $CREATE(10, 2);$
9 : $JMP\ 13;$
10 : $EVAL\ 1;$
11 : $RET;$
12 : $RET;$

13 : $JMR(\alpha : 14, \sigma : 16);$
14 : $WRITE\ \alpha;$
15 : $JMP\ 21;$
16 : $WRITE\ \sigma;$
17 : $CREATE(19, 2);$
18 : $JMP\ 13;$
19 : $CREATE(21, 1);$
20 : $JMP\ 13;$
21 : $RET;$

22 : $NODE(\alpha, 0);$
23 : $NODE(\alpha, 0);$
24 : $NODE(\sigma, 2);$
25 : $NODE(\alpha, 0);$
26 : $NODE\#;$
27 : $CREATE(32, 1, 29);$
28 : $JMP\ 2;$
29 : $CREATE(31, 2);$
30 : $JMP\ 13;$
31 : $RET;$
32 : $RET;$

Figure 19: Compilation of $R_1$ and $e = sh(\sigma(\alpha, \alpha), mi(\alpha))$.

## 5 Twin Unification Machine

In this section we extend the implementation of the lo reduction relation to the implementation of the deterministic unification algorithm for macro tree transducers of Definition 3.13. Since the former implementation only accepts gsd-expressions (cf. Definition 4.1) as input, the latter implementation also only accepts particular terms as input which are called *syntax-directed expressions.* A gsd-expression generalizes to a syntax-directed expression if variables of $FV$ are allowed to occur as zero-ary symbols.

**Definition 5.1** The set of *syntax-directed expressions (for short: sd-expressions) over $F$, $\Delta$, and $FV$,* denoted by $sdExp(F, \Delta, FV)$, is the smallest set $sdExp$ which is defined inductively as follows:

(i) $FV \subseteq sdExp$.

(ii) For every $\delta \in \Delta^{(k)}$ with $k \geq 0$ and for every $t_i \in sdExp$ with $i \in [k]$ : $\delta(t_1, \dots, t_k) \in sdExp$.

(iii) For every $f \in F^{(n+1)}$ with $n \geq 0$, $t \in T\langle\Delta\rangle(FV)$, and for every $t_i \in sdExp$ with $i \in [n]$ : $f(t, t_1, \dots, t_n) \in sdExp$. $\Box$

For example, the term $sh(\sigma(z_1, \alpha), mi(\sigma(z_2, z_1)))$ is an sd-expression over $F_1, \Delta_1$, and $FV$, whereas the term $sh(mi(\alpha), z_1)$ is not an sd-expression, because the first argument of the function call of $sh$ includes the function call of $mi$.

For extending the implementation of the lo reduction relation to the implementation of the deterministic unification algorithm for macro tree transducers, the reduction machine is enriched by additional mechanisms for the handling of free variables in sd-expressions, for backtracking, and for unification; the resulting abstract machine is called *twin unification machine.*

The handling of free variables and the way of backtracking are performed in a rather standard way, as, e.g., in [Loo93, War83]. Thus, the main contribution of the twin unification machine is the technique for handling unification. For this purpose, the twin unification machine consists of two runtime stacks. Each of them is responsible for the evaluation of one of the two sd-expressions $e_1$ and $e_2$ which should be $E$-unified. According to the deterministic unification algorithm for macro tree transducers, $e_1$ is evaluated into head normal form on the left runtime stack. After that, the control of the machine switches to the evaluation of $e_2$ into head normal form on the right runtime stack. Then the head symbols are compared and the computation continues with the unification of the subtrees in the same way.

We start this section with the definition of the instantaneous descriptions of the twin unification machine. This definition serves as a base for the following explanations of the additional mechanisms in Subsection 5.2 and hence, it is presented first. In Subsection 5.3, we present the machine's instructions and we finish the section with the definition of the compilation of rewrite rules and sd-expressions in Subsection 5.4.

## 5.1 Instantaneous Descriptions of the Twin Unification Machine

In this subsection we extend the instantaneous descriptions of the reduction machine and we define the instantaneous descriptions of the twin unification machine. An instantaneous description of the twin unification machine contains the following components:

- *program store:* It is a function $ps : PA \rightarrow Instr_U$, where $PA = \mathbb{N}$ and the set $Instr_U$ will be explained later.

  The program store of the twin unification machine differs from the program store of the reduction machine only in the set $Instr_U$ which results from $Instr_R$ by modifying some old instructions and by adding some new ones.

- *instruction pointer:* It is an element $ip \in PA$.

  The instruction pointer remains unchanged with respect to the reduction machine.

- *runtime stack:* It is an element $rs \in RS$, where $RS = SYMB^*$ and $SYMB = \{F, Y, C, CR\} \cup PA \cup Adr \cup \mathbb{N}$.

  In order to evaluate two sd-expressions $e_1$ and $e_2$, the instantaneous descriptions of the twin unification machine contain two runtime stacks $rs1$ and $rs2$. At every moment of the computation, exactly one runtime stack is active. Besides the $F$-blocks and $Y$-blocks that manage the environments in the reduction machine, the runtime stacks contain $C$-blocks and $CR$-blocks (choice-blocks and choice remote-blocks, respectively) for the management of backtracking in the twin unification machine.

  If a choice is performed, then a $C$-block is pushed to the active runtime stack and a $CR$-block is pushed to the other runtime stack. These two blocks contain the pieces of information that are necessary for backtracking. Furthermore, on top of the nonactive runtime stack, there is a *switch*-block which contains the pieces of information for switching from the active runtime stack to the nonactive one.

  The structure of an $F$-block is the same as in the reduction machine. But now the dynamic link $dl$ serves as saved environment pointer, i.e., the block which $dl$ points to, is not necessarily the block below the $F$-block.

  A *Y-block* has the following structure

  $$Y : ra : sep : sl$$

  where the additional component $sep \in \mathbb{N}$ denotes the saved environment pointer which points to the block that becomes the current environment after the deletion of the $Y$-block. Note that $sep$ will be an absolute address in the runtime stack, whereas $dl$ denotes a relative address in analogy to the reduction machine.

  A *C-block* has the following structure

  $$C : sip : sbp : i : lop : lt : lds : e_1 : \ldots : e_{lds}$$

  where

- $C$ is a tag.
- $sip \in PA$ is the saved instruction pointer which points to the instruction in the program at which the computation starts for the next alternative. This instruction is always a $JMR$-instruction.
- $sbp \in PA$ is the saved backtrack pointer which points to the next choice point below the current choice point.
- $i \in \mathbb{N}$ is the number of the alternative which is considered now.
- $lop, lt, lds \in \mathbb{N}$ are the lengths of the output pushdown, trail, and data stack, respectively.
- $e_1, \ldots, e_{lds} \in Adr$ are saved elements of the data stack which must be saved because of possible failures occuring in the unification phase.

A *CR-block* has the following structure

$$CR : sip : sep : sbp$$

where

- $CR$ is a tag.
- *sip*, *sep*, and *sbp* are the saved instruction pointer, environment pointer, and backtrack pointer, respectively.

The *switch-block* has the same structure as the $CR$-block without the tag.

- *graph:* It is the function $G : Adr \to GNodes$, where $Adr = \mathbb{N}$ and $GNodes$ contains, in addition to the constructor nodes in $TNodes$ of the reduction machine, *variable nodes* of the form

  $\langle VAR, a\rangle$ where $a \in Adr \cup \{?\}$.

  The graph component corresponds to the tree of the reduction machine. It can store a term over $\Delta$ and $FV$.

  The node $\langle VAR, ?\rangle$ represents an unbound variable and $\langle VAR, a\rangle$, where $a \in Adr$, embodies a variable that has been bound to the term whose graph representation starts with address $a$.

  Every variable in $FV$ is represented by at most one variable node. Thus, if a variable occurs more than once, then the tree becomes a graph by the mechanism of sharing.

- *state:* It is an element $st \in ST$, where $ST = \{1, 2\} \times \{u, n, e\}$.

  The first component of *st* denotes the active runtime stack. The second component is used during the unification phase. It indicates the current phase: $u$, $n$, and $e$ indicate that the machine is in the unification phase, not in the unification phase, and at the end of the unification phase, respectively.

- *stack of substitutions:* It is an element $ss \in SS$, where $SS = (Adr \cup \{?\})^*$.

  The stack of substitutions contains the graph addresses of the variables of the input terms. In particular, if $ss[i] \in Adr$, then $ss[i]$ denotes the graph address of $z_i$. It is used for the representation of the substitutions which occur in the second component of the derivation forms of $\overset{u}{\leadsto}_{\widehat{M}}$.

- *trail:* It is an element $tr \in TR$, where $TR = Adr^*$.

  The trail is a stack which contains the binding addresses of all free variables occurring in the evaluation.

- *environment pointer:* It is an element $ep \in EP$, where $EP = \mathbb{N}$.

  The environment pointer points to the topmost square of the current environment. It is needed, because there may be many blocks above the current environment which contain pieces of information for backtracking.

- *backtrack pointer:* It is an element $bp \in BP$, where $BP = \mathbb{N}$.

  The backtrack pointer points to the topmost square of the current choice point.

- *data stack:* It is an element $ds \in DS$, where $DS = (Adr \cup \mathbb{N})^*$.

  The data stack is used during the construction of the graph in the same way as it is used in the reduction machine. Furthermore, it is used in the unification phase in which natural numbers have to be remembered.

- *output pushdown:* It is an element $op \in OP$, where $OP = (\Delta \cup FV)^*$.

  The output tape in the reduction machine becomes an output pushdown, because it may be necessary to reset the tape partially in case of backtracking.

As in the reduction machine, the program store remains unchanged during the evaluation and thus, we will drop it from the denotation of an instantaneous description. An instantaneous description will be given by a tuple of the form

$$(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) \in ID_U$$

where

$$ID_U = PA \times RS \times [Adr \rightarrow GNodes] \times RS \times ST \times SS \times TR \times EP \times BP \times DS \times OP.$$

The stacks are assumed to grow to the left where a stack $s$ with $m$ elements is written in the form $s.1 : \ldots : s.m$. We also use an enumeration of the squares in $s$ from right to left such that $s[i] = s.(m{+}1{-}i)$ for $i \in [m]$. For $l \geq r$, we denote the part $s[l] : s[l{-}1] : \ldots : s[r]$ of $s$ by $st[l..r]$. The number of elements of a stack $s$ is denoted by $length(s)$. The output pushdown and the trail grow to the right. In analogy to the stacks, we denote the part $p[l] : \ldots : p[r]$ of $p$ by $p[l..r]$, where $r \geq l$ and $p$ is a trail or an output pushdown; the number of elements of $p$ is denoted by $length(p)$. The two components of a state $st$ are denoted by $st[1]$ and $st[2]$.

