[Rou] W.C. Rounds; Mappings and grammars on trees; Math. Syst. Theory 4(1970), 257-287.

[Tha] J.W. Thatcher; Tree automata: an informal survey; in: "Currents in the Theory of Computing", (A.V. Aho, Ed.), Prentice Hall, Englewood Cliffs (1973), 143-172.

[Vog] H. Vogler; Funktionale Programmierung mit primitiver Rekursion – formale Modelle zur Reduktionssemantik; Habilitationsschrift, RWTH Aachen, Oktober 1990.

References

[Aho] A.V. Aho; Nested stack automata; J. Assoc. Comput. Mach. 16 (1969), 383-406.

[AhoUll1] A.V. Aho, J.D. Ullman; The Theory of Parsing, Translation and Compiling, Vol. I and Vol. II; Prentice Hall, Englewood Cliffs 1973.

[AhoUll2] A.V. Aho, J.D. Ullman; Translations on a context free grammar; Inform. and Control 19 (1971), 439-475.

[CouFra] B. Courcelle, P. Franchi-Zannettacci; Attribute grammars and recursive program schemes I, II; Theoret. Comput. Sci. 17 (1982), 163-191 und 235-257.

[Eng1] J. Engelfriet; Bottom-up and top-down tree transformations – a comparison; Math. Syst. Theory 9 (1975), 198-231.

[Eng2] J. Engelfriet; Tree automata and tree grammars; Datalogisk Afdeling report, DAIMI FN-10, Aarhus University (1975).

[Eng3] J. Engelfriet; Some open questions and recent results on tree transducers and tree languages; in: Formal language theory; perspectives and open problems (ed. R.V. Book) New York, Academic Press.

[Eng4] J. Engelfriet; Tree transducers and syntax directed semantics; Memorandum Nr. 363, Technische Hogeschool Twente, 1981.

[Eng5] J. Engelfriet; Iterated pushdown automata and complexity classes; Proc. 15th STOC, April 1983, Boston, 365-373.

[EngSlu] J. Engelfriet, G. Slutzki; Extended macro grammars and stack controlled machines; J. Comput. System Sci. 29 (1984), 366-408.

[EngVog1] J. Engelfriet, H. Vogler; Macro tree transducer; J. Comput. System Sci. 31 (1985), 71-146.

[EngVog2] J. Engelfriet, H. Vogler; Pushdown machines for the macro tree transducer; Theor. Comput. Sci. 42 (1986), 251-368.

[EngVog3] J. Engelfriet, H. Vogler; Modular tree transducers; Aachener Informatik Berichte Nr. 88-22, RWTH Aachen, 1988; to be published in Theoret. Comput. Sci., 1991.

[Faß] H. Faßbender; Implementierung der call-by-need Auswertungsstrategie für Macro Tree-to-String Transducer auf Nested-Stack Maschinen mit geschachtelten Ausgabebändern; Diplomarbeit, RWTH Aachen, Dezember 1989.

[Hud] P. Hudak; Conception, evolution, and application of functional programming languages; ACM Computing Surveys, 21 (1989), 353-411.

[HueOpp] G. Huet, D.C. Oppen; Equation and rewrite rules, a survey; J. Assoc. Comput. Mach. 27 (1980), 797-821.

[Hup] U. Hupach; Rekursive Funktionen in mehrsortigen Algebren; Elektron. Informationsverarb. Kybernetik 15 (1978), 491-506.

[Ind] K. Indermark; Functional compiler description; Banach Center Publications 21 (1979), 223-232.

[Kla1] H.A. Klaeren; Eine Klasse von Algebren mit struktureller Rekursion und ihre Anwendung bei der abstrakten Software-Spezifikation; Dissertation, RWTH Aachen, 1980.

[Kla2] H.A. Klaeren; The SRDL specifcation experiment; in: "Workshop on Program Specification, 1981", LNCS 134 (1981), 282-293.

[Kla3] H.A. Klaeren; A constructive method for abstract algebraic specification; Theoret. Comput. Sci. 30 (1984), 139-204.

[Kla4] H.A. Klaeren; Ein algebraischer Ansatz zur Rekursionselimination; Habilitationsschrift, RWTH Aachen, 1988.

[KlaInd] H.A. Klaeren, K. Indermark; A new implementation technique for recursive function definitions; Aachener Informatik Berichte Nr. 87-10, RWTH Aachen, 1987.

[Klo] J.W. Klop; Term rewriting systems : a tutorial; Bulletin of the EATCS 32 (1987), 143-182.

[Lan] P.J. Landin; The mechanical evaluation of expressions; Computer Journal 6 (1964), 308-320.

[Loo] R. Loogen; Parallele Implementierung funktionaler Programmiersprachen; Informatik Fachberichte 232, Springer Verlag, 1990.

[LoMoKuRo] R. Loogen, J.J. Moreno-Navarro, H. Kuchen, M. Rodriguez-Artalejo; Graph-based implementation of a functional logic language; Aachener Informatik Berichte Nr. 89-20, RWTH Aachen, 1989.

[MorRod] J.J. Moreno-Navarro, M. Rodriguez-Artalejo; Logic programming with functions and predicates: The language BABEL; Dpto. de Informatica y Automatica UCM, Technical Report DIA/89/3, March 1989.

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

[RodVog] M. Rodriguez-Artalejo, H. Vogler; Note on A Narrowing Machine for Syntax Directed BABEL; Aachener Informatik Berichte Nr. 89-19, RWTH Aachen, 1989.

(3) $\stackrel{(e)}{\Longrightarrow}_N$ $q_2((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', Z_0\$, []))$

(4) $\stackrel{(a)(ii)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', q_3(x_1)(y_1, c)\$, []))$

(5) $\stackrel{(e)}{\Longrightarrow}_N$ $q_3((s, a\#(bq_3(x_2)(a,b)), [])(s', (y_1, c), []) \uparrow (\delta, Z_0\$, []))$

(6) $\stackrel{(a)(iii)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), [])(s', (y_1, c), []) \uparrow (\delta, y_2y_2\$, []))$

(7) $\stackrel{(c2)}{\Longrightarrow}_N$ $pr_2((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', (y_1, c), [])(\delta, [2]y_2\$, []))$

(8) $\stackrel{(d)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', ¢c\$, [])(s', (y_1, c), [])(\delta, [2]y_2\$, []))$

(9) $\stackrel{(b)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', c¢\$, [])(s', (y_1, c), [])(\delta, [2]y_2\$, []))$

(10) $\stackrel{(f3)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), []) \uparrow (s', (y_1, c), [0:c])(\delta, [2]y_2\$, []))$

(11) $\stackrel{(f4)}{\Longrightarrow}_N$ $p((s, a\#(bq_3(x_2)(a,b)), [])(s', (y_1, c), [0:c]) \uparrow (\delta, [2]y_2\$, [0:c]))$

(12) $\stackrel{(h)}{\Longrightarrow}_N$ $p((s, ac\#(bq_3(x_2)(a,b)), [])(s', (y_1, c), [0:c]) \uparrow (\delta, y_2\$, [0:c, 2:c]))$

(13) $\stackrel{(c1)}{\Longrightarrow}_N$ $p((s, acc\#(bq_3(x_2)(a,b)), [])(s', (y_1, c), [0:c]) \uparrow (\delta, \$, [0:c, 2:c]))$

(14) $\stackrel{(f1)}{\Longrightarrow}_N$ $p((s, acc\#(bq_3(x_2)(a,b)), []) \uparrow (s', (y_1, c)\$, [0:c]))$

(15) $\stackrel{(g)}{\Longrightarrow}_N$ $p((s, acc\#(bq_3(x_2)(a,b)), []) \uparrow (s', \$, [0:c]))$

(16) $\stackrel{(f1)}{\Longrightarrow}_N$ $p(\uparrow (s, acc\#(bq_3(x_2)(a,b))\$, []))$

(17) $\stackrel{(g)}{\Longrightarrow}_N$ $p(\uparrow (s, acc\#\$, []))$

(18) $\stackrel{(f2)}{\Longrightarrow}_N$ $\underline{acc}$ $\oplus$

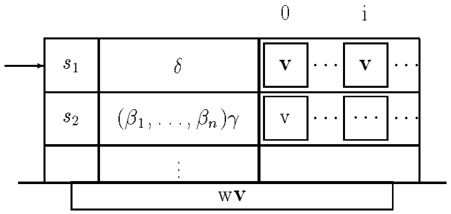
After the formalization of the call-by-need implementation, in the following section, we summarize the results of this paper and give some outviews to some possible future work.

8 Conclusions

In this paper we have formalized the call-by-name implementation of syntax directed functional programs on ct-ns transducers. In order to avoid the recomputation of parameter values and thus, to save computation time, we have introduced the attributed ct-ns transducers and have formalized the call-by-need implementation of syntax directed functional programs on such transducers.

In our current research we investigate narrowing machines on which a restricted type of functional logic programs can be implemented. As starting point, we have chosen the functional logic programming language BABEL [MorRod]; an inplementation of BABEL has been described in [LoMoKuRo]. We are interested in the implementation of the subset of BABEL in which the recursion (inherent in the functional part) is restricted to primitive recursion; we call this subset *"primitive recursive BABEL"*.

As a first step in our investigation, we have considered a subset of primitive recursive BABEL called *"syntax directed BABEL"* [RodVog] which generalizes syntax directed functional programming by allowing logical variables. In fact, ct-ns transducer can serve as a basis for an appropriate narrowing machine. In [RodVog] the ct-ns transducer has been equipped with the possibility for unification and backtracking which leads to the narrowing machine for syntax directed BABEL. Clearly, as for ct-ns transducers, particular storage cells of these narrowing machines can be attributed in order to avoid recomputation of parameters. More importantly, we will focus our research on the development of a narrowing machine which is based on nested stacks and which can serve for implementing primitive recursive BABEL.



If y_i appears again in expression δ, then we copy its value directly to the output tape.

Apart from the evaluation of a parameter variable the call-by-name implementation and the call-by-need implementation of ymts are the same.

After this informal description of the call-by-need implementation, in the next subsection, we construct the act-ns transducer in the described way.

7.2 Construction of the ACT-NS Transducer

The main difference between the call-by-name implementation and the call-by-need implementation is the evaluation of a parameter variable (cf. 5.(c) in Construction 7.1) which have been described in the previous subsection. In the following construction the call-by-need implementation is formalized.

Construction 7.1 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt.
We construct the act-ns transducer $N = (Q', \Sigma, \Gamma, \Delta, q_0', Z_0, R', n_0)$ as follows:

1. $n_0 := maxrank Q - 1$
2. $\Gamma := \{Z_0\} \cup \{\delta \mid$ either $\delta = \alpha$ or $\delta = [i]\alpha$ for some $i \in [n_0]$ and a well-bracketed postfix α of a right-hand side of a rule in $R\}$
3. $Q' := Q \cup \{p\} \cup \{pr_j \mid j \in [n_0]\}$
4. $q_0' := q_0$
5. Construction of R':
   For every $\alpha \in \Gamma$, $Z \in \{¢, \#, \lambda\}$, $d \subseteq [n_0]$ and $\sigma \in \Sigma_j$ with $j \geq 0$:

   (a) For every rule $q(\sigma(x_1,\ldots,x_j))(y_1,\ldots,y_n) \longrightarrow \beta$ in R,
   the rule $q(\sigma(x_1,\ldots,x_j), ZZ_0\$, \emptyset) \longrightarrow p(stay(\beta))$ is in R'.

   (b) For every $a \in \Delta$,
   the rule $p(\sigma(x_1,\ldots,x_j), Za\alpha\$, d) \longrightarrow p(stay(\alpha), write(a))$ is in R'.

   (c) For every $y_k \in \{y_1,\ldots,y_{n_0}\}$, the rules

   $$\begin{cases} (1)\ p(\sigma(x_1,\ldots,x_j), Zy_k\alpha\$, d) \longrightarrow p(stay(\alpha), writefrom(k)), & if\ k \in d \\ (2)\ p(\sigma(x_1,\ldots,x_j), Zy_k\alpha\$, d) \longrightarrow pr_k(stay([k]\alpha), movedown), & if\ k \notin d \end{cases}$$

   are in R'.

   (d) For every state $pr_i \in Q'$ with $i \in [k]$ and $(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$,
   the rule $pr_i(\sigma(x_1,\ldots,x_j), Z(\beta_1,\ldots,\beta_k)\alpha, d) \longrightarrow p(create(\beta_i))$ is in R'.

   (e) For every $q(x_i)(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$, the rule
   $p(\sigma(x_1,\ldots,x_j), Zq(x_i)(\beta_1,\ldots,\beta_k)\alpha\$, d) \longrightarrow q(stay((\beta_1,\ldots,\beta_k)\alpha), push(i, Z_0))$ is in R'.

   Moreover, the rules:

   (f1) $p(\sigma(x_1,\ldots,x_j), \$, d) \longrightarrow p(pop)$

   (f2) $p(\sigma(x_1,\ldots,x_j), \#\$, d) \longrightarrow \lambda$

   (f3) $p(\sigma(x_1,\ldots,x_j), ¢\$, d) \longrightarrow p(destruct)$

   (f4) $p(\sigma(x_1,\ldots,x_j), Z\alpha, d) \longrightarrow p(moveup)$, for every $\alpha \in \Gamma$

   are in R'.

   (g) For $(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$,
   the rule $p(\sigma(x_1,\ldots,x_j), Z(\beta_1,\ldots,\beta_k)\alpha\$, d) \longrightarrow p(stay(\alpha))$ is in R'.

   (h) For $[i]\alpha \in \Gamma$,
   the rule $p(\sigma(x_1,\ldots,x_j), Z[i]\alpha\$, d) \longrightarrow p(stay(\alpha), writein(i))$ is in R'. $\oplus$

We do not verify the correctness of the construction, because it is very similar to the construction of the ct-ns transducer in Section 4. The correctness is worked out in detail in [Faß].

At the end of this section we demonstrate the call-by-need implementation by an example.

Example 7.2 Consider the ymt M_1 of Example 3.3. The computation of M_1 with the input tree $\sigma(\gamma(\delta), \delta)$ (cf. Example 3.6) is simulated by the act-ns transducer N. One should compare this computation with the corresponding computation of the ct-ns transducer in Example 5.2. The ct-ns transducer in Example 5.2 needs <u>21 computation steps</u>. We will see that the act-ns transducer in the present example needs <u>18 computation steps</u>. Here the difference between the computation lengths of the call-by-name and the call-by-need implementations is not very strong, because the evaluation of the parameter variable y_2 is very short. But clearly, one could imagine easily an example in which the gain in efficiency is much bigger.

We present the computation in the same way as in Example 5.2. For better readability, in the environments we only list the values of the defined attributes:

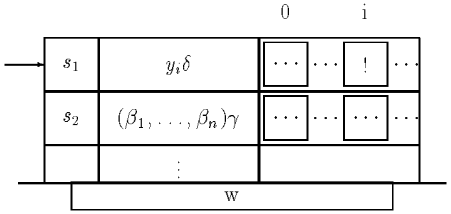
		$q_1(\uparrow(s, \#Z_0\$, []))$
(1)	$\overset{(a)(i)}{\Longrightarrow}_N$	$p(\uparrow(s, \#aq_2(x_1)(bq_3(x_2)(a,b))\$, []))$
(2)	$\overset{(b)}{\Longrightarrow}_N$	$p(\uparrow(s, a\#q_2(x_1)(bq_3(x_2)(a,b))\$, []))$

7 Call-by-need Implementation of YMTs on ACT-NS Transducers

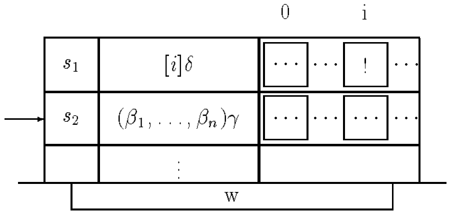
In this section, we formalize the call-by-need implementation of ymts on act-ns transducers. First, we explain the basic idea of this implementation. Then, we will formally define it and we will finish this section by giving an example.

