Compiler-based Implementation of Syntax-Directed Functional Programming

Katia Gladitz
Lehrstuhl für Informatik II, RWTH Aachen
Ahornstraße 55, W–5100 Aachen, Germany

Heinz Faßbender[*] and Heiko Vogler
Abt. Theoretische Informatik, Universität Ulm
Oberer Eselsberg, W-7900 Ulm, Germany

Ulmer Informatik-Berichte
Nr. 91 – 02

Abstract

We consider particular functional programs in which on the one hand the recursion is restricted to syntax-directed recursion and on the other hand simultaneous recursion and nesting of function calls in parameter positions of other functions is allowed. For such programs called syntax-directed functional programs, we formalize a compiler-based implementation of the call-by-name computation strategy. The machine involved in this implementation, called syntax-directed runtime-stack machine, is minimal in the sense that it computes exactly the class sdFun of functions which are expressible by syntax-directed functional programs. We verify this minimality property by showing a one-to-one correspondence between the implementation presented in this paper, and an interpreter-based implementation of syntax-directed functional programs on checking-tree nested-stack transducers. It is known from the literature that such transducers characterize in a formal sense the class sdFun.

[*]The work of this author has been supported by the Deutsche Forschungsgemeinschaft (DFG).

1 Introduction

In many situations it is appropriate, to describe the semantics of strings that are generated by some context-free grammar, in a syntax-directed way [Iro61, AU71, AU73, Knu68] (for short: sd way). Then, the meaning of a string is expressed in terms of the meanings of its substrings. For example, the translation of a high-level language program into machine code is often described in a syntax-directed way [Ind79]. Then, the translation of a program construct, e.g. an if_then_else statement, is described in terms of the translation of its constituents, i.e. the condition and the two alternatives of the if_then_else statement.

Up to now there exists various formalizations of the concept of sd translation:

- generalized sd translation schemes [AU73]
- attribute grammars [Knu68]
- denotational semantics [SS71, Gor79]
- affix grammars [Kos71]
- total deterministic macro tree-to-string transducers [CF82, Eng81, EV86]
- context-free hypergraph grammar based syntax-directed translation schemes [EH89]

In this paper we start from total deterministic macro tree-to-string transducers (for short: ymt) that are particular, left-linear, confluent, and noetherian term rewriting systems, and embed them into the context of functional programming. For this purpose we enrich the syntax of ymt by adding programming language like syntactic features. More precisely, all rewrite rules that specify the computation of one particular function, are collected into one equation, and the pattern matching inherent in term rewriting systems, is replaced by the usual case-construct. In this paper we call the result of this syntactic enrichment a *syntax-directed functional program* (for short: sd funprog).

Before showing an example of an sd funprog we explain informally the shape of such programs. An sd funprog is a finite list of function definitions over an input-output base where such a base consists of a ranked alphabet of input symbols and a usual alphabet of output symbols. The first argument z of a function is called its *recursion argument,* the other arguments are called *parameters.* Every function definition has the form of an equation; the right hand side of an equation is a case-construct which checks the label $\underline{root}(z)$ of the root of the recursion argument and switches to the appropriate case-alternative. For every input symbol there is exactly one case-alternative. A case-alternative is a string over function calls, parameters that appear on the left hand side of the equation, and output symbols. In particular, function calls may occur nested in parameter positions of other functions. Here, we only consider call-by-name computations of sd funprogs.

Figure 1 shows a piece of the concrete sd funprog $P_{\underline{trans}}$. Actually, it is the syntax-directed definition of the function $\underline{exptrans}$ which translates abstract syntax trees of high-level programs into code for some abstract machine. The one and only parameter of $\underline{exptrans}$ keeps a tree-structured address which has not yet been used in the code generated so far. Clearly, the input symbols of the programs are the labels of the abstract syntax trees (like the symbol "if_then_else" of rank three), and the output symbols are machine instructions, addresses, and interpunctuation symbols. In the figure we have only presented one case-alternative which treats recursion arguments of the form of if_then_else-statements. For the complete definition of $\underline{exptrans}$, we would have to add exactly one case-alternative for every other command of the programming laguage (like

```
$\underline{exptrans}\ (z, y_1)$ = CASE $\underline{root}(z)$ OF
                  ⋮
          if_then_else :          $\underline{exptrans}\ (\underline{sel}_1(z), y_1.1)$
                                  $JMC\ y_1.0;$
                                  $\underline{exptrans}\ (\underline{sel}_2(z), y_1.2)$
                                  $JMP\ y_1.4;$
                   $y_1.0$ :      $\underline{exptrans}\ (\underline{sel}_3(z), y_1.3)$
                   $y_1.4$ :
                  ⋮
          END
```

Figure 1: Part of an sd funprog for the translation of if_then_else into abstract machine code.

"while_do", ":=", "repeat_until"). The case-alternative shown in the figure contains three function calls, viz. $\underline{exptrans}\ (\underline{sel}_1(z), y_1.1)$, $\underline{exptrans}\ (\underline{sel}_2(z), y_1.2)$, and $\underline{exptrans}\ (\underline{sel}_3(z), y_1.3)$. In fact, these function calls occur independently from each other and not nestedly.

Up to now only an interpreter-based implementation of sd funprogs exists. The involved abstract machine called *checking-tree nested-stack transducer* (for short: *ct-ns transducer*) is a sequential machine with a nested-stack based memory called *checking-tree nested-stack* [Aho69, ES84]. As implementation devices, these machines are of particular interest, because in [EV86] it is shown in an indirect way that the class sdFun of *tree-to-string functions* which are computable by sd funprogs with call-by-name computation strategy, is characterized by ct-ns transducers. Thus, in particular, every function computable by a ct-ns transducer lies in sdFun. In [FV90] the interpreter-based implementation suggested in [EV86] has been formalized in a direct way. There, the complete case-alternative is written into one square of the storage of the ct-ns transducer. During runtime the case-alternative is interpreted symbol by symbol from left to right.

In this paper we formalize a compiler-oriented implementation of sd funprogs, i.e. we define an abstract machine, called *syntax-directed runtime-stack machine* (for short *sdrs machine*), and a translation function which maps every sd funprog into a program which is executable on the sdrs machine. This implementation is a restriction of the compiler-based implementation of first-order functional programs with call-by-name computation strategy in [Ind85] with respect to the special recursion structure of sd funprog. In the formalization of the sdrs machine we aim at a minimal set of instructions which is appropriate for the execution of sd funprog. Actually, we have achieved that functional programs which allow the specification of tree-to-string functions outside sdFun are not computable on the sdrs machine. Rather than proving this characterization formally we convince the reader about the validity of the characterization by comparing the sdrs machine with the ct-ns transducer.

The sdrs machine consists of a program storage PS, a program counter PC, an output tape OT, and a runtime stack RS. An instantaneous description of an sdrs machine is given in Figure 2. In fact, the main component of the sdrs machine is the runtime stack. It consists of F-blocks for the evaluation of function calls and Y-blocks for the evaluation of the parameter values. In Figure 2

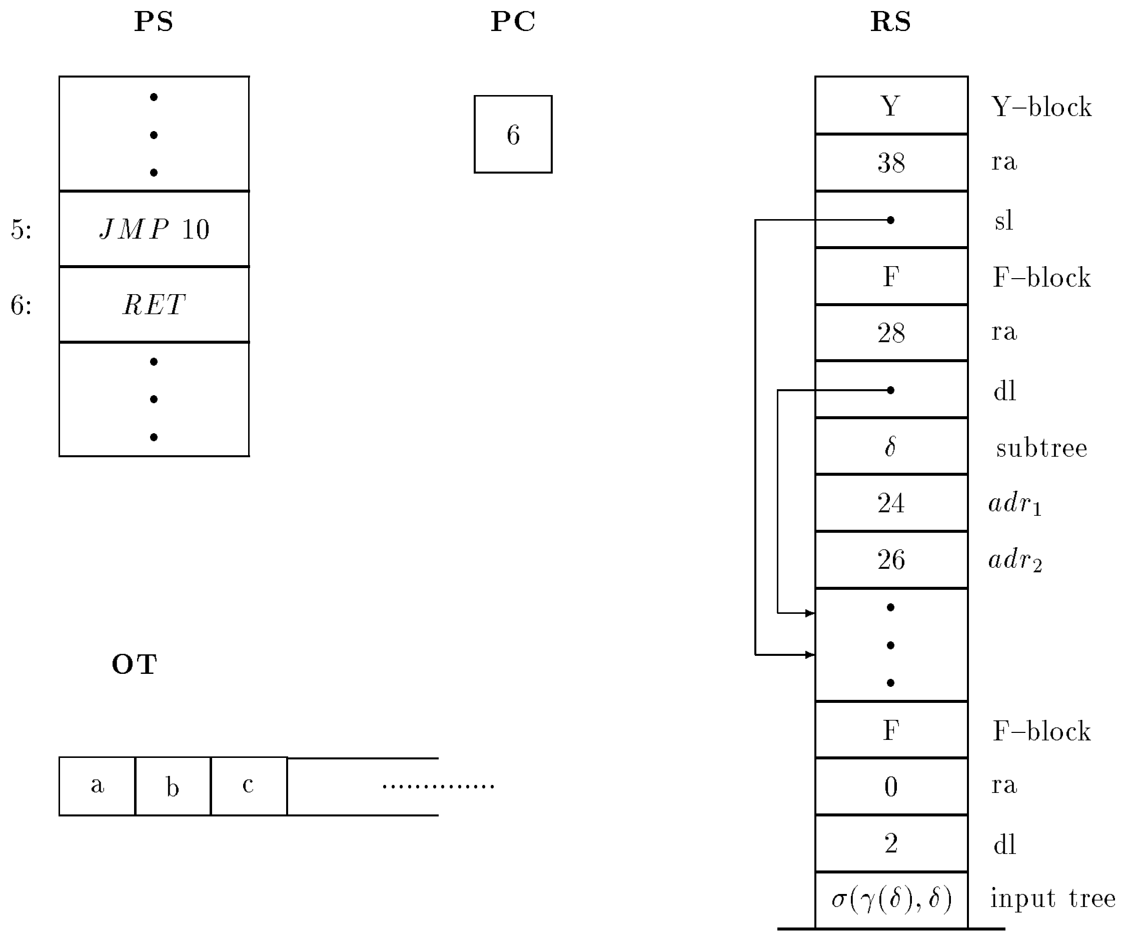


Figure 2: An instantaneous description of an sdrs machine

we have indicated three blocks: one Y-block on top; one F-block one position below the top, and one F-block at the bottom of the runtime stack. The block at the bottom represents the call of the main function F_{in} of the sd funprog at the beginning of the computation. We assume that F_{in} has just one argument (its recursion argument); in Figure 2 this argument is the tree $\sigma(\gamma(\delta), \delta)$.

The computation of a function call is illustrated in Figure 3. Suppose that

- "$G(\underline{sel}_2(z), par_1, \ldots, par_n)rest$" is a postfix of a case-alternative,
- the F-block bl is on top of RS, and
- the machine is about to simulate the function call of G with recursion argument $\underline{sel}_2(z)$ and parameter expressions $par_1, \ldots, par_n$.

Let $adr(piece)$ be the program address of the first instruction in the translation of some piece of the source program into sdrs machine code. Then, a new F-block bl' is pushed on top of the runtime stack by executing the instruction $CREATE(2, adr(par_1), \ldots, adr(par_n); adr(rest))$. The block bl' contains

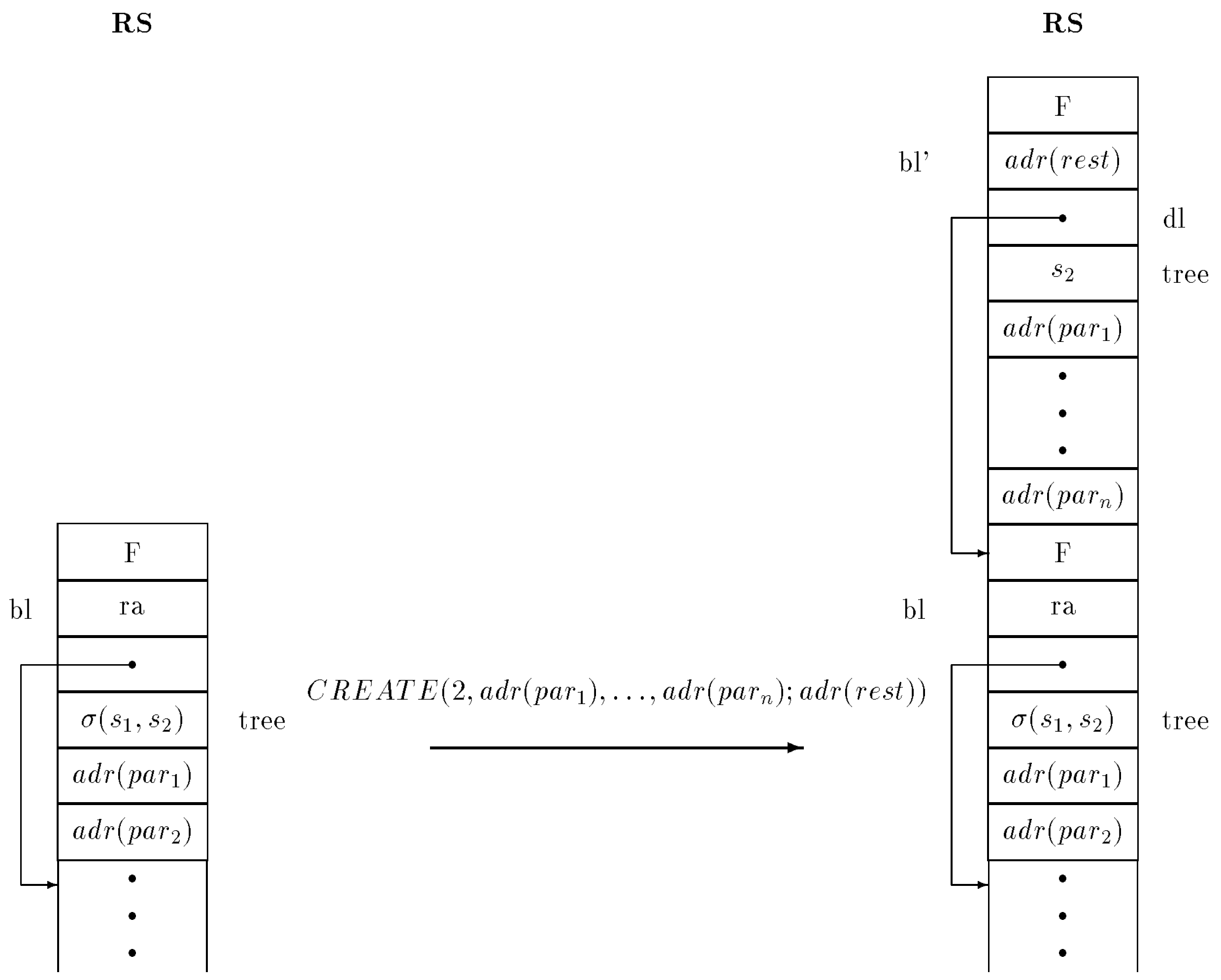


Figure 3: Computation of the function call $G(\underline{sel}_2(z), par_1, \ldots, par_n)$