## 5.2 From the Reduction Machine to the Twin Unification Machine

Now we explain how the twin unification machine uses its components to realize the handling of free variables in sd-expressions, backtracking, and unification.

Let $e_1$ and $e_2$ be two sd-expressions which should be $E$-unified.

## Graph Store, Twin Runtime Stack, and Switching

Since, on the one hand, free variables may occur more than once in $e_1$ and $e_2$ and, on the other hand, they are represented only once, the tree of the reduction machine becomes a *graph*. In the same way as in the reduction machine, the graph is constructed bottom-up by using the data stack at the beginning of the unification of $e_1$ and $e_2$. Furthermore, for the representation of free variables, the twin unification machine uses a stack of substitutions which points to the graph representations of the variables occurring in $e_1$ and $e_2$. In the trail the addresses of the graph representations of the variables are collected which are introduced during the computation.

The twin unification machine consists of *two runtime stacks* $rs1$ and $rs2$ where $rsi$ is active during the evaluation of $e_i$. The active runtime stack is indicated by the first component of the state. In the implementation, $e_1$ is evaluated into head normal form, i.e., its root is labeled by a constructor or a free variable. Then the control switches to the evaluation of $e_2$ until it is in head normal form, too.

*Switching* means that the active runtime stack becomes nonactive and the nonactive runtime stack becomes active. Furthermore, before switching is executed, relevant parts of the configuration of the machine must be saved. For this purpose, there is always a *switch-block* on top of the nonactive runtime stack; this switch-block contains the saved instruction pointer, saved environment pointer, and saved backtrack pointer. These pieces of information are needed for refreshing the configuration during the following switch.

## Indexing, Nondeterminism, and Backtracking

The *indexing* mechanism of the twin unification machine is the same as in the reduction machine. Note that it is slightly different from, e.g., the one which is used in the narrowing machine in [Loo93], where the different equations for the same function are checked sequentially starting from the first one, until a unifiable left hand side is found. In particular, for deterministic computations, this is inefficient, because it is clear from the beginning which alternative has to be chosen. This disadvantage is omitted in the twin unification machine by starting the code which is produced for a function definition, with a $JMR\ (\sigma_1 : m_1, \ldots, \sigma_r : m_r)$-instruction and by associating two different meanings to this instruction. In a deterministic computation, the $JMR\ (\sigma_1 : m_1, \ldots, \sigma_r : m_r)$-instruction jumps immediately to the code of the appropriate alternative, i.e., if the root of the current recursion argument is labeled by $\sigma_i$, then the machine jumps to address $m_i$ at which the code of the $i$-th right hand side starts.

In a nondeterministic computation, the $JMR$-instruction behaves differently. Recall that the only kind of *nondeterminism* occurs if a function call is computed and the root of its recursion argument is labeled by an unbound variable. In this case, an $F$-block has been pushed on the active runtime stack and its fourth component points to a $\langle VAR, ?\rangle$-node. Then the $JMR\ (\sigma_1 : m_1, \ldots, \sigma_r : m_r)$-instruction resembles the usual indexing scheme of the Warren abstract machine. It pushes a $C$-block and a $CR$-block to the active runtime stack and nonactive runtime stack, respectively, which contain the information that is needed for backtracking. Hence, there is always an $F$-block immediately below a C-block. Furthermore, the computation continues with the evaluation of the first alternative.

If the computation of the twin unification machine reaches an instantaneous description which corresponds to a nonsuccessful leaf in the ulo narrowing tree, then *backtracking* is executed by calling the *backtrack*-function (cf. Figure 20). This function restores the components of the machine corresponding to the pieces of information in the topmost $C$- and $CR$-blocks in the two runtime stacks. Depending on the current instantaneous description, we distinguish the following cases:

```
$backtrack : ID_U \to ID_U$
$backtrack\ (ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
            if $bp = 0$
            then $(0, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
            if $bp > 0$
            then if $st[1] = 1$
                   then if $rs1[bp..1] = C : sip : sbp : i : lop : lt : lds : e_1 : \ldots : e_{lds} : rs1^*$
                              and $rs2[(rs2.3)..1] = CR : sip2 : sep2 : sbp2 : rs2^*$
                          then $(sip, rs1', G', rs2', st', ss, tr', ep', bp, ds', op')$
                          where $rs1' = rs1[bp..1]$
                                  $G' = undo(G, tr[(lt+1)..length(tr)])$
                                  $rs2' = sip2 : sep2 : (rs2.3) : rs2[(rs2.3)..1]$
                                  $st' = (1, touch(n, \Lambda, lds))$
                                  $tr' = tr[1..lt]$
                                  $ep' = bp - 7 - lds$
                                  $ds' = e_1 : \ldots : e_{lds}$
                                  $op' = op[1..lop]$
                          if $rs1[bp..1] = CR : sip1 : sep1 : sbp1 : rs1^*$
                              and $rs2[(rs2.3)..1] = C : sip : sbp : i : lop : lt : lds : e_1 : \ldots : e_{lds} : rs2^*$
                          then $(sip, rs1', G', rs2', st', ss, tr', ep', bp', ds', op')$
                          where $rs1' = sip1 : sep1 : bp : rs1[bp..1]$
                                  $G' = undo(G, tr[(lt+1)..length(tr)])$
                                  $rs2' = rs2[(rs2.3)..1]$
                                  $st' = (2, touch(n, \Lambda, lds))$
                                  $tr' = tr[1..lt]$
                                  $ep' = (rs2.3) - 7 - lds$
                                  $bp' = rs2.3$
                                  $ds' = e_1 : \ldots : e_{lds}$
                                  $op' = op[1..lop]$
                   if $st[1] = 2$ then analogous
```

Figure 20: The function *backtrack*.

- If $bp = 0$, then there is no choice block on the active runtime stack. Hence, there is also no choice block on the other runtime stack, because $C$- and $CR$-blocks are created simultaneously. Thus, there is no alternative anymore, and the evaluation stops without success.

- If $bp > 0$ and the block $b$ of the active runtime stack which $bp$ points to, is a $C$-block, then there exists a $CR$-block $b'$ on the nonactive runtime stack which corresponds to $b$. The position of $b'$ is stored in the third square of the switch-block on top of the nonactive runtime stack. The function *backtrack* deletes every square of the active runtime stack above $b$ and every square of the nonactive runtime stack above $b'$. Note that there is no need for having stored an environment pointer in $b$, because there is always an $F$-block beneath $b$. Hence, the environment pointer is set to this $F$-block by decrementing the backtrack pointer by the length of $b$. Furthermore, the trail and the output pushdown are cut down to their old lengths $lt$ and $lop$, respectively, which are saved in $b$; the bindings saved right from position $lt$ in the trail, are dissolved by *undo*; the saved instruction pointer *sip* is loaded into the instruction pointer; the data stack is set to the elements stored in $b$; and the state is restored where its first component is not changed and the second component is yielded by the function *touch* (cf. Figure 22). Since the environment pointer and the instruction pointer of the nonactive runtime stack are changed, their values in the switch block must be modified with the pieces of information in the $CR$-block.

- If $bp > 0$ and the block $b$ of the active runtime stack which $bp$ points to, is a $CR$-block, then the nonactive runtime stack is responsible for the wrong choice; thus, an implicit switch must be executed by backtracking. For this purpose, a switch-block is pushed to the active runtime stack and the nonactive runtime stack becomes the active one. The definition of *backtrack* is symmetric to the previous case where the backtrack pointer and the saved backtrack pointer in the switch-block on the nonactive runtime stack exchange their roles.

## Unification and Occur Check

If $e_1$ as well as $e_2$ are in head normal form, i.e., if the labels $l_1$ and $l_2$ of the roots of $e_1$ and $e_2$, respectively, are both in $\Delta \cup FV$, then a decomposition step is implemented (cf. Definition 3.9 1. or 2.). The computation in this phase is almost directed by the output pushdown, the data stack, and the state. Note that the second component of the state indicates whether the machine is in the unification phase (i.e., $st[2] = u$), it is not unifying (i.e., $st[2] = n$), or it is at the end of a unification phase (i.e., $st[2] = e$).

In particular, if $l_i$ is a constructor, then it is written to the output pushdown if $i = 1$, or it is compared with the last symbol of the output pushdown if $i = 2$. This is realized by a $UNIFYCONSTR$-instruction which replaces the $WRITE$-instruction of the reduction machine. In a program of the twin unification machine, a $UNIFYCONSTR$-instruction is always followed by a $SWITCH$-instruction which, depending on the current state, executes a conditional switch.

In the other case, i.e., $l_i$ is a free variable $z_i$, the sequence

$$LOAD\ i;\ UNIFYVAR;\ SWITCH;$$

is executed, where $LOAD\ i$ pushes the graph address of $z_i$'s representation on the data stack, and the semantics of $UNIFYVAR$ and $SWITCH$ depend on $z_i$'s binding: If $z_i$ has been bound before, then its binding which is represented by the graph, is compared with the term which is produced on the other runtime stack. If $z_i$ has not been bound before, it will be bound to the term which is produced on the other runtime stack. According to the decomposition rules, this comparing or binding is performed node by node depth-first left-to-right. Clearly, since the two terms which have to be compared or bound, are only evaluated into head normal form, this comparison forces further evaluations of the subterms into head normal form. These evaluations are controlled by the data stack which pairwisely stores the addresses of the subterms, and the state which indicates the unification phase and which further indicates whether the *SWITCH*-instruction has to perform a switch to the other runtime stack, yes or no.

The transition from one state to the next one is computed by the auxiliary function *touch* (cf. Figure 22). It depends on the old state and the data stack. In Figure 21 all possible transitions between the states are summarized as a finite automaton, where we have dropped the transitions which stem from backtracking.