7.1 Informal Description of the Call-by-need Implementation

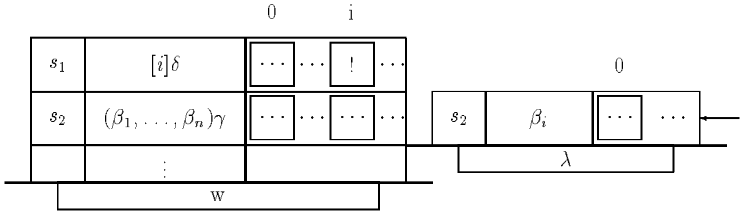
Consider again some arbitrary ymt M. We want to construct an act-ns transducer N which is equivalent to M and which evaluates a parameter variable at most once. For this purpose, if a parameter variable y_i is the prefix symbol in the current stack square, we have to distinguish, whether it has already been evaluated yes or no. If it has been evaluated, then its value is available in attribut i and we can copy this value from the attribut to the output tape of the current stack by means of the instruction writefrom(i). If it has not been evaluated, which is indicated by a '!' in attribute i, then y_i must be evaluated. The evaluation starts with a configuration of the following type:



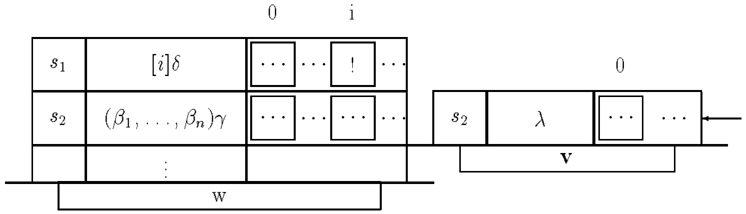
We replace y_i by a mark [i], because we have to know which parameter variable will be evaluated. Then we move down to the square which includes the parameter list.



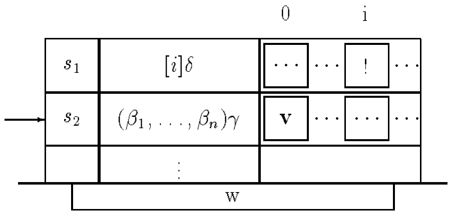
This parameter list includes the binding β_i of y_i. Thus, β_i must be evaluated on a newly created nested stack whose output tape contains the empty word λ.



Now assume that the value of β_i is the string v; due to our convention for write-instructions, v occurs in the output tape of the new nested stack after the completion of the evaluation of β_i.



The evaluation of β_i is finished and thus, the nested stack can be destructed. Simultaneously, the value of β_i is copied from the output tape of the new nested stack to attribute 0 of the current square in the outer stack.



Then we moveup and simultaneously, the value of attribute 0 of the current square is copied to attribute 0 of the square above. In this way, the value of β_i which is also the value of y_i, reaches attribute 0 of the top square of the current stack. By deleting the mark [i] and by copying the value of y_i to attribute i and to the end of the output tape of the current stack, the evaluation of y_i is completed.

$\cup\{(\theta_1\nu\theta_2,\ \theta_1\nu'\theta_2) \mid \nu = \uparrow(s, vZ\alpha\$, f),\ \nu' = \uparrow(s, vaZ\alpha\$, f)$
where $s \in T_\Sigma,\ Z \in \{\#, ¢\}, v \in \Delta^*, \alpha \in \Gamma, f \in Att_n\}$

j) $m(writefrom(j)) :=$
$\{(\theta_1\nu\theta_3\kappa\theta_2,\ \theta_1\nu'\theta_3\kappa\theta_2) \mid \nu = (s, vZ\alpha, f),\ \theta_3 = (s_1, \alpha_1, f_1)\cdots(s_k, \alpha_k, f_k),$
$\kappa = \uparrow(t, \gamma\$, g),\ \nu' = (s, vg(j)Z\alpha, f),$
where $s, t \in T_\Sigma,\ Z \in \{\#, ¢\},\ v \in \Delta^*,\ \alpha, \gamma \in \Gamma,\ f, g \in Att_n,$
$k \geq 0,\ for\ every\ l \in [k] : s_l \in T_\Sigma, \alpha_l \in \Gamma, f_l \in Att_n$
and $for\ every\ l \in [k-1] : s_{l+1}$ is a direct subtree of $s_l\}$
$\cup\{(\theta_1\nu\theta_2,\ \theta_1\nu'\theta_2) \mid \nu = \uparrow(s, vZ\alpha\$, f),\ \nu' = \uparrow(s, vf(j)Z\alpha\$, f)$
where $s \in T_\Sigma,\ Z \in \{\#, ¢\}, v \in \Delta^*, \alpha \in \Gamma, f \in Att_n\}$ $\oplus$

Based on this definition we are able to modify the definition of the ct-ns transducer in Definition 4.2. But now we only add the finite control to the storage configurations, because the output is realized on the act-ns.

Definition 6.2 An *attributed checking-tree nested-stack transducer* (for short: act-ns transducer) is an 8-tuple $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R, n)$ where:

(i) Q is the finite set of states.

(ii) Σ is the ranked alphabet of input symbols.

(iii) Δ is the alphabet of output symbols.

(iv) Γ is the alphabet of stack symbols.

(v) $q_0 \in Q$ is the initial state.

(vi) $Z_0 \in \Gamma$ is the initial stack symbol.

(vii) $n \geq 0$.

(viii) R is a finite set of rules of the form $q(\sigma(x_1, \ldots, x_j), \gamma, d) \longrightarrow \beta$
where $q \in Q,\ \sigma \in \Sigma_j\ with\ j \geq 0,\ \gamma \in Sym(\Gamma),\ d \subseteq [n],$
and $\beta \in \{\lambda\} \cup Q(aI(\Sigma, \Gamma, \Delta, n))$ $\oplus$

The nonnegative integer n provides the number of attributes which are associated with every stack square. In order to be able to use the appropriate write-instructions in the right-hand side of rules, there must be a means for testing whether a parameter variable has been evaluated already yes or no. For this purpose, the state in the left-hand side of a rule has as third argument a set $d \subseteq [n]$, and the rule is applicable only if the indices in d precisely indicate those parameter variables which have already been evaluated. We also note that the right-hand side of a rule is a state with an instruction, there are no terminal symbols preceeding the state.

We complete this section by defining the computation relation of act-ns transducers and the relation computed by an act-ns transducer.

6.3 Semantics of the ACT-NS Transducer

We modify the definition of the computation relation in Definition 4.4. In the following definition the decision, whether a rule may be applied, depends on the root of the current tree, the current stack symbol, and the attributes of the current square which have already been computed.

Definition 6.3 Let $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R, n)$ be an act-ns transducer.
The *computation relation* of M, denoted by $\Longrightarrow_M$, is defined as follows:
For $\beta_1, \beta_2 \in \Delta^* \cup Q(aSC(\Sigma, \Gamma, \Delta, n))$: $\beta_1 \Longrightarrow_M \beta_2$ iff:

(i) $\beta_1 = q(\theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma, f)\theta_2)$
for some $q \in Q,\ \theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma, f)\theta_2 \in aSC(\Sigma, \Gamma, \Delta, n)$.

(ii) If
$$\gamma = \begin{cases} w\gamma' & ,\ \ if\ \gamma' \in \{¢, \#\}\Gamma\{\$, \lambda\}\ and\ w \in \Delta^* \\ \gamma' & ,\ \ if\ \gamma' \in \Gamma\{\$, \lambda\} \end{cases}$$
and there exists a rule in R of the form either
(a) $q(\sigma(x_1, \ldots, x_j), \gamma', d) \longrightarrow q'(op),$
where $q' \in Q,\ op \in aI(\Sigma, \Gamma, \Delta, n)$ and $d = \{j \mid j \in [n]\ and\ f(j) \in \Delta^*\}$
or
(b) $q(\sigma(x_1, \ldots, x_j), \gamma', d) \longrightarrow \lambda$
where $d = \{j \mid j \in [n]\ and\ f(j) \in \Delta^*\}$

(iii) If the rule has the form (a), then $\beta_2 = q'(m(op)(\theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma, f)\theta_2))$.
If the rule has the form (b), then $\beta_2 = v$, where $\theta_1 = \theta_2 = \varepsilon$ and $\gamma = v\#\$$. $\oplus$

At the end of this section, we give the definition of the relation computed by an act-ns transducer and its computation forms (cf. Definition 4.5).

Definition 6.4 Let $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R, n)$ be an act-ns transducer.
The *relation computed by M,* denoted by $\tau(M)$, is the set

$$\{(s, w) \in T_\Sigma \times \Delta^* \mid\ q_0(\uparrow(s, \#Z_0\$, f)) \Longrightarrow^*_M w,\ where\ f \in Att_n,\ such\ that\ for\ every\ j \in [n]:\ f(j) = !\}$$

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

$$\{\xi \mid \xi \in \Delta^* \cup Q(aSC(\Sigma, \Gamma, \Delta, n))\ and\ q_0(\uparrow(s, \#Z_0\$, f)) \Longrightarrow^*_M \xi,\ for\ some\ s \in T_\Sigma\ and\ f \in Att_n,\ such\ that\ for\ every\ j \in [n]:\ f(j) = !\}$$ $\oplus$

In this section, we do not want to present an example, because the only purpose of having added the attribute mechanism is to be able to define a call-by-need implementation of ymts. Thus, we give an example after the definition of this implementation in the following section.

$push(i,\gamma)$: All attributes of the new square are initialized with '!'.

$moveup$: The value of attribute 0 of the current square is copied to attribute 0 of the square above.

$create(\gamma)$: All attributes of the new square are initialized with the values of the attributes of the current square.

$stay(\gamma)$, pop, $movedown$ do not change the values of the attributes.

$destruct$: The contents of the output tape of the destructed stack is copied to attribute 0 of the current square in the outer stack.

$write-instructions$: $j > 0, a \in \Delta$

- $writein(j)$: The value of attribute 0 of the current square is copied to attribute j and to the output tape of the current stack.
- $writefrom(j)$: The value of attribute j is copied to the output tape of the current stack.
- $write(a)$: a is written to the output tape of the current stack.

After this informal description of the act-ns transducer, we will define them formally in the next subsection.

6.2 Definition of the Attributed CT-NS Transducer

We start with the modification of the definition of the storage configurations, the instructions, and their meaning in Definition 4.1.

Definition 6.1 Let Γ be a fixed alphabet of stack symbols, Σ be a ranked alphabet of input symbols, Δ be an alphabet of output symbols, and let $n \geq 0$.

(i) The set of *storage configurations* of the act-ns, denoted by $aSC(\Sigma,\Gamma,\Delta,n)$, is the set

$aNS(\Sigma,\Gamma,\Delta,n)^* \uparrow aNS(\Sigma,\Gamma,\Delta,n)^+$ where $aNS(\Sigma,\Gamma,\Delta,n) = T_\Sigma \times Sym(\Gamma,\Delta) \times Att_n$, $Sym(\Gamma,\Delta) = \Gamma_\Delta \cup \Gamma_\Delta\$$ with $\Gamma_\Delta = \Gamma \cup \Delta^*¢\Gamma \cup \Delta^*\#\Gamma$, and $Att_n = \{f \mid f : \{0,\ldots,n\} \to \Delta^* \cup \{!\}\}$ for some symbols $\uparrow$, $\#$, ¢, $\$ \notin \Gamma$ and $! \notin \Delta$

(ii) The set of *instructions* on the act-ns, denoted by $aI(\Sigma,\Gamma,\Delta,n)$, is the set

$$\{push(i,\gamma), pop, moveup, movedown, create(\gamma), destruct, stay(\gamma), writein(j), write(a), writefrom(j) \mid i \in [maxrank\Sigma],\ \gamma \in \Gamma,\ j \in [n],\ a \in \Delta\}$$

(iii) For every $u \in aI(\Sigma,\Gamma,\Delta,n)$ we describe the *meaning of u,* denoted by $m(u)$, as a binary relation on $aSC(\Sigma,\Gamma,\Delta,n)$ as follows: Let $\theta_1,\theta_2 \in aNS(\Sigma,\Gamma,\Delta,n)^*$

a) $\mathrm{m}(push(i,\gamma)) :=$

$$\begin{aligned}\{(\theta_1\nu\theta_2, \theta_1\nu'\kappa\theta_2) \mid\ & \nu = \uparrow(s,\alpha\$,f),\ \nu' = (s,\alpha,f),\ \kappa = (s_i,\gamma\$,g),\\ & \text{where } s,s_i \in T_\Sigma,\ \alpha \in \Gamma_\Delta\ ,\ s_i \text{ is the i-th direct subtree of } s\\ & \text{and } f,g \in Att_n \text{ such that, for every } j \in [n]\ :\ g(j) =!\}\end{aligned}$$

b) $\mathrm{m}(pop) :=$

$$\begin{aligned}\{(\theta_1\nu\kappa\theta_2, \theta_1\nu'\theta_2) \mid\ & \kappa = \uparrow(s',\gamma\$,g),\ \nu = (s,\alpha,f),\ \nu' = \uparrow(s,\alpha\$,f)\\ & \text{where } s,s' \in T_\Sigma,\ \gamma \in \Gamma,\ \alpha \in \Gamma_\Delta,\\ & s' \text{ is a direct subtree of s and } f,g \in Att_n\}\end{aligned}$$

c) $\mathrm{m}(moveup) :=$

$$\begin{aligned}\{(\theta_1\nu\kappa\theta_2, \theta_1\nu'\kappa'\theta_2) \mid\ & \nu' = (s,\alpha,f),\ \nu = \uparrow\nu',\ \kappa = (t,\gamma,g),\ \kappa' = \uparrow(t,\gamma,g'),\\ & \text{where } \gamma \in Sym(\Gamma,\Delta)\backslash(\Delta^*\#\Gamma\ \cup\ \Delta^*\#\Gamma\$),\\ & s,t \in T_\Sigma\ ,\ \alpha \in \Gamma_\Delta\ ,\ f,g,g' \in Att_n \text{ with } g' = g[0/f(0)]\}\end{aligned}$$

d) $\mathrm{m}(movedown) :=$

$$\begin{aligned}\{(\theta_1\nu\kappa\theta_2, \theta_1\nu'\kappa'\theta_2) \mid\ & \nu = (s,\alpha,f),\ \nu' = \uparrow\nu,\ \kappa = \uparrow(s',\gamma,g),\ \kappa' = (s',\gamma,g),\\ & \text{where } s,s' \in T_\Sigma,\ \alpha \in \Gamma_\Delta,\ f,g \in Att_n\\ & \text{and } \gamma \in Sym(\Gamma,\Delta)\backslash(\Delta^*\#\Gamma\ \cup\ \Delta^*\#\Gamma\$)\}\end{aligned}$$