- the *tag* F
- the *return address* $adr(rest)$ at which the computation has to be resumed after the evaluation of the called function.
- a *dynamic link* dl which points to the topmost square of block bl
- the current *recursion argument* s_2 which is selected from the recursion argument of block bl
- the *addresses* $adr(par_1), \ldots, adr(par_n)$ *for the evaluation of the parameters* .

After the execution of the $CREATE$-instruction the program counter is set to the address for the evaluation of G by executing the instruction $JMP\ adr(G)$.

Besides the F-blocks, the runtime stack contains Y-blocks. Such a Y-block is pushed on top of the runtime stack, if a value of a parameter expression is needed. (Recall that the sdrs machine simulates the call-by-name computation strategy of sd funprogs.) A Y-block always consists of three squares:

- the *tag* Y
- the *return address* ra
- the *static link* sl which points to the F-block which contains the current environment, i.e., the bindings of the free variables of the parameter expressions. In the following we call this F-block and its tree the current block and current recursion argument, respectively.

The way in which F-blocks are pushed on top of the runtime stack and, in particular, the fact that the recursion argument of the called function always is a subtree of the current recursion argument, already indicates that the sdrs machine can only evaluate functions in the class sdFun. We are not going to prove this equivalence formally. Rather we compare this implementation with the interpreter-based implementation in [FV90] and we will show a one-to-one correspondence between the two implementations.

We have implemented the sdrs machine directly in C on a SPARC station SLC. On the other hand sd funprogs can be considered as very simple MIRANDA programs for which a commercial implementation is available. Not very strikingly, it turns out that our implementation runs faster than the MIRANDA-like implementation.

This paper is organized in seven sections where the second section contains preliminaries. In Section 3 we introduce the syntax and semantics of sd funprog. In Section 4, we formally define the sdrs machine. Section 5 deals with the call-by-name implementation of sd funprog on the sdrs machine. In Section 6 we recall the interpreter-based implementation of sd funprog on the ct-ns transducer and compare this implementation with the implementation of Section 5. Finally, Section 7 contains some concluding remarks and indicates further research topics.

2 Preliminaries

We recall some notations and basic definitions which will be used in the rest of the paper.

We denote the set of natural numbers by $\mathbb{N}$; it includes the number 0. For $j \in \mathbb{N}$, $[j]$ denotes the set $\{1, \ldots, j\}$; thus $[0] = \emptyset$.

For an alphabet A, we denote the set of word over A with length n by A^n. The set of *all words over* A is the set $\bigcup_{n \in \mathbb{N}} A^n$ and is denoted by A^*; ε denotes the empty word. As usual, a *ranked alphabet* is a pair $(\Sigma, \underline{rank}_\Sigma)$, where Σ is an alphabet and $\underline{rank}_\Sigma : \Sigma \longrightarrow \mathbb{N}$ is a total function. For $\sigma \in \Sigma$, $\underline{rank}_\Sigma(\sigma)$ is called *rank of* σ . The subset Σ_m of Σ consists of all symbols of rank m $(m \geq 0)$. Note that, for $i \neq j$, Σ_i and Σ_j are disjoint. If the ranks of the symbols are clear from the context, then we drop the function $\underline{rank}_\Sigma$ from the denotation of the ranked alphabet $(\Sigma, \underline{rank}_\Sigma)$ and simply write Σ.

Let $(\Sigma, \underline{rank}_\Sigma)$ be a ranked alphabet and let S be an arbitrary set. Then the set of (labeled) *trees over* $(\Sigma, \underline{rank}_\Sigma)$ *indexed by* S, denoted by $T_\Sigma(S)$ is defined inductively as follows:

(i) $S \subseteq T_\Sigma(S)$
(ii) for every $\sigma \in \Sigma_k$ with $k \geq 0$ and $t_1, \ldots, t_k \in T_\Sigma(S)$, $\sigma(t_1, \ldots, t_k) \in T_\Sigma(S)$.

We denote $T_\Sigma = T_\Sigma(\emptyset)$. For the rest of this paper we fix the set $Y = \{y_1, y_2, \ldots\}$ of the parameter variables; for $k \geq 0$, $Y_k = \{y_1, \ldots, y_k\}$.

For a set $U = \{u_1, \ldots, u_k\}$ of words with $k \geq 0$, and some word w in which the elements of U do not occur overlapped, we denote by $w[u_1/w_1, \ldots, u_k/w_k]$ the result of substituting w_i for every occurrence of u_i in w.

3 Syntax-Directed Functional Programs

In this section we introduce the concept of syntax-directed functional programs (for short: sd funprog) by giving the definition of its syntax and semantics. Then we illustrate this concept by a simple example. Before we start with the definition of the syntax, we fix the following preliminaries for the rest of this paper: We assume that

- Σ denotes the ranked alphabet $\{\sigma_1, \ldots, \sigma_\rho\}$ of *input symbols* for some $\rho \geq 1$.
- Δ denotes the alphabet of *output symbols* .

and we call (Σ, Δ) the *input-output base* of sd funprogs. The only reason of having these preliminaries is to avoid repetitions in the formal definitions. Although having fixed now the symbols of Σ, we feel free to choose different symbols in concrete examples.

3.1 Syntax of sd funprogs

As already mentioned in the introduction, an sd funprog consists of a finite system of equations, each of them specifying one function by means of a case-expression with case-alternatives. For the sake of a better understanding, the definitions of these two latter technical concepts are given preference to the definition of the syntax of sd funprogs.

Definition 3.1 Let $\mathcal{F}$ denote a ranked alphabet of function symbols such that, for every $F \in \mathcal{F}$, $rank_{\mathcal{F}}(F) \geq 1$. Moreover, let $\sigma \in \Sigma_m$ for some $m \geq 0$ and let $n \geq 0$. The set of *case-alternatives for σ with n parameters and function symbols from $\mathcal{F}$*, denoted by $CA(\sigma, n, \mathcal{F})$, is the smallest subset $CA \subseteq (\Delta \cup \mathcal{F} \cup \{\underline{sel}_j(z) \mid j \in \mathbb{N}\} \cup Y \cup \{(,),,\})^*$ such that:

(i) $\varepsilon \in CA$.

(ii) For every $a \in \Delta$ and $\beta \in CA$, $a\beta \in CA$.

(iii) For every $k \geq 0$, $G \in \mathcal{F}_{k+1}$, $\beta, \beta_1, \ldots, \beta_k \in CA$ and $j \in [m]$,
$G(\underline{sel}_j(z), \beta_1, \ldots, \beta_k)\beta \in CA$.

(iv) For every $i \in [n]$ and $\beta \in CA$, $y_i\beta \in CA$. $\oplus$

$CA(\mathcal{F})$ denotes the set $\bigcup_{\sigma \in \Sigma, n \in \underline{RANK}(\mathcal{F})} CA(\sigma, n, \mathcal{F})$ where $\underline{RANK}(\mathcal{F})$ is the set of all ranks of function symbols occurring in $\mathcal{F}$. The expressions $\beta_1, \ldots, \beta_k$ in (iii) are called *parameter expressions.* We note that parameter expressions may contain functions.

Definition 3.2 Let $\mathcal{F}$ again denote a ranked alphabet of function symbols as in the previous definition. For every $n \in \mathbb{N}$, the set of *case-expressions with n parameters,* denoted by $CASE-EXPR(n)$, is the smallest set which contains all strings of the form

$$\text{CASE } \underline{root}(z) \text{ OF } \sigma_1 : t_1 \ldots \sigma_\rho : t_\rho \text{ END}$$

where for every $i \in [\rho]$, $t_i \in CA(\sigma_i, n, \mathcal{F})$. The set of *case-expressions of $\mathcal{F}$*, denoted by $CASE-EXPR(\mathcal{F})$, is the set $\bigcup_{n \in \underline{RANK}(\mathcal{F})} CASE - EXPR(n)$. $\oplus$

Now we are able to present the definition of the syntax of sd funprogs.

Definition 3.3 An *sd funprog* P is a tuple $(\mathcal{F}, F_{in}, E)$ such that

- $\mathcal{F}$ is a ranked alphabet of function symbols such that, for every $F \in \mathcal{F}$, $\underline{rank}_{\mathcal{F}}(F) \geq 1$.
- $F_{in} \in \mathcal{F}$ with rank 1 is the initial function symbol.
- E is a finite set of equations of the form

$$F(z, y_1, \ldots, y_n) = \zeta$$

where $F \in \mathcal{F}_{n+1}$, z is a variable, and $\zeta \in CASE - EXPR(n)$. Moreover, for every $F \in \mathcal{F}$ there is exactly one such equation in E.

$\oplus$

Note that the variable z, called recursion variable, may only occur as first argument of a function F. Also note that the initial function F_{in} always has rank one. The only reason for these restrictions is the fact that the implementation on the sdrs machine becomes technically a bit easier. Finally we note that, due to the definition of case-expressions and of sd funprogs, for every function symbol $F \in \mathcal{F}$ and for every input symbol $\sigma \in \Sigma$ there is exactly one case-alternative. Thus, from the computational point of view, no computation can block because of a lack of an appropriate case-alternative. This is the reason why sd funprogs only compute total functions, but we will come back to this point in the next subsection. The class of sd funprogs over the input-output base (Σ, Δ) is denoted by $SDFP(\Sigma, \Delta)$. Now let us illustrate these definitions by an easy example.

Example 3.4 Consider the ranked alphabets $\mathcal{F} = \{F_1^{(1)}, F_2^{(2)}, F_3^{(3)}\}$ and $\Sigma = \{\sigma^{(2)}, \gamma^{(1)}, \delta^{(0)}\}$ of function symbols and input symbols, respectively. Let $\Delta = \{a, b, c\}$ be the set of output symbols and consider the set E with the following three equations:

$$\begin{array}{lcll}
F_1(z) & = & \text{CASE } \underline{root}(z) \text{ OF} & \\
 & & \sigma : aF_2(\underline{sel}_1(z), bF_3(\underline{sel}_2(z), a, b)) & (1.1) \\
 & & \gamma : a & (1.2) \\
 & & \delta : a & (1.3) \\
 & & \text{END} & \\
F_2(z, y_1) & = & \text{CASE } \underline{root}(z) \text{ OF} & \\
 & & \sigma : a & (2.1) \\
 & & \gamma : F_3(\underline{sel}_1(z), y_1, c) & (2.2) \\
 & & \delta : a & (2.3) \\
 & & \text{END} & \\
F_3(z, y_1, y_2) & = & \text{CASE } \underline{root}(z) \text{ OF} & \\
 & & \sigma : a & (3.1) \\
 & & \gamma : a & (3.2) \\
 & & \delta : y_2 y_2 & (3.3) \\
 & & \text{END} &
\end{array}$$

Then $(\mathcal{F}, F_1, E)$ is an sd funprog over the input-output base (Σ, Δ). The string

$$\text{CASE } \underline{root}(z) \text{ OF } \sigma : aF_2(\underline{sel}_1(z), bF_3(\underline{sel}_2(z), a, b)),\ \gamma : a,\ \delta : a \text{ END}$$

is a case-expression without parameter, and $aF_2(\underline{sel}_1(z), bF_3(\underline{sel}_2(z), a, b))$ in (1.1) is a case-alternative for σ without parameter and function symbols from $\mathcal{F}$. The function call $F_3(\underline{sel}_2(z), a, b)$ occurs nested in the first parameter of F_2. $\oplus$

Having formalized the syntax of sd funprogs we will introduce its call-by-name semantics in the next subsection.

3.2 Semantics of sd funprogs

Clearly, every computation of an sd funprog is directed by an input tree, and before a computation step is executed, the appropriate case-alternative has to be chosen according to the label of the current node of the input tree. Thus, to prepare the definition of the semantics, we first formalize this choice as the semantics of case-expressions.

