Acta Informatica 26, 131–192 (1988)

Acta Informatica

© Springer-Verlag 1988

High Level Tree Transducers and Iterated Pushdown Tree Transducers

Joost Engelfriet and Heiko Vogler*

University of Leiden, Department of Mathematics and Computer Science, P.O. Box 9512, NL-2300 RA Leiden, The Netherlands

Contents

1. Introduction 131
2. Preliminaries 137
3. Grammars with Storage, Simulation of Storage Types, and Pushdown Operators 141
4. High Level S Transducers and Iterated Pushdown S Transducers 154
5. Substitution of Applicative Terms 158
6. Applicative Terms and Iterated Pushdowns 168
7. Characterization of High Level Tree Transducers 179
8. Some Consequences 186
References 191

Summary. n-level tree transducers ($n \geqq 0$) combine the features of n-level tree grammars and of top-down tree transducers in the sense that the derivations of the tree grammars are syntax-directed by input trees. For running n, the sequence of n-level tree transducers starts with top-down tree transducers ($n=0$) and macro tree transducers ($n=1$). In this paper the class of tree-to-tree translations computed by n-level tree transducers is characterized by n-iterated pushdown tree transducers. Such a transducer can be considered as a regular tree grammar of which the derivations are syntax-directed by n-iterated pushdowns of trees; an n-iterated pushdown of trees is a pushdown of pushdowns of ... of pushdowns (n times) of trees. In particular, we investigate the total deterministic case, which is relevant for syntax-directed semantics of programming languages.

1. Introduction

In this paper *n-level tree transducers* are introduced as a natural extension of top-down tree transducers [Rou, Tha, Engl] ($n=0$) and macro tree transducers [Eng3, CouFra, EngVogl] ($n=1$). The underlying idea of this generalization

Offprint requests to: J. Engelfriet

* The work of the second author has been supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.). The second authors present address is Lehrstuhl für Informatik II, RWTH Aachen, Federal Republic of Germany

is the same as in the extension of regular tree grammars [Bra, GecSte] ($n=0$) and context-free tree grammars [EngSch, Fis] ($n=1$) to n-level tree grammars [Dam]. Actually, an n-level tree transducer can be considered as an n-level tree grammar of which the derivations are syntax-directed. Since top-down tree transducers are equivalent to the generalized syntax-directed translation schemes of [AhoUll1,2] (cf. [MarVer, Vog1]) and macro tree transducers generalize top-down tree transducers in the sense that the handling of context is possible, the high-level tree transducers also provide a metalanguage for describing parts of the semantics of programming languages. Using them, even context information of functional type (such as environments) can be handled, whereas for macro tree transducers the context information has to be of some basic type. The reader is refered to [Eng4] for a detailed discussion of tree transducers as formal models of metalanguages for syntax-directed semantics.

From the program schematic point of view an n-level tree transducer can be considered as a system of recursive function procedures that compute transformations on trees. Each function procedure has one "syntactic" parameter and it delivers a result of functional type up to level n (i.e., a basic element or a function of functions of ... of functions on basic elements). The name "syntactic" parameter originates from the use of (high-level) tree transducers as a metalanguage for describing syntax-directed semantics; its value is a (pointer to a) subtree of the input tree of the transducer. As an example, such a function procedure A may be of type $T_\Sigma \to ((T_\Delta \to T_\Delta) \times (T_\Delta \times T_\Delta \to T_\Delta) \to (T_\Delta \to T_\Delta))$, where T_Σ and T_Δ denote the sets of trees representing the basic values of syntactic objects and of semantic objects, respectively. Since the result of A is of level 2 (it is a function of functions), A may be part of an n-level tree transducer for any $n \geqq 2$. By designating a main function procedure of type $T_\Sigma \to T_\Delta$ (delivering a result of level 0), an n-level tree transducer computes a translation of trees. Thus, the initial value of the syntactic parameter of the main function procedure is the input tree of the transducer. Because of their relevance to syntax-directed semantics, our main interest is in the total deterministic n-level tree transducer, which computes a total function $T_\Sigma \to T_\Delta$. However, we also study the nondeterministic case, because it is easier to treat; then a relation in $T_\Sigma \times T_\Delta$ is computed. For the interested reader, an example of a (total deterministic) two-level tree transducer, relevant to the denotational semantics of programming languages, is presented at the end of the introduction.

The main result of this paper is the equivalence of the high-level tree transducers and the *iterated pushdown tree transducers*, in the sense that they compute the same class of translations. Such iterated pushdown tree transducers can be considered as systems of recursive function procedures that have one parameter consisting of a pushdown-like datastructure, and that deliver a result of basic type (level 0). More precisely, the parameter ranges over pushdowns of pushdowns of ... of pushdowns of (pointers to) the input tree of the transducer, and, hence, it is also called a syntactic parameter. Using this characterization result we also show that high-level tree transducers can compute the composition of any number of translations defined by attribute grammars [Knu]. (We note that it is shown in [CouFra] that macro tree transducers can simulate attribute grammars, viewed as tree transducers [EngFil, ChiMar, CouFra].)

Since in [EngVog3] the macro tree transducers are characterized by pushdown tree transducers (in which the syntactic parameter ranges over pushdowns of pointers to the input tree), the present paper should be considered as a direct continuation of the work in [EngVog3]. Our main result is inspired by the characterization of n-level tree grammars by n-iterated pushdown tree automata [DamGue2], and of n-level string grammars by n-iterated pushdown automata [DamGoe] (and we also give alternative proofs of these results). We use the same methods and tools as in [EngVog3], but many ideas and concepts in this paper are based on those in [Dam, DamGoe, Gue, DamGue1, DamGue2].

For the description of both high-level tree transducers and iterated pushdown tree transducers, we use the unifying concept of "grammar with storage", introduced in [Eng5, EngVog3]. A storage consists of configurations, predicates that test the configurations, and instructions that transform configurations. Roughly speaking, the type of grammar used determines the type of the results of the function procedures, whereas the type of storage used determines the type of their syntactic parameter. In this formalism of grammars with storage, n-level tree transducers are n-level tree grammars with a storage of type "tree", denoted by TR, and the n-iterated pushdown tree transducers are 0-level (or: regular) tree grammars with a storage of type "n-iterated pushdown of trees", denoted by P^n(TR) (where P is an operator on storage types called the pushdown operator). The configurations of TR are trees; for every such configuration, the label of the root can be tested and a direct subtree can be selected. The configurations of P^n(TR) are pushdowns of pushdowns of ... of pushdowns (n times) of (pointers to) trees, cf. [Gre, Mas, Eng6, DamGoe, EngVog3].

The main advantage of formulating both high-level tree transducers and iterated pushdown tree transducers in the framework of grammars with storage, is the fact that we can prove our characterization result *inductively* (rather than having to provide an immediate construction). To show this we sketch the main lines of the proof. In the first step, every n-level tree transducer (which is an n-level tree grammar with storage of type TR) is transformed into an equivalent 0-level tree grammar with a storage of type "n-level applicative term of trees", denoted by n-AT(TR). Since n-AT(TR) exactly reflects the rewriting mechanism inherent in n-level tree grammars, this first step is intuitively straightforward (cf. [DamGue1, 2]). For the second, major step we use the concept of simulation of storage types [Eng6, EngVog3] and the so-called "Justification Theorem" (cf. Theorem 4.18 of [EngVog3]). This theorem says that, whenever two storage types S_1 and S_2 are equivalent (for short: $S_1 \equiv S_2$) in the sense that S_1 can simulate S_2 and vice versa, then the class of translations computed by grammars with storage of type S_1 is equal to the class of translations computed by grammars (of the same type) with storage of type S_2. Thus, to prove the characterization it would suffice to show that n-AT(TR) and P^n(TR) are equivalent storage types. However, for technical reasons, we need a variation P_{bex} of P, where bex stands for "bounded excursion", and we show that $n\text{-AT}(S) \equiv P^n_{\text{bex}}(S)$ for an arbitrary storage of type S. This equivalence is proved inductively: for every storage of type S, $(n+1)\text{-AT}(S) \equiv P_{\text{bex}}(n\text{-AT}(S))$. The latter equivalence may be seen as the kernel of our proof: one functional level is replaced by one level

of pushdowns. Finally, we show that the 0-level tree grammars with storage of type $P^n_{\mathrm{bex}}(\mathrm{TR})$ compute the same class of translations as 0-level tree grammars with storage of type $P^n(\mathrm{TR})$.

This paper is organized in 8 sections of which the second contains preliminaries. In Sects. 3.1 and 3.2 the two concepts of "grammar with storage" and "storage type simulation" are recalled from [EngVog3], and in Sect. 3.3 the notion of coding is introduced as a special case of storage type simulation. In Sect. 3.4 the pushdown operators P and P_{bex} are defined, and it is shown that P_{bex} preserves the equivalence of storage types. The main devices of this paper are formalized in Sect. 4 (for arbitrary storage of type S): n-level S transducers and n-iterated pushdown S transducers. In Sect. 5 we demonstrate that the rewriting mechanism of n-level S transducers can be completely captured by the storage of type "n-level applicative term of S", i.e., n-AT(S), and in Sect. 6 the equivalence of n-AT(S) and $P^n_{\mathrm{bex}}(S)$ is shown. Section 7 contains the main result of the paper: the equivalence of n-level tree transducers and n-iterated pushdown tree transducers. Finally, in Sect. 8 some consequences of the results proved in this paper are pointed out, e.g., the characterization of n-level tree transducers by the n-fold composition of macro tree transducers, and hence by the composition of attribute grammars.

Conceptually, the paper may be divided into three parts. The first part (Sect. 3) concerns concepts of general automata theory. The reader who is already familiar with these concepts, can skip this part on first reading. The second part (Sect. 4–Sect. 6) contains the definitions of high-level S transducers and of iterated (bounded excursion) pushdown S transducers, respectively, and the proof of their equivalence. The third part (Sect. 7) is dedicated to the special case of high-level tree transducers: it contains the proof of their equivalence with iterated pushdown tree transducers.

An extended abstract of this paper can be found in [EngVog4]. Some of the proofs are presented in more detail in Part 4 of [Vog3].

Example. In this example we informally discuss a 2-level tree transducer M that performs type checking for programs of a small block-structured language, called CHECK. For a CHECK program, M checks whether the types in assignment statements are correct. It is well known that type checking cannot be captured in the derivations of context-free grammars; in the literature, usually attribute grammars are used to perform this analysis, cf., e.g., [Wat].

Of course, since tree transducers are schematic devices, which transform trees into trees, we can at most expect that, for the abstract syntax tree of a CHECK-program P, M generates a tree t_P such that the interpretation of t_P in an appropriate semantic domain yields the answer of checking P. However, to exclude the trivial solution, in which the whole checking is shifted into the semantic domain, we try to work as much as possible on the syntactic level.

The syntax of CHECK is given by the context-free grammar G_{CHECK}, which is specified by the following productions.

r_1: $P \rightarrow$ **program** D; S **end**
r_2: $D \rightarrow$ **var** I:T

r_3: $D \to D$; **var** $I{:}T$
r_4: $S \to I := I$
r_5: $S \to$ **begin** D; S **end**
r_6: $S \to S; S$
r_7: $I \to a$
r_8: $I \to b$
r_9: $T \to$ **int**
r_{10}: $T \to$ **bool**

where P, D, S, I, and T are nonterminals, and the other symbols are terminal. An example program P of CHECK is the following.

program var b: **bool**; **var** a: **int**;
begin var b: **int**; $b := a$ **end**;
$b := a$ **end**

Let T_Σ denote the set of abstract syntax trees of CHECK-programs. The abstract syntax tree t of P is shown in Fig. 1.

The first assignment $b := a$ of P, which occurs in an inner block, is correct, whereas the second statement gives a type conflict. Now we present the 2-level tree transducer M that performs type checking of CHECK-programs. M is described in the terminology of program schemes as a system of recursive function procedures over the following three semantic domains: TYPE = {int, bool, undef}, BOOL = {true, false}, and IDENT = {a, b}.

Function procedures of M:

Check-prog: $T_\Sigma \to \text{BOOL}$
Envir: $T_\Sigma \to ((\text{IDENT} \to \text{TYPE}) \to (\text{IDENT} \to \text{TYPE}))$
Inenv: $T_\Sigma \to (\text{IDENT} \to \text{TYPE})$
Check: $T_\Sigma \to ((\text{IDENT} \to \text{TYPE}) \to \text{BOOL})$
Id: $T_\Sigma \to \text{IDENT}$
Ty: $T_\Sigma \to \text{TYPE}$

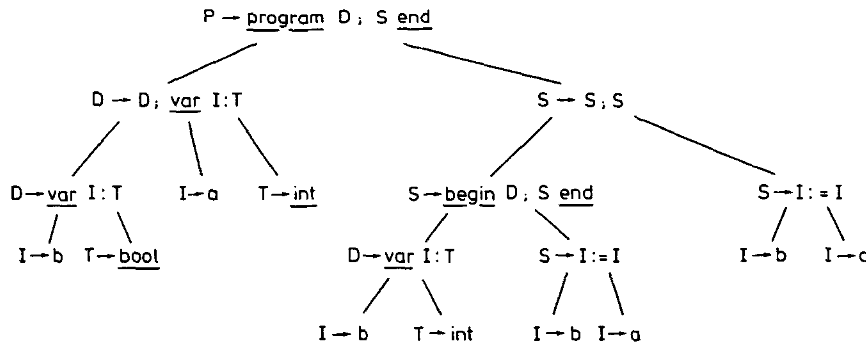

Fig. 1. Abstract syntax tree t of a CHECK-program P

For example, the function procedure Check has a syntactic parameter of type "abstract syntax tree" and delivers a result of functional level 2 and of type (IDENT→TYPE)→BOOL; a value of this result computes a boolean for each "environment", which is a function giving types to identifiers. Check-prog is the main function procedure of this system. In the definition of a function procedure, basic function symbols (of Δ) may be involved that are interpreted as operations over the semantic domains. We use the following basic function symbols: equal, and, cond, int, bool, undef, a, and b of which the interpretation should be obvious (e.g., cond is interpreted as function of type BOOL × TYPE × TYPE→TYPE and $\text{cond}(b, t_1, t_2) = \text{if } b \text{ then } t_1 \text{ else } t_2$). We note that the constants int, bool, and undef denote themselves.