e) $\mathrm{m}(create(\gamma)) :=$

$$\begin{aligned}\{(\theta_1\nu\theta_2, \theta_1\kappa\nu'\theta_2) \mid\ & \nu = \uparrow(s,\alpha,f),\ \nu' = (s,\alpha,f),\ \kappa = \uparrow(s,\lambda¢\gamma\$,f),\\ & \text{where } s \in T_\Sigma,\ \alpha \in Sym(\Gamma,\Delta) \text{ and } f \in Att_n\}\end{aligned}$$

f) $\mathrm{m}(destruct) :=$

$$\begin{aligned}\{(\theta_1\nu\kappa\theta_2, \theta_1\kappa'\theta_2) \mid\ & \kappa' = \uparrow(s,\gamma,g'),\ \kappa = (s,\gamma,g),\ \nu = \uparrow(s,\alpha,f),\\ & \text{where } s \in T_\Sigma,\ \gamma \in Sym(\Gamma,\Delta),\ \alpha = w¢\beta\$ \text{ with } w \in \Delta^*,\\ & \beta \in \Gamma \text{ and } f,g,g' \in Att_n \text{ with } g' = g[0/w]\}\end{aligned}$$

g) $\mathrm{m}(stay(\gamma)) :=$

$$\begin{aligned}\{(\theta_1\nu\theta_2, \theta_1\nu'\theta_2) \mid\ & \nu = \uparrow(s,Z\delta\$,f),\ \nu' = (s,Z\gamma\$,f),\\ & \text{where } s \in T_\Sigma,\ \delta \in \Gamma,\ f \in Att_n\\ & \text{and } Z \in \{\lambda\} \cup \{w¢ \mid w \in \Delta^*\} \cup \{w\# \mid w \in \Delta^*\}\}\end{aligned}$$

h) $\mathrm{m}(writein(j)) :=$

$$\begin{aligned}\{(\theta_1\nu\theta_3\kappa\theta_2,\ \theta_1\nu'\theta_3\kappa'\theta_2) \mid\ & \nu = (s,vZ\alpha,f),\ \theta_3 = (s_1,\alpha_1,f_1)\cdots(s_k,\alpha_k,f_k),\\ & \kappa = \uparrow(t,\gamma\$,g),\ \nu' = (s,vg(0)Z\alpha,f),\ \kappa' = \uparrow(t,\gamma\$,g[j/g(0)]),\\ & \text{where } s,t \in T_\Sigma,\ Z \in \{\#,¢\},\ v \in \Delta^*,\ \alpha,\gamma \in \Gamma,\\ & k \geq 0,\ for\ every\ l \in [k] : s_l \in T_\Sigma, \alpha_l \in \Gamma, f_l \in Att_n\\ & \text{and } for\ every\ l \in [k-1] : s_{l+1} \text{ is a direct subtree of } s_l\}\end{aligned}$$

$$\begin{aligned}\cup\{(\theta_1\nu\theta_2,\ \theta_1\nu'\theta_2) \mid\ & \nu = \uparrow(s,vZ\alpha\$,f),\ \nu' = \uparrow(s,vf(0)Z\alpha\$,f[j/f(0)]),\\ & \text{where } s \in T_\Sigma,\ Z \in \{\#,¢\}, v \in \Delta^*, \alpha \in \Gamma, f \in Att_n\}\end{aligned}$$

i) $\mathrm{m}(write(a)) :=$

$$\begin{aligned}\{(\theta_1\nu\theta_3\kappa\theta_2,\ \theta_1\nu'\theta_3\kappa\theta_2) \mid\ & \nu = (s,vZ\alpha,f),\ \theta_3 = (s_1,\alpha_1,f_1)\cdots(s_k,\alpha_k,f_k),\\ & \kappa = \uparrow(t,\gamma\$,g),\ \nu' = (s,vaZ\alpha,f),\\ & \text{where } s,t \in T_\Sigma,\ Z \in \{\#,¢\},\ v \in \Delta^*,\ \alpha,\gamma \in \Gamma,\ f,g \in Att_n,\\ & k \geq 0,\ for\ every\ l \in [k] : s_l \in T_\Sigma, \alpha_l \in \Gamma, f_l \in Att_n\\ & \text{and } for\ every\ l \in [k-1] : s_{l+1} \text{ is a direct subtree of } s_l\}\end{aligned}$$

6 Attributed CT-NS Transducer

After having formalized the call-by-name implementation of ymts on ct-ns transducers, we now turn to the call-by-need implementation of ymts on *attributed checking-tree nested-stack transducers,* for short act-ns transducer. There, every parameter variable of the right-hand side of a rule of the ymt is evaluated at most once. The act-ns transducer gains this efficiency by adding a finite list of attributes (also called registers) to every stack square. Then, as soon as the value of a parameter variable is computed, it is stored in the corresponding attribute where it is available for later use. In this section, we describe the way in which attributes are incorporated into ct-ns transducers.

6.1 Informal Description of the ACT-NS Transducer

An example configuration of an act-ns transducer is shown in Figure 6 (cf. also Figure 5) where every square contains an input subtree t_i, an expression γ_j, and some finite list f_j of attributes to be explained later.

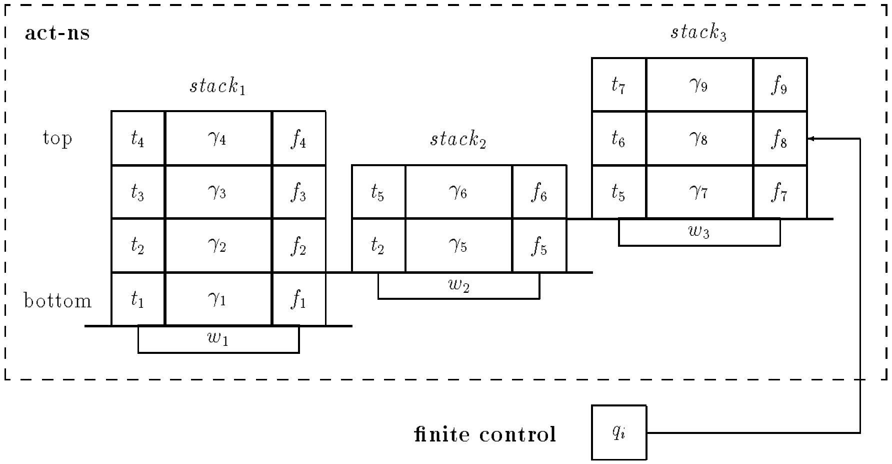


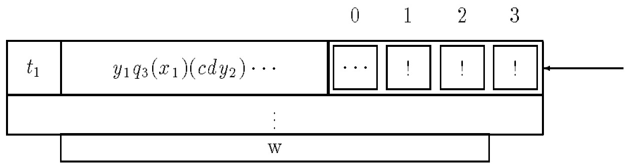
Figure 6: An Attributed Checking-Tree Nested-Stack Transducer.

There are two important differences between the ct-ns transducer and the act-ns transducer.

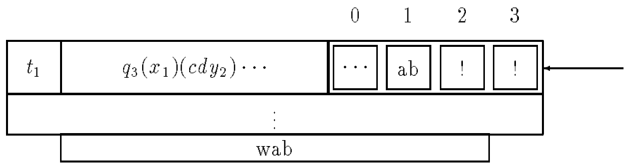
1. The act-ns transducer has no separate output tape, rather, every stack has its own output tape. In the configuration shown in Figure 6 the output tapes of the stacks $stack_1$, $stack_2$, and $stack_3$ contain w_1, w_2, and w_3, respectively. The final output of the machine is computed in the output tape of the outermost stack. At every point

of the computation, the act-ns transducer can only write to the output tape of the stack in which the current stack square lies. Thus, in our example, output would be transferred to the output tape of stack $stack_3$. In the call-by-need implementation the value of the parameter variable which is evaluated by a nested stack, is written into the output tape of this nested stack.

2. In the act-ns transducer, for every stack square sq, a finite list f_j of attributes is added to sq. The attribute in position 0 serves as a kind of unidirectional data bus on which strings are transported during moveup instructions (this will become more clear in Subsection 7.1). The attribute in position i (with $i \geq 1$) serves for storing the value of parameter variable y_i. If a parameter variable has not been evaluated, then the corresponding attribute includes the special symbol '!'. To illustrate the joint work of attributes and output tapes assume that, during the computation of the act-ns transducer which simulates a ymt, the postfix $y_1 q_3(x_1)(cdy_2)\cdots$ is stored in the current square. Moreover, assume that no parameter variable has been computed yet. In this case we have the following configuration of the current stack:



In the next computation steps y_1 is evaluated and its value, for example ab, is remembered in attribute 1 and written to the output tape of the current nested stack:

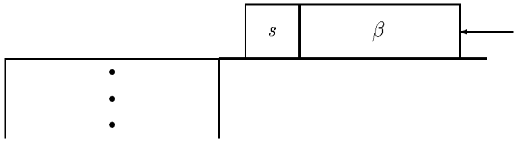


In the formal definition we denote the configurations in a linear form in analogy to the configurations of the ct-ns transducer. The contents of the output tapes are written left to the bottom squares. Thus, the configuration in Figure 6 has the following linear form:

$$q_i(\ (t_1, w_1\#\gamma_1, f_1)(t_2, w_2¢\gamma_5, f_5)(t_5, w_3¢\gamma_7, f_7) \uparrow (t_6, \gamma_8, f_8)(t_7, \gamma_9\$, f_9)$$
$$(t_5, \gamma_6\$, f_6)(t_2, \gamma_2, f_2)(t_3, \gamma_3, f_3)(t_4, \gamma_4\$, f_4))$$

At the end of the informal description of the act-ns transducer we explain the differences of the semantics of the instructions:

elements which might be considered as pruned working computation forms. One can think of these elements as illustrated in the following picture



The part of the outer stack above the current square is pruned. Such elements are needed to bind the free variables in β. Hence, the domain of g is defined by

$$dom(g) \ := \ WCF(N) \ \cup \ \{\xi \mid \xi = wp(\theta_1 \uparrow (s, ¢\beta\$)), \ where \ \ for \ some \ \theta_2 \in NS(\Sigma,\Gamma)^+ : \ wp(\theta_1 \uparrow (s, ¢\beta\$)\theta_2) \in WCF(N)\}$$

Intuitively, for a given working computation form, g constructs a computation form of $CS(M)$ by putting the expression which is stored in the right component of the current stack square into the left component of the corresponding closure and by recursively constructing the environment.

Definition 5.9 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt and N be the ct-ns transducer, which is constructed in Construction 5.1. If not stated otherwise, we let Z, Z', and Z^* be variables which are universally quantified over $\{\lambda, ¢, \#\}$.

The mapping $g : dom(g) \longrightarrow CF(CS(M))$ is defined as follows:

1. $g(w) \ := \ w$ for every $w \in \Delta^*$
2. $g(q_0(\uparrow (s, \#Z_0\$))) \ := \ (q_0(x_1), [x_1/s])$ for every $s \in T_\Sigma$
3. $g(wp(\theta)) \ := \ wg(p(\theta))$ for every $w \in \Delta^*$ and $\theta \in SC(\Sigma,\Gamma)$
4. $g(p(\uparrow (\sigma(s_1,\ldots,s_j), Z\beta\$))) \ := \ (\beta, [x_1/s_1,\ldots,x_j/s_j])$ for every $Z \in \{¢, \#\}$
5. $g(p(\theta_1(s', Z'(\beta_1,\ldots,\beta_n)\alpha) \uparrow (\sigma(s_1,\ldots,s_j), Z\beta'\$)\theta_2))$

$$:= \begin{cases} (a) \ (\beta', [x_1/s_1,\ldots,x_j/s_j, y_1/g(p(\theta_1 \uparrow (s', ¢\beta_1\$))),\ldots,y_n/g(p(\theta_1 \uparrow (s', ¢\beta_n\$)))]) \\ \quad \cdot g(p(\theta_1 \uparrow (s', Z'\alpha\$)\theta_2)), \hfill if \ Z = \lambda \\ (b) \ (\beta', [x_1/s_1,\ldots,x_j/s_j, y_1/g(p(\theta_1 \uparrow (s', ¢\beta_1\$))),\ldots,y_n/g(p(\theta_1 \uparrow (s', ¢\beta_n\$)))]), \\ \hfill if \ Z = ¢ \wedge \theta_2 = \lambda \\ (c) \ g(p(\theta_1(s', Z'(\beta_1,\ldots,\beta_n)\alpha) \uparrow (\sigma(s_1,\ldots,s_j), ¢\beta'\$))) \\ \quad \cdot g(p(\theta_1(s', Z'(\beta_1,\ldots,\beta_n)\alpha)\vartheta_1 \uparrow (s^*, Z^*\kappa\$)\theta_2')), \\ \quad if \ Z = ¢ \ and \ \theta_2 = \vartheta_1(s^*, Z^*\kappa\$)\theta_2', \ for \ some \ (s^*, Z^*\kappa\$), \ and \ \vartheta_1 \in NS(\Sigma,\Gamma) \end{cases}$$

$\oplus$

After the introduction of the technical tools, we are going to prove the two inclusions.

5.2.3 The Inclusions $\tau(M) \subseteq \tau(N)$ and $\tau(N) \subseteq \tau(M)$

To verify the first inclusion we have to show the following claim

> For every $s \in T_\Sigma$ and $w \in \Delta^*$, if there exists a computation $q_0(s) \Longrightarrow_M^* w$ by M, then there exists a computation $q_0(\uparrow (s, \#Z_0\$)) \Longrightarrow_N^* w$ by N. (1)

In order to prove Claim 1, we inspect every single computation step of the ymt and simulate it by a finite number of computation steps of N. This proof is divided into two parts. First, we show that every computation step of M can be simulated by a finite number of computation steps of CS(M) and then we show that every computation step of CS(M) can be simulated by a finite number of computation steps of N. Intuitively, these claims will be given in such a form that the involved single computation steps can be composed to a complete computation. Then, Claim 1 follows immediately.

To verify that every computation step of M can be simulated by a finite number of computation steps of CS(M), we prove the following claim by induction

> For every $\varphi, \psi \in CF(M)$: if $\varphi \Longrightarrow_M \psi$, then, for every $\alpha \in CF(CS(M))$ where $h(\alpha) = \varphi$, there exists a $\gamma \in CF(CS(M))$ such that $h(\gamma) = \psi$ and $\alpha \Longrightarrow_{CS(M)}^* \gamma$. (2)

In the second step we have to show that every derivation step of CS(M) can be simulated by a finite number of derivation steps of N. That means, that we have to show the following claim:

> For every $\varphi, \psi \in CF(CS(M))$: if $\varphi \Longrightarrow_{CS(M)} \psi$, then for every $\alpha \in WCF(N)$ with $g(\alpha) = \varphi$, there exists a $\gamma \in WCF(N)$ with $g(\gamma) = \psi$ and $\alpha \Longrightarrow_N^* \gamma$. (3)

This claim is shown by induction, too. We obtain the correctness of Claim 1 by a straightforward induction on the length of computation using Claims 2 and 3. So the inclusion $\tau(M) \subseteq \tau(N)$ is verified.