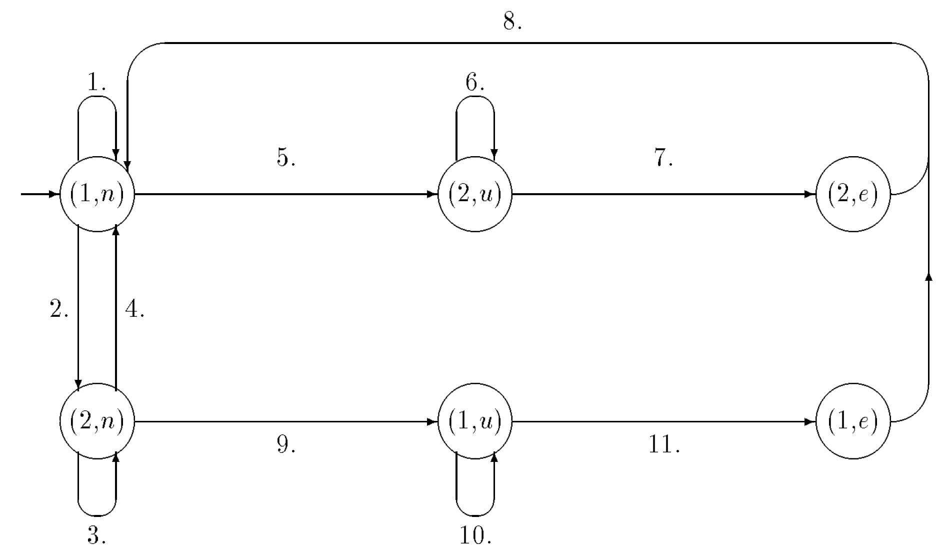


Figure 21: Finite automaton for state changes.

The automaton has the states $(1, n)$, $(1, u)$, $(2, n)$, $(2, u)$, $(1, e)$, and $(2, e)$. We have numbered each transition to explain in which case this transition is executed. The initial state is $(1, n)$, because we start with the left runtime stack and currently, no variable is unified.

1. The machine starts with the state $(1, n)$ (i.e., runtime stack $rs1$ is active and it is in nonunification mode) and it stays in this state until a *SWITCH*-instruction occurs.

2. If a *SWITCH*-instruction is executed and the data stack is empty, then runtime stack $rs2$ becomes active and the state of the machine is changed to $(2, n)$.

3. The computation continues on the runtime stack $rs2$ until a *SWITCH*-instruction is executed.

4. This transition is executed if a *SWITCH*-instruction is executed and the data stack is empty. Preceeding to the *SWITCH*-instruction, there is either a *UNIFYVAR*-instruction which has unified a variable with $\sigma$ where the rank of $\sigma$ is 0; or there is a

$UNIFYCONSTR$-instruction which has compared the topmost output pushdown symbol $\sigma$.

5. If a $SWITCH$-instruction is executed and the data stack is not empty, then the $SWITCH$-instruction is a part of the sequence $LOAD\ i; UNIFYVAR; SWITCH$. Then the machine switches from $rs1$ to $rs2$ and it changes to unification mode.
6. On the data stack, there are the graph addresses of the elements which have to be unified. The unification phase continues until the data stack is empty. Note that in this state $(2, u)$, a $SWITCH$-instruction has no effect.
7. If the data stack is emptied (this can happen either by a $UNIFYCONSTR$- or the $UNIFYVAR$-instruction), then the state is changed to $(2, e)$.
8. At the end of the unification phase, we have to ensure that runtime stack $rs1$ continues with the evaluation of the machine code according to the ulo narrowing relation. In this case, the $SWITCH$-instruction changes from both states $(2, e)$ and $(1, e)$ to $(1, n)$.

Transitions 9, 10, and 11 are dual to transitions 5, 6, and 7, respectively.

Note that Figure 21 is not symmetric, because the initial state has to be restored after a unification phase by transition 8.

During the unification of an expression $e$ and a free variable $z$, the auxiliary, boolean function *check* performs the occur check on the $(\Delta \cup FV)$-skeleton of $e$ (cf. Definition 3.8). This check is realized by comparing the graph addresses of $z$ and the addresses of the $(\Delta \cup FV)$-skeleton of $e$ which are stored on the data stack. For further explanations confer the explanations of the unification instructions in the following subsection.

## 5.3 Machine Instructions of the Twin Unification Machine

In this subsection we introduce the instructions of the twin unification machine and their semantics. The semantics of an instruction *inst* is a function

$$\mathcal{C}_U\,[\![\ inst\ ]\!]\ : ID_U \to ID_U.$$

### Auxiliary Functions

In the definition of the semantics of the twin unification machine's instructions, we need the following auxiliary functions which are defined in Figure 22.

- For a runtime stack $rs$, the function *env* yields the $F$-block which includes the current environment.
- For a runtime stack $rs$, the function *next* yields the $F$-block which includes the environment of the current environment.

```
$env : RS \to \mathbb{N}$
$env(rs) =$   $\textbf{if } rs.1 = F \textbf{ then } 1$
              $\textbf{if } rs.1 = Y \textbf{ then } 4 + rs.4$
$next : RS \to \mathbb{N}$
$next(rs) =$  $\textbf{if } rs.1 = F \text{ and } rs.(3 + rs.3) = F \textbf{ then } 3 + rs.3$
              $\textbf{if } rs.1 = F \text{ and } rs.(3 + rs.3) = Y \textbf{ then } 6 + rs.3 + rs.(6 + rs.3)$
              $\textbf{if } rs.1 = Y \text{ and } rs.(4 + rs.4) = F \textbf{ then } 4 + rs.4 + next(rs[(4 + rs.4)..1]) - 1$
$sel : Adr \times \mathbb{N} \times [Adr \to GNode] \to Adr$
$sel(adr, j, G) = \textbf{if } G(adr) = \langle CON, \sigma, a_1, \ldots, a_k\rangle \text{ and } j \in [k] \textbf{ then } a_j$
$new : [Adr \to GNodes] \times \mathbb{N} \to \mathcal{P}(Adr)$
$new(G, n) = \textbf{if } n = 0 \textbf{ then } \emptyset$
             $\textbf{if } n > 0 \text{ and } adr_1, \ldots, adr_n \text{ are the minimal addresses such that,}$
                      $\text{for every } i \in [n],\ G(adr_i) \text{ is not defined } \textbf{then } \{adr_1, \ldots, adr_n\}$
$undo : [Adr \to GNodes] \times Adr^* \to [Adr \to GNodes]$
$undo(G, list) = \textbf{if } list = \Lambda \textbf{ then } G$
                 $\textbf{if } list = adr : rest \textbf{ then } undo(G[adr/\langle VAR, ?\rangle], rest)$
$deref : Adr \times [Adr \to GNodes] \to Adr$
$deref(adr, G) = \textbf{if } G(adr) = \langle CON, \sigma, a_1, \ldots, a_k\rangle \textbf{ then } adr$
                 $\textbf{if } G(adr) = \langle VAR, ?\rangle \textbf{ then } adr$
                 $\textbf{if } G(adr) = \langle VAR, adr'\rangle \textbf{ then } deref(adr', G)$
$touch : \{n, u\} \times Adr^* \times \mathbb{N} \to \{n, u, e\}$
$touch(x, list, k) = \textbf{if } x = n \text{ and } list = \Lambda \text{ and } k = 0 \textbf{ then } n$
                     $\textbf{if } x = u \text{ and } list = \Lambda \text{ and } k = 0 \textbf{ then } e$
                     $\text{otherwise } u$
$check : Adr \times Adr \times [Adr \to GNodes] \to \{true, false\}$
$check(a_1, a_2, G) = \textbf{if } a_1 = a_2 \textbf{ then } true$
                      $\textbf{if } a_1 \neq a_2 \textbf{ then}$
                           $\textbf{if } G(a_2) = \langle CON, \sigma, b_1, \ldots, b_k\rangle \textbf{ then } false \vee \bigvee_{i \in [k]} check(a_1, b_i, G)$
                           $\textbf{if } G(a_2) = \langle VAR, b_1\rangle \textbf{ then } check(a_1, b_1, G)$
                           $\textbf{if } G(a_2) = \langle VAR, ?\rangle \textbf{ then } false$
```

Figure 22: Auxiliary functions.

- For a graph address $adr$, a natural number $j$, and a graph $G$, the function $sel$ yields the graph address of the $j$-th son of $G(adr)$ if it exists.
- For a graph $G$ and a natural number $n$, the function $new$ yields $n$ new graph addresses, i.e., addresses for which $G$ is not yet defined.
- For a graph $G$ and a list of graph addresses $list$, the function $undo$ yields the graph which results from overwriting every node in $G$ the address of which occurs in $list$, by $\langle VAR, ?\rangle$. This function is used in the definition of the function $backtrack$ where $list$ is always instantiated by a part of the trail.
- For a graph address $adr$ and a graph $G$, the function $deref$ yields the address which points to the root of the binding of $G(adr)$.

- For a tag $x \in \{n, u\}$, a list *list*, and a natural number $k$, the function *touch* yields the new unification state. In the definition of the instruction semantics, *list* is always instantiated by the data stack.
- The boolean function *check* realizes the occur check in the twin unification machine. It checks whether the variable which is represented in the graph at address $a_1$, occurs in the $(\Delta \cup FV)$-skeleton of the term the root of which is addressed by $a_2$.

**Jump and Switch Instructions (cf. Figure 23)**

- $JMP\ m$ only sets the instruction pointer on $m$.
- As in the reduction machine the indexing mechanism is realized by the $JMR$-instruction. Furthermore, after backtracking, the $JMR$-instruction initiates the computation of the next alternative. The $JMR$-instruction is only executed in two situations: either after an $F$-block has been created on the active runtime stack or after backtracking has been performed, i.e., the topmost block is a $C$-block. In the definition of the semantics of the $JMR$-instruction, there are the following four cases:

  1. If the topmost block of the active runtime stack is an $F$-block and the root of its recursion argument is labeled by constructor $\sigma_i$, then $JMR\ (\sigma_1 : m_1, \ldots, \sigma_r : m_r)$ sets the instruction pointer on $m_i$. Recall that, in the reduction machine, $JMR$ has only this semantics.
  2. If the topmost block of the active runtime stack is an $F$-block and its recursion argument is an unbound variable, then a $C$-block is pushed to the active runtime stack, a $CR$-block is pushed to the nonactive runtime stack, and the instantaneous description is prepared for evaluating the first alternative. Since it is possible that a choice point is created during the unification phase, there can be elements on the data stack which have to be saved in the $C$-block, because we have to be able to restore the instantaneous description after backtracking is performed.
  3. If the topmost block of the active runtime stack is a $C$-block and the entry $k$ of this $C$-block which indicates the alternative that has been computed at the time being, is smaller than the number $r$ of alternatives, then $JMR$ initiates the instantaneous description for the computation of alternative $k + 1$.
  4. If $k \geq r$, then the computation of the last alternative is finished and backtracking is started.