Definition 3.5 Let $P = (\mathcal{F}, F_{in}, E)$ be an sd funprog. The *semantics of case-expressions for $\mathcal{F}$*, denoted by $[\![\cdot]\!]_P^{case}$, is the function

$$[\![\cdot]\!]_P^{case} :\ CASE-EXPR(\mathcal{F}) \times T_\Sigma \longrightarrow CA(\mathcal{F})$$

with $[\![\text{ CASE } \underline{root}(z) \text{ OF } \sigma_1 : t_1, \ldots, \sigma_\rho : t_\rho \text{ END }]\!]_P^{case}\ (t) = t_j$, if $\underline{root}(t) = \sigma_j$. $\oplus$

The semantics of an sd funprog $P = (\mathcal{F}, F_{in}, E)$ is formalized by means of a binary computation relation on the set of potential computation forms.

Definition 3.6 Let $\mathcal{F}$ be a ranked alphabet of function symbols. The set of *potential computation forms,* denoted by *pCF,* is the smallest subset $\Omega \subseteq (\Delta \cup \mathcal{F} \cup T_\Sigma \cup \{(,),,\})^*$ such that

(i) $\varepsilon \in \Omega$.

(ii) For every $a \in \Delta$ and $\beta \in \Omega$, $a\beta \in \Omega$.

(iii) For every $F \in \mathcal{F}_{k+1}$ with $k \geq 0$, $t \in T_\Sigma$, $\beta \in \Omega$, and for every $i \in [k]$: $\beta_i \in \Omega$:

$$F(t, \beta_1, \ldots, \beta_k)\beta \in \Omega.$$

$\oplus$

In one computation step the leftmost occurrence of a function call $F(\sigma(s_1, \ldots, s_m), \beta_1, \ldots, \beta_n)$ is replaced by a modification of the case-alternative for σ in the equation for F. The modification consists of replacing every selection operator $\underline{sel}_j(z)$ by the direct subtree s_j of the current recursion argument, and by replacing every parameter variable y_i by the parameter expression β_i. Note that these parameter expressions do not contain parameter variables because this absence already holds for the whole function call. Now we are able to define the call-by-name computation relation of an sd funprog.

Definition 3.7 Let $P = (\mathcal{F}, F_{in}, E)$ be an sd funprog. The *call-by-name computation relation of P,* denoted by $\Longrightarrow_P$, is the smallest subset of $pCF \times pCF$ such that for every $\varphi_1, \varphi_2 \in pCF$: $\varphi_1 \Longrightarrow_P \varphi_2$ iff

(i) $\varphi_1 = wF(\sigma(s_1,\ldots,s_m),\beta_1,\ldots,\beta_n)\alpha$ for some $w \in \Delta^*$, $F \in \mathcal{F}_{n+1}$ with $n \geq 0$, $\sigma(s_1,\ldots,s_m) \in T_\Sigma$ with $m \geq 0$, $\beta_1,\ldots,\beta_n \in pCF$ and $\alpha \in pCF$.

(ii) $F(z, y_1,\ldots,y_n) \;=\; \zeta$ is an equation in E.

(iii) $\varphi_2 = w\ (\ [\![\ \zeta\ ]\!]_P^{case}(\sigma(s_1,\ldots,s_m)))\ [\underline{sel}_1(z)/s_1,\ldots,\underline{sel}_m(z)/s_m, y_1/\beta_1,\ldots,y_n/\beta_n]\alpha.$ $\oplus$

As usual we denote the reflexive, transitive closure of $\Longrightarrow_P$ by $\Longrightarrow_P^*$. We note that, if at all, any parameter expression is computed in a call-by-name fashion. However, we observe that the recursion argument does not have to be computed at all, because in every instance it is a subtree of the tree that is given to the initial function symbol at the beginning of the computation. Thus, the first argument of every function can also be considered as a call-by-value parameter. Actually, in the implementation of sd funprogs we will take advantage of this observation.

As described in the introduction, sd funprogs are in a one-to-one correspondence to total deterministic macro tree-to-string transducers. From the considerations of Section 3.3 of [EV86], it follows that for an arbitrary total deterministic macro tree-to-string transducer P and every input tree $s \in T_\Sigma$, there is a unique output string $w \in \Delta^*$ such that $F_{in}(s) \Longrightarrow_P^* w$. Thus, $\Longrightarrow_P$ induces a total function of type $T_\Sigma \longrightarrow \Delta^*$.

Definition 3.8 Let $P = (\mathcal{F}, F_{in}, E)$ be an sd funprog. The *function computed by P with call-by-name computation strategy* is the function $\tau(P)$: $T_\Sigma \longrightarrow \Delta^*$ defined by $\tau(P)(s) = w$ iff $F_{in}(s) \Longrightarrow_P^* w$.

The set of *computation forms of P,* denoted by $CF(P)$, is the set

$$\{\xi \mid \xi \in pCF \textit{ and } F_{in}(s) \Longrightarrow_P^* \xi,\ \textit{for some } s \in T_\Sigma\}$$

. $\oplus$

The class of functions computed by sd funprogs with call-by-name computation strategy is denoted by sdFun.

We finish this section by giving an example of a computation. The superscript at the computation relation $\Longrightarrow$ refers to the applied case-alternative.

Example 3.9 Consider the sd funprog P of 3.4. The following input tree $t \;=\; \sigma(\gamma(\delta),\delta)$ is given.

$$\begin{array}{ll}
 & F_1(\sigma(\gamma(\delta),\delta)) \\
\overset{(1.1)}{\Longrightarrow} & aF_2(\gamma(\delta), bF_3(\delta,a,b)) \\
\overset{(2.2)}{\Longrightarrow} & aF_3(\delta, bF_3(\delta,a,b), c) \\
\overset{(3.3)}{\Longrightarrow} & acc
\end{array}$$

Hence, $\tau(P)(\sigma(\gamma(\delta),\delta)) = acc.$

$\oplus$

In this section we have defined sd funprogs and explained them by an example. In the next two sections we will formalize a compiler-based implementation of the call-by-name computation strategy on a runtime stack machine: Section 4 introduces the machine, and Section 5 shows how to translate an sd funprog into code for this machine.

4 Syntax-Directed Runtime Stack Machine

Now we formalize the abstract machine on which the implementation of sd funprogs is based. These machines are called syntax-directed runtime stack machines, for short sdrs machines. The implementation presented here is compiler-based in the sense that every sd funprog is compiled into a flowchart over instructions which are executable on this machine.

The sdrs machine is defined by its instantaneous descriptions, its instruction set and the semantics of its instructions. An example of an instantaneous description of an sdrs machine is given in Figure 2 in the introduction. Since the definition of the sdrs machine is completely independent of the contents of the program store, this store is dropped from the instantaneous descriptions of the machine. Recall that the definitions refer to the input-output base (Σ, Δ) and that Σ is the set $\{\sigma_1, \ldots, \sigma_\rho\}$.

Definition 4.1 The (Σ, Δ)–*sdrs machine* is defined as follows:

- The set of *instantaneous descriptions*, denoted by $ID(\Sigma, \Delta)$, is the set PC $\times$ RS $\times$ OT, where
  - PC $= \mathbb{N}$ is the program counter.
  - RS $= SYMB^*$ with $SYMB = \{\text{F,Y}\} \cup \mathbb{N} \cup T_\Sigma$, is the runtime stack; a configuration h of the runtime stack is described as a sequence $h.1 : h.2 \ldots : h.q$ of squares $h.i \in SYMB$ where $h.1$ denotes the top square.
  - OT $= \Delta^*$ is the output tape.
- The set of *instructions* of the sdrs-machine, denoted by $I(\Sigma, \Delta)$, is the set containing all instructions of one of the following type:
  - Jump instructions:

    $JMP\ n$, $n \in$ PC
    $JMR\ (\sigma_1 : m_1, \ldots, \sigma_\rho : m_\rho)$, for every $i \in [\rho] : m_i \in$ PC

  - RS instructions:

    $CREATE(j, m_1, \ldots, m_n; ra)$, $j \in \mathbb{N}$, for every $i \in [n]$: $m_i \in$ PC, and $ra \in$ PC
    RET
    $EVAL\ i$, for every $i \in \mathbb{N}$

  - Output instruction:

    $WRITE\ a$, $a \in \Delta$

  - other instruction:

    $DUMMY$ $\oplus$

Programs for the sdrs machine are determinsitic flowcharts over the set of instructions in which the JMR-instructions realize the branching points.

Definition 4.2 The set of the *programs for the* (Σ, Δ)*-sdrs machine,* denoted by $Prog(\Sigma, \Delta)$, is the set

$$\{1 : \Gamma_1; \ \dots \ ; \ n : \Gamma_n \mid n \geq 1 \ \mathit{and\ for\ every}\ j \in [n] : \ \Gamma_j \in I(\Sigma, \Delta)\}$$

of strings over the alphabet $(\mathbb{N} \cup I(\Sigma, \Delta) \cup \{:, ;\})$. $\oplus$

We finish this subsection with an example of an sdrs machine program.

Example 4.3 Consider the input-output base (Σ, Δ) of Example 3.4. Then the following string is an (Σ, Δ)-sdrs machine program.

$$\begin{array}{lcl}
1 & : & JMR(\sigma : 2; \delta : 9) \ ; \\
2 & : & WRITE\ b \ ; \\
3 & : & CREATE(1, 5; 7) \ ; \\
4 & : & JMP\ 9 \ ; \\
5 & : & WRITE\ a \ ; \\
6 & : & RET \ ; \\
7 & : & WRITE\ b \ ; \\
8 & : & RET \ ; \\
9 & : & EVAL\ 2 \ ; \\
10 & : & JMP\ 11 \ ; \\
11 & : & RET \ ;
\end{array}$$

$\oplus$

Now we turn to the definition of the semantics of sdrs machine programs. First we give the definition of the semantics of the instructions, then we define the one-step semantics, and we end up with the iteration semantics. The semantics of a program will be defined by using particular input- and output-mappings and the iteration semantics.

We need the following five auxiliary functions to define the instruction semantics:

1. $env \ : \ \mathrm{RS} \longrightarrow \mathbb{N}$

$$env(h) = \begin{cases} 1 & , \ \mathit{if} \ \ h.1 = \mathrm{F} \\ 3 + h.3 & , \ \mathit{if} \ \ h.1 = \mathrm{Y} \end{cases}$$

Given a runtime stack h, this function yields the position of the top square of the F–block which contains the current environment, i.e., bindings of the recursion variable z and the parameter variables occuring in the postfix of the case-alternative which is currently computed. If the topmost block is an F–block, then it contains the bindings itself. If it is a Y–block, then the static link points to the appropriate F–block.

2. $next \ : \ \mathrm{RS} \longrightarrow \mathbb{N}$

$$next(h.1 : \ldots : h.q) = \begin{cases} 3 + h.3 & , \ \mathit{if}\ h.1 = \mathrm{F}\ \mathit{and}\ h.(3 + h.3) = \mathrm{F} \\ 5 + h.3 + h.(5 + h.3) & , \ \mathit{if}\ h.1 = \mathrm{F}\ \mathit{and}\ h.(3 + h.3) = \mathrm{Y} \\ 3 + h.3 + next(h') - 1 & , \ \mathit{if}\ h.1 = \mathrm{Y}\ \mathit{and}\ h.(3 + h.3) = \mathrm{F} \end{cases}$$

where $h' = h.(3 + h.3) : \ldots : h.q$

This function yields the position of the top square of the F–block which represents the environment of the current environment.

3. $cnr \;:\; \Sigma \longrightarrow \mathbb{N}$

   $cnr(\sigma) = l$, if $\sigma = \sigma_l$ for some $l \in [\rho]$

   This function yields the position of the constructor σ in the list $(\sigma_1, \ldots, \sigma_\rho)$ of input symbols that is given by the input alphabet of the (Σ, Δ)-sdrs machine.

4. $root \;:\; T_\Sigma \longrightarrow \Sigma$

   $root(t) = \sigma$, if $t = \sigma(t_1, \ldots, t_n)$ for some $\sigma \in \Sigma_n$ and $t_1, \ldots, t_n \in T_\Sigma$

   This function yields the root of the given tree.

5. $sel \;:\; T_\Sigma \times \mathbb{N} - \longrightarrow T_\Sigma$

$$sel(\sigma(t_1, \ldots, t_n), j) = \begin{cases} t_j & , \; if\ j \in [n] \\ undefined & , \; otherwise \end{cases}$$

   For a tree and a number j, this function determines the j-th subtree of the given tree, if it exists.