Definition of function procedures:
We use the formal (semantic) parameters y_{ID} and y_{EN} of type IDENT and IDENT→TYPE, respectively.

$$\begin{aligned}
&\text{Check-prog}[\mathbf{program}\ D;\ S\ \mathbf{end}] = \text{Check}[S]\ (\text{Envir}[D]\ (\text{Inenv}[\text{D}])),\\
&\text{Envir}[\mathbf{var}\ I\colon T]\ y_{\text{EN}}\ y_{\text{ID}} =\\
&\quad \text{cond}(\text{equal}(\text{Id}[I], y_{\text{ID}}), \text{Ty}[T], y_{\text{EN}}(y_{\text{ID}})),\\
&\text{Envir}[D;\ \mathbf{var}\ I\colon T]\ y_{\text{EN}}\ y_{\text{ID}} =\\
&\quad \text{cond}(\text{equal}(\text{Id}[I], y_{\text{ID}}), \text{Ty}[T], \text{Envir}[D]\ y_{\text{EN}}\ y_{\text{ID}}),\\
&\text{Inenv}[\mathbf{var}\ I\colon T]\ y_{\text{ID}} = \text{undef},\\
&\text{Inenv}[D;\ \mathbf{var}\ I\colon T]\ y_{\text{ID}} = \text{undef},\\
&\text{Check}[I_1 := I_2]\ y_{\text{EN}} = \text{equal}(y_{\text{EN}}(\text{Id}[I_1]), y_{\text{EN}}(\text{Id}[I_2])),\\
&\text{Check}[\mathbf{begin}\ D;\ S\ \mathbf{end}]\ y_{\text{EN}} = \text{Check}[S](\text{Envir}[D]\ y_{\text{EN}}),\\
&\text{Check}[S_1;\ S_2]\ y_{\text{EN}} = \text{and}\ (\text{Check}[S_1]\ y_{\text{EN}}, \text{Check}[S_2]\ y_{\text{EN}}),\\
&\text{Id}[a] = a, \text{ and } \text{Id}[b] = b,\\
&\text{Ty}[\mathbf{int}] = \text{int}, \text{ and } \text{Ty}[\mathbf{bool}] = \text{bool}.
\end{aligned}$$

Note that, in the usual way, the expression $\text{Envir}[\ldots]\ y_{\text{EN}}\ y_{\text{ID}}$ denotes $\text{Envir}(\ldots)(y_{\text{EN}})(y_{\text{ID}})$ (and similarly for $\text{Check}[\ldots]\ y_{\text{EN}}$). It should be clear that these function definitions are close to the usual denotational semantics notation. In Sect. 4 part of this example is given in our notation of tree transducers.

Finally, we show parts of the computation that our system M performs on the abstract syntax tree t of P. We use the following abbreviations: $t = r_1(\text{decl}_1, r_6(\text{stat}_1, \text{stat}_2))$, $\text{decl}_1 = r_3(r_2(r_8, r_{10}), r_7, r_9)$, $\text{stat}_1 = r_5(\text{decl}_2, \text{stat}_3)$, $\text{decl}_2 = r_2(r_8, r_9)$, $\text{stat}_2 = r_4(r_8, r_7)$, and $\text{stat}_3 = \text{stat}_2$.

First we type check the second statement stat_2 of P (using the abbreviations e_0 and e_1 for $\text{Inenv}[\text{decl}_1]$ and for $\text{Envir}[\text{decl}_1]\ e_0$, respectively):

$$\begin{aligned}
&\text{Check}[\text{stat}_2](\text{Envir}[\text{decl}_1]\ e_0)\\
&= \text{equal}(e_1(b), e_1(a))\\
&= \text{equal}(\text{cond}(\text{equal}(a, b), \text{int}, \text{cond}(\text{equal}(b, b), \text{bool}, \text{undef})),\\
&\qquad\quad \text{cond}(\text{equal}(a, a), \text{int}, \text{cond}(\text{equal}(b, a), \text{bool}, \text{undef}))).
\end{aligned}$$

Interpreting this tree yields false, which indicates the type conflict in stat_2.

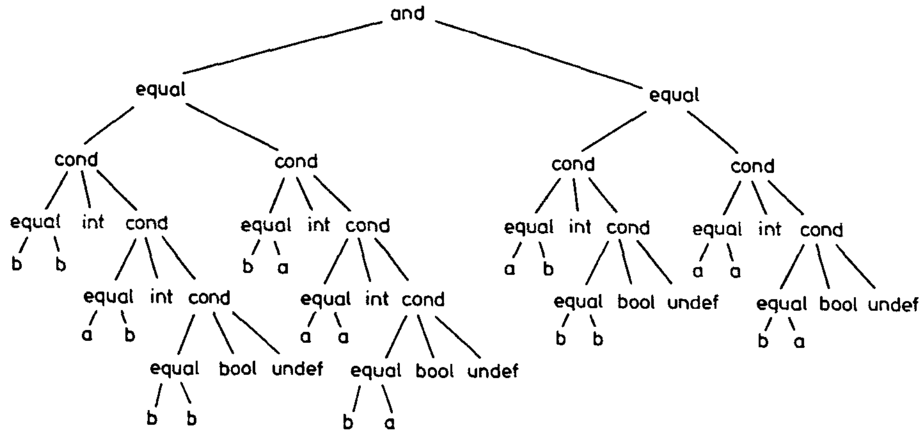

Fig. 2. Output tree t_P computed by checking P

In a similar way the first statement stat_1 of P is checked:

Check[stat_1] e_1 = equal(cond(equal(b, b), int,
cond(equal(a, b), int, cond(equal(b, b), bool, undef))),
cond(equal(b, a), int, cond(equal(a, a), int,
cond(equal(b, a), bool, undef)))), and this tree is interpreted by true.

Thus Check-prog[t] = Check[$r_6(\text{stat}_1, \text{stat}_2)$] e_1 = and (Check[stat_1] e_1, Check[stat_2] e_1) = t_P where the tree t_P is shown in Fig. 2. This means that M transforms t into t_P. Clearly, t_P is interpreted by false, and hence the 2-level tree transducer M detects a type conflict. □

2. Preliminaries

For most unexplained notions we refer the reader to [EngVog3]. That paper will from now on be cited as [EV]. However, for the convenience of the reader, we recall in Sect. 2.1 some of the most frequently used notions from [EV]. In Sect. 2.2 the concepts of derived types [Mai] and of applicative terms are defined formally.

2.1 Basic Notations

The set $\{0, 1, 2, \ldots\}$ of non-negative integers is denoted by nat. For every $k \geqq 0$, $[k]$ is the set $\{1, \ldots, k\} \subseteq \text{nat}$. The empty string is denoted by λ.

Let R, R_1, and R_2 be three relations. The inverse, the domain, and the range of R are denoted by R^{-1}, dom(R), and range(R), respectively. The composition of R_1 and R_2, denoted by $R_1 \circ R_2$, is the set $\{(x, y) \mid (x, z) \in R_1 \text{ and } (z, y) \in R_2$ for some $z\}$, and the n-fold composition of R is denoted by R^n. The transitive closure and the reflexive, transitive closure of R, are denoted by R^+ and R^*, respectively. The notations are extended in an obvious way to classes of relations.

Very often, the concept of *substitution* of objects into strings or trees is used. Let v be a string (or tree), let U and U' be arbitrary sets, and let θ

Fig. 3. Picture of the trees (a) $t_1 = \sigma(\sigma(\alpha(), b), a)$ and (b) $t_2 = \sigma(\sigma(\alpha(), \beta()), \alpha())$

be a mapping from U to the power set of U'. Then $v[u \leftarrow \theta(u); u \in U]$ denotes the set of strings (or trees) obtained from v by replacing every occurrence of u in v by an element of $\theta(u)$, where different occurrences may be replaced by different elements of $\theta(u)$. Almost in every situation, $\theta(u)$ is a singleton, which is specified by its element.

A *ranked set* is a pair $\langle \Delta, \text{rank} \rangle$, where Δ is a (possibly infinite) set of symbols and rank: $\Delta \rightarrow \text{nat}$ is a mapping. If Δ is finite, it is called a *ranked alphabet*. If rank is understood from the context, then $\langle \Delta, \text{rank} \rangle$ is also denoted by Δ. For every $n \geqq 0$, the set of symbols of Δ with rank n is denoted by Δ_n. For a ranked set $\langle \Delta, \text{rank} \rangle$ and a (usual) set A, the set of *trees over Δ indexed by A*, denoted by $T_\Delta(A)$, is the smallest set T which satisfies the following two properties. (i) $A \subseteq T$. (ii) If $\sigma \in \Delta$ with $\text{rank}(\sigma) = k$, and $t_1, \ldots, t_k \in T$, then $\sigma(t_1, \ldots, t_k) \in T$. The set of *trees over Δ*, denoted by T_Δ, is $T_\Delta(\varnothing)$. Figure 3 shows how we draw (indexed) trees in examples. Let Δ be a ranked alphabet containing the symbols σ, α, and β of rank 2, 0, and 0, respectively; let $A = \{a, b\}$. Figure 3a is a picture of the indexed tree $t_1 = \sigma(\sigma(\alpha(), b), a)$ in $T_\Delta(A)$, and Fig. 3b shows the tree $t_2 = \sigma(\sigma(\alpha(), \beta()), \alpha())$ in T_Δ.

The *yield* of a tree t in T_Δ, denoted by yield(t), is the concatenation of its leaves from the left to the right. Formally, (i) for $\sigma \in \Delta_0$, yield $(\sigma()) = \sigma$ and (ii) for $\sigma \in \Delta_k$ with $k \geqq 1$ and $t_1, \ldots, t_k \in T_\Delta$, yield $(\sigma(t_1, \ldots, t_k)) = \text{yield}(t_1) \ldots \text{yield}(t_k)$, where, for strings $w_1, \ldots, w_k$, their concatenation is denoted by $w_1 \ldots w_k$. The mapping yield is extended to relations $R \subseteq A \times T_\Delta$, where A is an arbitrary set, by defining $\text{yield}(R) = \{(s, w) \mid (s, t) \in R \text{ and } \text{yield}(t) = w\}$. This notation can be extended to classes of relations in an obvious way. Throughout this paper Ω denotes a countably infinite ranked set such that for every $n \geqq 0$, Ω_n is infinite.

We assume the reader to be familiar with regular grammars and context-free grammars, regular tree grammars, and context-free tree grammars (for formal definitions see [HopUll], [GecSte], and [EngSch], respectively). For context-free tree grammars we briefly recall the basic notions. A *context-free tree grammar* G is a tuple $(N, \Delta, A_{\text{in}}, R)$, where N and Δ are (disjoint) ranked alphabets of *nonterminals* and *terminals*, respectively, $A_{\text{in}} \in T_N$ is the *initial term*, and R is a finite set of *rules* of the form $A(y_1, \ldots, y_k) \rightarrow \zeta$, where $k \geqq 0$, $A \in N_k$, $\zeta \in T_{N \cup \Delta}(Y_k)$, and $Y_k = \{y_1, \ldots, y_k\}$. Note that we allow G to have an initial term rather than only an initial nonterminal. However, this does not increase the generating

power of context-free tree grammars. The set of *sentential forms* of G, denoted by $\mathrm{SF}(G)$, is the set $T_{N\cup\Delta}$. The *(OI-) derivation relation* of G, denoted by $=\!(G)\!\Rightarrow$, is a binary relation on $\mathrm{SF}(G)$ defined as follows. For $\xi_1, \xi_2 \in \mathrm{SF}(G)$, $\xi_1 =\!(G)\!\Rightarrow \xi_2$ iff there is a rule $A(y_1, \ldots, y_k) \rightarrow \zeta$ in R for some $k \geqq 0$, $A \in N_k$, and $\zeta \in T_{N\cup\Delta}(Y_k)$, there is a $\xi \in T_{N\cup\Delta}(\{z\})$ in which z occurs exactly once and z does not occur in a subtree of the form $B(\zeta_1, \ldots, \zeta_r)$ of ξ (with $B \in N_r$), and there are $\xi'_1, \ldots, \xi'_k \in \mathrm{SF}(G)$, such that $\xi_1 = \xi[z \leftarrow A(\xi'_1, \ldots, \xi'_k)]$ and $\xi_2 = \xi[z \leftarrow \zeta[y_i \leftarrow \xi'_i; i \in [k]]]$. The *language* generated by G, denoted by $L(G)$, is the set $\{t \in T_\Delta \mid A_{\mathrm{in}} =\!(G)\!\Rightarrow^* t\}$. A regular tree grammar is a context-free tree grammar such that every nonterminal has rank 0.

The classes of regular (or right-linear) grammars, context-free grammars, regular tree grammars, and context-free tree grammars are denoted by REG, CF, RT, and CFT, respectively. In general, for $X \in \{\mathrm{REG}, \mathrm{CF}, \mathrm{RT}, \mathrm{CFT}\}$, we use the same notations for specifying X-grammars as for a context-free tree grammar, i.e., an *X-grammar* is specified by a tuple $(N, \Delta, A_{\mathrm{in}}, R)$, where N and Δ are disjoint alphabets of nonterminals and terminals, respectively, A_{in} is the initial term, and R is a finite set of rules. Note that, depending on X, the sets N and Δ may or may not be ranked alphabets. For $X \in \{\mathrm{REG}, \mathrm{CF}, \mathrm{RT}\}$, as usual, the initial term is a single nonterminal. The set of sentential forms, the derivation relation, and the generated language of G are denoted by $\mathrm{SF}(G)$, $=\!(G)\!\Rightarrow$, and $L(G)$, respectively. The class of languages, which is generated by X-grammars, is also denoted by X. We also consider X-grammars with infinite sets of nonterminals and rules. The notions of sentential form, derivation relation, and generated language are defined in exactly the same way as for ordinary, finite X-grammars.

Every regular tree language over a ranked alphabet Δ can be accepted by a *total deterministic bottom-up tree automaton* $A = (P, \Delta, \delta, F)$, where P is a set of states, $F \subseteq P$ is the set of final states, and δ is a family $\{\delta_\sigma \mid \sigma \in \Delta\}$ of mappings such that, for every $\sigma \in \Delta_k$ with $k \geqq 0$, $\delta_\sigma \colon P^k \rightarrow P$. The function $\delta \colon T_\Delta \rightarrow P$ is defined in the usual way recursively on the structure of the tree in T_Δ, and $L(A) = \{t \in T_\Delta \mid \delta(t) \in F\}$. From the accepting point of view, regular tree languages are also called *recognizable tree languages* and the corresponding class RT of tree languages is also denoted by RECOG.

2.2 Applicative Terms

Here we formally define the concepts of derived types and applicative terms. Most of the definitions are taken over from [Dam].

Let Q be a set of *types*. Then, a string $w \in Q^*$ of length k is viewed as a mapping $w \colon [k] \rightarrow Q$. Thus, $w(i)$ is the i-th letter of w. *A Q-set* is a pair $\langle V, \mathrm{type}\rangle$, where V is a set and type: $V \rightarrow Q$ is a mapping. For every $q \in Q$, $V^q = \{\gamma \in V \mid \mathrm{type}(\gamma) = q\}$. For every $w \in Q^*$ of length k, we define $V^w = \{(\gamma_1, \ldots, \gamma_k) \mid \gamma_i \in V^{w(i)}$ for every $i \in [k]\}$, and thus $V^\lambda = \{()\}$. For two Q-sets $\langle V_1, \mathrm{type}_1\rangle$ and $\langle V_2, \mathrm{type}_2\rangle$, we write $V_1 \subseteq V_2$ iff $V_1^q \subseteq V_2^q$ for every $q \in Q$; and if V_1 and V_2 are disjoint, then $V_1 \cup V_2$ denotes the Q-set $\langle V_1 \cup V_2, \mathrm{type}\rangle$, where for every $\gamma \in V_i$ with $i \in \{1, 2\}$, $\mathrm{type}(\gamma) = \mathrm{type}_i(\gamma)$.