- Depending on the second component of the state, the $SWITCH$-instruction executes a conditional switch from the active runtime stack to the nonactive one. In the unification phase (i.e., if $st[2] = u$) no switch is performed. If a switch is performed, then a switch-block is pushed to the active runtime stack, the former switch-block is deleted on the other runtime stack, and the other runtime stack becomes the active one. We distinguish four cases:

1. $st = (1, n)$, i.e., we are either not in the unification phase, or we are at the beginning of a unification phase in the sense that the first two instructions of the sequence $LOAD\ i; UNIFYVAR; SWITCH$ have been executed already. In the first case (i.e., the machine is not in the unification phase which is indicated by an empty data stack) a switch is executed in order to cause $rs2$ to evaluate an output symbol which is compared with the topmost symbol of the output pushdown. In the second case (i.e., the machine is at the beginning of the unification phase which is indicated by a nonempty data stack) the component $st[2]$ has to be changed to $u$ (which is taken care of by the function *touch*); moreover, a switch is executed in order to unify a variable corresponding to the address on the data stack with the term which is evaluated by $rs2$.
2. $st = (2, n)$: the instruction semantics is defined analogous to the previous case and $rs1$ becomes the active runtime stack.
3. $st[2] = u$, i.e., the machine is unifying a variable. A switch is not executed. Only the instruction pointer is incremented.
4. $st[2] = e$, i.e., the unification phase is finished. We have to ensure that $rs1$ continues the evaluation. Hence, the machine switches only if $rs2$ is active. The component $st[2]$ is set to $n$.

## Runtime Stack Instructions (cf. Figure 24)

The runtime stack instructions have an influence only on the active runtime stack. Furthermore, in the definition of the instruction semantics, we must distinguish whether the environment pointer *ep* points to the topmost block, yes or no. In the former case, the instruction semantics are almost the same as in the reduction machine. In the latter case, the part of the runtime stack from the bottom to the square *ep* points to, is important for the definition of the instruction semantics. Since the dynamic link of an $F$-block serves as saved environment pointer and not any longer as link to the bottom of the $F$-block, a parameter $j$ is added to the $RET$-instruction which indicates how many squares have to be deleted from the runtime stack. In the case that $RET\ j$ deletes a $Y$-block, $j$ is equal 0.

- $CREATE\ (ra, j, m_1, \ldots, m_n)$ creates a new $F$-block on top of the active runtime stack almost in the same way as it is done in the reduction machine. The only difference is that the dynamic link serves as saved environment pointer.

- $EVAL\ i$ sets the instruction pointer to the $i$-th parameter address in the block which contains the current environment. Furthermore, it pushes a $Y$-block to the active runtime stack which contains the saved environment pointer as additional component.

- $RET\ j$ sets the instruction pointer to the return address of the block $b$ the environment pointer points to, and it sets the environment pointer to the saved environment pointer of $b$. Furthermore, $b$ is deleted only if it is the topmost block, i.e., $ep > bp$.

## Unification Instructions (cf. Figures 25 and 26)

- In the twin unification machine, the $WRITE\ \sigma$-instruction of the reduction machine is renamed to $UNIFYCONSTR\ (\sigma, k)$, where $k$ is the rank of $\sigma$, because it is used for unification, writing, and comparing. Its instruction semantics depends on the current state, where we distinguish the following cases:

  1. $st = (1, n)$, i.e., the first runtime stack is active and we are not in the unification phase. Then $\sigma$ is written on top of the output pushdown and the instruction pointer is incremented.
  2. $st = (2, n)$, then we check whether the symbol on top of the output pushdown is equal $\sigma$ or not. In the former case, only the instruction pointer is incremented, whereas in the latter case, backtracking is initiated.
  3. $st[2] = u$, i.e., a term the root of which is labeled by $\sigma$, has to be unified with the term represented by the graph address $gadr^*$ which is retrieved from the top of the data stack. If the graph node at $gadr$ is labeled by $\langle VAR, ?\rangle$, then the corresponding free variable $z_i$ is bound to $\sigma(z_j, \ldots, z_k)$, where $z_j, \ldots, z_k$ are new variables, and the unification phase continues. If the graph node at $gadr^*$ is labeled by $\sigma$, then the subterms have to be unified. For this purpose, the graph addresses of the subterms are pushed on the data stack and the unification phase continues. If the graph node at $gadr^*$ is labeled by a constructor which is different from $\sigma$, then backtracking is initiated.

  As mentioned before, the $UNIFYCONSTR\ (\sigma, k)$-instruction is always followed by a $SWITCH$-instruction which switches from the active runtime stack to the nonactive runtime stack in cases 1 and 2. Note that the case $st[2] = e$ does not occur.

- In the translation of a macro tree transducer and two sd-expressions, the $LOAD\ i$-instruction is always followed by a $UNIFYVAR$-instruction. Hence, it would be possible to glue $LOAD\ i$ and $UNIFYVAR$ together and to deal with one instruction $UNIFYVAR\ i$, instead. But, since the semantics of this instruction would be to complex, we prefer to have two instructions.

  1. If the machine is not in the unification phase, then $LOAD\ i$ prepares the machine to unify the variable $z_i$ with constructors or other variables evaluated by the other runtime stack. For this purpose, it pushes the graph address of $z_i$ on top of the data stack, where a new graph address is chosen if $z_i$ was not used before, i.e., $ss[i] = ?$.
  2. Otherwise, the topmost square of the data stack includes the graph address which should be unified with $z_i$. $LOAD\ i$ pushes the graph address of $z_i$ on top of the data stack together with the number 1 which indicates that there is <u>one</u> pair of terms which shall be unified; the terms are represented by the two topmost addresses on the data stack. Furthermore, $LOAD\ i$ writes $z_i$ to the output pushdown.

- A *UNIFYVAR*-instruction always follows a *LOAD* $i$-instruction which has stored at least the graph address of $z_i$ on top of the data stack. The instruction semantics of *UNIFYVAR* is defined recursively in the sense that in some cases of its definition, the instruction pointer is not changed, i.e., the same instruction is executed again in the next computation step. The definition of the instruction semantics depends on the current state. We distinguish the following cases:

  1. If the machine is not in the unification phase and $rs1$ is active, i.e., $st = (1, n)$, then only the instruction pointer is incremented which forces a switch to $rs2$ by the *SWITCH*-instruction in the next computation step.
  2. If the machine is not in the unification phase and $rs2$ is active, i.e., $st = (2, n)$, then $rs1$ has already written an output symbol $\sigma$ on top of the output pushdown; the variable which we are unifying at the moment, must be bound to $\sigma$. We compare $\sigma$ with the label of the graph node the address $adr^*$ of which can be retrieved from the top element of the data stack. If $G(adr^*)$ is an unbound *VAR*-node, then it is replaced by a $\sigma$-node and on the data stack, $adr^*$ is replaced by the addresses of the sons of the $\sigma$-node. If $G(adr^*)$ is a $\sigma$-node, then only $adr^*$ is replaced by the addresses of the sons of the $\sigma$-node on the data stack. In the last possible case, i.e., $G(adr^*)$ is a $\delta$-node and $\delta \neq \sigma$, backtracking is initiated.
  3. If the machine is in the unification phase, i.e., $st[2] = u$, then the topmost symbol on the data stack is a natural number $j$ which indicates the number of pairs on the data stack which shall be compared and unified.
     If $j > 0$, then there is at least one pair of addresses on the data stack the dereferenced addresses of which are $adr_1^*$ and $adr_2^*$. In this case the instruction pointer is not incremented until $j = 0$, i.e., the unification is finished. If $adr_1^* = adr_2^*$, then they are popped from the data stack and $j$ is decremented. Otherwise, we distinguish the following cases:
     (a) One of the two graph addresses $adr_i^*$ points to a *VAR*-node, the other one $adr_j^*$ points to a $\sigma$-node, and the occur check fails. In this case, the *VAR*-node is bound to the $\sigma$-node, the two addresses are popped from the data stack, and $j$ is decremented.
     (b) Both graph addresses point to a $\sigma$-node. Then the subgraphs must be compared. For this purpose, the two addresses are popped from the data stack, the addresses of the subgraphs are pushed pairwise on the data stack, and the number on top of the data stack becomes $j-1+k$, where $k$ denotes the rank of $\sigma$.
     (c) Both graph addresses point to a *VAR*-node. Then, the two *VAR*-nodes are bound to a new unbound *VAR*-node, the two addresses are popped from the data stack, and $j$ is decremented (cf. Definition 3.9 2.(a)).
     (d) If the occur check succeeds in Case (a), or if the comparison fails in Case (b), then backtracking is initiated.

     If $j = 0$, then the comparison phase on the data stack is finished. Hence, $j$ is popped from the data stack and the instruction pointer is incremented to continue the evaluation.

## Graph Instructions and Initialization (cf. Figure 27)

In the reduction machine, the trees for the representations of the recursion arguments are created by $NODE(\sigma, n)$-instructions and they are connected by a $NODE\#$-instruction which also initializes the other components of an instantaneous description.

In the twin unification machine, the $NODE(\sigma, n)$-instruction has the same semantics. The $NODE\#$-instruction only connects the representations of the recursion arguments. The initialization is realized by the $INIT\ ca$-instruction. Furthermore, there exists one more graph instruction $VAR\ i$ which creates the representation of the free variable $z_i$.

- By the $NODE\#$-instruction the graph addresses of all $n$ recursion arguments are connected to a graph the root of which is labeled by $\langle CON, \#, a_1, \ldots, a_n\rangle$.
- The $INIT\ ca$-instruction creates the initial F-block with return address 0 and the pointer to the #-node on both runtime stacks. Furthermore, it creates the initial switch-block on $rs2$ with the start address $ca$ of the second sd-expression, the saved environment pointer 4, and the saved backtrack pointer 0. The state is set to $(1, n)$. The environment pointer and the backtrack pointer are set to 4 and 0, respectively, and the data stack becomes empty.
- $VAR\ i$ creates a new node $\langle VAR, ?\rangle$ at a graph address $gadr$ for the representation of $z_i$, if $ss[i] =?$, i.e., this is the leftmost occurrence of $z_i$ in a recursion argument of the two sd-expressions which should be unified. Furthermore, $gadr$ is pushed to the data stack.