To verify the inverse inclusion we have to show the following claim:

> If there exists a derivation $q_0(\uparrow (s, \#Z_0\$)) \Longrightarrow_N^* w$ *where* $s \in T_\Sigma, w \in \Delta^*$ by N, then there exists a derivation $q_0(s) \Longrightarrow_M^* w$ by M. (4)

For this purpose, we verify the following two claims by induction:

> For every $\varphi, \psi \in WCF(N)$: if $\varphi \Longrightarrow_N \psi$ or $\varphi \Longrightarrow_N \nu \Longrightarrow_N \psi$ where $\nu \notin WCF(N)$, then: $g(\varphi) \Longrightarrow_{CS(M)} g(\psi)$. (5)

> For every $\varphi, \psi \in CF(CS(M))$: if $\varphi \Longrightarrow_{CS(M)} \psi$, then: $h(\varphi) = h(\psi)$ or $h(\varphi) \Longrightarrow_M h(\psi)$. (6)

Claim 4 is proved by induction on the length of derivations of N by using Claims 5 and 6. Thus, we obtain the correctness of the inclusion $\tau(N) \subseteq \tau(M)$ and we have finished the verification of the correctness of the construction. The inductive proofs of the various claims can be found in [Faß].

Now we are able to present the closure computation relation of M.

Definition 5.5 Let M=$(Q, \Sigma, \Delta, q_0, R)$ be a ymt.

The *closure computation relation,* denoted by $\Longrightarrow_{CS(M)}$, is the smallest binary relation $\Longrightarrow \subseteq \Delta^* C(M)^* \times \Delta^* C(M)^*$ such that:

for every $w \in \Delta^*$, $q \in Q_{n+1}$ with $n \geq 0$, $\sigma \in \Sigma_j$ with $j \geq 0$, $u \in \underline{Env}(q, \sigma)$, well-bracketed postfix γ of a (q, σ)-right-hand side, and $\delta \in C(M)^*$ either of the following four cases hold:

1. If $\gamma = \lambda$, then $w(\lambda, u)\delta \implies w\delta$.
2. If $\gamma = a\gamma'$ for some $a \in \Delta$, then $w(a\gamma', u)\delta \implies wa(\gamma', u)\delta$.
3. If $\gamma = p(x_k)(\beta_1, \ldots, \beta_m)\gamma'$ for some $p \in Q_{m+1}$, $k \in [j]$, $\beta_1, \ldots, \beta_m \in RHS(M)$, and if $u(x_k) = \pi(s_1, \ldots, s_r)$ and there is a rule $p(\pi(x_1, \ldots, x_r))(y_1, \ldots, y_m) \longrightarrow \beta$ in R, then $w(p(x_k)(\beta_1, \ldots, \beta_m)\gamma', u)\delta$
$$\implies w(\beta,\ [x_1/s_1, \ldots, x_r/s_r, y_1/(\beta_1, u), \ldots, y_r/(\beta_r, u)])(\gamma', u)\delta.$$
4. If $\gamma = y_k\gamma'$ with $k \in [n]$ and if $u(y_k) = \beta_k$, then $w(y_k\gamma', u)\delta \implies w\beta_k(\gamma', u)\delta$. ⊕

In analogy to Definition 3.5, we define the relation computed by $CS(M)$ and the set of computation forms for closure semantics as follows:

Definition 5.6 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt.

The *relation computed by $CS(M)$ with call-by-name computation strategy,* denoted by $\tau(CS(M))$, is the set

$$\{(s, w) \in T_\Sigma \times \Delta^* \mid (q_0(x_1), [x_1/s]) \Longrightarrow^*_{CS(M)} w\}$$

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

$$\{\xi \mid \xi \in \Delta^* C(M)^* \textit{ and } (q_0(x_1), [x_1/s]) \Longrightarrow^*_{CS(M)} \xi \textit{ for some } s \in T_\Sigma\}$$

⊕

We explain the previous definitions by giving an example of the closure semantics.

Example 5.7 For the sake of clarity, we only list in the environments the bindings of the free variables that appear in the expression in the left component. The computation of the ymt M_1 (cf. Example 3.3) with input tree $\sigma(\gamma(\delta), \delta)$ in Example 3.6 is simulated by $CS := CS(M_1)$ as follows

$$
\begin{array}{lll}
 & & (q_1(x_1), [x_1/s]) \\
(1) & \overset{(3)(i)}{\Longrightarrow}_{CS} & (aq_2(x_1)(bq_3(x_2)(a, b)), [x_1/s', x_2/\delta])(\lambda, []) \\
(2) & \overset{(2)}{\Longrightarrow}_{CS} & \underline{a}\ (q_2(x_1)(bq_3(x_2)(a, b)), [x_1/s', x_2/\delta])(\lambda, []) \\
(3) & \overset{(3)(ii)}{\Longrightarrow}_{CS} & \underline{a}\ (q_3(x_1)(y_1, c), [x_1/\delta, y_1/(bq_3(x_2)(a, b), [x_2/\delta])])(\lambda, [])(\lambda, []) \\
(4) & \overset{(3)(iii)}{\Longrightarrow}_{CS} & \underline{a}\ (y_2y_2, [y_2/(c, [])])(\lambda, [])(\lambda, [])(\lambda, []) \\
(5) & \overset{(4)}{\Longrightarrow}_{CS} & \underline{a}\ (c, [])(y_2, [y_2/(c, [])])(\lambda, [])(\lambda, [])(\lambda, []) \\
(6) & \overset{(2)}{\Longrightarrow}_{CS} & \underline{ac}\ (\lambda, [])(y_2, [y_2/(c, [])])(\lambda, [])(\lambda, [])(\lambda, []) \\
(7) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{ac}\ (y_2, [y_2/(c, [])])(\lambda, [])(\lambda, [])(\lambda, []) \\
(8) & \overset{(4)}{\Longrightarrow}_{CS} & \underline{ac}\ (c, [])(\lambda, [])(\lambda, [])(\lambda, [])(\lambda, []) \\
(9) & \overset{(2)}{\Longrightarrow}_{CS} & \underline{acc}\ (\lambda, [])(\lambda, [])(\lambda, [])(\lambda, [])(\lambda, []) \\
(10) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{acc}\ (\lambda, [])(\lambda, [])(\lambda, [])(\lambda, []) \\
(11) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{acc}\ (\lambda, [])(\lambda, [])(\lambda, []) \\
(12) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{acc}\ (\lambda, [])(\lambda, []) \\
(13) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{acc}\ (\lambda, []) \\
(14) & \overset{(1)}{\Longrightarrow}_{CS} & \underline{acc}
\end{array}
$$

The interested reader may compare this computation with the computation of the ct-ns transducer in Example 5.2. ⊕

After the definition of the closure semantics we introduce two mappings g and h, where g transforms every computation form of the constructed ct-ns transducer N into the corresponding computation form of the closure semantics, and h transforms every computation form of the closure semantics into the corresponding computation form of the ymt M.

5.2.2 The Mappings g and h

We start with the definition of h. The transformation realized by h is done by substituting for the free variables in the left component of a closure cl the corresponding expressions that are stored in the right component of cl.

Definition 5.8 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt.

The mapping $h : \Delta^* C(M)^* \longrightarrow \Omega^*$ is defined by

1. $h(w\gamma) := wh(\gamma)$ for every $w \in \Delta^*$ and $\gamma \in C(M)^*$
2. $h(\lambda) := \lambda$
3. $h((\beta, [x_1/s_1, \ldots, x_j/s_j, y_1/(\beta_1, u_1), \ldots, y_n/(\beta_n, u_n)])\gamma)$
$:= \beta[x_1/s_1, \ldots, x_j/s_j, y_1/h((\beta_1, u_1)), \ldots, y_n/h((\beta_n, u_n))]h(\gamma)$ for every $(\beta, [x_1/s_1, \ldots, x_j/s_j, y_1/(\beta_1, u_1), \ldots, y_n/(\beta_n, u_n)]) \in C(M)$ and $\gamma \in C(M)^*$. ⊕

Now we present the mapping g which is more complicate than h. We define g only on a special subset $WCF(N) \subset CF(N)$ of the computation forms of N, called the *working computation forms.* These are all computation forms, where the current square is the top square of a stack and the state in the finite control is the working state p. Furthermore, $WCF(N)$ contains the start and stop configurations of N. Now, in order to define the image of a working computation form under g, it is technically convenient to extend the domain of g by

Note that in (c) and (e) two instructions are applied in one rule. It is easy to simulate these rules by introducing (finitely many) auxiliary states and by applying the instructions one by one. The rule (f2) serves as final rule. Also note that N is deterministic in the sense that at no point during a computation of N there is a choice of rules. N is total in the sense that no computation blocks because of lack of rules.

Before we will explain the proof of the correctness of this construction we give an example:

Example 5.2 Consider the ymt M_1 of Example 3.3. The computation of M_1 with the input tree $\sigma(\gamma(\delta),\delta)$ (cf. Example 3.6) is simulated by the ct-ns transducer N. The type of the applied rule is written as a superscript to the computation arrow. If the applied rule is of type (a), we also indicate the number of the corresponding rule of the simulated ymt. We abbreviate the input tree $\sigma(\gamma(\delta),\delta)$ and its first direct subtree $\gamma(\delta)$ by s and s', respectively.

$$\begin{array}{lll}
 & & q_1(\uparrow(s,\#Z_0\$)) \\
(1) & \overset{(a)(i)}{\Longrightarrow}_N & p(\uparrow(s,\#aq_2(x_1)(bq_3(x_2)(a,b))\$)) \\
(2) & \overset{(b)}{\Longrightarrow}_N & \underline{a}\ p(\uparrow(s,\#q_2(x_1)(bq_3(x_2)(a,b))\$)) \\
(3) & \overset{(e)}{\Longrightarrow}_N & \underline{a}\ q_2((s,\#(bq_3(x_2)(a,b)))\uparrow(s',Z_0\$)) \\
(4) & \overset{(a)(ii)}{\Longrightarrow}_N & \underline{a}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',q_3(x_1)(y_1,c)\$)) \\
(5) & \overset{(e)}{\Longrightarrow}_N & \underline{a}\ q_3((s,\#(bq_3(x_2)(a,b)))(s',(y_1,c))\uparrow(\delta,Z_0\$)) \\
(6) & \overset{(a)(iii)}{\Longrightarrow}_N & \underline{a}\ p((s,\#(bq_3(x_2)(a,b)))(s',(y_1,c))\uparrow(\delta,y_2y_2\$)) \\
(7) & \overset{(c)}{\Longrightarrow}_N & \underline{a}\ pr_2((s,\#(bq_3(x_2)(a,b)))\uparrow(s',(y_1,c))(\delta,y_2\$)) \\
(8) & \overset{(d)}{\Longrightarrow}_N & \underline{a}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',\cent c\$)(s',(y_1,c))(\delta,y_2\$)) \\
(9) & \overset{(b)}{\Longrightarrow}_N & \underline{ac}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',\cent\$)(s',(y_1,c))(\delta,y_2\$)) \\
(10) & \overset{(f3)}{\Longrightarrow}_N & \underline{ac}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',(y_1,c))(\delta,y_2\$)) \\
(11) & \overset{(f4)}{\Longrightarrow}_N & \underline{ac}\ p((s,\#(bq_3(x_2)(a,b)))(s',(y_1,c))\uparrow(\delta,y_2\$)) \\
(12) & \overset{(c)}{\Longrightarrow}_N & \underline{ac}\ pr_2((s,\#(bq_3(x_2)(a,b)))\uparrow(s',(y_1,c))(\delta,\$)) \\
(13) & \overset{(d)}{\Longrightarrow}_N & \underline{ac}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',\cent c\$)(s',(y_1,c))(\delta,\$)) \\
(14) & \overset{(b)}{\Longrightarrow}_N & \underline{acc}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',\cent\$)(s',(y_1,c))(\delta,\$)) \\
(15) & \overset{(f3)}{\Longrightarrow}_N & \underline{acc}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',(y_1,c))(\delta,\$)) \\
(16) & \overset{(f4)}{\Longrightarrow}_N & \underline{acc}\ p((s,\#(bq_3(x_2)(a,b)))(s',(y_1,c))\uparrow(\delta,\$)) \\
(17) & \overset{(f1)}{\Longrightarrow}_N & \underline{acc}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',(y_1,c)\$)) \\
(18) & \overset{(g)}{\Longrightarrow}_N & \underline{acc}\ p((s,\#(bq_3(x_2)(a,b)))\uparrow(s',\$)) \\
(19) & \overset{(f1)}{\Longrightarrow}_N & \underline{acc}\ p(\uparrow(s,\#(bq_3(x_2)(a,b))\$)) \\
(20) & \overset{(g)}{\Longrightarrow}_N & \underline{acc}\ p(\uparrow(s,\#\$)) \\
(21) & \overset{(f2)}{\Longrightarrow}_N & \underline{acc}
\end{array}$$

We remark that the parameter variable y_2 is evaluated twice by means of the sequences (7)-(11) and (12)-(16) of computation steps. In Section 7 we will construct an attributed ct-ns transducer that avoids this recomputation. $\oplus$

After we have formally constructed the ct-ns transducer N which serves as a call-by-name implementation of the ymt M, we will prove in the next subsection the correctness of this construction.

5.2 Correctness of the Construction

We only want to describe the techniques, which are used to verify the correctness of the construction. The complete proof is contained in [Faß].

We have to show that the ymt M and the constructed ct-ns transducer N are equivalent with respect to their computed relations, i.e., we will prove that $\tau(M)=\tau(N)$. In order to aid the proof technically, we introduce another semantics of ymts which is based on the closure technique (cf. [Loo] pp. 46 and [Lan]) and which is already close to the ct-ns transducers. We call this semantics the closure semantics of ymts. And, one might say intuitively, that the closure semantics is more concrete than the computation relation, but less concrete than the implementation on ct-ns transducers. In the next subsection, we define the closure semantics and discuss the relevance for our main proof.

5.2.1 Closure Semantics of the YMT

For every ymt M, we define the closure semantics of M, denoted by $CS(M)$, by specifying sentential forms, a computation relation, and the relation computed by $CS(M)$ with call-by-name computation strategy. In general, a closure is a pair which consists of an expression e and an environment u which binds the free variables in e to a closure. In our situation, the expression e is a postfix of a right-hand side of a rule of M, and the free variables x_i and y_i are bound to trees and to sentential forms of $CS(M)$, respectively.

Definition 5.3 Let $M=(Q,\Sigma,\Delta,q_0,R)$ be a ymt.

The set of *closures of* M, denoted by $C(M)$, is the set

$$\{(\beta,u)\mid q(\sigma(x_1,\ldots,x_j))(y_1,\ldots,y_n)\longrightarrow\beta'\in R,\ \beta\ is\ a\ well-bracketed\ postfix\ of\ \beta',\ and\ u\in\underline{Env}(q,\sigma)\}$$

For every $q\in Q_{n+1},\ \sigma\in\Sigma_j:\underline{Env}(q,\sigma)$ denotes the set

$$\{f\mid f:X_j\cup Y_n\longrightarrow T_\Sigma\cup C(M)\ such\ that\ for\ every\ i\in[j]:\ f(x_i)\in T_\Sigma,\ and\ for\ every\ i\in[n]:\ f(y_i)\in C(M)\}$$

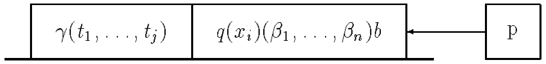
$\oplus$

We illustrate this definition by the following example.