If there is no confusion the Q-set $\langle V, \text{type}\rangle$ will also be denoted by V and the mapping "type" will be understood. The fact that an element $\gamma \in V$ has type q, i.e., $\text{type}(\gamma)=q$, is abbreviated by $\gamma: q$.

For a set V of symbols and an arbitrary set Φ, $V\langle\Phi\rangle$ denotes the set $\{\gamma\langle\phi\rangle \mid \gamma\in V, \phi\in\Phi\}$. If $\langle V, \text{type}\rangle$ is a Q-set, then $\langle V\langle\Phi\rangle, \text{type}'\rangle$ is also a Q-set, where for every $\gamma\langle\phi\rangle\in V\langle\Phi\rangle$, $\text{type}'(\gamma\langle\phi\rangle)=\text{type}(\gamma)$. The mapping type′ is also denoted by type. The same formalism is used, if V is a ranked set instead of a Q-set.

The concept of derived types allows to denote the type of functions and also of high level functionals. For every set Q of types, we define the set $D(Q)=Q^*\times Q$. Then the set of *derived types over* Q, denoted by $D^*(Q)$, is the set $\bigcup\{D^n(Q) \mid n\geqq 0\}$, where $D^0(Q)=Q$ and for every $n\geqq 0$, $D^{n+1}(Q)=D(D^n(Q))$. $D^+(Q)$ denotes the set $\bigcup\{D^n(Q) \mid n\geqq 1\}$.

The *level* of a derived type $\tau\in D^n(Q)$ with $n\geqq 0$, denoted by $\text{level}(\tau)$, is n. For any $D^*(Q)$-set $\langle V, \text{type}\rangle$, the level of $\gamma\in V$, denoted by $\text{level}(\gamma)$, is $\text{level}(\text{type}(\gamma))$. For $n\geqq 0$, the set $\{\gamma\in V \mid \text{level}(\gamma)=n\}$ is denoted by $V^{=n}$; similarly we use the denotation $V^{\leqq n}$.

We note that every derived type $\tau\in D^n(Q)$ with $n\geqq 0$ can be uniquely written as $\tau=(\alpha n, \ldots(\alpha 2, (\alpha 1, q))\ldots)$ with $\alpha i\in(D^{i-1}(Q))^*$ for every $1\leqq i\leqq n$ and $q\in Q$.

Applicative terms are of central importance in this paper. They represent the symbolic application of objects of derived types to arguments of appropriate type. Let V be a $D^*(Q)$-set. The $D^*(Q)$-set of *applicative terms over* V, denoted by $\text{AT}(V)$, is the smallest $D^*(Q)$-set AT which satisfies (i) and (ii).

(i) $V\subseteq \text{AT}$.

(ii) Let $\alpha\in D^n(Q)^*$ of length k with $n, k\geqq 0$. For every $j\in[k]$, let $\xi_j\in\text{AT}^{\alpha(j)}$. Let $\nu\in D^n(Q)$ and let $\xi_0\in\text{AT}^{(\alpha,\nu)}$. Then $\xi_0(\xi_1, \ldots, \xi_k)\in\text{AT}^{\nu}$.

It is easy to see from (i) and (ii) that every applicative term $\xi\in\text{AT}(V)$ has a *unique decomposition* $\xi=\gamma\bar{\xi}_r\ldots\bar{\xi}_1$ with $\gamma\in V$, $r\geqq 0$, and each $\bar{\xi}_i$ is a tuple of applicative terms. The element γ is called the *top* of ξ and is denoted by $\text{top}(\xi)$. If $\xi\in\text{AT}(V)^{\nu}$ for some $\nu\in D^n(Q)$, then $\gamma\in V^{\tau}$ for some $\tau=(\alpha r, \ldots(\alpha 1, \nu)\ldots)$ with $\alpha i\in D^{n+i-1}(Q)^*$, and for every $i\in[r]$, $\bar{\xi}_i\in\text{AT}(V)^{\alpha i}$.

From now on let $Q=\{q\}$. We illustrate the notions concerning applicative terms by an example.

Example. Let $\gamma: ((q, q)(\lambda, q), (qq, q))$, $\sigma: ((q, q), (\lambda, q))$, $\nu: (\lambda, (q, q))$, $\delta: (q, q)$, $\alpha: (\lambda, q)$, and $\beta: q$ be elements of a $D^*(Q)$-set V. Clearly, γ, σ, and ν have level 2, δ and α have level 1, and β has level 0. By part (i) of the definition of applicative terms, $\delta\in\text{AT}(V)^{(q,q)}$. Since $\beta\in\text{AT}(V)^q$, it follows from part (ii) that $\delta(\beta)\in\text{AT}(V)^q$. In the same way it can be checked that $\xi=\gamma(\delta, \sigma(\delta))(\alpha(), \nu()(\beta))\in\text{AT}(V)^q$, where $()$ is the empty list of parameters. The unique decomposition of ξ is $\gamma\bar{\xi}_2\bar{\xi}_1$ where $\bar{\xi}_2=(\delta, \sigma(\delta))$ and $\bar{\xi}_1=(\alpha(), \nu()(\beta))$; thus $\text{top}(\xi)=\gamma$. □

We *consider every* $D(Q)$-*set* $\langle V, \text{type}\rangle$ *as a ranked set* $\langle V, \text{rank}\rangle$: for $\gamma\in V$, if $\text{type}(\gamma)=(q^k, q)$ with $k\geqq 0$, then $\text{rank}(\gamma)=k$. Hence, for a $D(Q)$-set V and a Q-set W, $\text{AT}(V\cup W)^q$ is the set of trees over V indexed by W, i.e., $\text{AT}(V\cup W)^q = T_V(W)$. In particular $\text{AT}(V)^q=T_V$. Actually, this connection between applicative terms and trees can be generalized in a very natural way. Let $n\geqq 0$ and let V be a $D^*(Q)$-set. Then every applicative term over V of level n can be

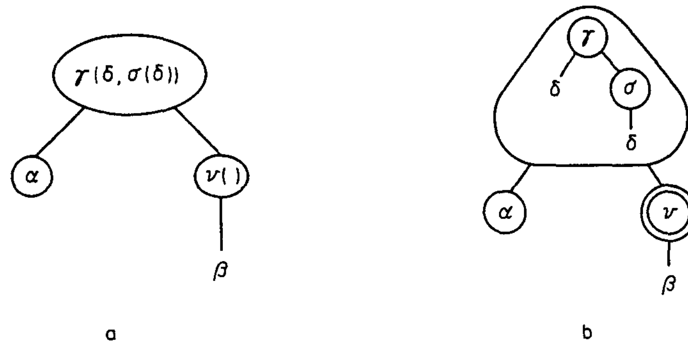

Fig. 4. Tree-form of an applicative term

represented as an indexed tree: the labels of its nodes are applicative terms over V each of level $n+1$, and it is indexed with symbols of V each of level n. This point of view is formalized as follows.

Definition. Let $n \geqq 0$ and let V be a $D^*(Q)$-set. The ranked set *associated with* (V, n) is $\langle \mathrm{AT}(V)^{=n+1}, \mathrm{rank}\rangle$ and, if $\xi \in \mathrm{AT}(V)^{=n+1}$ with $\xi: (\alpha, v)$ and the length of α is $k \geqq 0$, then $\mathrm{rank}(\xi) = k$. □

Lemma. *Let* $n \geqq 0$ *and let* V *be a* $D^*(Q)$*-set. Then* $\mathrm{AT}(V)^{=n} = T_\Psi(V^{=n})$, *where* $\Psi = \mathrm{AT}(V)^{=n+1}$, *the ranked set associated with* (V, n).

Proof. Follows immediately from the definition of applicative terms. □

The indexed tree, which corresponds to an applicative term ξ in the sense of the previous lemma, is called the *tree-form* of ξ (cf. [DamGue2]). As an example, we describe the tree-form of the applicative term $\xi = \gamma(\delta, \sigma(\delta))(\alpha(), v()(\beta))$ of level 0, of the previous example. The tree-form of ξ consists of a root of rank 2 labelled by $\xi_1 = \gamma(\delta, \sigma(\delta))$, and two direct subtrees. The left subtree only consists of a leaf labelled by $\xi_2 = \alpha$, and the right subtree consists of a root labelled by $\xi_3 = v()$ and a leaf labelled by β (note that β is an element of the "index" set $V^{=0}$). The tree-form of ξ is shown in Fig. 4a.

Of course, also the applicative terms ξ_1, ξ_2, and ξ_3 (of level 1) have a tree-form. Hence, ξ can be drawn as an indexed tree of indexed trees over V (cf. Fig. 4b; the number of circles surrounding an applicative term indicates its level!). In all subsequent examples we draw applicative terms in this "iterated" tree-form, i.e., also sub-applicative terms are drawn in tree-form. We note that, if V only contains symbols of level n, then the tree-form of an applicative term over V of level 0 is a tree of trees of ... of trees (n times) of symbols of V.

3. Grammars with Storage, Simulation of Storage Types, and Pushdown Operators

This section recalls from [EV] the two main tools which we use to prove the characterization of n-level tree transducers: grammars with storage (Sect. 3.1)

and storage type simulation (Sect. 3.2). We refer the reader for a broad discussion of these concepts to Sect. 3 and Sect. 4 of [EV]. In Sect. 3.3, we introduce the concept of coding of storage types as a special case of simulation. Finally, in Sect. 3.4, we recall from [EV] the definition of the pushdown operator, define a modification of it (called "bounded excursion"), and prove that this modified operator preserves the equivalence of storage types.

3.1 Grammars with Storage

The idea of the concept "grammars with storage" goes back to a suggestion of [Sco] in which he advocates the strict separation of the concepts of program and storage type. In Scott's sense a program is a flowchart which can test and transform the configurations of a storage by means of predicates and instructions, respectively. In [Eng5] this concept is generalized by allowing context-free grammars as program, and in [EngVog2, EV] macro grammars and context-free tree grammars serve this purpose. There the concept of "grammars with storage" is heavily exploited in order to achieve a pushdown tree transducer characterization for the macro tree transducer.

3.1. Definition. A *storage type* S is a tuple (C, P, F, m, I, E), where C is a non-empty set of *(S-)configurations*, P and F are sets of *(S-)predicate-* and *(S-)instruction symbols*, respectively, m interprets every $p \in P$ as a mapping $m(p): C \to \{\text{true}, \text{false}\}$ and every $f \in F$ as a partial function $m(f): C \to C$. Finally, I is a set of *input elements* and E is a set of *encodings*, where every encoding is a partial function $e: I \to C$. □

In the rest of this paper S denotes the storage type (C, P, F, m, I, E) if not specified otherwise. In the usual way m is extended to the set $\mathrm{BE}(P)$ of boolean expressions over P, where **true** and **false** are the boolean constants. We use "predicate" and "instruction" as shorthands for "predicate symbol" and "instruction symbol", respectively, and we say that S *contains an identity* if there is an instruction $f \in F$ such that $m(f)$ is the identity on C.

The tree storage type is of particular importance in this paper. It has trees as configurations and captures in its predicates and instructions the possibility of testing the label of the root of a tree and of selecting a direct subtree of a tree. Note that Ω is the infinite ranked set mentioned in Sect. 2.1.

3.2. Definition. The *tree storage type* TR is the storage type (C, P, F, m, I, E), where $C = T_\Omega$, $P = \{\text{root} = \sigma \mid \sigma \in \Omega\}$, $F = \{\text{sel}_i \mid i \geqq 1\}$, and for every $t = \delta(t_1, \ldots, t_k)$ in C with $\delta \in \Omega$ of rank $k \geqq 0$ and $t_1, \ldots, t_k \in T_\Omega$, $m(\text{root} = \sigma)(t) = (\delta = \sigma)$, $m(\text{sel}_i)(t) = t_i$ if $i \leqq k$, and undefined otherwise, $I = C$, and $E = \{e \mid e$ is the identity on T_Σ for some finite subset Σ of $\Omega\}$. □

A grammar with storage computes a translation from the set of input elements of the storage type to the set of terminal trees or strings of the grammar. Hence, we call such a device an $X(S)$-transducer, where X is the class of grammars used and S is the storage type. In the rest of this section, the modifier

X ranges over {REG, CF, RT, CFT}. However, after having recalled in Sect. 4 the definition of n-level tree grammars, we use Definitions 3.3–3.5 also for $X = n$-T, where n-T denotes the class of n-level tree grammars.

3.3. Definition. An $X(S)$-*transducer* M is a tuple $(N, e, \Delta, A_{in}, R)$, where

- N, Δ, and A_{in} are alphabets of nonterminals, terminals, and the initial term, respectively, as defined for X-grammars
- $e \in E$ is the encoding of M
- R is a finite set of rules; each rule is of the form $\Theta \rightarrow$ **if** b **then** ζ, where $\Theta \rightarrow \xi$ is a rule of a usual X-grammar, $b \in \mathrm{BE}(P)$, and ζ can be obtained from ξ by replacing every occurrence of a nonterminal B by $B\langle f \rangle$ for some $f \in F$, i.e., $\zeta \in \xi[B \leftarrow B\langle F \rangle; B \in N]$.

An $X(S)$-transducer is *deterministic* if for every $c \in C$ and every two different rules $\Theta \rightarrow$ **if** b_1 **then** ζ_1 and $\Theta \rightarrow$ **if** b_2 **then** ζ_2, $m(b_1)(c) = \text{false}$ or $m(b_2)(c) = \text{false}$. □

If r: $\Theta \rightarrow$ **if** b **then** ζ is a rule of an $X(S)$-transducer M and A is the nonterminal in Θ, then r is an (A, b)-*rule* of M, b is the *test* of r, and ζ is the *right-hand side term* of r. If $b = \mathbf{true}$, then we abbreviate r by $\Theta \rightarrow \zeta$. The construct $\Theta \rightarrow$ **if** b **then** ζ_1 **else** ζ_2 abbreviates the two rules $\Theta \rightarrow$ **if** b **then** ζ_1 and $\Theta \rightarrow$ **if not** b **then** ζ_2.

The translation computed by an $X(S)$-transducer is defined via the notion of associated grammar.

3.4. Definition. Let $M = (N, e, \Delta, A_{in}, R)$ be an $X(S)$-transducer. The X-*grammar* $G(M) = (N', \Delta, A, R')$ *associated with* M is defined by $N' = N\langle C \rangle$, A is any element of N, and R' is obtained as follows.

If $\Theta \rightarrow$ **if** b **then** ζ is in R, then for every $c \in C$ such that $m(b)(c) = \text{true}$ and such that every instruction occurring in ζ is defined on c, the rule $\Theta\langle c \rangle \rightarrow \zeta'$ is in R', where $\Theta\langle c \rangle = \Theta[A \leftarrow A\langle c \rangle; A \in N]$ and $\zeta' = \zeta[B\langle f \rangle \leftarrow B\langle m(f)(c) \rangle; B \in N, f \in F]$. □

Note that an associated grammar may have infinitely many nonterminals and infinitely many rules. Recall from Sect. 2.1 that, for an X-grammar G, $\mathrm{SF}(G)$ denotes the set of sentential forms of G.