## State Transitions

The transitions of the machine are determined by the code that is generated for the rewrite rules of the macro tree transducer $M$ and two sd-expressions $e_1$ and $e_2$. The machine execution starts with the instantaneous description

$$(1, \Lambda, G_\emptyset, \Lambda, (1, n), \underbrace{? : \ldots :?}_{k\ times}, \Lambda, 0, 0, \Lambda, \Lambda)$$

where $G_\emptyset$ and $k$ are assumed to be the empty graph and the number of variables occurring in $e_1$ and $e_2$, respectively.

The transition rule

$$(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) \vdash \mathcal{C}_U [\![ \, ps(ip) \, ]\!] \, (ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$$

is applied until one of the following conditions is true.

- $ip = 0$ and $ep = 0$ and $st = (2, n)$ (unifiable):

  The second runtime stack is the active one and $ep = 0$, i.e., the $RET$ 0-instruction after the translation of $e_2$ has been executed. This indicates that the evaluation has been successful. The result of the computation is the label $(e, \varphi)$ of the leftmost successful leaf in the ulo narrowing tree, where the output pushdown includes $e \in T\langle\Delta\rangle(FV)$ and the $(E_M, \Delta)$-unifier $\varphi$ is represented by the stack of substitutions and the graph.

- $ip = 0$ and ($ep \neq 0$ or $st \neq (2, n)$) (not unifiable):

  If the second condition in the conjunction holds, then $ip$ has been set to 0 by the *backtrack*-function, because there is no possible alternative. Hence, the traversal through the ulo narrowing tree is finished without finding a solution, i.e., $e_1$ and $e_2$ are not unifiable.

- $ps$ is not defined for $ip$ (syntax error):

  This case only occurs for programs which are not the result of the translation of a macro tree transducer and two sd-expressions.

As mentioned before, there may occur infinite computations, because there may be infinite branches in the ulo narrowing tree which are left to the leftmost solution.

```
$\mathcal{C}_U$ $[\![$ $JMP\ m$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    $(m, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$

$\mathcal{C}_U$ $[\![$ $JMR\ (\sigma_1 : m_1, \ldots, \sigma_r : m_r)$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[1] = 1$
    **then if** $rs1 = F : ra : dl : gadr : rs1^*$ and $gadr^* = deref(gadr, G)$
            **then if** $G(gadr^*) = \langle CONS, \sigma_i, a_1, \ldots, a_n \rangle$
                    **then** $(m_i, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
                    **if** $G(gadr^*) = \langle VAR, ? \rangle$ and $ds = e_1 : \ldots : e_m$ and $rs2 = ip2 : ep2 : bp2 : rs2^*$
                    **then** $(m_1, rs1', G', rs2', st, ss, tr : gadr^*, ep, ep + 7 + m, ds, op)$
                    where $rs1' = C : ip : bp : 1 : length(op) : length(tr) : m : e_1 : \ldots : e_m : rs1$
                            $G' = G[gadr^*/\langle CON, \sigma_1, a_1, \ldots, a_n \rangle, a_1/\langle VAR, ? \rangle, \ldots, a_n/\langle VAR, ? \rangle]$
                                where $\{a_1, \ldots, a_n\} = new(G, n)$
                            $rs2' = ip2 : ep2 : max(ep2, bp2) + 4 : CR : rs2$
            **if** $rs1 = C : sip : sbp : k : lop : lt : lds : e_1 : \ldots : e_{lds} : rs1^*$
                and $gadr = rs1[ep - 3]$ and $gadr^* = deref(gadr, G)$
            **then if** $k < r$
                    **then** $(m_{k+1}, rs1', G', rs2, st, ss, tr : gadr^*, ep, bp, ds, op)$
                    where $rs1' = C : sip : sbp : k + 1 : lop : lt : lds : e_1 : \ldots : e_{lds} : rs1^*$
                            $G' = G[gadr^*/\langle CON, \sigma_{k+1}, a_1, \ldots, a_n \rangle, a_1/\langle VAR, ? \rangle, \ldots, a_n/\langle VAR, ? \rangle]$
                                where $\{a_1, \ldots, a_n\} = new(G, n)$
                    **if** $k \geq r$
                    **then**  **if** $rs2 = ip2 : ep2 : bp2 : CR : rs2^*$
                            **then** $backtrack(ip, rs1^*, G, rs2^*, st, ss, tr, ep, sbp, ds, op)$
    **if** $st[1] = 2$ **then** analogous

$\mathcal{C}_U$ $[\![$ $SWITCH$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[2] = n$
    **then if** $st[1] = 1$
            **then if** $rs2 = sip : sep : sbp : rs2^*$
                    **then** $(sip, rs1', G, rs2^*, (2, touch(n, ds, 0)), ss, tr, sep, sbp, ds, op)$
                    where $rs1' = ip + 1 : ep : bp : rs1$
            **if** $st[1] = 2$
            **then if** $rs1 = sip : sep : sbp : rs1^*$
                    **then** $(sip, rs1^*, G, rs2', (1, touch(n, ds, 0)), ss, tr, sep, sbp, ds, op)$
                    where $rs2' = ip + 1 : ep : bp : rs2$
    **if** $st[2] = u$
    **then** $(ip + 1, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
    **if** $st[2] = e$
    **then if** $st[1] = 1$
            **then** $(ip + 1, rs1, G, rs2, (1, n), ss, tr, ep, bp, ds, op)$
    **then if** $st[1] = 2$
            **then if** $rs1 = sip : sep : sbp : rs1^*$
                    **then** $(sip, rs1^*, G, rs2', (1, n), ss, tr, sep, sbp, ds, op)$
                    where $rs2' = ip + 1 : ep : bp : rs2$
```

Figure 23: Jump and switch instructions of the twin unification machine.

```
$\mathcal{C}_U$ $[\![$ $CREATE\ (ra, j, m_1, \ldots, m_n)$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[1] = 1$
    **then if** $bp > ep$
        **then** $(ip + 1, rs1', G, rs2, st, ss, tr, bp + n + 4, bp, ds, op)$
        where $rs1' = F : ra : bp - ep + (n + 2) : gadr : m_1 : \cdots : m_n : rs1$
              $gadr = sel(rs1.(env(rs1[ep..1]) + 3 + bp - ep), j, G)$
        **if** $bp < ep$
        **then** $(ip + 1, rs1', G, rs2, st, ss, tr, ep + n + 4, bp, ds, op)$
        where $rs1' = F : ra : (n + 2) : gadr : m_1 : \cdots : m_n : rs1$
              $gadr = sel(rs1.(env(rs1) + 3), j, G)$
    **if** $st[1] = 2$ **then** analogous

$\mathcal{C}_U$ $[\![$ $EVAL\ i$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[1] = 1$
    **then if** $bp > ep$
        **then** $(ip', rs1', G, rs2, st, ss, tr, bp + 4, bp, ds, op)$
        where $ip' = rs1.(bp - ep + env(rs1[ep..1]) + 3 + i)$
              $rs1' = Y : ip + 1 : ep : next(rs1[ep..1]) + bp - ep : rs1$
        **if** $bp < ep$
        **then** $(ip', rs1', G, rs2, st, ss, tr, ep + 4, bp, ds, op)$
        where $ip' = rs1.(env(rs1) + 3 + i)$
              $rs1' = Y : ip + 1 : ep : next(rs1) : rs1$
    **if** $st[1] = 2$ **then** analogous

$\mathcal{C}_U$ $[\![$ $RET\ j$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[1] = 1$
    **then if** $bp > ep$
        **then if** $rs1[ep] = F$
            **then** $(rs1.(bp - ep + 2), rs1, G, rs2, st, ss, tr, ep', bp, ds, op)$
            where $ep' = ep - 2 - rs1.(bp - ep + 3)$
            **if** $rs1[ep] = Y$
            **then** $(rs1.(bp - ep + 2), rs1, G, rs2, st, ss, tr, rs1.(bp - ep + 3), bp, ds, op)$
        **if** $bp < ep$
        **then if** $rs1 = F : ra : dl : e_1 : \ldots : e_{j+1} : rs1^*$
            **then** $(ra, rs1^*, G, rs2, st, ss, tr, ep - 2 - dl, bp, ds, op)$
            **if** $rs1 = Y : ra : sep : sl : rs1^*$
            **then** $(ra, rs1^*, G, rs2, st, ss, tr, sep, bp, ds, op)$
    **if** $st[1] = 2$ **then** analogous
```

Figure 24: Runtime stack instructions of the twin unification machine.

```
$\mathcal{C}_U$ $[\![$ $UNIFYCONSTR$ $(\sigma, k)$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[2] = n$
    **then if** $st[1] = 1$
        **then** $(ip + 1, rs1, G, rs2, st, ss, tr, ep, bp, ds, op\sigma)$
        **if** $st[1] = 2$
        **then if** $op = op^*\sigma$
            **then** $(ip + 1, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
            **if** $op = op^*\delta$ and $\delta \neq \sigma$
            **then** $backtrack(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
    **if** $st[2] = u$
    **then if** $ds = gadr : ds^*$ and $gadr^* = deref(gadr, G)$
        **then if** $G(gadr^*) = \langle VAR, ?\rangle$
            **then** $(ip + 1, rs1, G', rs2, st', ss, tr : gadr^*, ep, bp, ds', op\sigma)$
            where $G' = G[gadr^*/\langle CON, \sigma, a_1, \ldots, a_k\rangle, a_1/\langle VAR, ?\rangle, \ldots, a_k/\langle VAR, ?\rangle]$
                      where $\{a_1, \ldots, a_k\} = new(G, k)$
                  $st' = (st[1], touch(u, ds^*, k))$
                  $ds' = a_1 : \ldots : a_k : ds^*$
            **if** $G(gadr^*) = \langle CON, \sigma, a_1, \ldots, a_k\rangle$
            **then** $(ip + 1, rs1, G, rs2, st', ss, tr, ep, bp, ds', op\sigma)$
            where $st' = (st[1], touch(u, ds^*, k))$
                  $ds' = a_1 : \ldots : a_k : ds^*$
            **if** $G(gadr^*) = \langle CON, \delta, a_1, \ldots, a_l\rangle$ and $\sigma \neq \delta$
            **then** $backtrack(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$

$\mathcal{C}_U$ $[\![$ $LOAD\ i$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $st[2] = n$
    **then if** $ss[i] = ?$
        **then** $(ip + 1, rs1, G', rs2, st, ss[i/gadr], tr, ep, bp, gadr : ds, op)$
        where $\{gadr\} = new(G, 1)$
              $G' = G[gadr/\langle VAR, ?\rangle]$
        **if** $ss[i] = gadr$
        **then** $(ip + 1, rs1, G, rs2, st, ss, tr, ep, bp, deref(gadr, G) : ds, op)$
    **if** $st[2] = u$ and $ds = gadr_1 : ds^*$
    **then if** $ss[i] = ?$
        **then** $(ip + 1, rs1, G', rs2, st, ss[i/gadr_2], tr, ep, bp, 1 : gadr_2 : ds, opz_i)$
        where $\{gadr_2\} = new(G, 1)$
              $G' = G[gadr_2/\langle VAR, ?\rangle]$
        **if** $ss[i] = gadr$
        **then** $(ip + 1, rs1, G, rs2, st, ss, tr, ep, bp, 1 : deref(gadr, G) : ds, opz_i)$
```