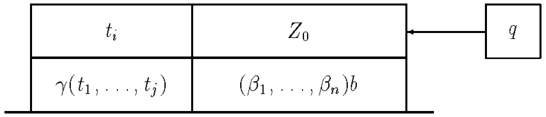
Example 5.4 In the third computation form of M_1 in Example 3.6 the part $q_3(\delta)(bq_3(\delta)(a,b),c)$ corresponds to the closure

$$(q_3(x_1)(y_1,c),[x_1/\delta,y_1/(bq_3(x_2)(a,b),[x_2/\delta])])$$

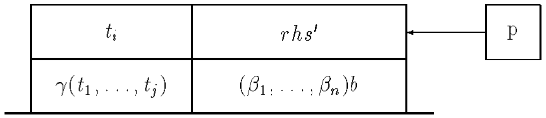
of M_1. The left component of the pair includes the right-hand side of rule (ii) of M_1 and the right component includes the bindings of its free variables. $\oplus$



is reduced to

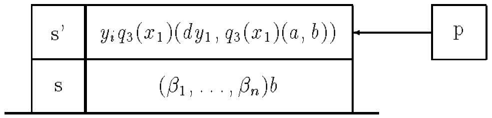


Then Z_0 is replaced by the right-hand side rhs' of a (q,σ)-rule of the ymt, if σ is the root of t_i, and the finite control changes to p.

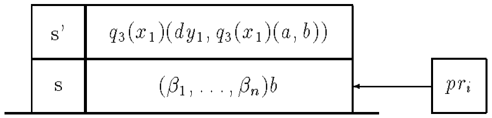


4. The first symbol of the current postfix is a parameter variable y_i.

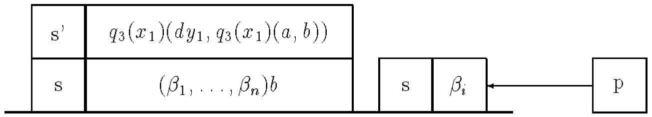
   Then the implementation has to compute the actual value w_i of y_i and to transfer it to the output tape. Due to the implementation of function calls (cf. 3), the square below the current square starts in its right component with a list $(\beta_1,\ldots,\beta_n)$; clearly, w_i is the normalform of β_i (note that β_i may contain function calls). Thus, in order to get access to β_i, the machine executes a movedown-instruction and stores the index i in the form of the state pr_i in the control. For instance, the configuration



is reduced to



In order to evaluate the normalform of β_i, a nested stack is created by executing the $create(\beta_i)$-instruction. Thus, the configuration above is reduced to



If a parameter variable appears in the right-hand side twice, it is evaluated twice. In the following sections we will formalize an extension of the ct-ns transducer, called attributed ct-ns transducer, which can serve as a call-by-need implementation of ymts. In this implementation a parameter variable is evaluated at most once.

In the rest of this section we construct the ct-ns transducer N that serves as a call-by-name implementation of the ymt M and prove the correctness of the construction.

Construction 5.1 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt.
We construct the ct-ns transducer $N = (Q', \Sigma, \Gamma, \Delta, q_0', Z_0, R')$ with $\tau(M) = \tau(N)$ as follows:

1. $\Gamma = \{Z_0\} \cup \{\alpha \mid \alpha$ is a well-bracketed postfix of a right-hand side of a rule in R$\}$
2. $Q' = Q \cup \{p\} \cup \{pr_j \mid j \in \{1,\ldots,maxrankQ-1\}\}$
3. $q_0' = q_0$
4. Construction of R':
   For every $\alpha \in \Gamma$, $Z \in \{¢, \#, \lambda\}$ and $\sigma \in \Sigma_j$ with $j \geq 0$:

   (a) For every rule $q(\sigma(x_1,\ldots,x_j))(y_1,\ldots,y_n) \longrightarrow \beta$ in R,
   the rule $q(\sigma(x_1,\ldots,x_j), ZZ_0\$) \longrightarrow p(stay(\beta))$ is in R'.

   (b) For every $a \in \Delta$,
   the rule $p(\sigma(x_1,\ldots,x_j), Za\alpha\$) \longrightarrow ap(stay(\alpha))$ is in R'.

   (c) For every $y_k \in \{y_1,\ldots,y_{maxrankQ-1}\}$,
   the rule $p(\sigma(x_1,\ldots,x_j), Zy_k\alpha\$) \longrightarrow pr_k(stay(\alpha), movedown)$ is in R'.

   (d) For every state $pr_i \in Q'$ *with* $i \in [k]$ *and* $(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$,
   the rule $pr_i(\sigma(x_1,\ldots,x_j), Z(\beta_1,\ldots,\beta_k)\alpha) \longrightarrow p(create(\beta_i))$ is in R'.

   (e) For every $q(x_i)(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$, the rule
   $p(\sigma(x_1,\ldots,x_j), Zq(x_i)(\beta_1,\ldots,\beta_k)\alpha\$) \longrightarrow q(stay((\beta_1,\ldots,\beta_k)\alpha), push(i, Z_0))$
   is in R'.

   Moreover, the rules

   (f1) $p(\sigma(x_1,\ldots,x_j), \$) \longrightarrow p(pop)$

   (f2) $p(\sigma(x_1,\ldots,x_j), \#\$) \longrightarrow \lambda$

   (f3) $p(\sigma(x_1,\ldots,x_j), ¢\$) \longrightarrow p(destruct)$

   (f4) $p(\sigma(x_1,\ldots,x_j), Z\alpha) \longrightarrow p(moveup)$, for every $\alpha \in \Gamma$

   are in R'.

   (g) For every $(\beta_1,\ldots,\beta_k)\alpha \in \Gamma$,
   the rule $p(\sigma(x_1,\ldots,x_j), Z(\beta_1,\ldots,\beta_k)\alpha\$) \longrightarrow p(stay(\alpha))$ is in R'.

5 Call-by-name Implementation of YMTs on CT-NS Transducers

In [EngVog2] it has been proved that ymts are equivalent to total deterministic ct-ns transducers. This proof is based on the characterization of ymts by total deterministic *pushdown*^{2} tree-to-string transducers (cf. Theorem 6.4 of [EngVog2]) and the equivalence of the storage types *pushdown*^{2} and *nested stack* (cf. Theorem 7.4 of [EngVog2]).

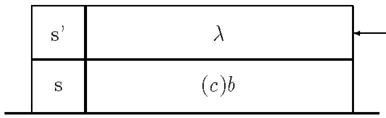
Here, we will not verify this equivalence again; rather, we will define a direct implementation of the call-by-name computation strategy for ymts on ct-ns transducers. After an informal description of this implementation we will formally define it and illustrate it by giving an example. At the end of this section we will indicate the correctness of the implementation.

5.1 Construction of the CT-NS Transducer

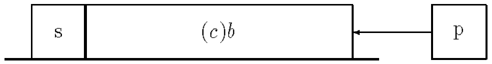
For a ymt M, the main idea of the implementation of M is to put postfixes of the right-hand side *rhs* of a rule of M on the nested stack and interpret *rhs* symbol by symbol from left-to-right. In Definition 3.1 of right-hand sides there are four construction mechanisms. Now we explain how the ct-ns transducer works in each of these cases and illustrate this by examples. If the output tape is not changed, it will be omitted. Note that in most cases the finite control includes the state p. Thus, we call p the *working state*.

1. The current postfix is the empty word λ.

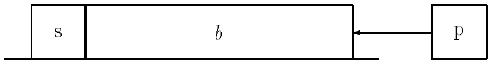
   Then the computation of the right-hand side is finished and hence, a pop-instruction drops the current square from the stack. For instance, the configuration



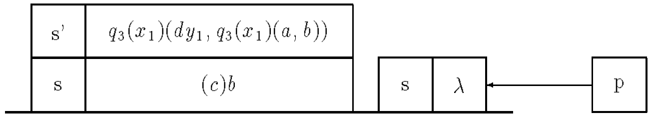
   is reduced to



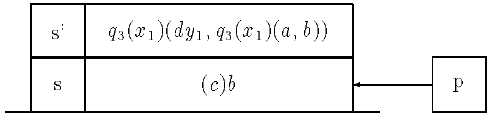
   In an additional step, the list (c) is dropped by means of a stay instruction. Then the current square contains again a postfix of *rhs* to which this interpretation procedure can be applied again:



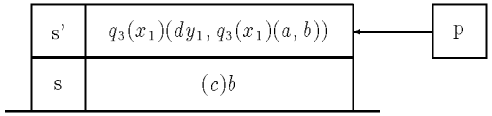
   If the current square is the only square of a nested stack, then a destruct-instruction is executed. For instance, the configuration



   is reduced to

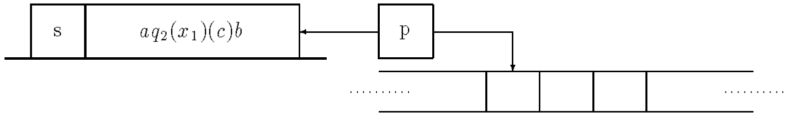


   Then a moveup instruction is executed yielding the configuration

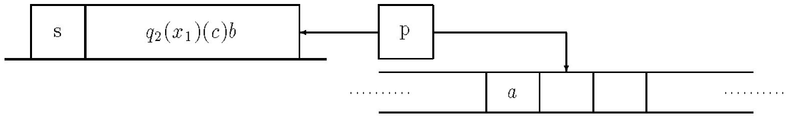


2. The first symbol of the current postfix is the output symbol $a \in \Delta$.

   Then a is deleted in the current square and written to the output tape. For instance, the configuration



   is reduced to



3. The current postfix begins with $q(x_i)(\beta_1, \ldots, \beta_n)$.

   Then the free variable x_i must be bound by the i-th subtree t_i of the tree in the current stack square. For this purpose $q(x_i)$ is deleted and a push(i,Z_0)-instruction is executed. Thus, in the new square the left component includes t_i and the right component includes Z_0. The state in the finite control changes to q. For instance, the configuration

Now we give an example of a ct-ns transducer:

Example 4.3 Let $Q = \{q_0, q_1\}$, $\Sigma = \{\sigma^{(1)}, \tau^{(1)}, \gamma^{(0)}\}$, $\Delta = \{a, b, c, d\}$, and $\Gamma = \{\delta, \kappa, Z_0\}$ be the sets of states, input symbols, output symbols, and stack symbols, respectively. Then $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R)$ is a ct-ns transducer where R consists of the following rules:

$$\begin{array}{llll}
q_0(\sigma(x_1), \#Z_0\$) & \longrightarrow & aq_0(push(1, \delta)) & \text{(i)} \\
q_0(\tau(x_1), \delta\$) & \longrightarrow & bq_0(push(1, \kappa)) & \text{(ii)} \\
q_0(\gamma, \kappa\$) & \longrightarrow & cq_0(create(\delta)) & \text{(iii)} \\
q_0(\gamma, \cent\delta\$) & \longrightarrow & q_0(movedown) & \text{(iv)} \\
q_0(\tau(x_1), \delta) & \longrightarrow & bbq_1(moveup) & \text{(v)} \\
q_1(\gamma, \cent\delta\$) & \longrightarrow & ccq_1(destruct) & \text{(vi)} \\
q_1(\gamma, \kappa\$) & \longrightarrow & dd & \text{(vii)}
\end{array}$$

⊕

We complete this section by defining the computation relation and the relation computed by a ct-ns transducer. At the end of the next subsection we will give two examples for computations of the ct-ns transducer defined in Example 4.3.

4.3 Semantics of the CT-NS Transducer

The semantics of the ct-ns transducer M is the relation $\tau(M) \subseteq T_\Sigma \times \Delta^*$ computed by M. This computation is based on the computation relation which is formalized in the next definition.

Definition 4.4 Let $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R)$ be a ct-ns transducer.

The *computation relation of M,* denoted by $\Longrightarrow_M$, is defined as follows:
For $\beta_1, \beta_2 \in \Delta^* \cup \Delta^* Q(SC(\Sigma, \Gamma))$: $\beta_1 \Longrightarrow_M \beta_2$ iff:

(i) $\beta_1 = wq(\theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma)\theta_2)$,
where $w \in \Delta^*$, $q \in Q$, and $\theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma)\theta_2 \in SC(\Sigma, \Gamma)$.

(ii) There exists a rule in R of the form either
(a) $q(\sigma(x_1, \ldots, x_j), \gamma) \longrightarrow vq'(op)$, where $v \in \Delta^*$, $q' \in Q$, $op \in I(\Sigma, \Gamma)$
or
(b) $q(\sigma(x_1, \ldots, x_j), \gamma) \longrightarrow v$, where $v \in \Delta^*$.

(iii) If the rule has the form (a), then $\beta_2 = wvq'(m(op)(\theta_1 \uparrow (\sigma(s_1, \ldots, s_j), \gamma)\theta_2))$.
If the rule has the form (b), then $\beta_2 = wv$. ⊕

The decision which rule can be applied depends on the current state, the root of the current tree, and the contents of the right component of the current stack square.

Definition 4.5 Let $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R)$ be a ct-ns transducer.

The *relation computed by M,* denoted by $\tau(M)$, is the set

$$\{(s, w) \in T_\Sigma \times \Delta^* \mid q_0(\uparrow (s, \#Z_0\$)) \Longrightarrow_M^* w\}.$$

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

$$\{\xi \mid \xi \in \Delta^* \cup \Delta^* Q(SC(\Sigma, \Gamma)) \textit{ and } q_0(\uparrow (s, \#Z_0\$)) \Longrightarrow_M^* \xi, \textit{ for some } s \in T_\Sigma\}.$$

⊕

Now, we illustrate the concept of ct-ns transducer by giving an example of a computation.

Example 4.6 Let $M = (Q, \Sigma, \Delta, \Gamma, q_0, Z_0, R)$ be the ct-ns transducer of Example 4.3 and consider the tree $\sigma(\tau(\gamma)) \in T_\Sigma$. Then M computes as follows:

$$\begin{array}{ll}
 & q_0(\uparrow (\sigma(\tau(\gamma)), \#Z_0\$)) \\
\overset{(i)}{\Longrightarrow}_M & \underline{a}\ q_0((\sigma(\tau(\gamma)), \#Z_0) \uparrow (\tau(\gamma), \delta\$)) \\
\overset{(ii)}{\Longrightarrow}_M & \underline{ab}\ q_0((\sigma(\tau(\gamma)), \#Z_0)(\tau(\gamma), \delta) \uparrow (\gamma, \kappa\$)) \\
\overset{(iii)}{\Longrightarrow}_M & \underline{abc}\ q_0((\sigma(\tau(\gamma)), \#Z_0)(\tau(\gamma), \delta) \uparrow (\gamma, \cent\delta\$)(\gamma, \kappa\$)) \\
\overset{(iv)}{\Longrightarrow}_M & \underline{abc}\ q_0((\sigma(\tau(\gamma)), \#Z_0) \uparrow (\tau(\gamma), \delta)(\gamma, \cent\delta\$)(\gamma, \kappa\$)) \\
\overset{(v)}{\Longrightarrow}_M & \underline{abcbb}\ q_1((\sigma(\tau(\gamma)), \#Z_0)(\tau(\gamma), \delta) \uparrow (\gamma, \cent\delta\$)(\gamma, \kappa\$)) \\
\overset{(vi)}{\Longrightarrow}_M & \underline{abcbbcc}\ q_1((\sigma(\tau(\gamma)), \#Z_0)(\tau(\gamma), \delta) \uparrow (\gamma, \kappa\$)) \\
\overset{(vii)}{\Longrightarrow}_M & \underline{abcbbccdd}
\end{array}$$

Hence, $(\sigma(\tau(\gamma)), abcbbccdd) \in \tau(M)$.