3.5. Definition. Let $M = (N, e, \Delta, A_{in}, R)$ be an $X(S)$-transducer and let $G(M) = (N', \Delta, A, R')$ be the associated X-grammar.

(i) The *set of sentential forms of* M, denoted by $\mathrm{SF}(M)$, is the set $\mathrm{SF}(G(M))$.

(ii) The *derivation relation of* M, $\Rightarrow_M \subseteq \mathrm{SF}(M) \times \mathrm{SF}(M)$, is defined by $\Rightarrow_M = \Rightarrow_{G(M)}$.

(iii) The *translation computed by* M, denoted by $\tau(M)$, is the set $\{(u, v) \mid u \in I, v \in \Phi, \text{ and } A_{in}\langle e(u) \rangle \Rightarrow_M^* v\}$ where Φ is, depending on X, the set of strings or trees over Δ, and for every $c \in C$, $A_{in}\langle c \rangle = A_{in}[A \leftarrow A\langle c \rangle; A \in N]$.

(iv) M is **total**, if $\mathrm{dom}(\tau(M)) = \mathrm{dom}(e)$. □

Two $X(S)$-transducers are called equivalent if they compute the same translation. The class of translations computed by (total deterministic) $X(S)$-transducers is denoted by $X(S)$ $(D_t X(S))$. If the involved transducers have only one nonter-

minal, then the class of computed translations is indexed by "one", e.g., $D_tX_{one}(S)$ denotes the class of translations computed by total deterministic $X(S)$-transducers that have one nonterminal.

By considering the trivial storage type S_0, which has only one configuration, no predicates, and the identity as instruction, an $X(S_0)$-transducer can be regarded again as a generating device, and in fact as an X-grammar.

3.6. Definition. The *trivial storage type* S_0 is the tuple $(\{c\}, \varnothing, \{\text{id}\}, m, \{c\}, \{m(\text{id})\})$, where $m(\text{id})(c)=c$. □

3.7. Lemma. For $X\in\{\text{REG, CF, RT, CFT}\}$, range $(X(S_0))=X$.

Proof. Lemma 3.9 of [EV]. □

3.2 Simulation of Storage Types

The second main tool for proving the desired characterization of n-level tree transducers is the simulation of storage types, which is formalized as a relation $\leqq$. We recall the formal definition of S-flowcharts (for predicates and instructions) and of $\leqq$ from [EV].

3.8. Definition. S_{id} denotes the storage type $(C, P, F\cup\{\text{id}\}, m', I, E)$, where $\text{id}\notin P\cup F$, m' restricted to $P\cup F$ is equal to m, and $m'(\text{id})$ is the identity on C. □

3.9. Definition. An *S-flowchart* is a deterministic $\text{REG}(S_{\text{id}})$-transducer such that all its rules have the form $A\to\textbf{if } b \textbf{ then } B\langle\phi\rangle$, where A and B are nonterminals, $b\in\text{BE}(P)$, and $\phi\in F\cup\{\text{id}\}$. □

Sometimes it is convenient to apply two instructions ϕ_1 and ϕ_2 in one rule. Hence, we allow a rule of the form $A\to\textbf{if } b \textbf{ then } B\langle\phi_1;\phi_2\rangle$, which denotes the rules $A\to\textbf{if } b \textbf{ then } [B;\phi_2]\langle\phi_1\rangle$ and $[B;\phi_2]\to B\langle\phi_2\rangle$, with a new nonterminal $[B;\phi_2]$.

Since the encoding and the terminal alphabet of an S-flowchart ω are not relevant for our purposes, we denote the corresponding components in ω by $-$. The class of S-flowcharts is denoted by FC(S). We need two different variations of S-flowcharts: one for the simulation of predicates and another for the simulation of instructions. To describe the flowcharts for instructions we define the notion of a path containing an instruction.

3.10. Definition. Let $\omega=(N,-,-,A_{\text{in}},R)\in\text{FC}(S)$ and let $n\geqq1$. If, for every $i\in[n]$, $A_{i-1}\to\textbf{if } b_i \textbf{ then } A_i\langle\phi_i\rangle$ is in R, then $\pi=\langle A_0, b_1, A_1, \phi_1, \ldots, b_i, A_i, \phi_i, \ldots, b_n, A_n, \phi_n\rangle$ is a *path of* ω *from* A_0 *to* A_n. π *contains an instruction* if there is an $i\in[n]$ such that $\phi_i\in F$. □

The set of paths of ω *from* A *to* B is denoted by PATH(ω, A, B), where $A, B\in N$.

3.11. Definition. Let $\omega=(N,-,-,A_{\text{in}},R)$ be an S-flowchart.

(i) ω is an *S-flowchart for predicates* if $\{\text{true, false}\}\subseteq N$ and the left-hand side of every rule is different from true and false.

(ii) ω is an *S-flowchart for instructions* if stop$\in N$, the left-hand side of every rule is different from stop, and every path in PATH$(\omega, A_{\text{in}}, \text{stop})$ contains an instruction. □

The classes of S-flowcharts for predicates and for instructions are denoted by P-FC(S) and F-FC(S), respectively.

3.12. Definition. Let $\omega\in$P-FC$(S)\cup$F-FC(S) with initial nonterminal A_{in}.

(i) ω induces an *operation* on C, denoted by oper(ω), which is the relation $\{(c_1, c_2)\in C\times C \mid A_{\text{in}}\langle c_1\rangle \Rightarrow(\omega)\Rightarrow^* x\langle c_2\rangle$ with $x\in D\}$ where $D=\{\text{true}, \text{false}\}$ if $\omega\in$P-FC(S), and $D=\{\text{stop}\}$ otherwise.

(ii) If $\omega\in$P-FC(S), then the *predicate* induced by ω, denoted by pred(ω), is the relation $\{(c_1, x)\in C\times\{\text{true}, \text{false}\} \mid A_{\text{in}}\langle c_1\rangle\Rightarrow(\omega)\Rightarrow^* x\langle c_2\rangle$ for some $c_2\in C\}$. □

Note that oper(ω) and pred(ω) are partial functions. Now we recall the definition of the simulation relation from [EV]. It is based on the direct simulation relation (Definition 4.6 of [EV]). If not specified otherwise, then, for $i\in\{1, 2\}$, S_i is the storage type $(C_i, P_i, F_i, m_i, I_i, E_i)$.

3.13. Definition. Let S_1 and S_2 be two storage types.

S_1 is *directly simulated by* S_2, for short $S_1\leqq_d S_2$, if $I_1\subseteq I_2$ and there is a partial function $h: C_2\to C_1$ called the *representation function* such that

1. for every $e_1\in E_1$ there is an $e_2\in E_2$ such that
   1.1.1. dom(e_1)=dom(e_2)
   1.1.2. $e_2(I_2)\subseteq$dom(h)
   1.2. for every $u\in$dom(e_2), $h(e_2(u))=e_1(u)$,
2. for every $p\in P_1$ there is an $\omega\in$P-FC(S_2) such that
   2.1.1. for every $c_2\in$dom(h): oper$(\omega)(c_2)$ is defined
   2.1.2. oper(ω)(dom(h))$\subseteq$dom(h)
   2.2. for every $c_2\in$dom(h): $h(\text{oper}(\omega)(c_2))=h(c_2)$ and pred$(\omega)(c_2)=m_1(p)(h(c_2))$,
3. for every $f\in F_1$ there is an $\omega\in$F-FC(S_2) such that
   3.1.1. for every $c_2\in$dom(h): $m_1(f)(h(c_2))$ is defined iff oper$(\omega)(c_2)$ is defined
   3.1.2. oper(ω)(dom(h))$\subseteq$dom(h)
   3.2. for every $c_2\in$dom(h) such that $m_1(f)(h(c_2))$ is defined, $h(\text{oper}(\omega)(c_2))=m_1(f)(h(c_2))$. □

If $h(c_2)=c_1$ for $c_1\in C_1$ and $c_2\in C_2$, then we say that "c_1 is represented by c_2" or "c_2 represents c_1", and if an encoding e_1 of E_1 and an encoding e_2 of E_2 satisfy requirement 1 of the previous definition, then we say that "e_1 is simulated by e_2". Similar shorthands are used in the case that requirement 2 and 3 hold.

Note that even for a flowchart ω for predicates we have defined an operation oper(ω), because sometimes the representing configurations have to be transformed before a simulating test can be applied. However, in many simulation proofs of this paper, the flowchart ω, which simulates a predicate p, contains only one rule of the form $A_{\text{in}}\to$ **if** b **then** true$\langle\text{id}\rangle$ **else** false$\langle\text{id}\rangle$, where b is

a boolean expression. Then we only construct this b and say that "p is simulated by b".

Very often it is essential that the simulated storage type uses only a finite number of predicates and instructions, and one encoding. This property is captured in the notion of finite restriction.

3.14. Definition. A *finite restriction of* S is a storage type $U=(C, P_f, F_f, m_f, I, \{e\})$, where P_f and F_f are finite subsets of P and F, respectively, m_f is m restricted to $P_f \cup F_f$, and $e \in E$. □

3.15. Definition. (i) S_1 is *simulated by* S_2, denoted by $S_1 \leqq S_2$, if for every finite restriction U of S_1, $U \leqq_d S_2$.

(ii) If $S_1 \leqq S_2$ and $S_2 \leqq S_1$, then S_1 and S_2 are *equivalent*, denoted by $S_1 \equiv S_2$. □

The relation $\leqq$ is reflexive (Theorem 4.10 of [EV]) and transitive (Theorem 4.20 of [EV]). Hence $\equiv$ is an equivalence relation on the class of storage types.

The following theorem formalizes the intuitively clear consequence of a particular storage type simulation for the classes of transducers, which work on these storage types.

3.16. Theorem. *"Justification Theorem" (Theorem* 4.18 *of* [EV]): *For* $X \in \{\text{REG}, \text{CF}, \text{RT}\}$, *if* $S_1 \leqq S_2$, *then* $X(S_1) \subseteq X(S_2)$ *and* $D_t X(S_1) \subseteq D_t X(S_2)$. □

3.3 Coding of Storage Types

The coding of storage types is a special case of simulation in the sense that the form of the flowcharts for the simulation of predicates and instructions is very restricted. Actually, the control of these flowcharts is superfluous, because they only specify one boolean expression and one instruction, respectively.

3.17. Definition. S_1 is *directly coded* by S_2, denoted by $S_1 \leqq_{dc} S_2$, if $S_1 \leqq_d S_2$ and

- in requirement 2 of Definition 3.13, the flowchart ω contains only one rule, of the form $A_{\text{in}} \rightarrow$ **if** b **then** true $\langle \text{id} \rangle$ **else** false $\langle \text{id} \rangle$ for some $b \in \text{BE}(P_2)$, and
- in requirement 3 of Definition 3.13, the flowchart ω contains only one rule, of the form $A_{\text{in}} \rightarrow \text{stop}\langle g \rangle$ for some $g \in F_2$. □

If we prove $S_1 \leqq_{dc} S_2$ for two storage types, then we will only specify the boolean expression b and the instruction g, which determine the flowcharts for the simulation of a predicate p and an instruction f, respectively. In this situation we say that "p is coded by b" and "f is coded by g".

Coding of storage types is obtained from direct coding in the same way as simulation is obtained from direct simulation.

3.18. Definition. (i) S_1 is *coded* by S_2, denoted by $S_1 \leqq_c S_2$, if for every finite restriction U of S_1, $U \leqq_{dc} S_2$.

(ii) If $S_1 \leqq_c S_2$ and $S_2 \leqq_c S_1$, then S_1 and S_2 are *coding equivalent*, denoted by $S_1 \equiv_c S_2$. □

Obviously, $\leqq_c$ is reflexive. Since boolean expressions are closed under the construction of boolean expressions, i.e., $\mathrm{BE}(\mathrm{BE}(P)) = \mathrm{BE}(P)$, it is also easy to see that $\leqq_c$ is transitive.

The coding relation is stronger than the simulation relation in the sense that if $S_1 \leqq_c S_2$, then $S_1 \leqq S_2$. The fact that, in the coding of storage types, a predicate and an instruction are coded by one boolean expression and by one instruction, respectively, has the consequence that in the justification theorem the property of a transducer having only one nonterminal is preserved.

3.19. Theorem. *If* $S_1 \leqq_c S_2$, *then* $\mathrm{RT}_{\mathrm{one}}(S_1) \subseteq \mathrm{RT}_{\mathrm{one}}(S_2)$ *and* $D_t\mathrm{RT}_{\mathrm{one}}(S_1) \subseteq D_t\mathrm{RT}_{\mathrm{one}}(S_2)$.

Proof. The construction is straightforward. In an $\mathrm{RT}(S_1)$-transducer with one nonterminal every predicate and instruction of S_1 has to be replaced by the corresponding S_2-predicate and S_2-instruction, respectively, provided by the coding $S_1 \leqq_c S_2$. □

3.4 Pushdown Operators on Storage Types

Now we recall from [EV] the formal definition of the concept of the pushdown operator on storage types [Gre, Eng5, Eng6, EV]. Given a storage type S, the "pushdown of S" is again a storage type of which the configurations are pushdowns of a special form: every square contains besides a usual pushdown symbol also a configuration of S. We also recall from [EV] the definition of a variation of P, viz. the bounded excursion pushdown operator P_{bex}. We prove that P_{bex} is monotonic with respect to the simulation relation (cf. Lemma 3.23), and hence, P_{bex} preserves the equivalence of storage types. This property is needed (in Sect. 6) in the inductive proof of the equivalence of the storage types "n-level applicative term of S" and "n-iterated bounded excursion pushdown of S" (cf. the discussion in the Introduction).

3.20. Definition. Let S be the storage type (C, P, F, m, I, E). The *pushdown of* S, denoted by $P(S)$, is the storage type (C', P', F', m', I', E'), where
$C' = (\Gamma \times C)^+$ and Γ is a fixed infinite set of pushdown symbols (intuitively, the top of the pushdown is at the left),

$P' = \{\mathrm{top} = \gamma \mid \gamma \in \Gamma\} \cup \{\mathrm{test}(p) \mid p \in P\}$,
$F' = \{\mathrm{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{pop}\}$
$\quad \cup \{\mathrm{stay}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{stay}(\gamma) \mid \gamma \in \Gamma\} \cup \{\mathrm{id}\}$,

for every $c' = (\delta, c)\beta$ with $\delta \in \Gamma$, $c \in C$, and $\beta \in C' \cup \{\lambda\}$
$m'(\mathrm{top} = \gamma)(c') \quad = \text{true iff } \delta = \gamma$
$m'(\mathrm{test}(p))(c') \quad = m(p)(c)$
$m'(\mathrm{push}(\gamma, f))(c') = (\gamma, c_1)(\delta, c)\beta$, if $m(f)$ is defined on c and $c_1 = m(f)(c)$, and undefined otherwise
$m'(\mathrm{pop})(c') \quad = \beta$ if $\beta \neq \lambda$ and undefined otherwise
$m'(\mathrm{stay}(\gamma, f))(c') = (\gamma, c_1)\beta$ if $m(f)$ is defined on c and $c_1 = m(f)(c)$, and undefined otherwise

$m'(\text{stay}(\gamma))(c') = (\gamma, c)\beta$
$m'(\text{id})(c') \quad = (\delta, c)\beta,$
$I' = I$, and
$E' = \{\lambda u \in I \cdot (\gamma_0, e(u)) \mid \gamma_0 \in \Gamma, e \in E\}$. □

The mapping test: $P \to \{\text{test}(p) \mid p \in P\}$ is uniquely extended to BE(P) such that it is a boolean homomorphism.