Figure 25: Unification instructions of the twin unification machine.

```
$\mathcal{C}_U$ $[\![ \; UNIFYVAR \; ]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    if $st[2] = n$
    then if $st[1] = 1$
        then $(ip+1, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
        if $st[1] = 2$ and $ds = adr : ds^*$ and $adr^* = deref(adr, G)$ and $op = op^*\sigma$
        then if $G(adr^*) = \langle VAR, ? \rangle$
            then $(ip+1, rs1, G', rs2, st, ss, tr : adr^*, ep, bp, ds', op)$
            where $G' = G[adr^*/\langle CON, \sigma, a_1, \ldots, a_k \rangle, a_1/\langle VAR, ? \rangle, \ldots, a_k/\langle VAR, ? \rangle]$
                    where $\{a_1, \ldots, a_k\} = new(G, k)$
                  $ds' = a_1 : \ldots : a_k : ds^*$
            if $G(adr^*) = \langle CON, \sigma, a_1, \ldots, a_k \rangle$
            then $(ip+1, rs1, G, rs2, st, ss, tr, ep, bp, ds', op)$
            where $ds' = a_1 : \ldots : a_k : ds^*$
            if $G(adr^*) = \langle CON, \delta, a_1, \ldots, a_k \rangle$ and $\delta \neq \sigma$
            then $backtrack(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
    if $st[2] = u$ and $ds = j : ds^*$
    then if $j > 0$ and $ds = j : adr_1 : adr_2 : \widetilde{ds}$ and for every $i \in [2] : adr_i^* = deref(adr_i, G)$
        then if $adr_1^* = adr_2^*$
            then $(ip, rs1, G, rs2, st, ss, tr, ep, bp, j-1 : \widetilde{ds}, op)$
            if $adr_1^* \neq adr_2^*$
            then if $G(adr_1^*) = \langle VAR, ? \rangle$ and $G(adr_2^*) = \langle CON, \sigma, a_1, \ldots, a_k \rangle$
                    and $check(adr_1^*, adr_2^*, G) = false$
                then $(ip, rs1, G', rs2, st, ss, tr : adr_1^*, ep, bp, j-1 : \widetilde{ds}, op)$
                where $G' = G[adr_1^*/\langle VAR, adr_2^* \rangle]$
                if $G(adr_1^*) = \langle CON, \sigma, a_1, \ldots, a_k \rangle$ and $G(adr_2^*) = \langle VAR, ? \rangle$
                    and $check(adr_2^*, adr_1^*, G) = false$
                then $(ip, rs1, G', rs2, st, ss, tr : adr_2^*, ep, bp, j-1 : \widetilde{ds}, op)$
                where $G' = G[adr_2^*/\langle VAR, adr_1^* \rangle]$
                if $G(adr_1^*) = \langle CON, \sigma, a_1, \ldots, a_k \rangle$ and $G(adr_2^*) = \langle CON, \sigma, b_1, \ldots, b_k \rangle$
                then $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds', op)$
                where $ds' = j - 1 + k : a_1 : b_1 : \ldots : a_k : b_k : \widetilde{ds}$
                if $G(adr_1^*) = \langle VAR, ? \rangle$ and $G(adr_2^*) = \langle VAR, ? \rangle$
                then $(ip, rs1, G', rs2, st, ss, tr : adr_1^* : adr_2^*, ep, bp, j-1 : \widetilde{ds}, op)$
                where $G' = G[adr_1^*/\langle VAR, nadr \rangle, adr_2^*/\langle VAR, nadr \rangle]$
                        where $\{nadr\} = new(G, 1)$
                otherwise $backtrack(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op)$
        if $j = 0$
        then $(ip+1, rs1, G, rs2, (st[1], touch(u, ds^*, 0)), ss, tr, ep, bp, ds^*, op)$
```

Figure 26: UNIFYVAR-instruction of the twin unification machine.

```
$\mathcal{C}_U$ $[\![$ $NODE(\sigma, n)$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $ds = a_n : \cdots : a_1 : ds'$
    **then** $(ip+1, rs1, G[gadr/\langle CON, \sigma, a_1, \ldots, a_n\rangle], rs2, st, ss, tr, ep, bp, gadr : ds', op)$
    where $\{gadr\} = new(G, 1)$
$\mathcal{C}_U$ $[\![$ $NODE\#$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $ds = a_n : \cdots : a_1$
    **then** $(ip+1, rs1, G[gadr/\langle CON, \#, a_1, \ldots, a_n\rangle], rs2, st, ss, tr, ep, bp, gadr, op)$
    where $\{gadr\} = new(G, 1)$
$\mathcal{C}_U$ $[\![$ $INIT\ ca$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $ds = gadr$
    **then** $(ip+1, F : 0 : 2 : gadr, G, ca : 4 : 0 : F : 0 : 2 : gadr, (1, n), ss, tr, 4, 0, \Lambda, op)$
$\mathcal{C}_U$ $[\![$ $VAR\ i$ $]\!]$ $(ip, rs1, G, rs2, st, ss, tr, ep, bp, ds, op) =$
    **if** $ss[i] =?$
    **then** $(ip+1, rs1, G[gadr/\langle VAR, ?\rangle], rs2, st, ss[i/gadr], tr, ep, bp, gadr : ds, op)$
    where $\{gadr\} = new(G, 1)$
    **if** $ss[i] = gadr$
    **then** $(ip+1, rs1, G, rs2, st, ss, tr, ep, bp, gadr : ds, op)$
```

Figure 27: Graph instructions and initialization of the twin unification machine.

## 5.4 Compilation of Rewrite Rules and SD-Expressions

In this subsection we present the compilation of the rewrite rules of the macro tree transducer $M = (F, \Delta, R)$ together with two sd-expressions $e_1$ and $e_2$, into code of the twin unification machine. The compilation is very close to the compilation into code of the reduction machine (cf. Subsection 4.3). Hence, we only give explanations of the differences to the compilation in Subsection 4.3; however, in the figures we show the complete translation.

In the description of the compilation schemes, we use, for every $i \in \mathbb{N}$, the following metavariables: $r_i \in RHS(F, \Delta)$; $e, e_i \in sdExp(F, \Delta, FV)$; $t, t_i \in T\langle\Delta\rangle(FV)$; and $\alpha, \alpha.i$ are tree-structured addresses.

The compilation scheme *trans* (cf. Figure 28) results from the one in Figure 12 by adding to every $RET$-instruction a parameter which is the number of parameter variables of the corresponding translated function. Furthermore, the gsd-expression $e$ is replaced by the two sd-expressions $e_1$ and $e_2$.

```
trans(R, e_1, e_2) =
              JMP r + 1;
      1 :     functrans({f_1(σ_j(x_1, ..., x_rank(σ_j)), y_1, ..., y_n_1) → r_j1 | j ∈ [ρ]}, 1) RET n_1;
              ⋮
      r :     functrans({f_r(σ_j(x_1, ..., x_rank(σ_j)), y_1, ..., y_n_r) → r_jr | j ∈ [ρ]}, r) RET n_r;
      r + 1 : goaltrans(e_1, e_2, r + 1)
```

Figure 28: Compilation scheme *trans.*

The compilation scheme *functrans* (cf. Figure 29) does not differ from the one in Figure 13.

```
functrans({f_i(σ_j(x_1, ..., x_rank(σ_j)), y_1, ..., y_n_i) → r_ji | j ∈ [ρ]}, i) =
                  JMR(σ_1 : i.1, ..., σ_ρ : i.ρ);
      i.1 :       rhstrans(r_1i, i.1)  JMP i.(ρ + 1);
                  ⋮
      i.(ρ − 1) : rhstrans(r_(ρ−1)i, i.(ρ − 1))  JMP i.(ρ + 1);
      i.ρ :       rhstrans(r_ρi, i.ρ)
      i.(ρ + 1) :
```

Figure 29: Compilation scheme *functrans.*

The compilation scheme *rhstrans* (cf. Figure 30) results from the one in Figure 14 by replacing every $WRITE\ \sigma_j$-instruction by a $UNIFYCONSTR\ (\sigma_j, rank(\sigma_j))$-instruction which is followed by a $SWITCH$-instruction. Furthermore, the parameter 0 is added to every $RET$-instruction.