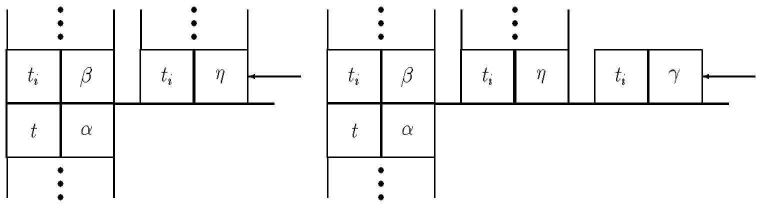
Next, consider the input tree $\sigma(\gamma)$. Then M computes as follows:

$$\begin{array}{ll}
 & q_0(\uparrow (\sigma(\gamma), \#Z_0\$)) \\
\overset{(i)}{\Longrightarrow}_M & aq_0((\sigma(\gamma), \#Z_0) \uparrow (\gamma, \delta\$))
\end{array}$$

Since there is no rule in R with left-hand side $q_0(\gamma, \delta\$)$, the computation of M stops. ⊕

After we have recalled the ymt in Section 3 and the ct-ns transducer in this section, we will implement the call-by-name computation strategy for ymts on ct-ns transducers in the next section.

2.)



(1) **(2)**

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

(1) **(2)**

4.2 Definition of the CT-NS Transducer

Let us start with the definition of the storage configurations, the instructions, and their meaning.

Definition 4.1 Let Γ be the alphabet of stack symbols and let Σ be the ranked alphabet of input symbols.

(i) The set of *storage configurations* of the ct-ns, denoted by $SC(\Sigma,\Gamma)$, is the set $NS(\Sigma,\Gamma)^* \uparrow NS(\Sigma,\Gamma)^+$ where $NS(\Sigma,\Gamma) = T_\Sigma \times Sym(\Gamma)$, and $Sym(\Gamma) = \Gamma' \cup \Gamma'\$$ with $\Gamma' = \Gamma \cup \cent\Gamma \cup \#\Gamma$ for some symbols $\uparrow$, $\#$, $\cent$, $\$ \notin \Gamma$.

(ii) The set of *instructions* on the ct-ns, denoted by $I(\Sigma,\Gamma)$, is the set

$$\{push(i,\gamma), pop, moveup, movedown, create(\gamma), destruct, stay(\gamma) \mid i \in \{1,\ldots,maxrank\Sigma\}, \gamma \in \Gamma\}$$

(iii) For every $u \in I(\Sigma,\Gamma)$, we describe the *meaning of u*, denoted by $m(u)$, as a binary relation on $SC(\Sigma,\Gamma)$ as follows:

Let $\theta_1,\theta_2 \in NS(\Sigma,\Gamma)^*$

a) $m(push(i,\gamma)) :=$
$\{(\theta_1\nu\theta_2,\ \theta_1\nu'\kappa\theta_2) \mid \nu = \uparrow(s,\alpha\$),\ \nu' = (s,\alpha),\ \kappa = \uparrow(s_i,\gamma\$),$
where $s, s_i \in T_\Sigma,\ \alpha \in \Gamma'$ and s_i is the i-th direct subtree of s $\}$

b) $m(pop) :=$
$\{(\theta_1\nu\kappa\theta_2,\ \theta_1\nu'\theta_2) \mid \kappa = \uparrow(s',\gamma\$),\ \nu = (s,\alpha),\ \nu' = \uparrow(s,\alpha\$)$
where $s, s' \in T_\Sigma,\ \gamma \in \Gamma,\ \alpha \in \Gamma'$ and s' is a direct subtree of s$\}$

c) $m(moveup) :=$
$\{(\theta_1\nu\kappa\theta_2,\ \theta_1\nu'\kappa'\theta_2) \mid \nu = \uparrow(s,\alpha),\ \nu' = (s,\alpha),\ \kappa = (s',\gamma),\ \kappa' = \uparrow\kappa,$
where $s, s' \in T_\Sigma,\ \alpha \in \Gamma', \gamma \in Sym(\Gamma)\backslash(\#\Gamma \cup \#\Gamma\$)\}$

d) $m(movedown) :=$
$\{(\theta_1\nu\kappa\theta_2,\ \theta_1\nu'\kappa'\theta_2) \mid \nu = (s,\alpha),\ \nu' = \uparrow(s,\alpha),\ \kappa = \uparrow(s',\gamma),\ \kappa' = (s',\gamma),$
where $s, s' \in T_\Sigma,\ \alpha \in \Gamma', \gamma \in Sym(\Gamma)\backslash(\#\Gamma \cup \#\Gamma\$)\}$

e) $m(create(\gamma)) :=$
$\{(\theta_1\nu\theta_2,\ \theta_1\kappa\nu'\theta_2) \mid \nu = \uparrow(s,\alpha),\ \nu' = (s,\alpha),\ \kappa = \uparrow(s,\cent\gamma\$),$
where $s \in T_\Sigma$ *and* $\alpha \in Sym(\Gamma)\}$

f) $m(destruct) :=$
$\{(\theta_1\nu\kappa\theta_2,\ \theta_1\kappa'\theta_2) \mid \kappa' = \uparrow(s,\alpha),\ \kappa = (s,\alpha),\ \nu = \uparrow(s,\cent\gamma\$),$
where $s \in T_\Sigma,\ \gamma \in \Gamma$ *and* $\alpha \in Sym(\Gamma)\}$

g) $m(stay(\gamma)) :=$
$\{(\theta_1\nu\theta_2,\ \theta_1\nu'\theta_2) \mid \nu = \uparrow(s,Z\alpha\$),\ \nu' = \uparrow(s,Z\gamma\$),$
where $s \in T_\Sigma,\ \alpha \in \Gamma$ *and* $Z \in \{\lambda, \cent, \#\}\}$ $\oplus$

Based on this definition we are able to define the ct-ns transducer by adding the finite control and the output tape to the storage configurations:

Definition 4.2 A *checking-tree nested-stack transducer* (for short: ct-ns transducer) is a 7-tuple $M = (Q,\Sigma,\Delta,\Gamma,q_0,Z_0,R)$ where:

(i) Q is the finite set of states.

(ii) Σ is the ranked alphabet of input symbols.

(iii) Δ is the alphabet of output symbols.

(iv) Γ is the alphabet of stack symbols.

(v) $q_0 \in Q$ is the initial state.

(vi) $Z_0 \in \Gamma$ is the initial stack symbol.

(vii) R is a finite set of rules of the form $q(\sigma(x_1,\ldots,x_j),\gamma) \longrightarrow \beta$, where $q \in Q,\ \sigma \in \Sigma_j$ *with* $j \geq 0,\ \gamma \in Sym(\Gamma)$, and $\beta \in \Delta^* \cup \Delta^* Q(I(\Sigma,\Gamma))$. $\oplus$

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. 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 5, 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.

In the sequel we write configurations of the ct-ns transducer in a linear form very close to the one in [Aho]. For this purpose, we use new symbols $\#$, ¢, and \$ that may accompany symbols of the stack alphabet Γ; they have the following meaning:

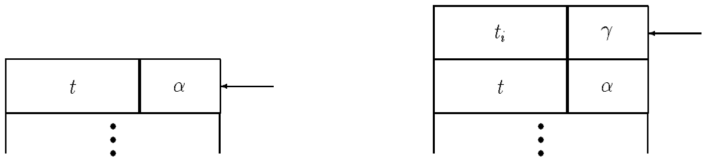
- $\#$ is written left to the bottom square of the outermost stack.
- ¢ is written left to the bottom square of a nested stack.
- \$ is written right to the top square of a stack.

The new symbol $\uparrow$ marks the current stack square and is written to the left of it. In the linear form the contents of the output tape is written to the left, followed by the current state and the configuration of the ct-ns. Thus, the configuration of Figure 5 is denoted in linear form as follows

$$\underline{abcba}\; q_i((\sigma(s_1, \ldots, s_j), \#\alpha)(s_1, ¢\delta)(s'_2, ¢\zeta) \uparrow (s''_1, \eta)(s''_2, \psi\$)(s'_2, \varepsilon\$)(s_1, \beta)(s'_1, \gamma\$)).$$

A ct-ns transducer is able to perform 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)

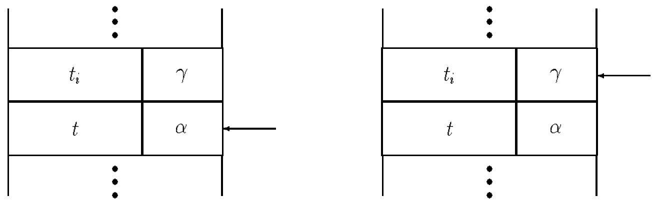


(1) **(2)**

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

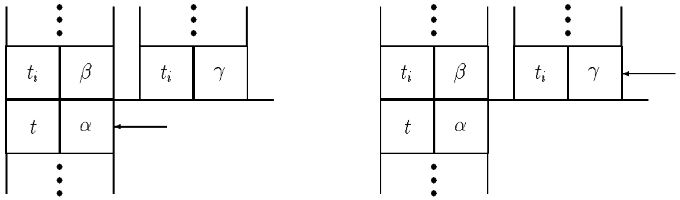
We have to distinguish two possible configurations:

1.)



(1) **(2)**

2.)

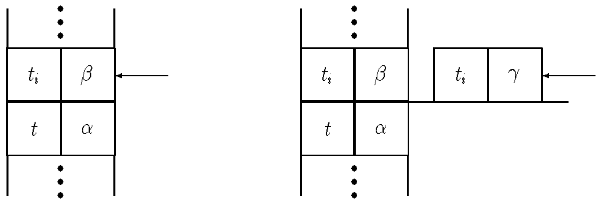


(1) **(2)**

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

Also here, we have to distinguish two possibilities:

1.)



(1) **(2)**

$$
\begin{aligned}
& q_1(\sigma(\gamma(\delta),\delta)) \\
\overset{(i)}{\Longrightarrow}_{M_1} \quad & a q_2(\gamma(\delta))(b q_3(\delta)(a,b)) \\
\overset{(ii)}{\Longrightarrow}_{M_1} \quad & a q_3(\delta)(b q_3(\delta)(a,b),c) \\
\overset{(iii)}{\Longrightarrow}_{M_1} \quad & acc
\end{aligned}
$$

Hence, $(\sigma(\gamma(\delta),\delta), acc) \in \tau(M_1)$. $\oplus$

In this section we have recalled the ymts by defining them formally and by explaining them by an example. In the next two sections we will give a space efficient implementation of ymts with call-by-name computation strategy on an abstract machine called *checking-tree nested-stack transducer*.

4 Checking-Tree Nested-Stack Transducer

In this section we recall an abstract machine, called *checking-tree nested-stack transducer,* (for short: ct-ns transducer) from [EngVog2]; there, such transducers are called nested-stack tree-to-string transducers. Sequential machines that have a nested-stack based memory were first introduced in [Aho] (cf. also [EngSlu]).

We start this section with an informal description of the structure of a ct-ns transducer and continue with the formal definition. Because of technical convenience, we will only define nondeterministic ct-ns transducers which compute relations rather than functions. However, for every ymt M, the implementation of M will lead to a total deterministic ct-ns transducer, which computes for every input exactly one output. In analogy to Section 3 we will introduce the computation relation and the relation computed by a ct-ns transducer. At the end of this chapter we will explain the behaviour of a ct-ns transducer by an example.

4.1 Informal Description of the CT-NS Transducer

A ct-ns transducer consists of a finite control, an output-tape and a storage, which is realized as a checking-tree nested-stack, for short: ct-ns. An example of a configuration of a ct-ns transducer is given in Figure 5.

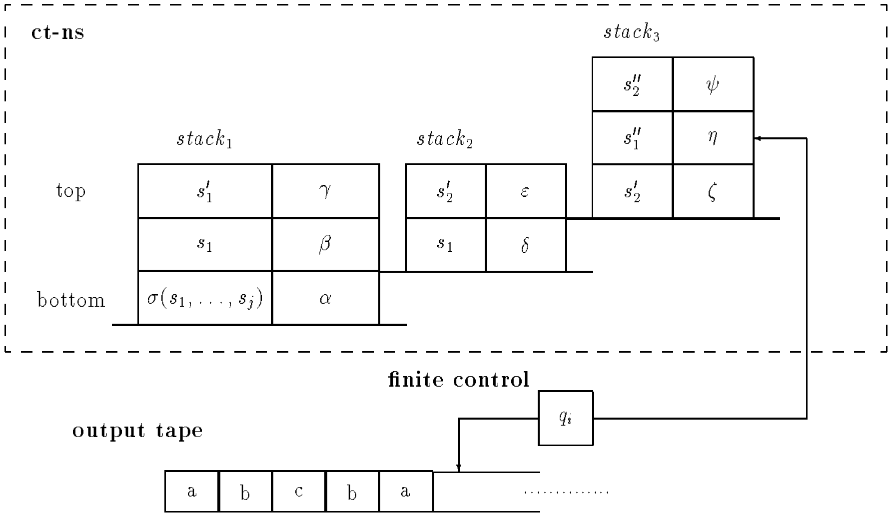


Figure 5: A Checking-Tree Nested-Stack Transducer.

Intuitively, a ct-ns consists of a finite amount of stacks that are nested in each other, e.g., the nested stack in Figure 5 consists of three stacks $stack_1$, $stack_2$, and $stack_3$; $stack_3$

In the definition of ymt we have chosen the terminology of [EngVog2]; in particular, we talk about states. Clearly, having syntax directed functional programming as motivation in mind, states represent the functions of the programs. Also note that the initial state always has rank one and that, in right-hand sides of rules, recursion variables may only occur as first argument of states. The only reason for these restrictions is the fact that the implementation on ct-ns transducers becomes technically a bit easier; it would be no principal problem to implement ymts with an initial state of arbitrary rank greater than 0 and with free occurrences of recursion variables in right-hand sides.

Next, we introduce some denotations. The trees t over Σ are called *recursion arguments* for the ymt M. The expressions β_i in case (iii) of the definition of $RHS(q, \sigma, \Delta)$ are called *parameter expressions*. The set of right-hand sides of a ymt M, denoted by $RHS(M)$, is the set $\bigcup_{\sigma \in \Sigma, q \in Q} RHS(q, \sigma, \Delta)$.

Now let us illustrate these definitions by one simple example.