Naturally, the pushdown operator can be iterated: $P^0(S) = S$ and for every $n \geqq 0$, $P^{n+1}(S) = P(P^n(S))$. We denote $P(S_0)$ by P. For more remarks, in particular for a broad discussion of the storage types $P(\text{TR})$ and $P^2(\text{TR})$, we refer the reader to [EV]. Here we only mention that the operator P is monotonic with respect to the simulation relation $\leqq$ (cf. Theorem 4.22 of [EV]), i.e., for two storage types S_1 and S_2, if $S_1 \leqq S_2$, then $P(S_1) \leqq P(S_2)$.

As mentioned in the introduction, we have to use a modification of the pushdown operator in order to give an inductive proof of the main characterization result. The appropriate modification is the operator P_{bex} on storage types, where "bex" stands for bounded excursion: for every storage type of the form $P_{\text{bex}}(S)$ with an arbitrary S, the number of excursions that can be initiated from each square of a configuration of $P_{\text{bex}}(S)$ has to be bounded. This property of bounded excursion was introduced by van Leeuwen [vLe] in order to show that his preset pushdown automata, when restricted to be bounded excursion, accept all EOL languages. We recall the formal definition of P_{bex} from Sect. 5.2 of [EV].

3.21. Definition. The *bounded-excursion pushdown of S*, denoted by $P_{\text{bex}}(S)$, is the storage type (C', P', F', m', I', E') where

$C' = (\Gamma \times C \times \text{nat} \times \text{nat})^+,$
$P' = \{\text{top} = \gamma \mid \gamma \in \Gamma\} \cup \{\text{test}(p) \mid p \in P\},$
$F' = \{\text{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\text{pop}\}$
$\quad \cup \{\text{stay}(\gamma) \mid \gamma \in \Gamma\} \cup \{\text{stay}\},$

and for every $c' = (\delta, c, i, k)\beta$ with $\delta \in \Gamma$, $c \in C$, $i, k \geqq 0$, and $\beta \in C' \cup \{\lambda\}$,

$m'(\text{top} = \gamma)(c') \quad = (\delta = \gamma),$
$m'(\text{test}(p))(c') \quad = m(p)(c),$
$m'(\text{push}(\gamma, f))(c') = (\gamma, m(f)(c), 0, k)(\delta, c, i+1, k)\beta$ if $m(f)$ is defined on c and if $i+1 \leqq k$, and undefined otherwise,
$m'(\text{pop})(c') \quad = \beta$, if $\beta \neq \lambda$, and undefined otherwise,
$m'(\text{stay}(\gamma))(c') \quad = (\gamma, c, i+1, k)\beta$, if $i+1 \leqq k$, and undefined otherwise,
$m'(\text{stay})(c') \quad = (\delta, c, i+1, k)\beta$, if $i+1 \leqq k$, and undefined otherwise,
$I' = I,$
$E' = \{\lambda u \in I \cdot (\gamma_0, e(u), 0, mx) \mid \gamma_0 \in \Gamma, e \in E, mx \geqq 0\}$. □

We call the third component and the fourth component of every pushdown square s the *excursion counter* and the *excursion bound* of s, respectively. An

excursion which only consists of a stay or a stay(γ) instruction, is called a trivial excursion. Note that the excursion bound is not changed by the application of instructions. Hence, for an $X(P_{bex}(S))$-transducer M, the excursion bound for every pushdown square, which will occur during a computation, is fixed by the encoding of M.

Intuitively, it is clear that $P_{bex}(S)$ is a weaker storage type than $P(S)$ in the sense that $P_{bex}(S)$ is simulated by $P(S)$. The idea of the simulation is that for every square s of a $P_{bex}(S)$-configuration, the value of the excursion counter of s is stored in the pushdown symbol of s and updated appropriately. We leave the formal construction to the reader.

3.22. Lemma. $P_{bex}(S) \leqq P(S)$.

In the rest of this section we prove the *monotonicity of the operator* P_{bex} with respect to the simulation relation, i.e., for two storage types S_1 and S_2, if $S_1 \leqq S_2$, then $P_{bex}(S_1) \leqq P_{bex}(S_2)$.

Although the operators P and P_{bex} seem to be closely related, the proof of the monotonicity of P_{bex} is much more involved than the proof of the same property of P (cf. Theorem 4.22 of [EV]). The essential problem arises in the simulation of the predicate symbols. To make this difficulty clear, we first provide some notations. Let U' be a finite restriction of $P_{bex}(S_1)$. Clearly, U' induces a finite restriction U on S_1, and since $S_1 \leqq S_2$, U is directly simulated by S_2. Assume that h is the involved representation function. In particular, U' contains only a finite number of predicates test$(p_1), \ldots,$ test(p_r), where $p_1, \ldots, p_r$ are the predicates of U. Then, for every such predicate p_j, there is an S_2-flowchart $\omega(p_j)$ which simulates p_j.

In the proof of the monotonicity of P, every predicate test(p_j) is simulated by a $P(S_2)$-flowchart, which is obtained from the S_2-flowchart $\omega(p_j)$ by replacing every S_2-instruction f by stay(γ, f) with an appropriate γ. A similar straightforward technique cannot be used here, because instructions that are performed on one particular square increment its excursion counter, and hence, in general, the counter cannot be bounded during the execution of $\omega(p_j)$. A solution for this problem is to replace every S_2-instruction f of $\omega(p_j)$ by push$(\#, f)$ with a dummy symbol $\#$, and to pop the sequence of squares with $\#$ from the pushdown, as soon as the execution of this modification $\omega(p_j)'$ of $\omega(p_j)$ is finished, i.e., a truth value is computed. Clearly, the excursion counter of every "dummy square" is bounded by 1. However, after the execution of the $P_{bex}(S_2)$-flowchart $\omega(p_j)'$, also the excursion counter of the topmost pushdown square was incremented (by the first push instruction). Now assume that another predicate of the form test(p) has to be simulated starting with the result of the previous simulation. Then, after simulation, the excursion counter of the topmost square was again incremented. Since $P_{bex}(S_2)$ must be able to simulate an arbitrary number of successive predicates, again the excursion counter cannot be bounded. Roughly speaking, we solve this problem by executing for every pushdown square the flowcharts $\omega(p_1)', \ldots, \omega(p_r)'$ only once and by storing the results into the corresponding pushdown symbol. This causes only a finite number of additional excursions, proportional to r. Then, a predicate is simulated just by looking at the information stored in the pushdown symbol.

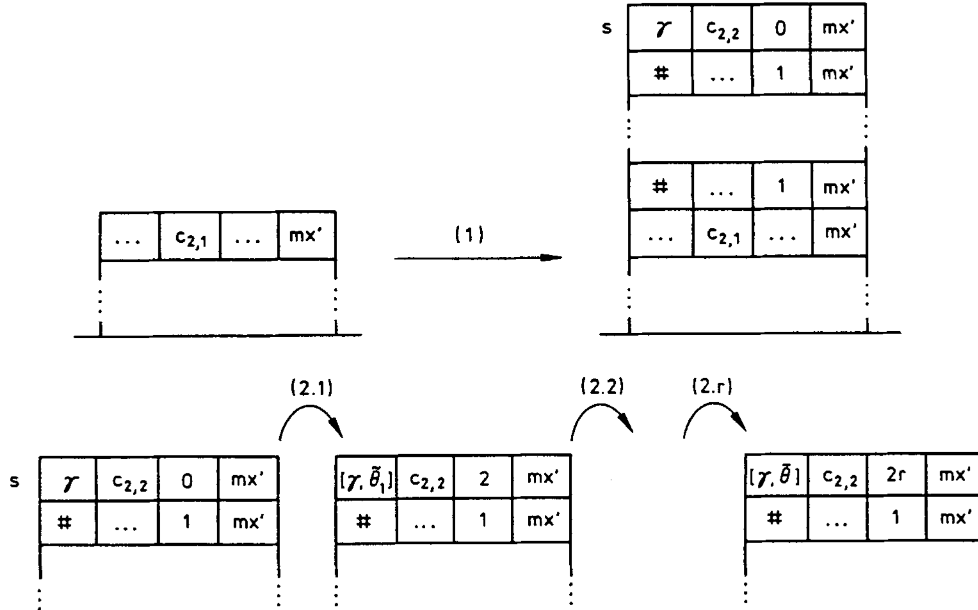

Fig. 5. Simulation of push(γ, f) by a $P_{\mathrm{bex}}(S_2)$-flowchart; (1) execution of $\omega(f)'$, $(2\cdot j)$ execution of $\omega(p_j)'$ for $j\in[r]$; $\tilde{\theta}_1=(\mathrm{pred}(\omega(p_1))(c_{2,2}), \mathrm{true}, \ldots, \mathrm{true})$ and $\tilde{\theta}=(\mathrm{pred}(\omega(p_1))(c_{2,2}), \ldots, \mathrm{pred}(\omega(p_r))(c_{2,2}))$

We want to make this more precise. Let $c'_{1,1}$ be a configuration of U' and let $(\delta, c_{1,1}, v, mx)$ be an arbitrary pushdown square of $c'_{1,1}$ (δ: pushdown symbol, $c_{1,1}$: S_1-configuration, v: excursion counter, mx: excursion bound). Let $c_{2,1}$ be a representation of $c_{1,1}$, i.e., $h(c_{2,1})=c_{1,1}$. Then, in every representation $c'_{2,1}$ of $c'_{1,1}$, we add the sequence $\tilde{\theta}$ of truth values to δ, where $\tilde{\theta} =(\mathrm{pred}(\omega(p_1))(c_{2,1}), \ldots, \mathrm{pred}(\omega(p_r))(c_{2,1}))$. Note that, since p_j is simulated by $\omega(p_j)$, it follows from requirement 2.2 of Definition 3.13 that $\mathrm{pred}(\omega(p_j))(c_{2,1}) =m(p_j)(c_{1,1})$, where m is the meaning function of U.

But how are the sequences $\tilde{\theta}$ installed in the pushdown squares? Immediately after the application of a push instruction the sequence of truth values is computed, that corresponds to the S_2-configuration of the new topmost square. So, let us have a closer look at the simulation of a push instruction. For this purpose we let $c'_{1,1}=(\delta, c_{1,1}, v, mx)\beta$ and assume that the application of the instruction push(γ, f) to $c'_{1,1}$ yields the configuration $c'_{1,2}=(\gamma, c_{1,2}, 0, mx)(\delta, c_{1,1}, v+1, mx)\beta$. Since $U\leq_d S_2$, there is an S_2-flowchart $\omega(f)$ which simulates f. The simulation of push(γ, f) is divided into two steps (cf. Fig. 5). First, the new S_2-configuration $c_{2,2}$, which represents $c_{1,2}$, is computed via a modification $\omega(f)'$ of $\omega(f)$. In $\omega(f)'$ every S_2-instruction g is replaced by push$(\#, g)$. Thus a whole sequence of pushdown squares of the form $(\#, c, 1, mx')$ is produced, where the value of the excursion counter of the topmost one is 0, until finally $c_{2,2}=\mathrm{oper}(\omega(f))(c_{2,1})$ is computed (mx' is the new excursion bound, whose value is discussed in a few seconds). Let us denote the topmost square of the

produced $P_{\mathrm{bex}}(S_2)$-configuration by s. (Note that it follows immediately from this way of calculating $\omega(f)$ that the instruction pop of U' can be simulated by a sequence of pops until a symbol is found, which is not equal to #.)

In the second step of the simulation of push(γ, f), the truth values $\mathrm{pred}(\omega(p_1))(c_{2,2}), \ldots, \mathrm{pred}(\omega(p_r))(c_{2,2})$ have to be computed and added to the pushdown symbol γ in square s. For this purpose, the flowcharts $\omega(p_1)', \ldots, \omega(p_r)'$ are executed one after the other, where $\omega(p_j)'$ is obtained from $\omega(p_j)$ as explained above for $\omega(f)$. First, $\omega(p_1)'$ is started. If, for some $j \in [r]$, the execution of $\omega(p_j)'$ is finished and a nonterminal $B \in \{\text{true, false}\}$ is reached, then the information B is reported down to s by popping the #-rubbish and by entering B into the j-th component of the sequence of truth values, which is being prepared in the square s. Note that the computation of the truth values adds $2r$ more excursions starting from s (a push and a stay for each p_j). Actually, the new excursion bound mx' is $mx+2r+1$, where the addition of 1 has only a technical reason. In this way the symbol of every pushdown square (except in the rubbish part) contains the appropriate sequence of truth values.

However, this does not hold "in the beginning": if c_2' is a representation of a result of the encoding of U', then we cannot assume that the truth values of the $\omega(p_j)'s$ are already present. But, of course, we must be able to simulate predicates like test(p_j) on c_2'. How do we solve this problem? We split the simulation of test(p_j) into two cases. If the sequence $\tilde{\theta}$ of truth values is already present in c_2', then test (p_j) is simulated just by looking at the j-th component of $\tilde{\theta}$. If it is not yet there, then test(p_j) is computed in the same way as it is computed when a new square is created by a push instruction (see above). Hence, the value of the excursion counter of the bottom square must also be allowed to exceed mx by $2r+1$. But this possibility of making $2r$ extra excursions from the bottom square can be misused: the simulation of an instruction ϕ of U' may be defined on c_2' (in which $\tilde{\theta}$ is not yet computed), although ϕ is not defined on the $P_{\mathrm{bex}}(S_1)$-configuration, which is represented by c_2'; e.g., if $mx=0$, then U' should not apply any push, stay(γ), or stay to any result of the encoding of U'. In order to avoid this misuse, we also have to split the simulation of the instructions push(γ, f), stay(γ), and stay into two cases (similar to the two cases of the simulation of test(p_j)). This has the effect that the $2r+1$ extra excursions are always consumed before the "real" simulation starts.

3.23. Lemma. *For $i \in \{1, 2\}$, let $S_i = (C_i, P_i, F_i, m_i, I_i, E_i)$ be two storage types. If $S_1 \leqq S_2$, then $P_{\mathrm{bex}}(S_1) \leqq P_{\mathrm{bex}}(S_2)$.*

Proof. Assume that $S_1 \leqq S_2$. Let U' be a finite restriction of $P_{\mathrm{bex}}(S_1)$ with encoding $e_1' = \lambda u \in I_1 \cdot (\gamma_0, e_1(u), 0, mx)$ for some encoding $e_1 \in E_1$ and some $mx \geqq 0$. Let $\{p_1, \ldots, p_r\}$ and $\Gamma_f = \{\gamma_1, \ldots, \gamma_\kappa\}$ with $r, \kappa \geqq 1$ be the finite sets of predicates of S_1 and of pushdown symbols, respectively, occurring in U' (note that an element p_j of the first set occurs in U' in the form test(p_j)). U' induces a finite restriction U on S_1, and since $S_1 \leqq S_2$, $U \leqq_d S_2$. Let $h: C_2 \to C_1$ be the involved representation function and let m_1 be the meaning function of U. We show that $U' \leqq_d P_{\mathrm{bex}}(S_2)$.