For two sd-expressions $e_1$ and $e_2$, the function *goaltrans* (cf. Figure 32) constructs the graphs of recursion arguments in $e_1$ and $e_2$, it connects them by the $NODE\#$-instruction, and it produces the initial configuration by the $INIT$-instruction. Furthermore, *goaltrans*

$$\begin{array}{lcll}
rhstrans(\sigma_j(r_1,\dots,r_n),\alpha) & = & \multicolumn{2}{l}{UNIFYCONSTR\ (\sigma_j, rank(\sigma_j)); SWITCH;} \\
 & & \multicolumn{2}{l}{rhstrans(r_1,\alpha.1)\dots rhstrans(r_n,\alpha.n)} \\
rhstrans(y_i,\alpha) & = & \multicolumn{2}{l}{EVAL\ i;} \\
rhstrans(f_i(x_j,r_1,\dots,r_n),\alpha) & = & & CREATE(\alpha.(n+1), j, \alpha.1,\dots,\alpha.n); \\
 & & & JMP\ i; \\
 & & \alpha.1: & rhstrans(r_1,\alpha.1)\ RET\ 0; \\
 & & & \vdots \\
 & & \alpha.n: & rhstrans(r_n,\alpha.n)\ RET\ 0; \\
 & & \alpha.(n+1): &
\end{array}$$

Figure 30: Compilation scheme *rhstrans.*

translates $e_1$ and $e_2$ into code of the twin unification machine by the function *exptrans*. It works in a similar way as the translation scheme in Figure 15 and it uses the function *count* the modification of which is described in Figure 31.

$$\begin{array}{l}
count : sdExp(F,\Delta,FV) \rightarrow \mathbb{N} \\
count(e) = \textbf{if } e = \sigma_j(e_1,\dots,e_n) \textbf{ then } \sum_{i=1}^{n} count(e_i) \\
\qquad\qquad\ \ \textbf{if } e = f_i(t, e_1\dots,e_n) \textbf{ then } 1 + \sum_{i=1}^{n} count(e_i) \\
\qquad\qquad\ \ \textbf{if } e = z_i \textbf{ then } 0
\end{array}$$

Figure 31: Auxiliary function *count.*

$$\begin{array}{lrl}
goaltrans(e_1,e_2,\alpha) = & & makegraph(e_1)\ makegraph(e_2) \\
 & & NODE\#;\ INIT\ \alpha.2; \\
 & \alpha.1: & exptrans(e_1, 1, \alpha.1)\ SWITCH;\ RET\ 0; \\
 & \alpha.2: & exptrans(e_2, count(e_1)+1, \alpha.2)\ RET\ 0;
\end{array}$$

Figure 32: Compilation scheme *goaltrans.*

The compilation scheme *maketree* in Figure 16 is replaced by the compilation scheme *makegraph* (cf. Figure 33) which is defined in the same way, but there occurs one additional case, i.e., the occurrence of a free variable $z_i$. In this case, *makegraph* does not produce any code, whereas *makenodes* produces a $VAR\ i$-instruction.

The compilation scheme *exptrans* (cf. Figure 34) results from the one in Figure 18 by replacing every $WRITE\ \sigma_i$-instruction by a $UNIFYCONSTR\ (\sigma_i, rank(\sigma_i))$-instruction which is followed by a $SWITCH$-instruction. Furthermore, the parameter 0 is added to every $RET$-instruction. The occurrence of a variable $z_i$ is translated into the code sequence $LOAD\ i; UNIFYVAR; SWITCH$.

In Figure 35 the translation of the rules in $R_1$ (cf. Figure 5) and the two sd-expressions $e_1 = sh(z_1,\alpha)$ and $e_2 = mi(\sigma(z_2,\alpha))$ is shown, where we abbreviate $UNIFYCONSTR$ by $UNIFYCON$. The left column includes the $JMP$ 25-instruction to the translation of $e_1$ and the translation of the rules for $sh$. The column in the middle includes the translation of the rules for $mi$ and finally, the right column includes the translation of $e_1$ and $e_2$.

$$
\begin{array}{ll}
makegraph(\sigma_j(e_1,\dots,e_n)) & = makegraph(e_1)\dots makegraph(e_n) \\
makegraph(f_i(t,e_1,\dots,e_n)) & = makenodes(t)\ makegraph(e_1)\dots makegraph(e_n) \\
makegraph(z_i) & = \Lambda \\
\hline
makenodes(\sigma_j(t_1,\dots,t_n)) & = makenodes(t_1)\dots makenodes(t_n)\ NODE(\sigma_j, rank(\sigma_j)); \\
makenodes(z_i) & = VAR\ i;
\end{array}
$$

Figure 33: Tree construction schemes *makegraph* and *makenodes.*

$$
\begin{array}{llll}
exptrans(\sigma_i(e_1,\dots,e_n),j,\alpha) & = & & UNIFYCONSTR\ (\sigma_i, rank(\sigma_i)); SWITCH; \\
 & & & exptrans(e_1, j, \alpha.1) \\
 & & & \vdots \\
 & & & exptrans(e_n, j+\sum_{k=1}^{n-1} count(e_k), \alpha.n) \\
exptrans(f_i(t,e_1,\dots,e_n),j,\alpha) & = & & CREATE(\alpha.(n+1), j, \alpha.1,\dots,\alpha.n); \\
 & & & JMP\ i; \\
 & \alpha.1: & & exptrans(e_1, j+1, \alpha.1)\ RET\ 0; \\
 & & & \vdots \\
 & \alpha.n: & & exptrans(e_n, j+1+\sum_{k=1}^{n-1} count(e_k), \alpha.n)\ RET\ 0; \\
 & \alpha.(n+1): & & \\
exptrans(z_i,j,\alpha) & = & & LOAD\ i;\ UNIFYVAR;\ SWITCH;
\end{array}
$$

Figure 34: Compilation scheme *exptrans.*

The computation of this program describes the implementation of a depth-first left-to-right traversal over the ulo narrowing tree in Figure 7.

```
 1 : $JMP$ 25;
 2 : $JMR(\alpha : 3, \sigma : 5)$;
 3 : $EVAL$ 1;
 4 : $JMP$ 13;
 5 : $CREATE(13, 1, 7)$;
 6 : $JMP$ 2;
 7 : $UNIFYCON$ $(\sigma, 2)$;
 8 : $SWITCH$;
 9 : $CREATE(11, 2)$;
10 : $JMP$ 14;
11 : $EVAL$ 1;
12 : $RET$ 0;
13 : $RET$ 1;

14 : $JMR(\alpha : 14, \sigma : 18)$;
15 : $UNIFYCON$ $(\alpha, 0)$;
16 : $SWITCH$;
17 : $JMP$ 24;
18 : $UNIFYCON$ $(\sigma, 2)$;
19 : $SWITCH$;
20 : $CREATE(22, 2)$;
21 : $JMP$ 14;
22 : $CREATE(24, 1)$;
23 : $JMP$ 14;
24 : $RET$ 0;

25 : $VAR$ 1;
26 : $VAR$ 2;
27 : $NODE$ $(\alpha, 0)$;
28 : $NODE$ $(\sigma, 2)$;
29 : $NODE$ #;
30 : $INIT$ 38;
31 : $CREATE(36, 1, 33)$;
32 : $JMP$ 2;
33 : $UNIFYCON$ $(\alpha, 0)$;
34 : $SWITCH$;
35 : $RET$ 0;
36 : $SWITCH$;
37 : $RET$ 0;
38 : $CREATE(40, 2)$;
39 : $JMP$ 14;
40 : $RET$ 0;
```

Figure 35: Compilation of $R_1$ and the sd-expressions $sh(z_1, \alpha)$ and $mi(\sigma(z_2, \alpha))$.

## 6 Comparison with the BABEL System

In order to give an impression of the efficiency of the twin unification machine, we compare its implementation with the implementation of the BABEL system [Loo93, Win94]; both implementations are written in the programming language C and they run on a SPARC-station SLC. We have chosen the BABEL system for a comparison, because we intend to incorporate the implementation of our deterministic unification algorithm for macro tree transducers into the implementation of BABEL (cf. Section 7).

In the BABEL system the equality is defined by predefined rules which are similar to the decomposition rules in the present paper. For the evaluation of function calls, the user can decide whether he uses an eager narrowing strategy or a lazy narrowing strategy. However, the predefined rules for the equality are handled differently; the BABEL system evaluates an equation $t = s$ as follows:

- It derives $t$ by the chosen narrowing strategy until it is in normal form, i.e., an element of $T\langle\Delta\rangle(FV)$.
- It derives $s$ by the chosen narrowing strategy until it is in normal form, i.e., an element of $T\langle\Delta\rangle(FV)$.
- It tries to unify the two normal forms by applying the rules for the equality.

Since the BABEL system does not apply decomposition rules during the evaluation of $t$ and $s$, it is possible that it does not find solutions which are computed by the twin unification machine. Consider, e.g., the following two pairs of sd-expressions:

- $t = sh(\sigma(\alpha, \alpha), \alpha)$ and $s = mi(\sigma(z_2, \sigma(z_1, \alpha)))$
  Here, $t$ and $s$ are not $E_{M_1}$-unifiable. This result is yielded by the twin unification machine, whereas the BABEL system behaves as follows: In the eager evaluation mode and in the lazy evaluation mode, the BABEL system does not terminate, because it evaluates both terms in normal forms which are not unifiable and there do exist infinitely many such normal forms.
- $t = mi(\sigma(z_1, z_2))$ and $s = sh(\sigma(\sigma(\alpha, \alpha), \alpha), \alpha)$
  The substitution $\varphi = [z_1/\sigma(\alpha, \alpha), z_2/\alpha]$ is the only $E_{M_1}$-unifier of $t$ and $s$. The twin unification machine exactly yields this substitution and stops with the answer that there does not exist any other solution. The BABEL system behaves as follows: In the eager evaluation mode and in the lazy evaluation mode, the BABEL system yields the substitution $\varphi$; and it does not terminate if the user asks for other solutions because of the same reasons as in the previous example.

Hence, with respect to the terminology which we have introduced in the discussion about Figure 4 in Subsection 2.4, our deterministic unification algorithm for macro tree transducers is better than the deterministic $E$-unification algorithm which is inherent in the implementation of the BABEL system.

Besides these differences with respect to the behaviours of the implementations, there are also differences with respect to the runtimes in the case where both implementations behave in the same way. In Figure 37 we document the runtimes of the BABEL system for every of the two possible narrowing strategies and the runtimes of the twin unification machine for the unification of the following four pairs of sd-expressions:

1. $t_1 = sh(\sigma(\sigma(\alpha,\alpha),\sigma(\alpha,\alpha)), mi(\sigma(\sigma(\sigma(\alpha,\alpha),\alpha),\alpha)))$ and $s_1 = z_1$:

   Actually, this $E_{M_1}$-unification is a reduction in disguise. It can simply be performed by the BABEL system by evaluating $t_1$. Depending on the structure of the goals of the twin unification machine, it starts with the goal $equ(t_1, s_1)$, where $s_1$ is a variable $z_1$. Surely, in the twin unification machine, there is an overhead when it is merely used as a reduction machine, i.e., switching after every evaluation into head normal form of $t_1$ and binding of $z_1$, which is not performed in the BABEL system. Nevertheless, the overhead is extremly low which can be shown by comparing the runtimes in Figure 37.