Definition 4.4 For every instruction $\Gamma \in I(\Sigma, \Delta)$, *the instruction semantics of* Γ, denoted by $\mathcal{C}[\![\, \Gamma \,]\!]$, is a function of type $ID(\Sigma, \Delta) \longrightarrow ID(\Sigma, \Delta)$ defined as follows where $(m, h, w) \in ID(\Sigma, \Delta)$:

1. $\mathcal{C}[\![\, JMP\ n \,]\!]\,(m, h, w) \;:=\; (n, h, w)$

   $\mathcal{C}[\![\, JMR\ (\sigma_1 : m_1, \ldots, \sigma_\rho : m_\rho) \,]\!]\,(m, h, w) \;:=\; (m_\kappa, h, w)$

   where $\kappa = cnr(root(h.4))$

2. $\mathcal{C}[\![\, CREATE(j, m_1, \ldots, m_n; ra) \,]\!]\,(m, h, w) \;:=$

   $(m + 1, \mathrm{F} : ra : (n + 2) : sel(h.(env(h) + 3), j) : m_1 : \ldots : m_n : h, w)$

   $\mathcal{C}[\![\, RET \,]\!]\,(m, h, w) \;:=$ if $h = \mathrm{F} : ra : dl : tree : adr_1 : \ldots : adr_{dl-2} : h'$ then (ra, h', w) if $h = \mathrm{Y} : ra : sl : h'$ then (ra, h', w)

   $\mathcal{C}[\![\, EVAL\ i \,]\!]\,(m, h, w) \;:=\; (h.(env(h) + 3 + i), \mathrm{Y} : m + 1 : next(h) : h, w)$

3. $\mathcal{C}[\![\, WRITE\ a \,]\!]\,(m, h, w) \;:=\; (m + 1, h, wa)$

4. $\mathcal{C}[\![\, DUMMY \,]\!]\,(m, h, w) \;:=\; (m + 1, h, w)$ $\oplus$

Before we define the one-step semantics, we informally explain the instruction semantics.

1. *JMP n* sets the program counter to n.

   Using the instruction $JMR\ (\sigma_1 : m_1, \ldots, \sigma_\rho : m_\rho)$ (jump on root) the program "jumps" to the program place corresponding to the label of the root of the current recursion argument. This tree is contained in the fourth square (counted from the top) of the current F–block. The functions *cnr* and *root* help to determine this symbol.

2. $CREATE(j, m_1, \ldots, m_n; ra)$ pushes a new F–block on top of the runtime stack with the tag F, the return address ra, the dynamic link $n+2$, the j-th subtree of the current recursion argument and the addresses $m_1, \ldots, m_n$ for the evaluation of the parameter variables. The program counter is set to the address of the next instruction.
   The RET instruction deletes the topmost block and sets the program counter to the return address of the deleted block.
   By means of the $EVAL\ i$ instruction a Y–block for the evaluation of the parameter variable y_i is pushed on the runtime stack and the program counter is set to the address for the computation of the parameter variable y_i. Note that this address is contained in the F–block which is specified by $env(h)$.

3. The instruction $WRITE\ a$ appends the output symbol a to the end of the output tape and increases the program counter.

4. The $DUMMY$ instruction only increases the program counter. The other components of the sdrs machine are not changed. This instruction is only introduced for the computation of an empty case-alternative of sd funprog.

After this explanation we define the one-step semantics and the iteration semantics of sdrs machine programs.

Definition 4.5 Let $P = 1 : \Gamma_1; \ldots; \mathrm{n} : \Gamma_n \ \in \ Prog(\Sigma, \Delta)$ be an (Σ, Δ)-sdrs machine program. The *one-step semantics* $\mathcal{E}[\![\ P\ ]\!]$ *of* P is the function $\mathcal{E}[\![\ P\ ]\!] : ID(\Sigma, \Delta) \longrightarrow ID(\Sigma, \Delta)$ where

$$\mathcal{E}[\![\ P\ ]\!]\,(m,h,w) = \begin{cases} \mathcal{C}[\![\ \Gamma_m\ ]\!]\,(m,h,w) & , \ if\ m \in [n] \\ (0,h,w) & , \ otherwise \end{cases}$$

The *iteration semantics* $\mathcal{I}[\![\ P\ ]\!]$ *of* P is the function $\mathcal{I}[\![\ P\ ]\!] : ID(\Sigma, \Delta) \longrightarrow ID(\Sigma, \Delta)$ where

$$\mathcal{I}[\![\ P\ ]\!]\,(m,h,w) = \begin{cases} (m,h,w) & , \ if\ m = 0 \\ \mathcal{I}[\![\ P\ ]\!]\,(\mathcal{E}[\![\ P\ ]\!]\,(m,h,w)) & , \ otherwise \end{cases}$$

$\oplus$

Now we introduce the input-mapping and output-mapping for the sdrs machine.

Definition 4.6 The input-mapping is the function $input : T_\Sigma \longrightarrow ID(\Sigma, \Delta)$ defined by

$$input(t) = (1, \mathrm{F} : 0 : 2 : t, \varepsilon)$$

.

The output-mapping is the function $output : ID(\Sigma, \Delta) \longrightarrow \Delta^*$ defined by

$$output(0,h,w) = w$$

. $\oplus$

The initial instantaneous description (m, h, w) of the sdrs machine is provided by the input-mapping. The program counter m is set to the starting address 1, the runtime stack h only contains one F–block with tag F, return address 0, dynamic link 2, and the given input tree t, and the output tape w contains the empty word ε. The output-mapping projects the string written on the output tape. Now, the semantics of a program can be defined in terms of the input- and output-mappings, and the iteration semantics.

Definition 4.7 Let P be a (Σ, Δ)-sdrs machine program.

The semantics of P is the function $\mathcal{M}[\![\ P \ ]\!]$: $T_\Sigma - \longrightarrow \Delta^*$ defined by

$$\mathcal{M}[\![\ P \ ]\!] := \mathit{input} \circ \ \mathcal{I}[\![\ P \ ]\!] \circ \ \mathit{output}$$

where $\circ$ denotes the composition operation to be read from the left to the right. $\oplus$

Note that $\mathcal{M}[\![\ P \ ]\!]$ might be a partial function. Although, if P is an sdrs machine program which is generated by a translation of an sd funprog, then $\mathcal{M}[\![\ P \ ]\!]$ is total. After the introduction of the sd funprogs and the sdrs machine we define the translation of sd funprogs into sdrs machine programs in the next section.

5 Translation of sd funprogs into sdrs machine programs

For the sake of simplicity of the translation from $SDFP(\Sigma, \Delta)$ to $Prog(\Sigma, \Delta)$, we always use tree-structured addresses in the generated programs. Such an address is a string of nonnegative integers seperated 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. Before formalizing programs with tree-structured addresses, we present an example of such a program.

Example 5.1 Let $\Sigma = \{\sigma^{(2)}, \delta^{(0)}\}$ be a ranked alphabet and $\Delta = \{a, b\}$ be a usual alphabet. Then

```
1       :  JMR(σ : 1.1, δ : 1.2) ;
1.1     :  WRITE b ;
           CREATE(1, 1.1.1; 1.1.2) ;
           JMP 1.2 ;
1.1.1   :  WRITE a ;
           RET ;
1.1.2   :  WRITE b ;
           RET ;
1.2     :  EVAL 2 ;
1.3     :  JMP 1.4 ;
1.4     :  RET ;
```

is a (Σ, Δ)-sdrs machine program with tree-structured addresses. $\oplus$

We assume that a load program exists, which transforms programs with tree-structured addresses into programs with usual addresses as required in Definition 4.2. This load program transforms, e.g., the program in Example 5.1 into the program shown in Example 4.3.

Definition 5.2 The *set of* (Σ, Δ)*-sdrs machine programs with tree-structured addresses,* denoted by $Prog - t(\Sigma, \Delta)$, is the set

$$\{w_1 : \Gamma_1 : \ldots ; w_n : \Gamma_n \mid n \geq 1,\ for\ every\ i \in [n] : \Gamma_i \in I'(\Sigma, \Delta),\ w_i \in (\mathbb{N}^*)^*\},$$

of strings where $I'(\Sigma, \Delta)$ denotes the set of instructions that include tree-structured addresses as parameters and it is required that no address appears more than once as a label of an instruction in a program. $\oplus$

The translation from $SDFP(\Sigma, \Delta)$ to $Prog - t(\Sigma, \Delta)$ is defined inductively over the syntax of the sd funprog (cf. Definitions 3.1, 3.2, and 3.3).

Definition 5.3 The translation $\underline{trans}$ is a function of type $SDFP(\Sigma, \Delta) \longrightarrow Prog - t(\Sigma, \Delta)$. Let $P = (\mathcal{F}, F_{in}, E)$ be an sd funprog with $\mathcal{F} = \{F_1, \ldots, F_r\}$ for some $r \geq 0$, $F_{in} = F_1$, and

$$E = \{F_i(z, y_1, \ldots, y_{n_i}) = \zeta_i \mid i \in [r]\}.$$

Then

$$\begin{aligned} \underline{trans}(P) = 1 \ &: \ \underline{ct}(\zeta_1, 1) \ \ RET \ ; \\ &\vdots \\ r \ &: \ \underline{ct}(\zeta_r, r) \ \ RET \ ; \end{aligned}$$

where $\underline{ct}$ is the function which translates a case-expression into a piece of an sdrs machine program. The second parameter of $\underline{ct}$ is a tree-structured address α such that for every string $\beta \in \mathbb{N}^*$, the address $\alpha\beta$ is not yet used in the program generated so far. $\underline{ct}$ is defined as follows:

$$\underline{ct} \ : \ CASE - EXPR(\mathcal{F}) \ \times \ \mathbb{N}^* \ \longrightarrow \ Prog - t(\Sigma, \Delta)$$

with $\underline{ct}(\text{ CASE } \underline{root}(z) \text{ OF } \sigma_1 : t_1 \ldots \sigma_\rho : t_\rho \text{ END }, \alpha) =$

$$\begin{array}{lcl} & & JMR \ (\sigma_1 : \alpha.1, \ldots, \sigma_\rho : \alpha.\rho); \\ \alpha.1 & : & \underline{et} \ (t_1, \alpha.1) \ \ JMP \ \alpha.(\rho+1) \ ; \\ \alpha.2 & : & \underline{et} \ (t_2, \alpha.2) \ \ JMP \ \alpha.(\rho+1) \ ; \\ & & \vdots \\ \alpha.\rho & : & \underline{et} \ (t_\rho, \alpha.\rho) \\ \alpha.(\rho+1) : & & \end{array}$$

where $\underline{et}$ is the function which translates a case-alternative into a sequence of instructions. It is defined inductively over the structure of the case-alternatives as follows:

$$\underline{et} \ : \ CA(\mathcal{F}) \times \mathbb{N}^* \ \longrightarrow \ Prog - t(\Sigma, \Delta)$$

with $\underline{et} \ (\varepsilon, \alpha) \ = \ DUMMY \ ;$

for $a \in \Delta, \ \beta \in CA(\mathcal{F})$:

$$\underline{et}(a\beta, \alpha) = WRITE \ a; \ \underline{et}(\beta, \alpha.1)$$

for y_i with $i \geq 1$, $\beta \in CA(\mathcal{F})$:

$$\underline{et}(y_i\beta, \alpha) = EVAL \ i; \ \underline{et} \ (\beta, \alpha.1)$$

for $F_i \in \mathcal{F}_{n+1}, i \geq 1, \ n \in \mathbb{N}$, for every $k \in [n]$: $\beta_k \in CA(\mathcal{F})$, $\beta \in CA(\mathcal{F})$, and $j \geq 1$:

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{et}(F_i(\underline{sel}_j(z), \beta_1, \ldots, \beta_n)\beta, \alpha) = CREATE(j, \alpha.1, \ldots, \alpha.n; \alpha.(n+1)) \ ;} \\ & JMP \ i \ ; \\ \alpha.1: & \underline{et} \ (\beta_1, \alpha.1) \ \ RET \ ; \\ & \vdots \\ \alpha.n: & \underline{et} \ (\beta_n, \alpha.n) \ \ RET \ ; \\ \alpha.(n+1): & \underline{et} \ (\beta, \alpha.(n+1)) \end{array}$$

$\oplus$

The generated program always starts with the translation of the case-expression of the equation for F_1, which is also called the main equation. The first instruction in the translation of any case-expression is the JMR-instruction. Depending on the root symbol of the current recursion argument, it jumps to the corresponding code which is produced by the translation function $\underline{et}$. This latter function is defined recursively on the strucure of case-alternatives:
An empty case-alternative is translated into the $DUMMY$ instruction . If the given case-alternative is prefixed by an output symbol a, then the instruction $WRITE\ a$ is produced. If the given case-alternative is prefixed by a parameter variable y_i, then the instruction $EVAL\ i$ must be executed. This instruction produces a new Y–block at the top of the runtime stack for the evaluation of this parameter variable. If the case-alternative starts with a function call $F_i(sel_j(z), \beta_1, \ldots, \beta_n)$, then a $CREATE$-instruction pushes an F–block on top of the runtime stack. The succeeding $JMP\ i$ instruction jumps to the first instruction of $\underline{ct}(\zeta_i, i)$. Furthermore, the code for the evaluation of the parameter expressions $\beta_1, \ldots, \beta_n$ is produced. The starting addresses for the evaluation of parameter expressions are given as parameters to the mentioned $CREATE$-instruction.

Now we want to illustrate the translation for an sd funprog into an sdrs-machine program by means of an example.

Example 5.4 Consider the sd funprog P of Example 3.4. We do not want to develop its translation step by step. Rather, we show the top-level of this development for the translation of the case-expression of the function $\mathcal{F}_2$ and we show the complete translation of P in Figure 4.

$\underline{ct}($ CASE $\underline{root}(z)$ OF $\sigma\ :\ a, \gamma\ :\ F_3(\underline{sel}_1(z), y_1, c), \delta\ :\ a$ END $, 2) =$

$$\begin{array}{ll}
2\ : & JMR(\sigma : 2.1, \gamma : 2.2, \delta : 2.3)\ ; \\
2.1 : & \underline{et}(a, 2.1) \\
 & JMP\ 2.4\ ; \\
2.2 : & \underline{et}(F_3(\underline{sel}_1(z), y_1, c), 2.2) \\
 & JMP\ 2.4\ ; \\
2.3 : & \underline{et}(a, 2.3) \\
2.4 : & RET\ ;
\end{array}$$

The RET instruction and Label 2 is produced by $\underline{trans}(P)$.