Example 3.3 Consider the ranked alphabets $Q = \{q_1^{(1)}, q_2^{(2)}, q_3^{(3)}\}$ and $\Sigma = \{\sigma^{(2)}, \gamma^{(1)}, \delta^{(0)}\}$ of states and input symbols, respectively. Let $\Delta = \{a, b, c, d\}$ be the set of output symbols. Then the system $M_1 = (Q, \Sigma, \Delta, q_1, R)$ is a ymt where R is the set of the following rules:

$$
\begin{array}{lll l}
q_1(\sigma(x_1, x_2)) & \longrightarrow & a q_2(x_1)(b q_3(x_2)(a, b)) & \text{(i)} \\
q_2(\gamma(x_1))(y_1) & \longrightarrow & q_3(x_1)(y_1, c) & \text{(ii)} \\
q_3(\delta)(y_1, y_2) & \longrightarrow & y_2 y_2 & \text{(iii)} \\
q_1(\gamma(x_1)) & \longrightarrow & a q_1(x_1) & \text{(iv)} \\
q_2(\delta)(y_1) & \longrightarrow & c y_1 c & \text{(v)} \\
q_3(\sigma(x_1, x_2))(y_1, y_2) & \longrightarrow & y_1 y_2 & \text{(vi)} \\
q_1(\delta) & \longrightarrow & cc & \text{(vii)} \\
q_2(\sigma(x_1, x_2))(y_1) & \longrightarrow & y_1 q_1(x_2) & \text{(viii)} \\
q_3(\gamma(x_1))(y_1, y_2) & \longrightarrow & y_2 q_2(x_1)(d q_1(x_1) c) & \text{(ix)}
\end{array}
$$

$\oplus$

We note that, in rule (iii) of the previous example, the parameter variable y_2 appears more than once. In the following sections this property will play an important role, because we want to construct an implementation of ymts that does not evaluate a parameter variable twice (Section 7).

3.2 Semantics of the YMT

The semantics of a ymt M is the function which is computed by M. The computation is based on a computation relation that incorporates a computation strategy. In this paper, we only formalize the call-by-name strategy for ymts and discuss the usual refinement that yields the call-by-need strategy.

In this subsection, we first present the formal definitions of the call-by-name computation relation and the function which is computed by this relation. Then we give an example of a computation for the ymt which was introduced in the previous subsection.

Definition 3.4 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt and let $\Omega = \Delta \cup Q(T_\Sigma) \cup \{(,), ,\}$ where $Q(T_\Sigma) = \{q(t) \mid q \in Q, t \in T_\Sigma\}$.

The *call-by-name computation relation of M,* denoted by $\Longrightarrow_M$, is a subset of $\Omega^* \times \Omega^*$ and is defined as follows:

For every $\varphi_1, \varphi_2 \in \Omega^*$: $\varphi_1 \Longrightarrow_M \varphi_2$ iff:

(i) $\varphi_1 = w q(\sigma(s_1, \ldots, s_j))(\beta_1, \ldots, \beta_n)\alpha$ where $w \in \Delta^*$, $q \in Q_{n+1}$ with $n \geq 0$, $\sigma \in \Sigma_j$ with $j \geq 0$, for every $i \in [j] : s_i \in T_\Sigma$, for every $i \in [n] : \beta_i \in \Omega^*$, and $\alpha \in \Omega^*$.

(ii) $q(\sigma(x_1, \ldots, x_j))(y_1, \ldots, y_n) \longrightarrow \beta$ is the (q, σ)-rule in R.

(iii) $\varphi_2 = w\beta[x_1/s_1, \ldots, x_j/s_j, y_1/\beta_1, \ldots, y_n/\beta_n]\alpha$. $\oplus$

As usual we denote the transitive, reflexive closure of $\Longrightarrow_M$ by $\Longrightarrow_M^*$.

In one computation step the leftmost occurrence of a state $q \in Q$ together with its recursion argument $\sigma(s_1, \ldots, s_j)$ and its parameter list $(\beta_1, \ldots, \beta_n)$ is replaced by a modification of the right-hand side β of the (q, σ)-rule. The modification consists of the replacement of the recursion variables $x_1, \ldots, x_j$ and the parameter variables $y_1, \ldots, y_n$ by the direct subtrees $s_1, \ldots, s_j$ and by the parameter expressions $\beta_1, \ldots, \beta_n$, respectively. Thus, the free variables in β are always replaced by subtrees or expressions and hence, there are no free variables in the intermediate computation forms.

It follows from the considerations of Section 3.3 of [EngVog1] that for an arbitrary ymt $M = (Q, \Sigma, \Delta, q_0, R)$ and every input tree $s \in T_\Sigma$, there is a unique output string $w \in \Delta^*$ such that $q_0(s) \Longrightarrow_M^* w$. Thus, in a straightforward way, $\Longrightarrow_M$ induces a total function of type $T_\Sigma \rightarrow \Delta^*$. However, in order to be compatible with the semantics definition of ct-ns transducers given in Section 4, we formalize this function as a relation.

Definition 3.5 Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a ymt.

The *relation computed by M with call-by-name computation strategy,* denoted by $\tau(M)$, is the set

$$\{(s, w) \in T_\Sigma \times \Delta^* \mid q_0(s) \Longrightarrow_M^* w\}.$$

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

$$\{\xi \mid \xi \in \Omega^* \textit{ and } q_0(s) \Longrightarrow_M^* \xi, \textit{ for some } s \in T_\Sigma\}.$$

$\oplus$

After this formal definition we give an example. The superscript at the computation arrow refers to the rule applied at this computation step.

Example 3.6 Consider the ymt M_1 of Example 3.3 and the recursion argument $\sigma(\gamma(\delta), \delta)$. Then the computation looks as follows:

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* $t_1, \ldots, t_k \in T_\Sigma(S)$, $\sigma(t_1, \ldots, t_k) \in T_\Sigma(S)$.

If in Case (ii) $k = 0$, then we write σ rather than $\sigma()$. $T_\Sigma(\emptyset)$ is abbreviated by T_Σ.

The *height* of a tree t, denoted by $h(t)$, is defined inductively as follows:

(i) *for every* $\sigma \in \Sigma_0$, $h(\sigma) = 1$
(ii) *for every* $\sigma \in \Sigma_k$ *with* $k \geq 1$ *and* $t_1, \ldots, t_k$, $h(\sigma(t_1, \ldots, t_k)) = 1 + \max_{1 \leq j \leq k}\{h(t_j)\}$.

Substitutions

For a set $Z_k = \{z_1, \ldots, z_k\}$ of symbols with $k \geq 0$, a word w with occurrences of elements of Z_k, and words $w_1, \ldots, w_k$, we denote by $w[z_1/w_1, \ldots, z_k/w_k]$ the result of substituting w_i for every occurrence of z_i in w.

3 Macro Tree-to-String Transducer

In this section we recall the concept of total deterministic macro tree-to-string transducer (for short: ymt) from [Eng4,EngVog2] by giving the definitions of syntax and semantics. Recall from the introduction that such special term rewriting systems formalize the reduction semantics of syntax directed functional programming.

3.1 Syntax of the YMT

We start with the definition of the syntax of ymts. For the sake of better readability, we define the set of right-hand sides of rewrite rules separately.

Definition 3.1 Let Q and Σ be ranked alphabets and let Δ be a usual alphabet.
For every $q \in Q_{n+1}$ with $n \geq 0$ and $\sigma \in \Sigma_j$ with $j \geq 0$, the set of (q, σ)-*right-hand sides*, denoted by $RHS(q, \sigma, \Delta)$, is the smallest set RHS, such that:

(i) $\lambda \in RHS$.

(ii) For every $a \in \Delta$ and $\beta \in RHS : a\beta \in RHS$.

(iii) For every $k \geq 0$, $p \in Q_{k+1}$, $i \in [j]$, and $\beta, \beta_1, \ldots, \beta_k \in RHS$:
$$p(x_i)(\beta_1, \ldots, \beta_k)\beta \in RHS.$$

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

Definition 3.2 A *ymt* is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, R)$, where

(i) Q is the ranked alphabet of states.

(ii) Σ is the ranked alphabet of input symbols.

(iii) Δ is the alphabet of output symbols.

(iv) $q_0 \in Q_1$ is the initial state.

(v) R is the finite set of rewrite rules; for every $q \in Q_{n+1}$ with $n \geq 0$, $\sigma \in \Sigma_j$ with $j \geq 0$, there is exactly one rule of the form $q(\sigma(x_1, \ldots, x_j))(y_1, \ldots, y_n) \longrightarrow \beta$ in R, where $\{x_1, \ldots, x_j\} \subseteq X$, $\{y_1, \ldots, y_n\} \subseteq Y$, and $\beta \in RHS(q, \sigma, \Delta)$. $\oplus$

The unique rule with left-hand side $q(\sigma(x_1, \ldots, x_j))(y_1, \ldots, y_n)$ is called the $(q, \sigma) - rule$ of M.

because there can be function calls involved. This is the usual deficiency of the call-by-name computation strategy. Consider, e.g., the following syntax directed functional program $P(exp)$:

$$\begin{array}{l} exp(x, y_1) = \underline{\text{case}}\ x\ \underline{\text{of}} \\ \qquad \sigma(x_1, x_2) : exp(x_1, exp(x_2, y_1)); \\ \qquad \alpha \qquad\quad\; : \sigma(y_1, y_1) \\ \quad \underline{\text{end}} \end{array}$$

and the rewrite rules of the corresponding ymt

$$\begin{array}{ll} exp(\sigma(x_1, x_2))(y_1) & \longrightarrow exp(x_1)(exp(x_2)(y_1)) \\ exp(\alpha)(y_1) & \longrightarrow \sigma(y_1, y_1). \end{array}$$

Note that, in the right-hand side of the first rule, a recursive function call occurs nested in the parameter position of another function call. Now, suppose that the right-hand side rhs_2 of the second rule has just been put onto the nested stack and the ct-ns transducer starts with the interpretation of rhs_2. Then, after having transferred "σ" and "(" to the output tape, the first symbol, i.e., "y_1", requires the computation of the first component ξ_1 of the list stored one square below. After having finished this computation and having transferred "," to the output tape, again ξ_1 has to be computed. How can we avoid this recomputation?

In this paper we shall introduce the concept of *attributed checking-tree nested-stack transducer* (for short: act-ns transducer) to solve this problem. There, with every square of the nested stacks, a finite but sufficiently large amount of attributes (or registers) is associated in which the values of substitution variables y_j are stored immediately after their computation. Then, if such a value is needed again, it does not have to be reevaluated, but can just be transferred from the corresponding attribute to the output tape. We note that the lifetime of the additional registers is the same as that of the square to which they are associated. In this way we are able to formalize a call-by-need implementation of a syntax directed functional programming language on the act-ns transducer.

This paper is organized in seven sections where the second section contains preliminaries. In Section 3 we recall the concept of ymt which forms the starting point of our formal investigation. To say this explicitly, in the rest of this paper we shall not define the syntax and denotational semantics of a syntax directed functional programming language; this programming concept only served as motivational background for the discussion of the introduction. In Section 4, after a detailed informal discussion of the concept of ct-ns transducer, we formally define this concept and provide an example. Section 5 deals with the call-by-name implementation of ymts on ct-ns transducers. In Section 6 we introduce the extension of the ct-ns transducers by attributes and formally define the concept of act-ns transducer. The construction of the call-by-need implementation of ymts on act-ns transducers is presented in Section 7. Finally, Section 8 contains some concluding remarks and indicates further research topics.

2 Preliminaries

We recall and collect some notations, basic definitions, and terminology which will be used in the rest of the paper. Nevertheless, the reader is assumed to be familiar with the basic concepts of tree language theory (see, e.g., [Rou,Tha,Eng1,Eng2]).

General notations

We denote the set of nonnegative integers by $\mathbb{N}$. The empty set is denoted by $\emptyset$. For $j \in \mathbb{N}$, $[j]$ denotes the set $\{1, \ldots, j\}$; thus $[0] = \emptyset$. For a finite set A, $\mathcal{P}(A)$ is the *set of subsets* of A and $|A|$ denotes the *cardinality* of A. For two sets A, B, a *function* $f : A \longrightarrow B$ is a relation from A into B, such that for every $a \in A$ there exists at most one $b \in B$ with $f(a) = b$. For a function $f : A \longrightarrow B$, we call A the *domain of* f and denote it by *dom(f)*. If two functions f and g from A into B are different only for finitely many elements $a_1, \ldots, a_n$ of the domain A and if, for every $j \in [n] : g(a_j) = b_j$, then we denote g by $f[a_1/b_1, \ldots, a_n/b_n]$. An *environment* is a function with a finite domain. If the domain of the environment u is the set $\{a_1, \ldots, a_n\}$ and if, for every $j \in [n] : u(a_j) = b_j$, then we denote u by $[a_1/b_1, \ldots, a_n/b_n]$. For two sets A and B, the set $\{ab \mid a \in A \textit{ and } b \in B\}$ is denoted by AB. If A resp. B consists only of a, we denote AB by aB resp. Aa.

In this paper, we use two sets of substitution variables, viz., the sets $X = \{x_i \mid i \geq 1\}$ and $Y = \{y_i \mid i \geq 1\}$. The x_i's and y_i's are called *recursion variables* and *parameter variables,* respectively. For every $k \geq 0$, we denote $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_k\}$ by X_k and Y_k, respectively.

Alphabets and Words

An *alphabet* is a finite set. If A is an alphabet and $n \in \mathbb{N}$, then we call the set

$$A^n := \begin{cases} \{\lambda\} & , \textit{if } n = 0 \\ \{a_1 a_2 \cdots a_n \mid \textit{for every } j \in [n] : a_j \in A\} & , \textit{if } n > 0 \end{cases}$$

the set of *words over* A *with length* n. The *empty word* is denoted by λ. The set of *words over* A is the set $\bigcup_{n \in \mathbb{N}} A^n$ and is denoted by A^*. The set $A^+ := A^* \backslash \{\lambda\}$ is called the *set of the nonempty words over* A.

Ranked Alphabets and Trees

A pair $(\Sigma, \underline{rank}_\Sigma)$ is called *ranked alphabet,* if 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; $maxrank\Sigma$ denotes the maximal rank of an element 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. We can define a ranked alphabet $(\Sigma, \underline{rank}_\Sigma)$ either by enumerating the finite subsets Σ_m that are not empty, or by giving a set of symbols that are indexed with their (unique) rank. For example, if $\Sigma = \{a, b, c\}$ and $\underline{rank}_\Sigma : \Sigma \longrightarrow \mathbb{N}$ with $\underline{rank}_\Sigma(a) = 0$, $\underline{rank}_\Sigma(b) = 2$ and $\underline{rank}_\Sigma(c) = 7$, then we can describe $(\Sigma, \underline{rank}_\Sigma)$ either by $\Sigma_0 = \{a\}$, $\Sigma_2 = \{b\}$ and $\Sigma_7 = \{c\}$ or by $\{a^{(0)}, b^{(2)}, c^{(7)}\}$. If the ranks of the symbols are

$\Delta = \Sigma \cup \{(,),,\}$. In order to indicate also notationally the difference between recursion argument and parameters, we shall consider the curried form of every function; thus, finally, shovel has type $T_\Sigma \to (\Delta^* \to \Delta^*)$, and the ymt $M_{ts}(P(shovel))$ that describes the reduction semantics of $P(shovel)$ has the following rewrite rules:

$$\begin{array}{ll} shovel(\sigma(x_1,x_2))(y_1) & \longrightarrow shovel(x_1)(\sigma(x_2,y_1)) \\ shovel(\alpha)(y_1) & \longrightarrow y_1. \end{array}$$

This is an example in which the resulting string represents a tree, but ymts are also used for the description of functions which transform trees into usual strings, i.e., in a syntax directed translation. In Figure 4 a rewrite rule for the translation of the if_then_else statement of a programming language into a code sequence of an abstract machine is described (cf. [Ind]). Note that the right-hand side of the rule is a string over function calls (like "$\underline{exptrans}(x_1)(y_1.1)$"), machine instructions (like "JMC"), labels (like "$y_1.0$"), and other program symbols taken from the set $\{.,:,;,0,1,2,3,4\}$.

$$\begin{array}{lrl} \underline{exptrans}\ (if\_then\_else(x_1,x_2,x_3))(y_1) \longrightarrow & & \underline{exptrans}\ (x_1)(y_1.1) \\ & & JMC\ y_1.0; \\ & & \underline{exptrans}\ (x_2)(y_1.2) \\ & & JMP\ y_1.4; \\ & y_1.0 & :\underline{exptrans}\ (x_3)(y_1.3) \\ & y_1.4 & : \end{array}$$

Figure 4: Rewrite rule of a ymt for the translation of if_then_else.