Define the set TEST of all sequences $\tilde{\theta}$ of truth values with length r, i.e., $\mathrm{TEST} = \{\text{true, false}\}^r$. We can assume without loss of generality that

$\Gamma_f \times \text{TEST} \subseteq \Gamma$. We abbreviate the boolean expression $\text{top}=\gamma_1$ **or**...**or** $\text{top}=\gamma_\kappa$ by "initial".

Let $mx' = mx + (2r+1)$. Then define the representation function $h'\colon C_2' \to C_1'$, where C_i' is the set of configurations of $P_{\text{bex}}(S_i)$, as follows.

$\text{dom}(h') = \{(\gamma_0, c, 0, mx') \mid c \in \text{dom}(h)\}$
$\cup \{s_{n+1}\beta_n s_n \ldots \beta_1 s_1 \mid n \geq 0,$
for every $i \in [n+1]$, $s_i = ([\gamma_i, \tilde{\theta}_i], c_i, v_i, mx')$
with $\gamma_i \in \Gamma_f$, $\tilde{\theta}_i \in \text{TEST}$, $c_i \in \text{dom}(h)$,
and for every $j \in [r]$, $m_1(p_j)(h(c_i)) = (\tilde{\theta}_i)_j$ where $(\tilde{\theta}_i)_j$ is the j-th component of $\tilde{\theta}_i$,
and $2r+1 \leq v_i \leq mx'$,
and for every $i \in [n]$, $\beta_i \in (\{\#\} \times C_2 \times \{1\} \times \{mx'\})^*\}$.

Let $c' = s_{n+1}\beta_n s_n \ldots \beta_1 s_1 \in \text{dom}(h')$ be as above. Then $h'(c') = s'_{n+1} s'_n \ldots s'_1$, where for every $i \in [n+1]$, $s'_i = (\gamma_i, h(c_i), v_i - (2r+1), mx)$. Finally, for $c' = (\gamma_0, c, 0, mx') \in \text{dom}(h')$, $h'(c') = (\gamma_0, h(c), 0, mx)$.

We show that h' satisfies the requirements 1–3 of Definition 3.13. Before doing so, we introduce the $P_{\text{bex}}(S_2)$-flowchart CSEQ which *c*omputes the *seq*uence of truth values of $p_1, \ldots, p_r$ as described in the discussion preceding this lemma. CSEQ can be viewed as a kind of macro, which will be later embedded into other flowcharts.

For every $j \in [r]$, let the S_2-flowchart $\omega(p_j)$, which simulates p_j, be determined by $(N_j, -, -, A^j_{\text{in}}, R_j)$. Then $\text{CSEQ} = (\tilde{N}, -, -, q_1, \tilde{R})$, where

$$\tilde{N} = \{q_j \mid j \in [r]\} \cup \{[B, j, i] \mid B \in \bigcup\{N_j \mid j \in [r]\}, j \in [r], \text{ and } i \in \{0, 1\}\}$$
$$\cup \{[B, j] \mid B \in \bigcup\{N_j \mid j \in [r]\} \text{ and } j \in [r]\}$$
$$\cup \{\text{end}\}$$

and $\tilde{R}$ is determined by (i)–(iii).

(i) "simulating $\omega(p_j)$": For every $j \in [r]$

– if $A^j_{\text{in}} \to$ **if** b **then** $B\langle g\rangle$ in R_j, then $q_j \to$ **if** test(b) **then** ζ is in $\tilde{R}$, where $\zeta = [B, j, 1]\langle\text{push}(\#, g)\rangle$ if $g \neq \text{id}$, and $[B, j, 0]\langle\text{id}\rangle$ otherwise.
The third component in $[B, j, i]$ indicates whether an excursion is made during the simulation of $\omega(p_j)$ or not.

– if $A \to$ **if** b **then** $B\langle g\rangle$ in R_j, then for every $i \in \{0, 1\}$
$[A, j, i] \to$ **if** test(b) **then** ζ is in $\tilde{R}$,
where $\zeta = [B, j, 1]\langle\text{push}(\#, g)\rangle$ if $g \neq \text{id}$, and $[B, j, i]\langle\text{id}\rangle$ otherwise.

(ii) "reporting truth values downwards": For every $j \in [r]$ and $B \in \{\text{true}, \text{false}\}$,

$[B, j, 1] \to$ **if** top $= \#$ **then** $[B, j, 1]\langle\text{pop}\rangle$ **else** $[B, j]\langle\text{id}\rangle$ and
$[B, j, 0] \to [B, j]\langle\text{stay}\rangle$ are in $\tilde{R}$.

(iii) "entry of truth values": For every $B \in \{\text{true}, \text{false}\}$, j with $1 < j < r$, $\gamma \in \Gamma_f$, and $\tilde{\theta} \in \text{TEST}$,

$[B, 1] \to$ **if** top $= \gamma$ **then** $q_2\langle\text{stay}([\gamma, \tilde{\theta}_1])\rangle$,
where $\tilde{\theta}_1 = (B, \text{true}, \ldots, \text{true}) \in \text{TEST}$,
$[B, j] \to$ **if** top $= [\gamma, \tilde{\theta}]$ **then** $q_{j+1}\langle\text{stay}([\gamma, \tilde{\theta}(j \leftarrow B)])\rangle$, and
$[B, r] \to$ **if** top $= [\gamma, \tilde{\theta}]$ **then** end$\langle\text{stay}([\gamma, \tilde{\theta}(r \leftarrow B)])\rangle$

are in $\tilde{R}$, where $\tilde{\theta}(j \leftarrow B)$ is obtained from $\tilde{\theta}$ by replacing the j-th component by B. $\tilde{\theta}(r \leftarrow B)$ is obtained similarly.

This completes the definition of CSEQ. It is an easy observation that the result of the application of CSEQ to a configuration $(\gamma, c_{2,1}, v, mx')\beta$ of $P_{\text{bex}}(S_2)$ with $c_{2,1} \in \text{dom}(h)$ is $([\gamma, \tilde{\theta}], c_{2,1}, v+2r, mx')\beta$, where for every component $\tilde{\theta}_j$ of $\tilde{\theta}$, $\tilde{\theta}_j = \text{pred}(\omega(p_j))(c_{2,1}) = m_1(p_j)(h(c_{2,1}))$. By using this flowchart we now prove that h' satisfies the requirements 1–3.

Requirement 1. The encoding $e'_2 = \lambda u \in I_2 \cdot (\gamma_0, e_2(u), 0, mx')$ of $P_{\text{bex}}(S_2)$, where e_1 is simulated by e_2, satisfies the requirements.

Requirement 2. Every predicate ϕ of U' is simulated by a $P_{\text{bex}}(S_2)$-flowchart for predicates, which is determined by its set R of rules.

$\phi = (\text{top} = \gamma)$: $A_{\text{in}} \to$ **if** initial **then** $q_1\langle\text{id}\rangle$ **else** look$\langle\text{id}\rangle$ is in R, $\tilde{R} \subseteq R$, end $\to$ look$\langle\text{stay}\rangle$ is in R, and look $\to$ **if** b **then** true$\langle\text{id}\rangle$ **else** false$\langle\text{id}\rangle$ is in R, where the test b is the disjunction of all predicates top $= [\gamma, \tilde{\theta}]$ for $\tilde{\theta} \in$ TEST.

$\phi = \text{test}(p_j)$: $A_{\text{in}} \to$ **if** initial **then** $q_1\langle\text{id}\rangle$ **else** look$\langle\text{id}\rangle$ is in R, $\tilde{R} \subseteq R$, and the rules end $\to$ look$\langle\text{stay}\rangle$ and look $\to$ **if** b **then** true$\langle\text{id}\rangle$ **else** false$\langle\text{id}\rangle$ are in R, where b is the disjunction of all the predicates top $= [\gamma, \tilde{\theta}]$ with $\gamma \in \Gamma_f$ and $\tilde{\theta} \in$ TEST such that the j-th component of $\tilde{\theta}$ is "true".

Requirement 3. Every instruction ϕ of U' is simulated by a $P_{\text{bex}}(S_2)$-flowchart for instructions, which is determined by its set R of rules.

$\phi = \text{push}(\gamma, f)$: Let $\omega(f) = (N_f, -, -, A_{\text{in}}, R_f)$ be the S_2-flowchart which simulates f. We assume that $\text{CSEQ}_{\text{new}} = (\tilde{N}_{\text{new}}, -, -, q_{1,\text{new}}, \tilde{R}_{\text{new}})$ is a copy of CSEQ such that $\tilde{N}_{\text{new}}$ and $\tilde{N}$ are disjoint; end_{new} is the copy of the nonterminal "end" of CSEQ. Intuitively, CSEQ and CSEQ_{new} are used for the computation of the truth values for the topmost square before (if necessary) and after the application of ϕ, respectively.

- $A_{\text{in}} \to$ **if** initial **then** $q_1\langle\text{id}\rangle$ **else** sim$\langle\text{id}\rangle$ is in R, $\tilde{R} \subseteq R$, and end $\to$ sim$\langle\text{stay}\rangle$ is in R.
- If $A \to$ **if** b **then** $B\langle g\rangle$ in $\omega(f)$,

  Case 1. if $g \neq \text{id}$, and

  Case 1.1. if $B \neq \text{stop}$, then

  $\Psi \to$ **if** test(b) **then** $B\langle\text{push}(\#, g)\rangle$ is in R,

  where $\Psi = \text{sim}$ if $A = A_{\text{in}}$, and $\Psi = A$ otherwise.

  Case 1.2. if $B = \text{stop}$, then

  $\Psi \to$ **if** test(b) **then** $q_{1,\text{new}}\langle\text{push}(\gamma, g); \text{stay}\rangle$

  is in R with Ψ as in Case 1.1.

  Case 2. if $g = \text{id}$, and

  Case 2.1. if $B \neq \text{stop}$, then

  $\Psi \to$ **if** test(b) **then** $B\langle\text{id}\rangle$ is in R and Ψ as in Case 1.1.

  Case 2.2. if $B = \text{stop}$, then

  $A \to$ **if** test(b) **then** $q_{1,\text{new}}\langle\text{stay}(\gamma)\rangle$ is in R.

  (Note that $\omega(f)$ cannot only contain the rule $A_{\text{in}} \to$ **if** b **then** stop$\langle\text{id}\rangle$. This is forbidden by definition of flowcharts for instructions.)

– $\tilde{R}_{\text{new}} \subseteq R$
– $\text{end}_{\text{new}} \to \text{stop}\langle\text{id}\rangle$ is in R.

$\phi = \text{pop}$: $A_{\text{in}} \to A\langle\text{pop}\rangle$ and
$A \to$ **if** top $= \#$ **then** $A\langle\text{pop}\rangle$ **else** stop$\langle$id$\rangle$ are in R.

$\phi = \text{stay}(\gamma)$: $A_{\text{in}} \to$ **if** initial **then** $q_1\langle\text{id}\rangle$ **else** sim$\langle$id$\rangle$ is in R, $\tilde{R} \subseteq R$, and, for every $\delta \in \Gamma_f$ and $\tilde{\theta} \in \text{TEST}$, the rules end $\to$ sim$\langle$stay$\rangle$ and sim $\to$ **if** top $= [\delta, \tilde{\theta}]$ **then** stop$\langle$stay$([\gamma, \tilde{\theta}])\rangle$ are in R.

$\phi = \text{stay}$: $A_{\text{in}} \to$ **if** initial **then** $q_1\langle\text{id}\rangle$ **else** sim$\langle$id$\rangle$ is in R, $\tilde{R} \subseteq R$, and the rules end $\to$ sim$\langle$stay$\rangle$ and sim $\to$ stop$\langle$stay$\rangle$ are in R.

Actually, Case 2.2 in $\phi = \text{push}(\gamma, f)$ causes the presence of the 1 in $mx' = mx + (2r+1)$. In order to avoid the misuse of this extra excursion on those pushdowns, for which the test "initial" is true, the value of the excursion counter is incremented by 1 before the real simulation of an instruction or predicate ϕ is started (beginning with the nonterminal sim). This incrementation takes place in the "bridge rules" end $\to$ sim$\langle$stay$\rangle$ and end $\to$ look$\langle$stay$\rangle$, respectively. □

4. High Level S Transducers and Iterated Pushdown S Transducers

In this section the two main devices of the paper are defined: for every $n \geqq 0$ and every storage type S, we define the n-level S transducer and the n-iterated pushdown S transducer. For $S = \text{TR}$, the n-level tree transducer and the n-iterated pushdown tree transducer, respectively, are obtained. An n-level S transducer is an $X(S)$-transducer where X is the class of n-level tree grammars. These tree grammars were introduced in [Dam] to model ALGOL68-like programs with finite mode. We recall the definition of this grammar class and give three examples of high level S transducers to illustrate this concept. An n-iterated pushdown S transducer is an $X(P^n(S))$-transducer where X is the class of 0-level (i.e., regular) tree grammars and $P^n(S)$ is the n-iterated pushdown of S (cf. Definition 3.20).

We slightly modify the definition of n-level tree grammars given in [Dam] by allowing an initial applicative term rather than only an initial nonterminal. By the usual technique of adding an extra rule with the initial term as right-hand side, it can be shown that the definitions are equivalent.

For terminology concerning types and applicative terms see Sect. 2.2. Recall that, for convenience, there is only one basic type, i.e., $Q = \{q\}$.

To manage the substitution involved in high level tree grammars, we define a set of parameters Y in such a way that for every derived type in $D^*(Q)$ there are infinitely many parameters of this type. Then we can use such a formal parameter, say of type τ, to represent and substitute applicative terms of type τ.

4.1. Definition. (i) The $D^*(Q)$-set of *parameters*, denoted by Y, is the tuple $\langle Y, \text{type}\rangle$, where $Y = \{y_{i,\tau} \mid i \geqq 1 \text{ and } \tau \in D^*(Q)\}$ and type: $Y \to D^*(Q)$ is defined by type $(y_{i,\tau}) = \tau$.

(ii) For $n \geqq 0$ and $\alpha \in D^n(Q)^*$ of length $k \geqq 0$, the list of parameters $(y_{1,\alpha(1)}, \ldots, y_{k,\alpha(k)})$ is denoted by y_α.

(iii) For $n \geqq 0$, $\alpha \in D^n(Q)^*$ of length k, and applicative terms $\zeta_1, \ldots, \zeta_k$ of type $\alpha(1), \ldots, \alpha(k)$, respectively, $[y_\alpha \leftarrow (\zeta_1, \ldots, \zeta_k)]$ abbreviates $[y_{1,\alpha(1)} \leftarrow \zeta_1, \ldots, y_{k,\alpha(k)} \leftarrow \zeta_k]$, to be used in substitutions. □

Now we are ready to define the n-level tree grammars.