By applying the load program to $\underline{trans}(P)$ we obtain the translation of P which is presented in Figure 4.

For a better understanding we have ommitted the $DUMMY$-instructions, but note that a 'complete' translation would contain them. We finish this example with the description of the computation of the sdrs machine for the input tree $\sigma(\gamma(\delta), \delta)$ in Figure 5 (cf. the computation of the sd funprog in Example 3.9). The execution of the program is documented as follows:

- Every state of the sdrs machine is enumerated.
- Every block of the runtime stack is written down separately in one line. Thus the uppermost line corresponds to the top block of the runtime stack.
- The instruction of the program to be evaluated next is listed separately.
- We only list the components of the sdrs machine which have been changed during the last computation step.

For a better understanding of the behaviour of the machine, we give some detailed informations. The step transforming the state with the number n to the state with the number $n+1$, is called step n.

- State 1 is produced by application of the input mapping to the input tree $\sigma(\gamma(\delta),\delta)$.
- Step 1 tests the top-constructor and jumps to the program address for the execution of the case-alternative for σ. Then the output symbol a is written to the output tape.
- In step 3 a new F–block for the evaluation of the function call $F_2(\underline{sel}_1(z), bF_3(\underline{sel}_2(z), a, b))$ is created on top of the runtime stack. Note that the actual value of z is $\sigma(\gamma(\delta),\delta)$.
- Then a new test on the top-constructor is executed.
- In step 6 another F–block for the evaluation of the function call $F_3(\underline{sel}_1(z), y_1, c)$ is created. Note that the actual values of z and y_1 are $\gamma(\delta)$ and $bF_3(\delta, a, b)$, respectively.
- In this configuration the value of the parameter variable y_2, which is c, is evaluated and then the output symbol c is written down to the output tape.
- In step 11 the Y–block can be deleted, because the evaluation of the value of y_2 has been finished.
- Then the second occurrence of y_2 in the case-alternative (3.3) of Example 3.4 must be evaluated . Thus, again a new Y–block is generated , c is written down to the output tape, and then the Y–block is deleted.
- The second function call is finished and the corresponding F–block is deleted in step 15.
- The first function call is finished and the corresponding F–block is deleted in step 17.
- In step 19 the input–block is deleted and the stop label 0 is written to the program counter.
- The work stops with the output acc. $\oplus$

Function	No.	Code	Constructor
F_1	1	$JMR(\sigma : 2, \gamma : 14, \delta : 16)$;	
	2	$WRITE\ a$;	σ
	3	$CREATE(1, 5; 13)$;	
	4	JMP 18 ;	
	5	$WRITE\ b$;	
	6	$CREATE(2, 8, 10; 13)$;	
	7	JMP 30 ;	
	8	$WRITE\ a$;	
	9	RET ;	
	10	$WRITE\ b$;	
	11	RET ;	
	12	RET ;	
	13	JMP 17 ;	
	14	$WRITE\ a$;	γ
	15	JMP 17 ;	
	16	$WRITE\ a$;	δ
	17	RET ;	
F_2	18	$JMR(\sigma : 19, \gamma : 21, \delta : 28)$;	
	19	$WRITE\ a$;	σ
	20	JMP 29 ;	
	21	$CREATE(1, 23, 25; 27)$;	γ
	22	JMP 30 ;	
	23	$EVAL$ 1 ;	
	24	RET ;	
	25	$WRITE\ c$;	
	26	RET ;	
	27	JMP 29 ;	
	28	$WRITE\ a$;	δ
	29	RET ;	
F_3	30	$JMR(\sigma : 31, \gamma : 33, \delta : 35)$;	
	31	$WRITE\ a$;	σ
	32	JMP 37 ;	
	33	$WRITE\ a$;	γ
	34	JMP 37 ;	
	35	$EVAL$ 2 ;	δ
	36	$EVAL$ 2 ;	
	37	RET ;	

Figure 4: Translation of the sd funprog of Example 3.4

No.	ProgramCounter	RuntimeStack	OutputTape	Instruction
1	1	F: $0:2:\sigma(\gamma(\delta),\delta)$	ε	$JMR(\sigma:2,\gamma:14,\delta:16)$
2	2			$WRITE\ a$
3	3		a	$CREATE(1,5,13)$
4	4	F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$JMP\ 18$
5	18			$JMR(\sigma:19,\gamma:21,\delta:28)$
6	21			$CREATE(1,23,25;27)$
7	22	F: $27:4:\delta:23:25$ F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$JMP\ 30$
8	30			$JMR(\sigma:31,\gamma:33,\delta:35)$
9	35			$EVAL\ 2$
10	25	Y: $36:7$ F: $27:4:\delta:23:25$ F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$WRITE\ c$
11	26		ac	RET
12	36	F: $27:4:\delta:23:25$ F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$EVAL\ 2$
13	25	Y: $37:7$ F: $27:4:\delta:23:25$ F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$WRITE\ c$
14	26		acc	RET
15	37	F: $27:4:\delta:23:25$ F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		RET
16	27	F: $13:3:\gamma(\delta):5$ F: $0:2:\sigma(\gamma(\delta),\delta)$		$JMP\ 29$
17	29			RET
18	13	F: $0:2:\sigma(\gamma(\delta),\delta)$		$JMP\ 17$
19	17			RET
20	0	ε		

Figure 5: Computation of the sdrs machine program in Figure 4 with input tree $\sigma(\gamma(\delta),\delta)$

6 Comparison of the sdrs machine and the ct-ns transducer

After having developed a compiler-based implementation of sd funprogs on the basis of the sdrs machine, we now briefly recall from [FV90] the checking-tree nested-stack transducer (for short: ct-ns transducer), which serves as an interpreter-based implementation of sd funprogs. Recall from the introduction that ct-ns transducers exactly compute the functions of the class sdFun. In the second subsection we compare both implementations and, by establishing a one-to-one correspondence between the computational behaviour of the sdrs machine and the ct-ns transducer, we verify that sdrs machines also compute exactly the class sdFun.

6.1 Informal description of ct-ns transducer

An instantaneous description of the ct-ns transducer is provided by the state of the finite control, the contents of the output-tape, and the configuration of the checking-tree nested-stack (for short: ct-ns). An example of an instantaneous description is given in Figure 6.

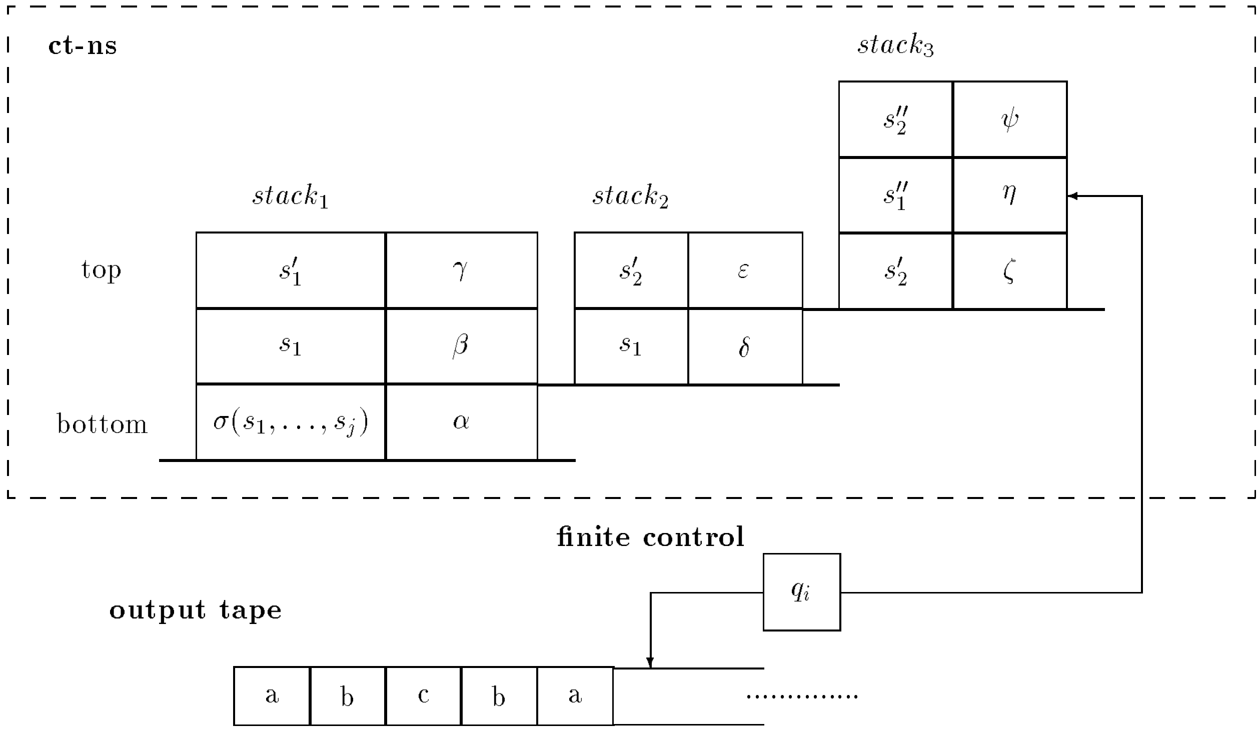
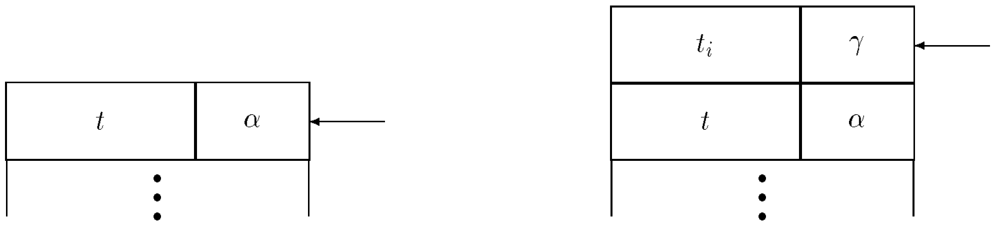


Figure 6: A Checking-Tree Nested-Stack Transducer.

Intuitively, a ct-ns consists of a finite amount of stacks that are nested in each other, e.g., the nested-stack in Figure 6 consists of three stacks $stack_1$, $stack_2$, and $stack_3$; $stack_3$ is nested between the two squares of $stack_2$, $stack_2$ is nested between the bottom square of $stack_1$ and the next square above; $stack_1$ is called outermost stack. We note that the nesting relation of stacks is acyclic. One of the squares is designated to be the current square. Each square of the ct-ns consists of two components. The left component includes a subtree of the checking-tree and the right component includes a symbol of the stack alphabet. The tree in the left component of a stack square is a direct subtree of the tree in the square below, e.g., in Figure 6, s'_2 is a direct subtree of s_1. The complete checking-tree $\sigma(s_1, \ldots, s_j)$ is stored in the bottom square of the outermost stack. (The reader is refered to Definition 3.17 and Definition 7.1 of [EV86] for a formal definition of the checking-tree nested-stack; there it is denoted by NS(TR).) The configurations of the ct-ns can be transformed by means of the instructions $push(i, \gamma)$, *pop*, *moveup*, *movedown*, $create(\gamma)$, *destruct*, and $stay(\gamma)$, where γ is an element of the stack alphabet and i is a positive integer. The meaning of these instructions is described in the following figures, where we assume that t_i is the i-th direct subtree of t.

- $push(i, \gamma)$: (1) $\Longrightarrow$ (2), *pop* : (2) $\Longrightarrow$ (1)

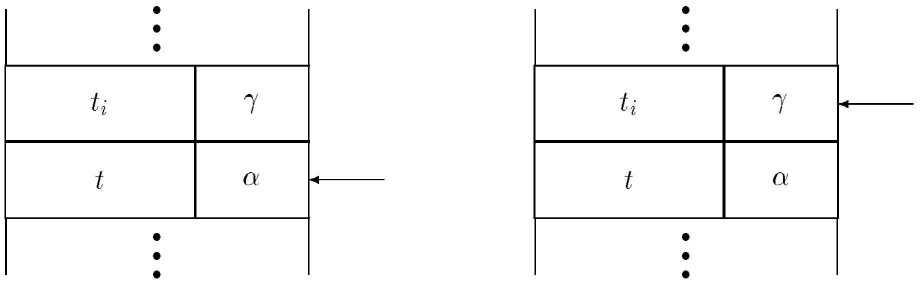


(1) (2)

- *moveup* : (1) $\Longrightarrow$ (2), *movedown* : (2) $\Longrightarrow$ (1)

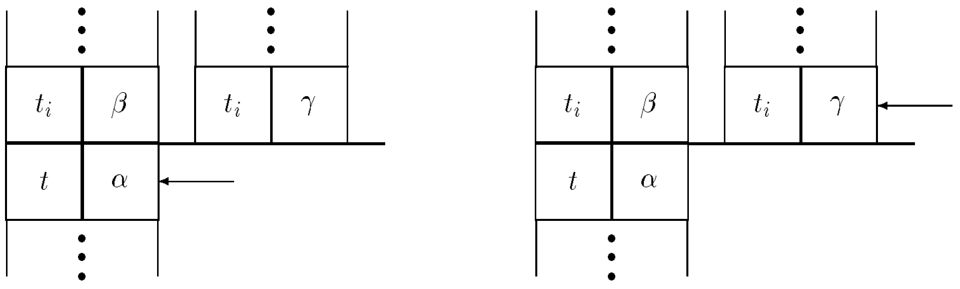
  We have to distinguish two possible configurations:

1.)