Up to this point we have discussed the subclass of functional programming in which we are interested, and we have indicated the appropriate term rewriting systems for the formalization of its reduction semantics. Now let us turn to the implementational aspect. In this paper we formalize a call-by-need implementation of a syntax directed functional programming language on an abstract machine called *attributed checking-tree nested-stack transducer* (for short: act-ns transducer). The construction of this implementation is based on the characterization of extended (nondeterministic) macro tree-to-string transducers with call-by-name computation strategy by *checking-tree nested-stack transducers* (for short: ct-ns transducer). This construction preserves the property total determinism; in this case, the extension is not needed, i.e., ymts with call-by-name computation strategy are characterized by total deterministic ct-ns transducer (cf. Theorem 7.6 of [EngVog2]). In [EngVog2], the characterization of ymts has been proved in an indirect way: first, such transducers are characterized by total deterministic $pushdown^2$ tree transducers (cf. Theorem 6.5 and Theorem 8.3) and second, the equivalence of the storage types $pushdown^2$ and $nested\ stack$ has been shown (cf. Theorem 7.4, cf. also [Eng5]). Here we present a direct call-by-name implementation of syntax directed functional programming by simulating ymts on nested-stack tree-to-string transducers; the latter transducers are equivalent to ct-ns transducers.

Now let us first briefly discuss the concept of the call-by-name implementation on ct-ns transducer; afterwards we shall indicate how the attribution of particular storage segments can be used to refine the call-by-name implementation to the call-by-need implementation. Thereby, the well-known problem of recomputation of parameter values is avoided.

A ct-ns transducer is a machine that consists of a finite control, an output tape, and a storage called checking-tree nested-stack [EngSlu]. The storage is based on a nested stack [Aho] which generalizes the concept of pushdown by allowing (1) the inspection of squares which are not at top position and (2) the creation of nested stacks between the currently inspected square and the square below it; conversely, a nested stack with one square only can be destructed. In a checking-tree nested-stack configuration, besides some information taken from a finite set, every square contains a pointer to a node of the checking-tree. The walking on the checking-tree and the actions on the nested-stack are synchronized in the following sense: if a square sq' is pushed onto the currently inspected square sq (which has to be in top position), then sq' contains a pointer to a specified son of the node which corresponds to the pointer stored in sq; vice versa, if a square is popped, then the father of the previously visible node is accessible. We refer the reader to Sections 3.3 and 7 of [EngVog2] for a detailed discussion of the storage type checking-tree nested-stack.

A syntax directed functional program P can be implemented on a ct-ns transducer. The recursion argument serves as checking-tree and, during the computation, every square sq_1 contains a postfix post(rhs) of the right-hand side rhs of a rewrite rule of $M_{ts}(P)$ as information (recall that $M_{ts}(P)$ is the ymt which formalizes the reduction semantics of P). The ct-ns transducer works through rhs symbol by symbol from left to right and performs actions corresponding to the leftmost symbol of post(rhs):

1. Interpunctuation symbols (i.e., "(", ")", and ",") and output symbols are immediately transferred to the output tape.

2. If post(rhs) starts with a construct like $con = f(x_i)(\xi_1,\ldots,\xi_k)$ where f is a recursive function call and if the i-th son of the current node of the checking-tree is labeled by some σ, then con is replaced by $(\xi_1,\ldots,\xi_k)$ and a new square sq_2 with the information $rhs(f,\sigma)$ is pushed onto the current nested stack where $rhs(f,\sigma)$ is the right-hand side of the rule with left-hand side $f(\sigma(x_1,\ldots,x_j))(y_1,\ldots,y_k)$; moreover, sq_2 contains a pointer to the i-th son of the node corresponding to the pointer stored in sq_1.

3. If post(rhs) starts with y_j, then the machine inspects the square sq_0 below sq_1; now the treatment of recursive function calls (see (2)) guarantees that the information of sq_0 starts with a list $(\xi_1,\ldots,\xi_r)$ of trees which represent the parameter values of some function calls. Since post(rhs) starts with y_j, the value of ξ_j has to be computed and transferred to the output tape. For the computation of the value of ξ_j, a nested-stack is created between sq_0 and the square sq_{-1} below; this new square contains ξ_j as information and the same pointer as stored in sq_0.

Clearly, if a parameter y_j occurs more than once in the right-hand side of a rule of $M_{ts}(P)$, then every occurrence denotes the same value, i.e., string. However, the ct-ns transducer evaluates the corresponding string as often as y_j occurs, and this reevaluation may be costly of an element t of T_Σ where the dotted and the thick lines indicate the right-spine of t and the left-subtrees of t, respectively. The dual notions left-spine and right-subtree are used in the corresponding way.

Now we want to program the unary function *reverse* of type $T_\Sigma \to T_\Sigma$ in a functional way by using primitive recursion. The function should reverse the order of the left-subtrees of its argument, i.e., $reverse(\sigma(t_1, \sigma(t_2, \sigma(t_3, \alpha)))) = \sigma(t_3, \sigma(t_2, \sigma(t_1, \alpha)))$. Figure 2 shows the program $P(reverse)$ that fulfills this requirement; actually, $P(reverse)$ is an $SRDL^*$-program. The function *reverse* is defined by composition of the previously defined functions

$$
\begin{array}{lll}
shovel(x, y_1) & = & \underline{\text{case}}\ x\ \underline{\text{of}} \\
 & & \quad \sigma(x_1, x_2) : shovel(x_1, \sigma(x_2, y_1)); \\
 & & \quad \alpha \qquad\quad\;\; : y_1 \\
 & & \underline{\text{end}} \\
 & & \\
mirror(x) & = & \underline{\text{case}}\ x\ \underline{\text{of}} \\
 & & \quad \sigma(x_1, x_2) : \sigma(mirror(x_2), x_1); \\
 & & \quad \alpha \qquad\quad\;\; : \alpha \\
 & & \underline{\text{end}} \\
 & & \\
reverse(x) & = & shovel(mirror(x), \alpha)
\end{array}
$$

Figure 2: Program $P(reverse)$ for reversing the order of right-subtrees.

mirror and shovel. The function mirror has the type $T_\Sigma \to T_\Sigma$ and thus, the argument is also the recursion argument. Mirror considers the right-spine of its recursion argument $\sigma(t_1, \sigma(t_2, \ldots \sigma(t_n, \alpha) \ldots))$ and yields the tree $\sigma(\sigma(\ldots \sigma(\alpha, t_n) \ldots, t_2), t_1)$ as result; note that the left-subtrees $t_1, t_2, \ldots, t_n$ of t are not affected. The binary function shovel has the type $T_\Sigma \times T_\Sigma \to T_\Sigma$ where the first argument is the recursion argument and the second argument is called *(context) parameter*. Shovel considers the left-spine of its argument and shovels every right-subtree onto its second argument. Note that shovel is defined by the scheme of primitive recursion with parameter substitution, because $\sigma(x_2, y_1)$ occurs in the parameter position of the recursive call of shovel. $\oplus$

A special instance of the concept of functional programming with primitive recursion arises, when the recursion of every function which occurs in the program, is controlled by a tree which is provided to the program at the beginning of the computation. In other words, functions should be programmed simultaneously by one generalized scheme of primitive recursion. For $P(reverse)$ this restriction does not hold, because the recursion of the function shovel is controlled by a tree which is the result of the computation of the function mirror; i.e., *reverse* is programmed in a modular way that involves more than one recursion scheme. But clearly, e.g., the function shovel alone constitutes an example program of the special instance under concern (cf. Figure 3). This instance of the concept of functional programming with primitive recursion can be considered as a generalization of the well-known concept of syntax directed translation [AhoUll1,AhoUll2] in the sense that context information can be handled

$$
\begin{array}{lll}
shovel(x, y_1) & = & \underline{\text{case}}\ x\ \underline{\text{of}} \\
 & & \quad \sigma(x_1, x_2) : shovel(x_1, \sigma(x_2, y_1)); \\
 & & \quad \alpha \qquad\quad\;\; : y_1 \\
 & & \underline{\text{end}}
\end{array}
$$

Figure 3: Syntax directed functional program $P(shovel)$.

(cf. [Eng4] where this amalgamation of concepts is discussed in detail): in $P(shovel)$, the second argument of shovel keeps the context information for the recursion argument; e.g., the value of $shovel(\sigma(t_1, t_2), t)$ depends on the value t of the context for the recursion argument $\sigma(t_1, t_2)$ with respect to the complete input tree. Having this origin, we call this special instance the concept of *syntax directed functional programming*. The main subject of the present paper is the call-by-need implementation of syntax directed functional programming. We note that syntax directed functional programming is relevant for compiler construction, in particular, for the phase of code generation (cf. [Ind]).

As usual the semantics of functional programs can be described in two different ways: denotational and operational. In its turn, operational semantics is either specified as reduction semantics or machine semantics, and in this paper we choose the former specification method. In a natural way, the reduction semantics of syntax directed functional programs can formally be described by special term rewriting systems called *total deterministic macro tree transducers* [Eng3,CouFra,EngVogl]. In fact, the transcription from a syntax directed functional program P into the corresponding total deterministic macro tree transducer $M(P)$ is trivial: every line of the case statement of P is represented by one rewrite rule. Thus, for our example $P(shovel)$, the transducer $M(P(shovel))$ has the following two rewrite rules

$$
\begin{array}{ll}
shovel(\sigma(x_1, x_2), y_1) & \longrightarrow shovel(x_1, \sigma(x_2, y_1)) \\
shovel(\alpha, y_1) & \longrightarrow y_1.
\end{array}
$$

This macro tree transducer is total deterministic in the sense that, for every function and every input symbol (or: pattern), there is exactly one corresponding rule.

In general, a right-hand side of a rule (e.g., $shovel(x_1, \sigma(x_2, y_1))$) is a tree that may contain output symbols (like σ), recursive function calls (like *shovel*), and substitution variables (like x_1, x_2, and y_1). In the usual way these rules are applied using pattern matching (cf. [Klo,HueOpp] for surveys about term rewriting systems).

Recall that, in our concept of syntax directed functional programming, we only consider tree functions. But, since the machine that we shall present for the implementation of a program P, has a usual output tape, it is more appropriate to consider the right-hand side of $M(P)'s$ rewrite rules and the trees in the parameter positions of functions in the usual way as strings in which particular symbols like "(", ")", and "," may occur. Note that, because of the subtree selection mechanism by means of substitution variables x_i, the recursion argument still has to be considered as a tree. From this point of view, total deterministic macro tree transducers turn into *total deterministic macro tree-to-string transducers* (for short: ymts), and, e.g., shovel would have the type $T_\Sigma \times \Delta^* \to \Delta^*$ for the alphabet

A Call-by-need Implementation of Syntax Directed Functional Programming

Heinz Faßbender* and Heiko Vogler
Lehrstuhl für Informatik II, RWTH Aachen
Ahornstraße 55, 5100 Aachen, West Germany

Aachener Informatik-Berichte
Nr. 90–22

Abstract

This paper contributes to the field of functional programming languages. We investigate the implementation of a restricted type of functional programming, called *syntax directed functional programming;* the target of this implementation is an abstract machine that is based on nested stacks. In fact, the technical kernel of this paper is a refinement of an automata theoretical result that, roughly speaking, investigates the well-known relationship "recursion = iteration + stack" in the framework of tree transducers. More precisely, in the underlying result the class of functions computed by total deterministic macro tree-to-string transducers with the call-by-name computation strategy is characterized by total deterministic checking-tree nested-stack transducers. Note that total deterministic macro tree-to-string transducers are term rewriting systems by means of which the reduction semantics of syntax directed functional programming languages can be described.

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

1 Introduction

The main concern of functional programming is the specification of functions over some data type (cf. [Hud] for a survey of functional programming) and the most important specification method is recursion. However, it can be observed that in a lot of practical example programs, recursion is only used in the restricted form of primitive recursion over some freely generated data type like lists or, in general, tree-structured objects (cf. [Pet] for primitive recursive functions over natural numbers; cf. [Hup,Kla3,EngVog3] for primitive recursive functions over trees). Actually, in this paper we shall only consider functions that are defined over the initial term algebra, i.e., both the arguments and the results of such functions are trees. And hence, we call them *tree functions*. Although, from the practical point of view, this choice is a restriction, it is justified for the more theoretically oriented investigation presented in this paper.

We shall call the subclass of functional programming which obeys both restrictions, viz., the choice of the initial term algebra as data type for functions and the restriction to primitive recursion, the concept of *functional programming with primitive recursion*. This concept is realized in the programming language $SRDL^*$ (cf. [Vog]) which is a generalization of the language $SRDL$ [Kla1,Kla2,Kla4,KlaInd] ($SRDL$ is a short hand for *s*tructural *r*ecursive *d*efinition *l*anguage). The generalization consists in the availability of a more flexible version of the basic scheme of primitive recursion (cf. [Pet,Hup,Kla3,EngVog3] for this basic scheme); in particular, it allows for simultaneous recursion, parameter substitution, and nested recursive function calls (cf. [EngVog3] for the definition of these generalizations and for the fact that they do not extend the class of definable functions). Let us illustrate the concept of functional programming with primitive recursion by a simple example.

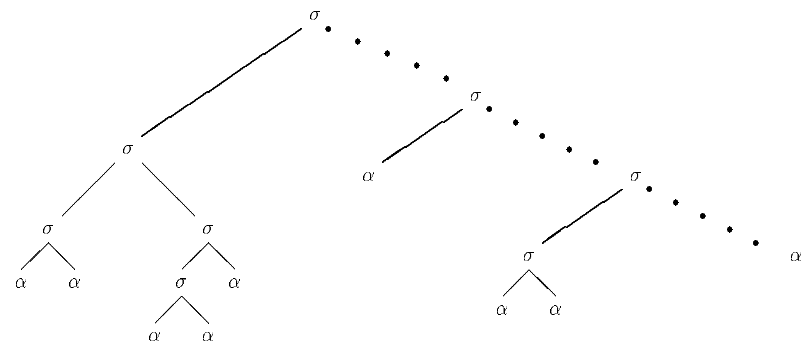


Figure 1: Element $t = \sigma(t_1, \sigma(t_2, \sigma(t_3, \alpha))) \in T_\Sigma$ with left-subtrees $t_1 = \sigma(\sigma(\alpha,\alpha), \sigma(\sigma(\alpha,\alpha),\alpha))$, $t_2 = \alpha$, and $t_3 = \sigma(\alpha,\alpha)$.

Example 1.1 Consider the set T_Σ of binary trees over the ranked alphabet $\Sigma = \{\sigma, \alpha\}$ where σ and α are input symbols of rank 2 and 0, respectively. Figure 1 shows an example