2. $t_2 = sh(\sigma(\sigma(\alpha,\alpha),\sigma(\alpha,\alpha)), mi(\sigma(\sigma(\sigma(\alpha,\alpha),\alpha),\alpha)))$ and $s_2 = \sigma(z_2, \sigma(z_1, \sigma(\alpha, \sigma(z_2, z_1))))$:

   This is an example where $t_2$ is a gsd-expression and $s_2$ is an sd-expression without function calls. Hence, the derivation is deterministic (i.e., there is never a choice point on a runtime stack) and decomposition steps are applied. The good performance of the twin unification machine is due to the used indexing scheme (cf. the paragraph "Jump and Switch Instructions" in Subsection 5.3).

3. $t_3 = mi(z_1)$ and $s_3 = sh(z_1, \alpha)$:

   This is an example for which there exist infinitely many solutions. The runtimes in Figure 37 are the runtimes for finding the first solution $[z_1/\alpha]$.

4. $t_4 = exp(\gamma(\gamma(\gamma(\gamma(\gamma(\gamma(\gamma(\gamma(\alpha)))))))), \alpha)$ and $s_4 = z_1$:

   The rewrite rules of the set of rewrite rules $R_{exp}$ of the macro tree transducer $M_{exp} = (\{exp^{(2)}\}, \{\gamma^{(1)}, \alpha^{(0)}\}, R_{exp})$ which computes an exponential function are shown in Figure 36.

$$\begin{array}{lcl} exp(\alpha, y_1) & \rightarrow & \gamma(y_1) \\ exp(\gamma(x_1), y_1) & \rightarrow & exp(x_1, exp(x_1, y_1)) \end{array}$$

Figure 36: Rewrite rules of the macro tree transducer $M_{exp}$.

This is an example where the BABEL system with eager narrowing strategy is faster than the twin unification machine. Since there are a lot of function calls in parameter positions which must be evaluated all, it is clear that an implementation of an innermost strategy which evaluates function calls in parameter positions more efficiently, is faster than an implementation of an outermost strategy.

| unifiable terms | **BABEL** eager narrowing | lazy narrowing | **Twin Unification Machine** |
|---|---|---|---|
| $(t_1, s_1)$ | 0.14 sec | 0.30 sec | $\leq$ 0.01 sec |
| $(t_2, s_2)$ | 0.15 sec | 0.31 sec | $\leq$ 0.01 sec |
| $(t_3, s_3)$ | 0.18 sec | 0.29 sec | $\leq$ 0.01 sec |
| $(t_4, s_4)$ | 0.11 sec | 2.62 sec | 0.45 sec |

Figure 37: Runtimes of the BABEL system and the twin unification machine.

## 7 Conclusion and Related Work

In this paper we have presented an efficient implementation of the deterministic unification algorithm for macro tree transducers. This implementation is an extension of the implementation of the leftmost outermost reduction relation for macro tree transducers in [GFV91]. The deterministic unification algorithm for macro tree transducers is induced by a combination of leftmost outermost narrowing and interleaving decomposition rules. The main point of the presented implementation is an efficient implementation of the decomposition rules.

Up to now, the only existing implementation of ulo narrowing has been presented in [Faß93]. This implementation is an extension of the implementation of leftmost outermost reduction on a checking-tree nested-stack transducer in [FV92a]. But, since this implementation is nondeterministic, it is not very interesting from the more practical point of view.

In existing approaches of the implementation of narrowing, decomposition rules are not implemented explicitely. In [Han90] the consistency check of constructors is realized by the *rejection rule* which is applied during *innermost basic narrowing.* In BABEL [MR92] decomposition rules are predefined rules which are added to every BABEL program. In implementations of BABEL [KLMR90, MKLR90, Loo93] decomposition rules are implemented like rules of the implemented BABEL program. Hence, neither the occur check, nor the binding mode is implemented. Clearly, this omission can lead to infinite computations. The only check which is realized in these implementations, is the local consistency check. But, the combination of this check with an eager narrowing strategy is not very effectful, because the local consistency check is only applied when a possibly infinite narrowing derivation is finished. Thus, the local consistency check interleaves only narrowing derivations in implementations of lazy narrowing if the decomposition rules are implemented in the same way as program rules. But, this has not been done in the implementation of the BABEL system [Win94]. Furthermore, in [HLW92, LLR93, MKM$^{+}$93] it has been pointed out how difficult it is to implement lazy narrowing in the case of a full functional logic programming language (in opposite to the implementation of ulo narrowing for macro tree transducers in the present paper). This problem is also be confirmed by the comparison in the previous section.

In our further research, we will integrate the presented implementation in the implementation of particular BABEL programs, where the guards have the form $e_1 = e_2$ for two sd-expressions $e_1$ and $e_2$, and the functions which occur in guards, are described by macro tree transducers. We hope that this integration will increase the efficiency of the implementation of BABEL. Furthermore, we will investigate the extension of the presented implementation for the class of primitive recursive tree functions.

### Acknowledgement

The authors would like to thank Markus Schauler who developed the implementation in C on a SPARCstation SLC, and Rita Loogen and Stefan Winkler who helped us to understand the implementation of BABEL.

## References

[CF82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I, II. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

[DJ91] N. Dershowitz and J.P. Jouannaud. Notations for rewriting. *Bulletin of the EATCS*, 43:162–172, 1991.

[Ech88] R. Echahed. On completeness of narrowing strategies. In *CAAP'88*, pages 89–101. Springer-Verlag, 1988. LNCS 299.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory; perspectives and open problems*. New York, Academic Press, 1980.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–145, 1985.

[EV86] J. Engelfriet and H. Vogler. Pushdown machines for the macro tree transducer. *Theoretical Computer Science*, 42:251–368, 1986.

[EV91] J. Engelfriet and H. Vogler. Modular tree transducers. *Theoretical Computer Science*, 78:267–304, 1991.

[Faß93] H. Faßbender. Implementation of a universal unification algorithm for macro tree transducers. In *FCT'93*, pages 222–233. Springer-Verlag, 1993. LNCS 710.

[Fay79] M. Fay. First-order unification in an equational theory. In *Proceeding of the 4th workshop on automated deduction, Austin*, pages 161–167, 1979.

[FHVV93] Z. Fülöp, F. Herrmann, S. Vagvölgyi, and H. Vogler. Tree transducers with external functions. *Theoretical Computer Science*, 108:185–236, 1993.

[Fri85] L. Fribourg. SLOG : A logic programming language interpreter based on clausual superposition and rewriting. In *Proceedings of the IEEE International Symposium on logic programming*, pages 172–184. IEEE Computer Society Press, 1985.

[FV92a] H. Faßbender and H. Vogler. An implementation of syntax directed functional programming on nested-stack machines. *Formal Aspects of Computing*, 4:341–375, 1992.

[FV92b] H. Faßbender and H. Vogler. A universal unification algorithm based on unification-driven leftmost outermost narrowing. Technical Report 92-07, University of Ulm, Fakultät für Informatik, D-89069 Ulm, Germany, 1992. *to appear in Acta Cybernetica'94*.

[GFV91] K. Gladitz, H. Faßbender, and H. Vogler. Compiler-based implementation of syntax directed functional programming. Technical Report 91-10, Aachen University of Technology, Fachgruppe Informatik, Ahornstr. 55, D-52056 Aachen, Germany, 1991.

[Han90] M. Hanus. Compiling logic programs with equality. In *PLILP'90*, pages 387–401. Springer-Verlag, 1990. LNCS 456.

[HLW92] W. Hans, R. Loogen, and S. Winkler. On the interaction of lazy evaluation and backtracking. In *PLILP'92*, pages 355–369. Springer-Verlag, 1992. LNCS 631.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*. Academic Press, New York, 1980.

[Hue80] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27:797–821, 1980.

[Hul80] J.M. Hullot. Canonical forms and unification. In *Proceedings of the 5th conference on automated deduction*, pages 318–334. Springer-Verlag, 1980. LNCS 87.

[Hup78] U. Hupach. Rekursive Funktionen in mehrsortigen Algebren. *Elektron. Informationsverarb. Kybernetik*, 15:491–506, 1978.

[Kla84] H.A. Klaeren. A constructive method for abstract algebraic specification. *Theoretical Computer Science*, 30:139–204, 1984.

[KLMR90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodriguez-Artalejo. Graph-based implementation of a functional logic language. In *ESOP'90*, pages 271–290. Springer-Verlag, 1990. LNCS 432.

[Lan75] D.S. Lankford. Canonical inference. Technical Report ATP-32, Department of Mathematics and Computer Science, University of Texas at Austin, 1975.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second, extended edition.

[LLR93] R. Loogen, F. Lopez-Fraguas, and M. Rodriguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *PLILP'93*, pages 184–200, 1993. LNCS 714.

[Loo93] R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.

[Mat70] Y. Matiyasevich. Diophantine representation of recursively enumerable predicates. In *Proceedings of the Second Scandinavian Logic Symposium*. North-Holland, 1970.

[MH92] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing. In *ALP'92*, pages 244–258. Springer-Verlag, 1992. LNCS 632.

[MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodriguez-Artalejo. Lazy narrowing in a graph machine. In *ALP'90*, pages 298–317. Springer-Verlag, 1990. LNCS 463.

[MKM+93] J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. In *PLILP'93*, pages 167–183, 1993. LNCS 714.

[MR92] J.J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic-programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[Pad87] P. Padawitz. Strategy-controlled reduction and narrowing. In *RTA'87*, pages 242–255. Springer-Verlag, 1987. LNCS 256.

[Pét57] R. Péter. *Rekursive Funktionen.* Akademiai Kiado, Budapest, 1957. English translation: Academic Press, New York, 1967.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 20:23–41, 1965.

[Sie89] J.H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

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

[Win94] S. Winkler. The BABEL system. Personal Communication, 1994.

[You88] J.H. You. Solving equations in an equational language. In *ALP'88*, pages 245–254. Springer-Verlag, 1988. LNCS 343.

[You89] J.H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–341, 1989.