(1) (2)

2.)

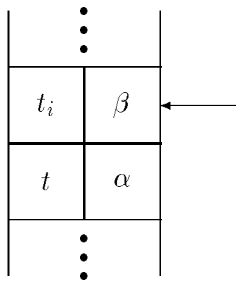


(1) (2)

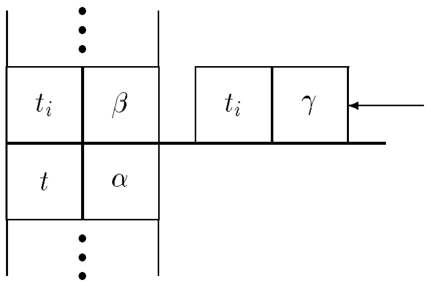
- $create(\gamma)$: (1) $\Longrightarrow$ (2), $destruct$: (2) $\Longrightarrow$ (1)

  Also here, we have to distinguish two possibilities:

1.)

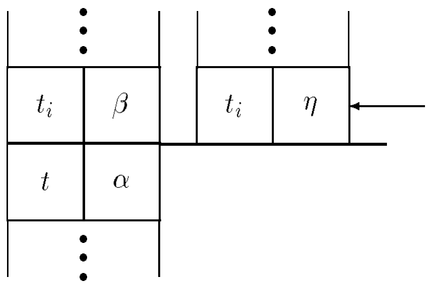


(1)

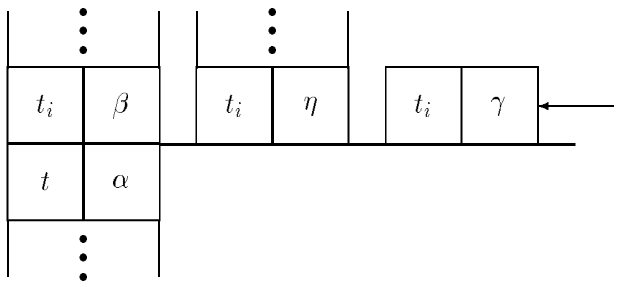


(2)

2.)

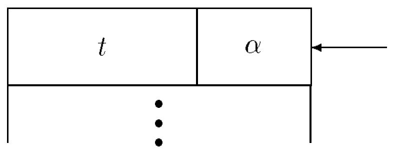


(1)

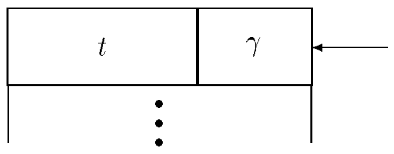


(2)

- $stay(\gamma)$: (1) $\Longrightarrow$ (2)



(1)



(2)

A ct-ns transducer contains a finite set of rules each of which has the following form:

$$q(\sigma(x_1, \ldots, x_j), \gamma) \longrightarrow \beta$$

where q denotes a state of the finite control, $\sigma(x_1, \ldots, x_j)$ and γ denote representations of the two components of the current stack square. The right-hand-side β is a string over output symbols followed by at most one construct (p, ϕ) where p is a state and ϕ is an instruction of the ct-ns. The rule can be applied to an instantaneous description of the ct-ns transducer, if q is the state of the finite control and the current stack square contains a tree with root label σ in its first component and the symbol γ in its second component.

Example 6.1 Consider the instantaneous description of the ct-ns transducer in Figure 6 and the following rule:

$$q_i(\sigma(x_1, \ldots, x_j), \eta) \longrightarrow abcq_l(moveup),$$

where q_i and q_l are states and a, b, c are output symbols. This rule can be applied to the instantaneous description in Figure 6, if we assume that σ denotes the root of the tree s_1'' in the current square. The application of this rule changes the instantaneous description by

- switching the finite control to state q_l.
- applying the instruction *moveup* to the ct-ns and
- appending the string *abc* to the output tape. $\oplus$

The formal definition of the computation relation of a ct-ns transducer can be found in [EV86, FV90].

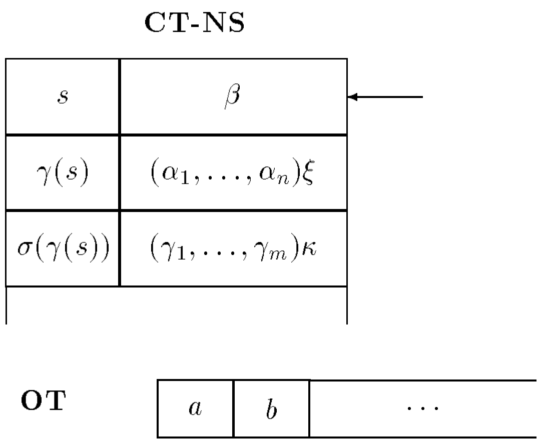
6.2 Comparison of the implementations

In [FV90] a direct interpreter-based implementation of sd funprogs on ct-ns transducers is defined. In this subsection we simultaneously recall this implementation and compare it with the implementation of sd funprog on the sdrs machine as developed in this paper.

In the interpreter-based implementation the whole case-alternative is put into the right component of the current square, and it is interpreted symbol by symbol from left to right. This interpretation mechanism replaces the program-controled mechanism of the sdrs machine. We divide the comparison of the interpretation mechanism and the program-controled mechanism into the four cases which are possible for case-alternatives (cf. Definition 3.1):

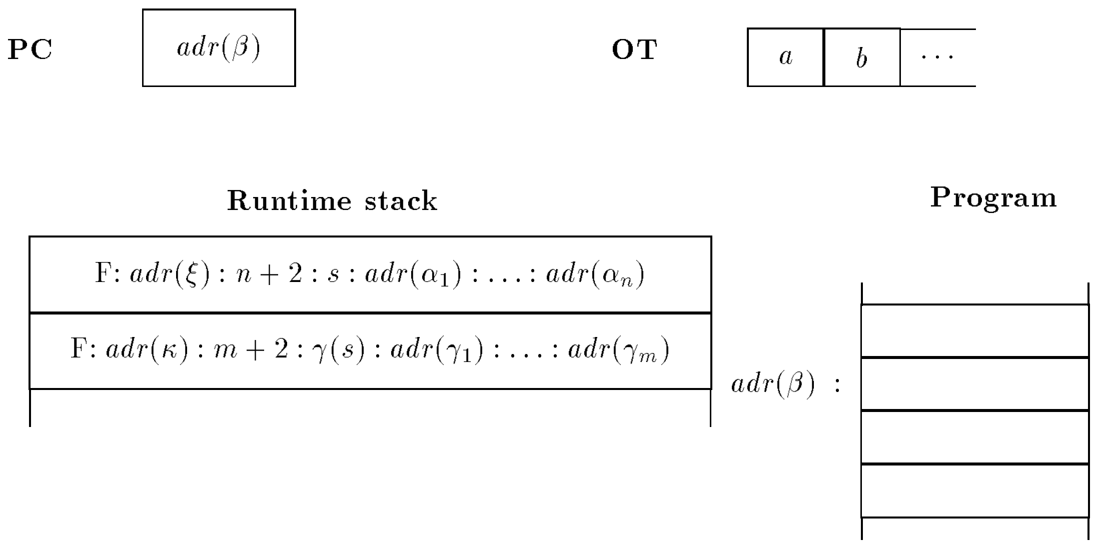
1. $\beta \;=\; a\delta$, where $a \in \Delta$
2. $\beta \;=\; F_i(\underline{sel}_j(z), \beta_1, \ldots, \beta_k)\delta$
3. $\beta \;=\; y_i\delta$
4. $\beta \;=\; \varepsilon$

We consider the following instantaneous description of the ct-ns transducer as starting point of our comparison where p is the working state:


CT-NS
s
β
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\sigma(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$
OT
a
b
$\cdots$

We will see that this instantaneous description is sufficiently general for the description of the behaviour of the ct-ns transducer in each of the four cases.

The corresponding instantaneous description of the sdrs machine is shown in the next figure:

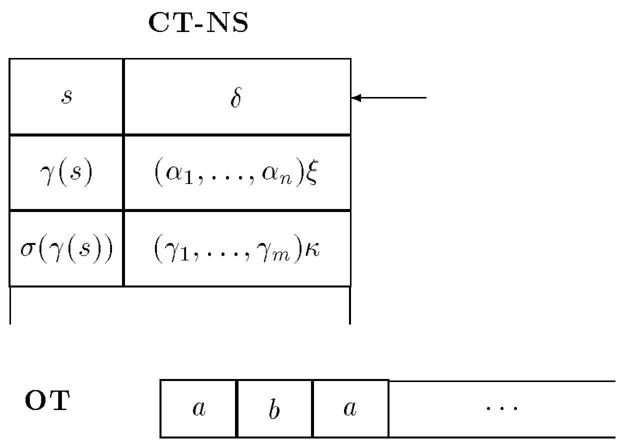

PC
$adr(\beta)$
OT
a
b
$\cdots$
Runtime stack
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$
Program
$adr(\beta)$:

The program counter contains the start address for the evaluation of the current expression β. The top F–block consists of the tag F, the return address $adr(\xi)$ for the evaluation of the expression ξ, the dynamic link $n+2$, the recursion argument s and the start addresses $adr(\alpha_1), \ldots, adr(\alpha_n)$ for the evaluation of the parameters $\alpha_1, \ldots, \alpha_n$. The contents of the output tape is identical to the output tape of the ct-ns transducer.

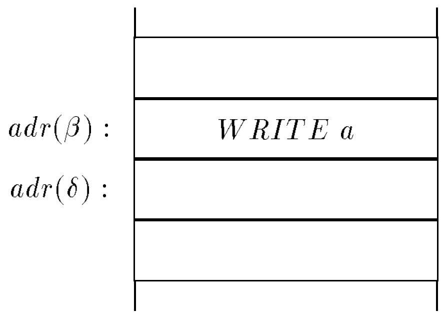
Now, we illustrate the instantaneous description of the ct-ns transducer after interpretation of the prefix symbol of β and the instantaneous description of the sdrs machine after the execution of the code which arises from the translation of the prefix symbol of β. If the output tape is not changed, it is omitted. The finite control of the ct-ns transducer mostly is in the working state p; if this holds, then we omit the control from the figures.

Case 1: $\beta = a\delta$

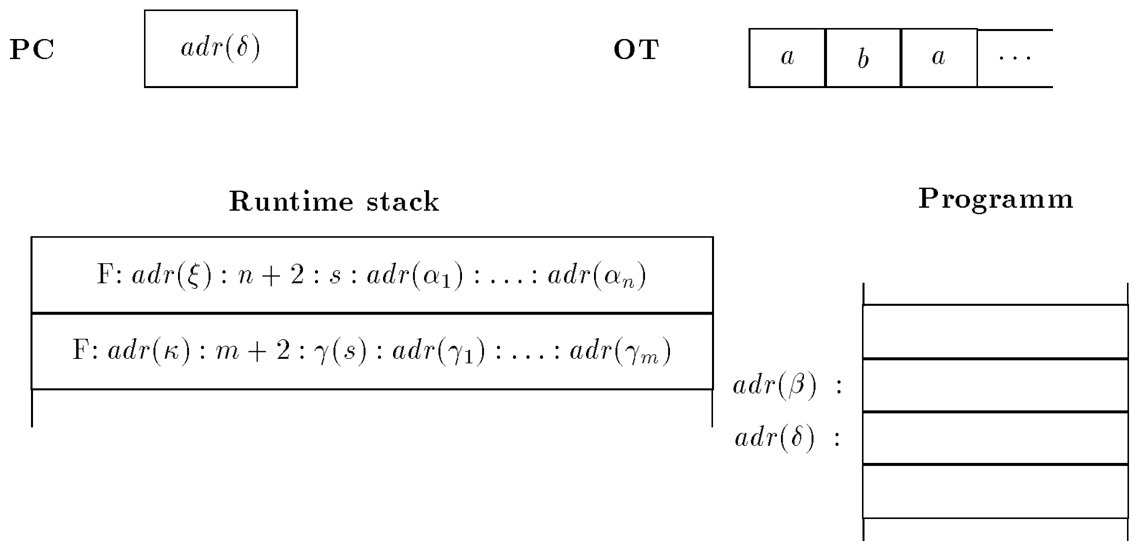
The output symbol a is deleted in the current square of the ct-ns and it is written to the output tape. Thus, we reach the following instantaneous description:


CT-NS
s
δ
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\sigma(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$
OT
a
b
a
$\cdots$

In this situation the program storage of the sdrs machine has the following structure:

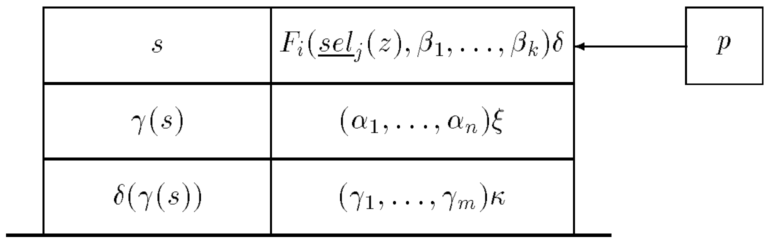


After executing $WRITE\ a$ we obtain the following instantaneous description:

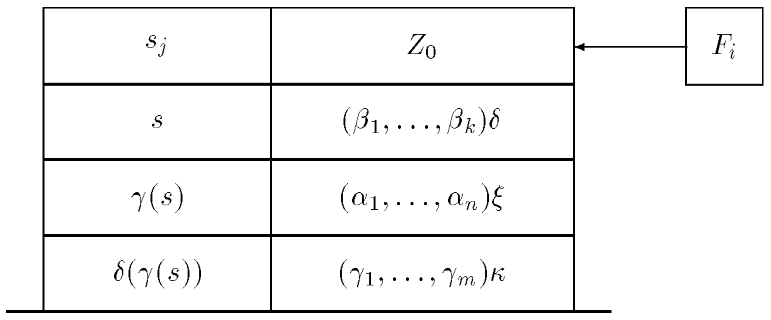

PC
$adr(\delta)$
OT
a
b
a
$\cdots$
Runtime stack
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$
Programm
$adr(\beta)$:
$adr(\delta)$:

Case 2: $\beta \; = \; F_i(\underline{sel}_j(z), \beta_1, \ldots, \beta_k)\delta$

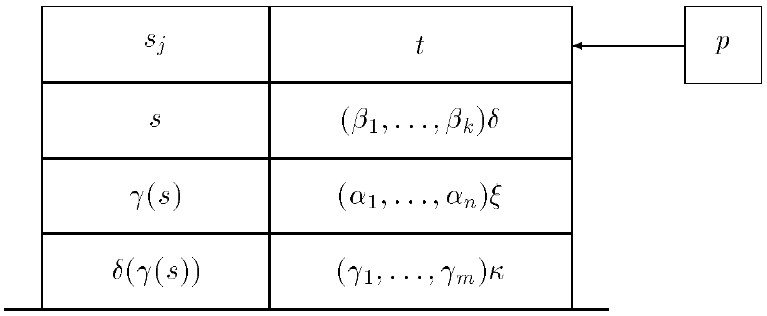
In the ct-ns transducer $F_i(\underline{sel}_j(z), \beta_1, \ldots, \beta_k)\delta$ is replaced by $(\beta_1, \ldots, \beta_k)\delta$, and a $push(j, Z_0)$-instruction is executed. Thus, in the new square the left component includes the j-th subtree s_j of s and the right component includes Z_0. Furthermore, the state of the finite control changes to F_i. For instance, the instantaneous description


s
$F_i(\underline{sel}_j(z), \beta_1, \ldots, \beta_k)\delta$
p
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\delta(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$

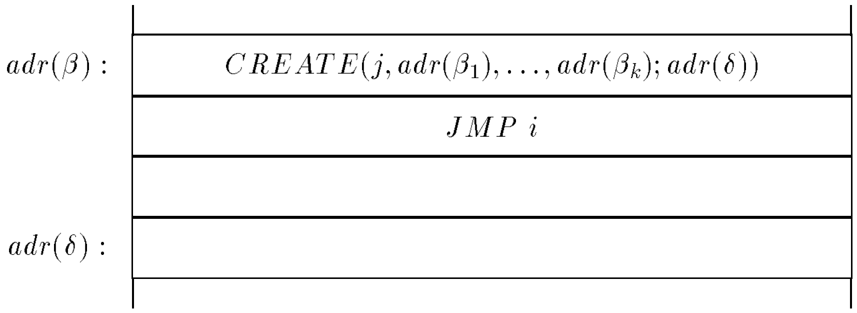
is changed into the following instantaneous description:


s_j
Z_0
F_i
s
$(\beta_1, \ldots, \beta_k)\delta$
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\delta(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$

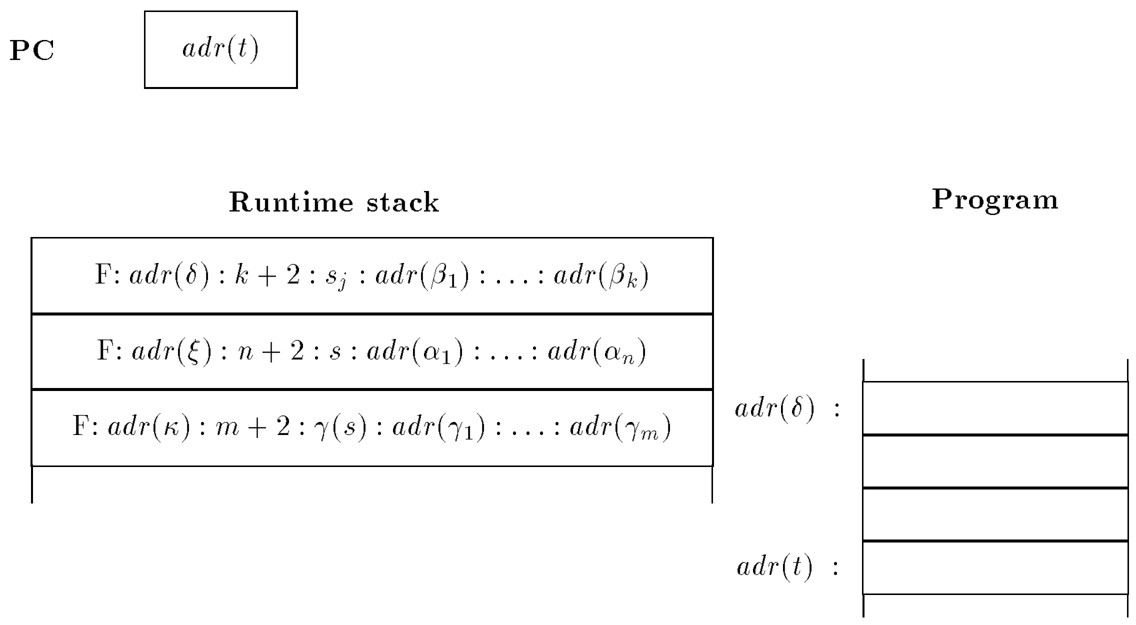
If, for example the root of s_j is σ and, in the sd funprog which is under consideration, t is the case-alternative for σ of F_i, then Z_0 is replaced by t, and the finite control changes back to the working state p. Note that the parameter list in the square below the new top square is not deleted, because it contains the bindings of the parameter variables in t. We obtain the following instantaneous description:


s_j
t
p
s
$(\beta_1, \ldots, \beta_k)\delta$
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\delta(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$

In this situation the program storage of the sdrs machine has the following structure:


$adr(\beta)$:
$CREATE(j, adr(\beta_1), \ldots, adr(\beta_k); adr(\delta))$
$JMP\ i$
$adr(\delta)$:

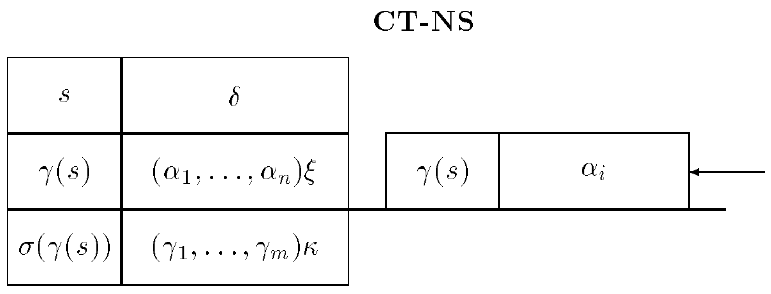
By executing the $CREATE$ and JMP instructions we obtain the following instantaneous description:


PC
$adr(t)$
Runtime stack
Program
F: $adr(\delta) : k+2 : s_j : adr(\beta_1) : \ldots : adr(\beta_k)$
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$
$adr(\delta)$:
$adr(t)$:

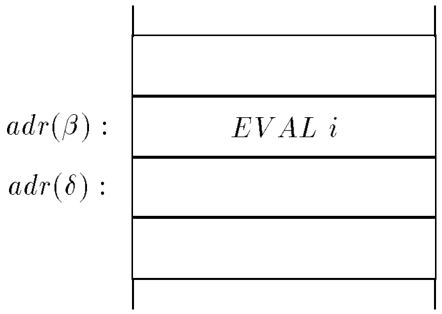
A new F–block with return address $adr(\delta)$, the dynamic link $k+2$, the recursion argument s_j, the addresses $adr(\beta_1), \ldots, adr(\beta_k)$ for the evaluation of the parameters $\beta_1, \ldots, \beta_k$ is created at top of the runtime stack. Then the program "jumps" to the start address of F_i. There a JMR instruction is executed. Thus, the program jumps to the start address for the evaluation of the corresponding case-alternative $adr(t)$.

Case 3: $\beta = y_i\delta$

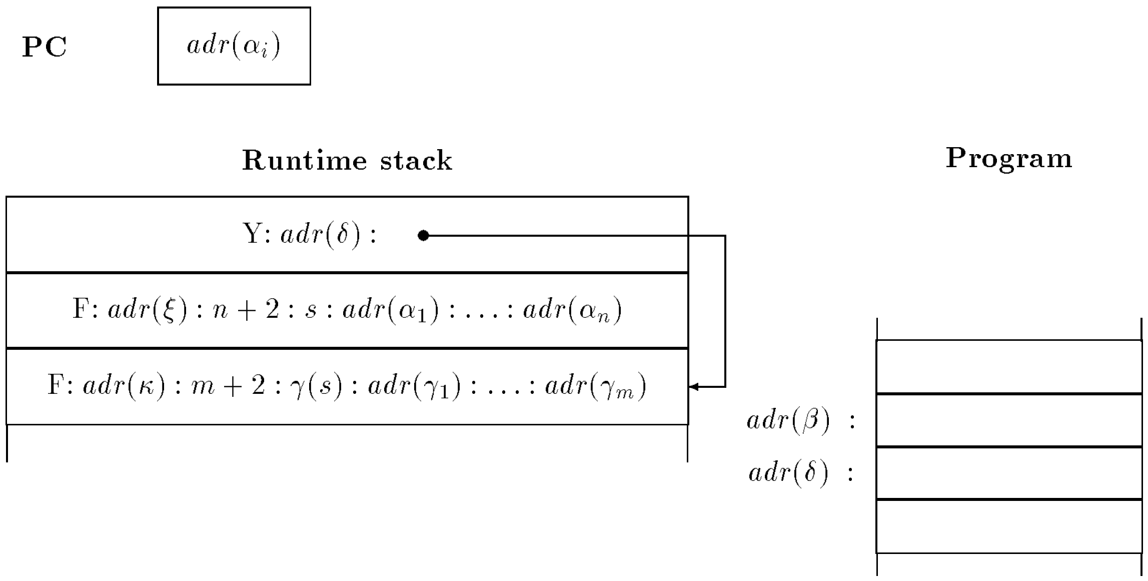
The parameter list $(\alpha_1, \ldots, \alpha_n)$ contained in the square below the current stack square, represents the bindings of the parameter variables in β. Thus, we find the binding of y_i in the square below and we have to evaluate the expression α_i. We delete y_i in the top square, we movedown to the parameter list, and we create a new stack for the evaluation of α_i. Note that this new square contains $\gamma(s)$ as checking-tree. Then, we obtain the following instantaneous description:


CT-NS
s
δ
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\gamma(s)$
α_i
$\sigma(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$

In this situation the program storage of the sdrs machine has the following structure:



By executing the $EVAL\ i$ instruction we obtain the following configuration:


PC
$adr(\alpha_i)$
Runtime stack
Program
Y: $adr(\delta)$:
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$
$adr(\beta)$:
$adr(\delta)$:

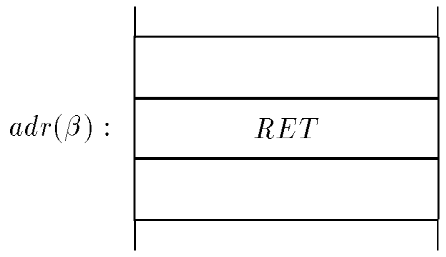
The address $adr(\alpha_i)$ for the evaluation of α_i is written to the program counter and a new Y–block is created on top of the runtime stack. This Y–block contains the return address $adr(\delta)$ and the static link to the environment of α_i. In other words, the static link points to the F-block in which the appropriate addresses are stored for the computation of the values of parameter variables that may occur in α_i.