4.2. Definition. Let $n \geqq 0$. An *n-level tree grammar* G is a tuple $(N, \Delta, A_{\text{in}}, R)$, where

- N is a finite $D^*(Q)$-set of *nonterminals* such that for every $A \in N$, level $(A) \leqq n$
- Δ is a finite $D(Q)$-set (i.e., ranked alphabet) of *terminals* (with $N \cap \Delta = \emptyset$),
- $A_{\text{in}} \in \text{AT}(N)^q$ is called the *initial term*, and
- R is a finite set of *rules*, each of the form

  (*) $A\, y_{\alpha m} \ldots y_{\alpha 2}\, y_{\alpha 1} \to \zeta$,

  where $A \in N^\tau$ with $\tau = (\alpha m, \ldots (\alpha 2, (\alpha 1, q)) \ldots)$, $m \geqq 0$, and $\zeta \in \text{AT}(N \cup \Delta \cup Y)^q$. Since $y_{\alpha m} \ldots y_{\alpha 2}\, y_{\alpha 1}$ follows uniquely from type(A), we can abbreviate $A\, y_{\alpha m} \ldots y_{\alpha 2}\, y_{\alpha 1}$ by $A^\wedge$. Note that $A^\wedge$ and ζ are of level 0. □

The class of n-level tree grammars is denoted by n-T.

Next we define the derivation relation $=\!(G)\!\Rightarrow$ of an n-level tree grammar G for the outside-in (OI) mode (this was shown to be sufficient in Corollary 6.6 of [Dam]; see also [DamGue1]). $=\!(G)\!\Rightarrow$ is a binary relation on the set SF(G) of sentential forms of G, which is the set of applicative terms of type q built up out of nonterminals and terminals. A derivation step $\xi_1 =\!(G)\!\Rightarrow \xi_2$ according to a rule $A^\wedge \to \zeta$ consists of two phases. First, an outermost occurrence of A in ξ_1 is replaced by ζ, where outermost means: not occurring in a parameter position of another nonterminal. Second, the actual parameters are substituted for the parameters occurring in ζ.

4.3. Definition. Let $G = (N, \Delta, A_{\text{in}}, R)$ be an n-level tree grammar with $n \geqq 0$.

(i) The set of *sentential forms* of G, denoted by SF(G), is the set $\text{AT}(N \cup \Delta)^q$.

(ii) The *(OI-) derivation relation of* G, denoted by $=\!(G)\!\Rightarrow$, is a binary relation on SF(G) and is defined as follows.

For $\xi_1, \xi_2 \in \text{SF}(G)$, $\xi_1 =\!(G)\!\Rightarrow \xi_2$ iff
there is a rule $A^\wedge \to \zeta$ in R, and $A \in N^\tau$ with
$\tau = (\alpha m, \ldots (\alpha 2, (\alpha 1, q)) \ldots) \in D^m(Q)$ for some $m \geqq 0$,
there is a $\xi \in \text{AT}(N \cup \Delta \cup \{y_{1,q}\})^q$, where $y_{1,q}$ occurs exactly once in ξ, but not in a sub(-applicative)-term η of ξ with top$(\eta) \in N$, and
for every $i \in [m]$, there is a $\xi'_i \in \text{AT}(N \cup \Delta)^{\alpha i}$,
such that $\xi_1 = \xi[y_{1,q} \leftarrow A\, \xi'_m \ldots \xi'_2\, \xi'_1]$
and $\xi_2 = \xi[y_{1,q} \leftarrow \zeta[y_{\alpha i} \leftarrow \xi'_i;\, i \in [m]]]$.

(iii) The *language generated by* G, denoted by $L(G)$, is the set $\{t \in T_\Delta \mid A_{\text{in}} =\!(G)\!\Rightarrow^* t\}$. □

Note that $T_\Delta = \text{AT}(\Delta)^q$. The class of tree languages generated by n-level tree grammars is also denoted by n-T. Obviously, 0-T and 1-T are the classes of regular tree languages and context-free tree languages, respectively.

4.4. Proposition. RT $=0$-T *and* CFT $=1$-T.

We now define the n-level S transducer as an $X(S)$-transducer with $X=n$-T (cf. Definition 3.3).

4.5. Definition. For $n \geqq 0$, an *n-level S transducer* is an n-$T(S)$-transducer. □

We write "*n-level tree transducer*" rather than "n-level TR transducer". We could also have called them n-level top-down tree transducers, in the sense that every X(TR)-transducer works on its input-tree in a top-down fashion, by definition of the storage type TR.

Since every context-free tree grammar can be viewed as 1-level tree grammar and vice versa, it is immediately clear that, for every storage type S, CFT$(S)=1$-$T(S)$. Note that, in our definition, a context-free tree grammar can start from an initial tree consisting of nonterminals.

4.6. Fact. CFT$(S)=1$-$T(S)$ and RT$(S)=0$-$T(S)$. Totality, determinism, and the number of nonterminals are preserved.

Hence, in particular, 0-T(TR) and 1-T(TR) coincide with the class of tree translations induced by top-down tree transducers [Tha, Rou, Eng1] (Corollary 3.20 of [EV]) and by (OI-) macro tree transducers [Eng3, CouFra, EngVog1] (Theorem 3.19 of [EV]), respectively.

By consider the trivial storage type, n-$T(S_0)$-transducers turn back into generating devices.

4.7. Lemma. range $(n\text{-}T(S_0)) = n\text{-}T$.

The following three examples illustrate the concept of $X(S)$-transducer in the situation that X is the class of 2-level tree grammars. In particular, in the third example, part of the CHECK-example (cf. Introduction) is formulated in the notational framework of high level tree transducers.

4.8. Example. (i) The storage type count-down is the tuple (nat, {null}, {dec}, m, nat, $\{\lambda n \in \text{nat} \cdot n\}$) and for every $n \geqq 0$, $m(\text{null})(n) = \text{true}$ iff $n=0$, and $m(\text{dec})(n) = n-1$ if $n \geqq 1$.

Define the (nondeterministic) 2-level count-down transducer $M = (N, e, \Delta, A_{\text{in}}, R)$ as follows.

$N = \{A_{\text{in}}, A, B, +\}$ with $A_{\text{in}}: q$, $A: ((q, q), (q, q))$, $B: (q, q)$, $+: ((q, q)(q, q), (q, q))$,
$e = \lambda n \in \text{nat} \cdot n$,
$\Delta = \{\sigma, \delta, \$, a, b\}$ with $\sigma: (q^3, q)$, $\delta: (q^2, q)$, and $\$, a, b: (\lambda, q)$,

and R contains the following rules.

$$A_{\text{in}} \to A\langle\text{dec}\rangle(B\langle\text{dec}\rangle)(a()),$$

and the same rule for b instead of a,

$$A(y_{1,(q,q)})(y_{1,q}) \to \textbf{if not null then}$$
$$A\langle\text{dec}\rangle(+\langle\text{dec}\rangle(B\langle\text{dec}\rangle, y_{1,(q,q)}))(\delta(a(), y_{1,q})),$$

and the same rule for b instead of a,

$$A(y_{1,(q,q)})(y_{1,q}) \rightarrow \textbf{if } \text{null } \textbf{then } \delta(\$(), y_{1,(q,q)}(y_{1,q})),$$
$$+(y_{1,(q,q)}, y_{2,(q,q)})(y_{1,q}) \rightarrow \sigma(y_{1,(q,q)}(y_{1,q}), \$(), y_{2,(q,q)}(y_{1,q})), \text{ and}$$
$$B(y_{1,q}) \rightarrow y_{1,q}.$$

Let us now look at a computation of M which starts with $A_{\text{in}}\langle 3\rangle$. We abbreviate the tree $\delta(b(), \delta(b(), a()))$ by t. The derivation relation $=\!(M)\!\Rightarrow$ is abbreviated by $\Rightarrow$.

$$\begin{aligned}
A_{\text{in}}\langle 3\rangle &\Rightarrow A\langle 2\rangle(B\langle 2\rangle)(a())\\
&\Rightarrow A\langle 1\rangle(+\langle 1\rangle(B\langle 1\rangle, B\langle 2\rangle))(\delta(b(), a()))\\
&\Rightarrow A\langle 0\rangle(+\langle 0\rangle(B\langle 0\rangle, +\langle 1\rangle(B\langle 1\rangle, B\langle 2\rangle)))(t)\\
&\Rightarrow \delta(\$(), +\langle 0\rangle(B\langle 0\rangle, +\langle 1\rangle(B\langle 1\rangle, B\langle 2\rangle))(t))\\
&\Rightarrow \delta(\$(), \sigma(B\langle 0\rangle(t), \$(), +\langle 1\rangle(B\langle 1\rangle, B\langle 2\rangle)(t)))\\
&\Rightarrow^2 \delta(\$(), \sigma(t, \$(), \sigma(B\langle 1\rangle(t), \$(), B\langle 2\rangle(t))))\\
&\Rightarrow \delta(\$(), \sigma(t, \$(), \sigma(t, \$(), B\langle 2\rangle(t))))\\
&\Rightarrow \delta(\$(), \sigma(t, \$(), \sigma(t, \$(), t))).
\end{aligned}$$

It can be proved by induction that yield $(\tau(M)) = \{(n, v) \mid n \geqq 1 \text{ and } v = (\$w)^n \text{ with } w \in \{a, b\}^n\}$.

(ii) Define the deterministic 2-level tree transducer $M = (N, e, \Delta, A_{\text{in}}, R)$ as follows (cf. Lemma 7.1 of [Dam]).
$N = \{A_{\text{in}}, A\}$ with $A_{\text{in}}: q$ and $A: ((q, q), (q, q))$, e is the identity on T_Σ where $\Sigma = \{\sigma, \alpha\}$ and $\sigma \in \Sigma_1$, $\alpha \in \Sigma_0$, and
$\Delta = \{f, a\}$ with $f: (q, q)$ and $a: (\lambda, q)$.

Since we believe that the reader is now familiar with the use and the interpretation of subscripts of parameters, we abbreviate $y_{1,q}$ and $y_{1,(q,q)}$ by y and z, respectively. Then R contains the following rules.

$$\begin{aligned}
A_{\text{in}} &\rightarrow \textbf{if } \text{root} = \sigma \textbf{ then } A\langle \text{sel}_1\rangle(A\langle \text{sel}_1\rangle(f))(a()),\\
A_{\text{in}} &\rightarrow \textbf{if } \text{root} = \alpha \textbf{ then } f(f(a())),\\
A(z)(y) &\rightarrow \textbf{if } \text{root} = \sigma \textbf{ then } A\langle \text{sel}_1\rangle(A\langle \text{sel}_1\rangle(z))(y),\\
A(z)(y) &\rightarrow \textbf{if } \text{root} = \alpha \textbf{ then } z(z(y)).
\end{aligned}$$

If we interpret monadic trees as strings, then it can be proved by induction that $\tau(M) = \{(\sigma^n\alpha, f^m a) \mid n \geqq 0 \text{ and } m = 2^{2^n}\}$. M is total, because $\text{dom}(\tau(M)) = T_\Sigma = \text{dom}(e)$ (cf. Definition 3.5(iv)). Since, for every macro tree transducer (i.e., 1-T(TR) transducer), the height of an output tree is exponentially bounded in the height of the corresponding input tree (Theorem 3.24 of [EngVog1]), $\tau(M)$ is an example of a translation, which cannot be induced by macro tree transducers, i.e.,

$$\tau(M) \in 2\text{-}T(\text{TR}) \text{ but } \tau(M) \notin 1\text{-}T(\text{TR}).$$

(iii) At the end of the introduction, we have shown a system of recursive function procedures that was used to perform the type checking in a small block-structured programming language called CHECK. Here, we repeat two equations of this system in the notational framework of high-level tree transduc-

ers. According to our typing conventions, the type of the result of Check is changed into (IDENT → TYPE) → (λ → BOOL). The considered equations are the following:

$$\text{Envir}[D; \textbf{var } I\colon T]\; y_{\text{EN}}\, y_{\text{ID}} = \text{cond}(\text{equal}(\text{Id}[I], y_{\text{ID}}), \text{Ty}[T], \text{Envir}[D]\; y_{\text{EN}}\, y_{\text{ID}}), \text{ and}$$
$$\text{Check}[\textbf{begin } D; S \textbf{ end}]\; y_{\text{EN}} = \text{Check}[S](\text{Envir}[D]\; y_{\text{EN}}).$$

They turn into the following rules of a 2-level tree transducer:

$$\text{Envir}(y_{\text{EN}})(y_{\text{ID}}) \to \textbf{if } \text{root} = r_3 \textbf{ then } \text{cond}(\text{equal}(\text{Id}\langle \text{sel}_2\rangle, y_{\text{ID}}), \text{Ty}\langle \text{sel}_3\rangle, \text{Envir}\langle \text{sel}_1\rangle(y_{\text{EN}})(y_{\text{ID}})), \text{ and}$$
$$\text{Check}(y_{\text{EN}})() \to \textbf{if } \text{root} = r_5 \textbf{ then } \text{Check}\langle \text{sel}_2\rangle(\text{Envir}\langle \text{sel}_1\rangle(y_{\text{EN}}))(),$$

respectively. □

After having introduced high-level S transducers, we now define iterated pushdown S transducers in the terminology of grammars with storage.

4.9. Definition. For every $n \geqq 0$, an *n-iterated pushdown S transducer* is a 0-level $P^n(S)$ transducer. □

An n-iterated pushdown TR transducer is called an *n-iterated pushdown tree transducer*. For the sake of convenience, 0-level $P^n_{\text{bex}}(S)$ transducers are also called n-iterated pushdown S transducers. However, if the difference between the unbounded and the bounded variation becomes important, then the precise denotation will be used.

Now we can formalize the main aim of the present paper, which is the characterization of n-level tree transducers by means of n-iterated pushdown tree transducers: for every $n \geqq 0$,

$$n\text{-}T(\text{TR}) = 0\text{-}T(P^n_{\text{bex}}(\text{TR})) \quad \text{(Theorem 7.1) and}$$
$$D_t\, n\text{-}T(\text{TR}) = D_t\, n\text{-}T(P^n(\text{TR})) \quad \text{(Theorem 7.12).}$$

For $n = 1$, i.e., for macro tree transducers, both characterizations are already proved in [EV].

4.10. Theorem. $1\text{-}T(\text{TR}) = 0\text{-}T(P_{\text{bex}}(\text{TR}))$ *and* $D_t 1\text{-}T(\text{TR}) = D_t 0\text{-}T(P(\text{TR}))$.