Case 4: $\beta = \varepsilon$
The top square of the ct-ns includes the empty word ε. That means, the evaluation of the case-alternative is finished and thus, this square can be popped. The parameter list in the new current square contains the bindings of the parameter variables for the previous function call and hence, this list is not needed any longer. We get the following instantaneous description of the ct-ns:

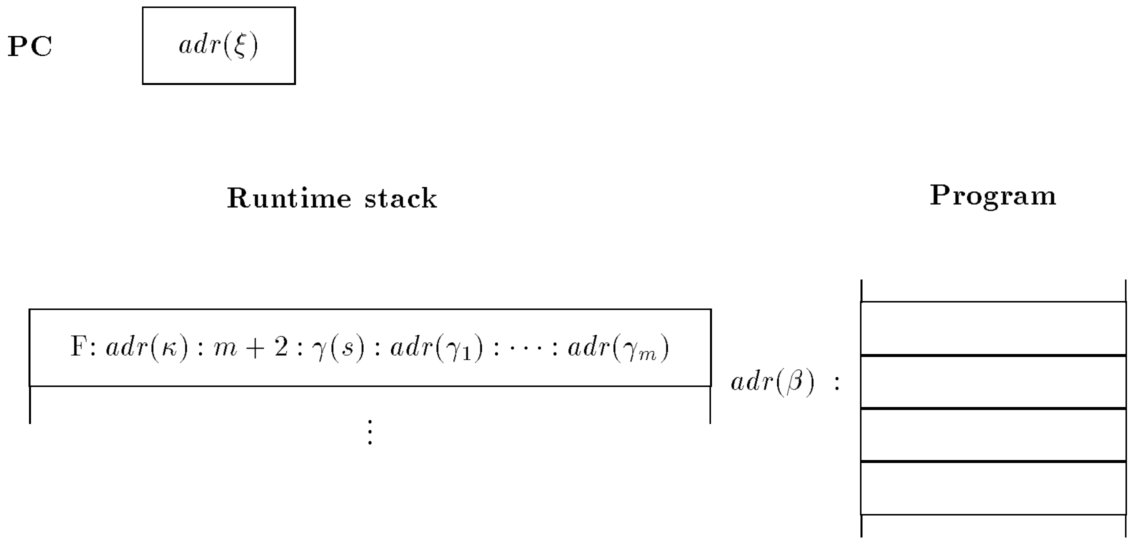
CT-NS

$\gamma(s)$	ξ
$\sigma(\gamma(s))$	$(\gamma_1, \ldots, \gamma_m)\kappa$

In the sdrs machine the program storage has the following structure:

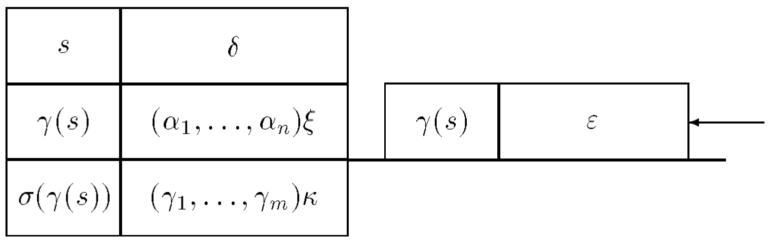


By executing the *RET* instruction we obtain the following configuration:


PC
$adr(\xi)$
Runtime stack
Program
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \cdots : adr(\gamma_m)$
$adr(\beta)$:

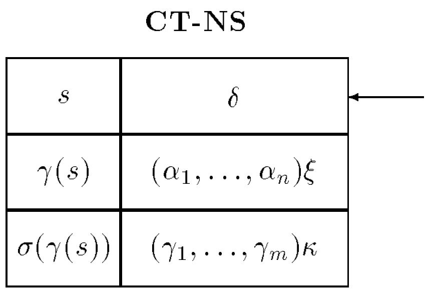
This ends the discussion of the four possible cases of the structure of case-alternatives. However, the attentive reader might have recognized that there is one possible configuration which is not yet included in these four cases. It is an instantaneous description of the ct-ns transducer in which the current square of the ct-ns is the bottom square of a nested-stack and its right component contains the empty word ε. For example:

a) **CT-NS**

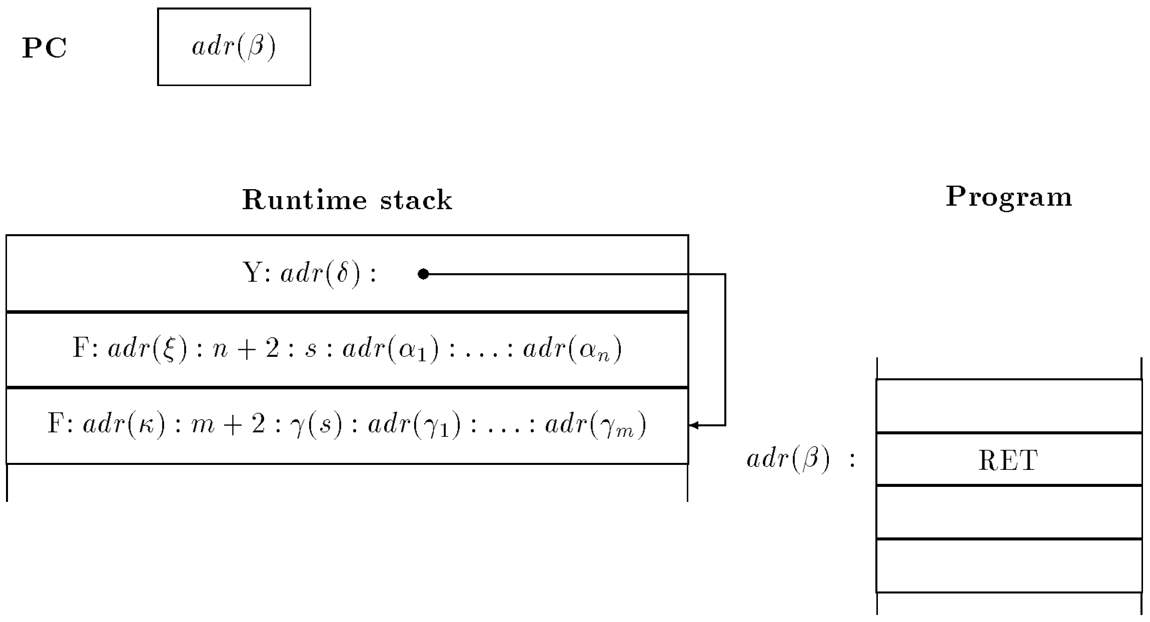

s
δ
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\sigma(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$
$\gamma(s)$
ε

The nested-stack has been created for the evaluation of a parameter expression and now this evaluation is finished. Thus, the nested stack is no longer needed. Therefore, a *destruct*-instruction is executed and, since the new current square is not a top square of a stack, a *moveup*-instruction is executed. We obtain the following instantaneous description:

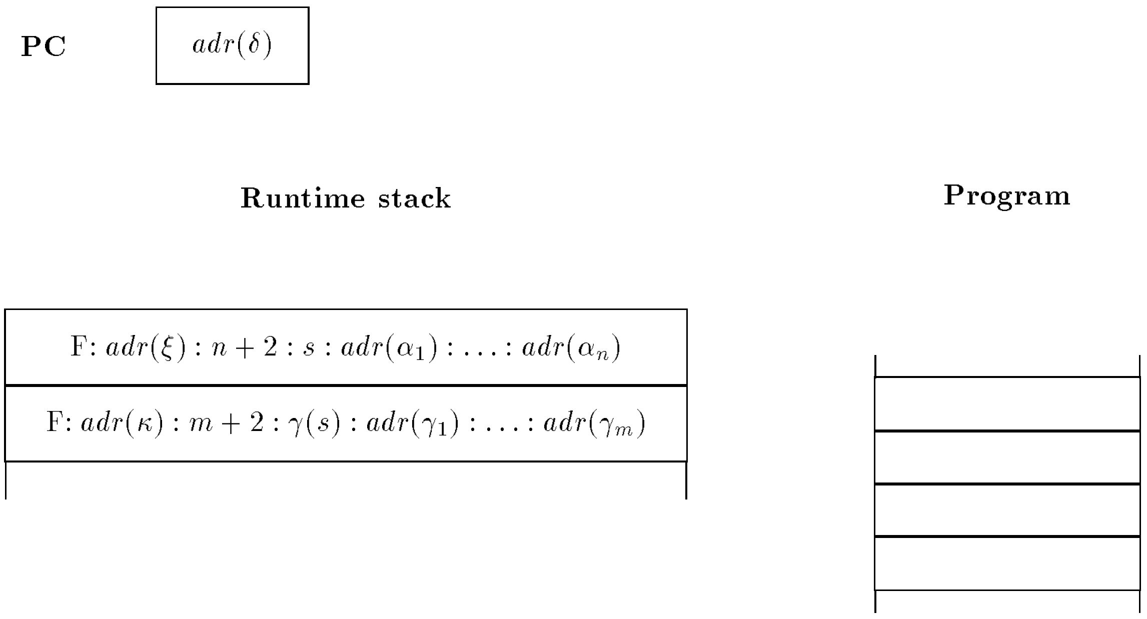
b)


CT-NS
s
δ
$\gamma(s)$
$(\alpha_1, \ldots, \alpha_n)\xi$
$\sigma(\gamma(s))$
$(\gamma_1, \ldots, \gamma_m)\kappa$

As illustrated in Case 3, a nested stack is created in the ct-ns transducer iff a Y–block is created in the sdrs machine. Thus, the following instantaneous description of the sdrs machine corresponds to instantaneous description a) of the ct-ns:


PC
$adr(\beta)$
Runtime stack
Program
Y: $adr(\delta)$:
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$
$adr(\beta)$:
RET

The Y-block has been pushed to the runtime stack for the evaluation of a parameter expression. Now, this evaluation is finished. Thus, the Y-block is deleted by the RET-instruction at address $adr(\beta)$. We get the following instantaneous description of the sdrs machine which corresponds to instantaneous description b) of the ct-ns:


PC
$adr(\delta)$
Runtime stack
Program
F: $adr(\xi) : n+2 : s : adr(\alpha_1) : \ldots : adr(\alpha_n)$
F: $adr(\kappa) : m+2 : \gamma(s) : adr(\gamma_1) : \ldots : adr(\gamma_m)$

At the end of this section we summarize the comparison in the following table:

	ct-ns transducer	sdrs machine
Instructions	*push*	*CREATE*
	pop	*RET* (the top block is an F–block)
	movedown and *create*	EVAL
	destruct and *moveup*	*RET* (the top block is a Y–block)
function-call	push a square	create an F–block
y_i-variable	create a nested stack	create a Y–block
working mode	interpreter-based machine	compiler-based machine

Thus we can conclude a strong similarity between the interpreter-based implementation of sd funprog on ct-ns transducer and the compiler-based implementation on the sdrs machine. Therefore, we can claim that the sdrs machine is a minimal machine for the implementation of sd funprog, because of the characterization of sd funprog by the ct-ns transducer in [EV86]. In particular, it is not possible to implement tree-to-string functions on the sdrs machine which are not in the class of functions that can be computed by sd funprogs.

7 Conclusion

In this paper we have formalized a compiler-based implementation of syntax-directed functional programming on an abstract runtime stack machine. We have compared this implementation with an interpreter-based implementation on ct-ns transducer which was formalized in [FV90]. By means of this comparison it was found out that there is a one-to-one correspondence between the two implementations. In this sense, the sdrs machine is a minimal among all machines on which sd funprog can be implemented.

In [Faß89, FV90] an extension of the implementation on the ct-ns transducer to a call-by-need implementation of sd funprogs was formalized. In [Faß89] the implementation on the sdrs machine is modified to a call-by-need implementation. There, we have to ensure that each parameter variable is evaluated at most once. For this purpose, the value of a parameter expression is written to a local output tape which is associated to the corresponding Y-block. If this Y-block is popped from the runtime stack, the contents of the local output tape replaces the address for the computation of the parameter in the F-block. Then, if the value of the parameter is needed once again, the machine picks it from the F-block.

In our current research we investigate sd funprogs which are enriched by features of logic programming languages, i.e., free variables in goals for which an answer has to be computed. We study narrowing machines on which such functional logic programming languages can be implemented. A note for an interpreter-based machine for such a language is presented in [RV89].

References

[Aho69] A.V. Aho. Nested stack automata. *Journal of Assoc. Comput. Mach.*, 16:383–406, 1969.

[AU71] A.V. Aho and J.D. Ullman. Translations on a context free grammar. *Inform. and Control*, 19:439–475, 1971.

[AU73] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. I and Vol. II.* Prentice-Hall, 1973.

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

[EH89] J. Engelfriet and L.M. Heyker. The term-generating power of context-free hypergraph grammars and attribute grammars. Technical Report 89-17, Rijksuniversiteit te Leiden, Vakgroep Informatica, 1989.

[Eng81] J. Engelfriet. Tree transducers and syntax directed semantics. Technical Report Memorandum 363, Technische Hogeschool Twente, 1981.

[ES84] J. Engelfriet and G. Slutzki. Extended macro grammars and stack controlled machines. *Journal of Computer and System Sciences*, 29:366–408, 1984.

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

[Faß89] H. Faßbender. Implementierung der call-by-need Auswertungsstrategie für macro tree-to-string transducer auf nested-stack Maschinen mit geschachtelten Ausgabebändern, December 1989. Diplomarbeit, RWTH Aachen.

[FV90] H. Faßbender and H. Vogler. A call-by-need implementation of syntax directed functional programming. Technical Report 22, Aachen University of Technology, Fachgruppe Informatik, Ahornstr. 55, W-5100 Aachen, FRG, 1990.

[Gor79] M.J. Gordon. *The denotational description of programming languages, an introduction.* Springer-Verlag, 1979.

[Ind79] K. Indermark. Functional compiler description. *Banach Center Publications*, 21:223–232, 1979.

[Ind85] K. Indermark. Grundlagen funktionaler Programmiersprachen. Lecture Notes at the Aachen University of Technology, 1985.

[Iro61] E.T. Irons. A syntax directed compiler for ALGOL 60. *Comm. Assoc. Comput. Mach.*, 4:51–55, 1961.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968. correction: Math. Syst. Theory 5 (1971) 95-96.

[Kos71] C.H.A. Koster. Affix grammars. In *Proc. of the IFIP working conf. on ALGOL68 implementation*, 1971. Amsterdam, North-Holland.

[RV89] M. Rodriguez-Artalejo and H. Vogler. Note on a narrowing machine for syntax directed BABEL. Technical Report 19, Aachen University of Technology, Fachgruppe Informatik, Ahornstr. 55, W-5100 Aachen, FRG, 1989.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Computers and Automata*, pages 19–46. Wiley, New York, 1971.