Proof. In Theorem 8.3 and Theorem 8.2 of [EV] it is proved that $\text{CFT}(\text{TR}) = \text{RT}(P_{\text{bex}}(\text{TR}))$ and $D_t\text{CFT}(\text{TR}) = D_t\text{RT}(P(\text{TR}))$, respectively. Then, the present theorem follows from Fact 4.6 of this paper. □

5. Substitution of Applicative Terms

In Theorem 1 of [DamGue1] the equivalence of n-level tree grammars (called level-n grammars there) and level-n stack automata is proved. A level-n stack automaton M, introduced in [DamGue1], works on an input tree like a finite top-down tree automaton [Rou, Tha, Engl]. However, every state which occurs in a computation of M has an auxiliary storage, of which the configurations

are applicative terms. The kernel of the transitions of M is the substitution of applicative terms of different types at different levels. Since the control of M can be viewed as a regular tree grammar, one might say that the main feature of n-level tree grammars is the substitution of applicative terms. In this section we support this claim by verifying it for n-level S transducers (recall that range $(n\text{-}T(S_0)) = n\text{-}T$, Lemma 4.7). For this purpose, the storage of level-n stack automata is generalized to a storage type operator, called "n-level applicative term of S" (for short: n-AT(S)). Then the above mentioned claim is formally expressed in the equation $n\text{-}T(S) = 0\text{-}(n\text{-AT}(S))$ (cf. Theorem 5.13), which says that the substitution power inherent in n-level S transducers is entirely captured by the storage type n-AT(S). For $S = S_0$ this is the result of [DamGue1]; n-AT(S_0) was formalized as a storage type in [DamGue2], called n-TREEPD. We note that the level-n stack transducer of [DamGue1] differs from our n-level tree transducer: the second is an $n\text{-}T(\text{TR})$-transducer (equivalent, as will be shown, to the $0\text{-}T(n\text{-AT}(\text{TR}))$-transducer), and the first is equivalent to the $0\text{-}T(\text{TR} \times n\text{-AT}(S_0))$-transducer, where $\times$ is the obvious product operation on storage types (see [DamGue2], where arbitrary $0\text{-}T(\text{TR} \times S)$-transducers are studied).

As a reservoir of symbols of derived types, we fix for the rest of this paper the $D^*(Q)$-set $\langle \Xi, \text{type}_\Xi \rangle$ such that, for every $\tau \in D^*(Q)$, $\text{type}_\Xi^{-1}(\tau)$ is an infinite set. See Definition 4.1 for terminology on parameters.

5.1. Definition. Let $n \geq 1$. The *n-level applicative term of S*, denoted by n-AT(S), is the storage type (C', P', F', m', I', E'), where

$$C' = \text{AT}(\Xi^{\leq n}\langle C \rangle)^q \text{ and } \Xi^{\leq n} = \{\gamma \in \Xi \mid \text{level}(\gamma) \leq n\},$$
$$P' = \{\text{top} = \gamma \mid \gamma \in \Xi^{\leq n}\} \cup \{\text{test}(p) \mid p \in P\},$$
$$F' = \{\text{push}(\zeta) \mid \zeta \in \text{AT}(\Xi^{\leq n}\langle F \rangle \cup Y)^q\},$$

and for $c' = \delta\langle c \rangle \xi_k \ldots \xi_1 \in C'$ such that $\text{type}_\Xi(\delta) = (\alpha k, \ldots (\alpha 1, q) \ldots)$, and $\xi_i \in \text{AT}(\Xi^{\leq n}\langle C \rangle)^{\alpha i}$ for every $i \in [k]$,

$$m'(\text{top} = \gamma)(c') = \text{true iff } \delta = \gamma,$$
$$m'(\text{test}(p))(c') = m(p)(c),$$

for $\xi = \zeta[y_{\alpha i} \leftarrow \xi_i;\ i \in [k]][f \leftarrow m(f)(c); f \in F]$,

$$m'(\text{push}(\zeta))(c') = \xi, \text{ if } \xi \in C', \text{ and undefined otherwise,}$$

$$I' = I, \text{ and}$$
$$E' = \{\lambda u \in I \cdot \eta\langle e(u) \rangle \mid \eta \in \text{AT}(\Xi^{\leq n})^q \text{ and } e \in E\}$$
$$\text{with } \eta\langle e(u) \rangle = \eta[\gamma \leftarrow \gamma\langle e(u) \rangle;\ \gamma \in \Xi^{\leq n}]. \quad \square$$

The storage type n-AT(S_0) is denoted by n-AT. Note that the application of an instruction of the form push(ζ) to $\delta\langle c \rangle \xi_k \ldots \xi_1$ is closely related to the application of the rule $r: \delta^\wedge \to \zeta$ of an n-level tree grammar, where the symbols of Ξ are viewed as nonterminals (cf. Definition 4.3): the second phase of the derivation step via r, which is the substitution of applicative terms, is captured by the application of push(ζ).

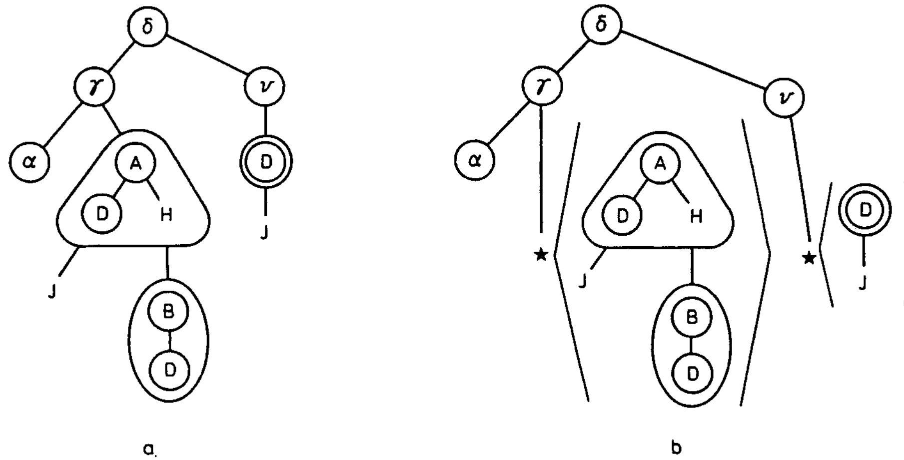

Fig. 6. One-to-one correspondence between sentential forms of (a) a terminal restricted n-level S transducer and (b) a 0-level n-AT(S) transducer with the only nonterminal $*$. The S-configurations are dropped for convenience

Now we formalize and prove the relationship between n-level S transducers and 0-level n-AT(S) transducers. However, before discussing the general case, we study a particular case, in which the relationship between the transducer classes is quite apparent: every sentential form of a transducer of one class can be viewed as a sentential form of a transducer of the other class and vice versa. The two involved transducer types are the following: "terminal restricted" n-level S transducers and 0-level n-AT(S) transducers with only one nonterminal. An n-level S transducer is terminal restricted, if in the right-hand side of every rule no terminal symbol occurs in the argument positions of a nonterminal or a parameter. In the rest of this section, for every $k \geq 0$, $X_k = \{x_1, \ldots, x_k\}$ is the Q-set of auxiliary substitution variables.

5.2. Definition. Let $n \geq 1$ and let $M = (N, e, \Delta, A_{\text{in}}, R)$ be an n-level S transducer. M is *terminal restricted* if for every rule in R, the right-hand side is of the form $t[x_i \leftarrow \zeta_i;\ i \in [k]]$, where $t \in \text{AT}(\Delta \cup X_k)^q$, $k \geq 0$, and for every $i \in [k]$, $\zeta_i \in \text{AT}(N\langle F\rangle \cup Y)^q$. □

It is an easy observation that every derivable sentential form ξ of a terminal restricted n-level S transducer has the form $t[x_i \leftarrow \xi_i;\ i \in [k]]$, where t is an applicative term over terminals and the substitution variables $x_1, \ldots, x_k$, and $\xi_1, \ldots, \xi_k$ are applicative terms over $N\langle C\rangle$, where N is the set of nonterminals of the transducer. Now the one-to-one correspondence between the sentential forms of the involved transducers becomes obvious (cf. Fig. 6; the nonterminals have the types A: $((q, q)(\lambda, q), (qq, q))$, B: $((q, q), (\lambda, q))$, D: $(\lambda, (q, q))$, H: (λ, q), and J: q; the ranks of the terminals are obvious). Roughly speaking, the applicative terms $\xi_1, \ldots, \xi_k$ of ξ are viewed as configurations of n-AT(S) and vice versa. Hence, the nonterminals of the terminal restricted n-level S transducer correspond to the Ξ-symbols of the 0-level n-AT(S) transducer. More precisely, $\xi = t[x_i \leftarrow \xi_i;\ i \in [k]]$ is a derivable sentential form of a terminal restricted n-level S transducer iff $\xi' = t[x_i \leftarrow *\langle \xi_i\rangle;\ i \in [k]]$ is a derivable sentential form of a 0-level n-AT(S) transducer with the only nonterminal $*$.

In the next definition the relation between two particular transducers (one of each of the discussed classes) is formalized in such a way that the described one-to-one correspondence holds. Then, the equivalence of such transducers is an immediate consequence (cf. Lemma 5.4).

5.3. Definition. Let $n \geqq 1$. Let $M_1 = (N_1, e_1, \Delta, A^1_{in}, R_1)$ be a terminal restricted n-level S transducer and let $M_2 = (\{*\}, e_2, \Delta, *, R_2)$ be a 0-level n-AT(S) transducer with one nonterminal $*$ (of type q).

M_1 and M_2 are *related* if

- $e_2 = \lambda u \in I \cdot A^1_{in}\langle e_1(u)\rangle$
- $R_2 = \{* \to \textbf{if } \mathrm{top} = A \textbf{ and } \mathrm{test}(b) \textbf{ then } t[x_i \leftarrow *\langle \mathrm{push}(\zeta_i)\rangle;\ i \in [k]] \mid A^\wedge \to \textbf{if } b \textbf{ then } t[x_i \leftarrow \zeta_i;\ i \in [k]]$ is in R_1 for some $t \in \mathrm{AT}(\Delta \cup X_k)^q$ and $\zeta_i \in \mathrm{AT}(N_1\langle F\rangle \cup Y)^q\}$. □

5.4. Lemma. *Let M_1 and M_2 be defined as in Definition 5.3. If M_1 and M_2 are related, then $\tau(M_1) = \tau(M_2)$.*

Now the general case is treated. First, we show that the terminal restriction of n-level S transducers does not decrease their transformational power (cf. Lemma 7.12 of [Dam]). In case S has an identity id, the proof is easy: just replace, in the right-hand side of a rule, each terminal δ by $\bar{\delta}\langle \mathrm{id}\rangle$, where $\bar{\delta}$ is a new nonterminal, and add the obvious rule for $\bar{\delta}$. Unfortunately, this does not work in general, because it is not clear which instruction to use in place of id. This forces us to work out a more complicated construction. To explain our construction, let M be an arbitrary n-level S transducer. For every nonterminal of M of level greater than 1, we prolongate the list of parameters of level 1 by the number of terminal symbols of M. The terminal symbols are kept (in the form of new nonterminals) in the extra parameter positions. Then, the non-desired terminals, which occur in the argument position of nonterminals and parameters in the right-hand side of a rule of M, are represented by the appropriate new parameters. To apply this trick also to a nonterminal $A: \tau$ of level 0 or of level 1, we first replace A everywhere by the applicative term $A'()()$ and $A'()$, respectively, where $A': (\lambda, (\lambda, \tau))$ and $A': (\lambda, \tau)$, respectively (note that the applicative terms have again type τ). Clearly, for this purpose we have to assume that $n \geqq 2$. But note that the characterization of 1-level tree transducers by 0-level (1-iterated) pushdown tree transducers is already given in Theorem 4.10. Hence, to obtain our main result also for $n = 1$, there is no need to prove 1-$T(S)$ = 0-T(1-AT(S)). However, 1-$T(S)$ = 0-T(1-AT(S)) will be proved explicitly in Sect. 6 (cf. Corollary 6.6).

5.5. Lemma. *Let $n \geqq 2$. For every n-level S transducer there is an equivalent terminal restricted n-level S transducer. Determinism and totality are preserved.*

Proof. Let $M_1 = (N_1, e, \Delta, A_{in,1}, R_1)$ be an n-level S transducer. We use $()^r$ with $r \geqq 0$, to abbreviate a sequence $()()\ldots()$ of r empty lists. First, we construct an n-level S transducer $M'_1 = (N'_1, e, \Delta, A'_{in,1}, R'_1)$, in which every nonterminal has at least level 2.

- $N'_1 = (N_1 - N_1^{\leqq 1}) \cup \{A' \mid A \in N_1^{\leqq 1}\}$ is the $D^*(Q)$-set such that, if $A: \tau$ with

level$(\tau)=i\leqq 1$, then A' is of level 2 with type (λ, τ) if $i=1$ and $(\lambda, (\lambda, \tau))$ if $i=0$ (note that there are $2-i$ λ's)

- $A'_{\text{in},1}=A_{\text{in},1}[A\leftarrow A'()^{r(A)};\ A\in N_1^{\leqq 1}]$ where $r(A)=2-\text{level}(A)$
- if $A^\wedge\to$ **if** b **then** ζ is in R_1, then $\Theta^\wedge\to$ **if** b **then** $\zeta[A\langle f\rangle\leftarrow A'\langle f\rangle()^{r(A)};$ $A\in N_1^{\leqq 1}, f\in F]$ is in R'_1, where $\Theta=A'$ if $A\in N_1^{\leqq 1}$, and $\Theta=A$ otherwise, and $r(A)$ as above.

It is obvious that $\tau(M_1)=\tau(M'_1)$ and that determinism and totality are preserved. Thus, we may now assume that every nonterminal of M_1 has at least level 2. Next we construct the terminal restricted n-level S transducer $M_2=(N_2, e, \Delta, A_{\text{in},2}, R_2)$, equivalent to M_1. Let $\Delta=\{\delta_1, \ldots, \delta_r\}$ for some $r\geqq 1$, and for every $i\in[r]$, δ_i: $\tau(i)$.

We define inductively the mapping ψ: $D^*(Q)\to D^*(Q)$ that inserts uniformly at subtypes of level 1 the sequence $\tau(1), \ldots, \tau(r)$.

(i) For every $\tau=(\alpha, \nu)\in D^\rho(Q)$ with $\rho>2$ and $\alpha=\alpha(1)\ldots\alpha(k)$ for some $k\geqq 0$, $\psi(\tau)=(\psi(\alpha(1))\ldots\psi(\alpha(k)), \psi(\nu))$.

(ii) For every $\tau=(\alpha, \nu)\in D^\rho(Q)$ with $\rho=2$, $\psi(\tau)=(\tau(1)\ldots\tau(r)\,\alpha, \nu)$.

(iii) For every $\tau=(\alpha, \nu)\in D^\rho(Q)$ with $\rho\leqq 1$, $\psi(\tau)=\tau$.

Now we define N_2, $A_{\text{in},2}$, and R_2.

- $N_2=\{\bar{\gamma}\mid\gamma\in N_1\cup\Delta\}$ is the $D^*(Q)$-set such that, if $\gamma\in N_1$, then type$(\bar{\gamma})=\psi(\text{type}(\gamma))$; if $\gamma\in\Delta$ with $\gamma=\delta_i$ for some $i\in[r]$, then type $(\bar{\gamma})=\tau(i)$.
- $A_{\text{in},2}=\phi(A_{\text{in},1})$ where ϕ: AT$(N_1)\to$ AT(N_2) is the mapping defined as follows.

For $\xi\in\text{AT}(N_1)$ with the decomposition $A\tilde{\xi}_m\ldots\tilde{\xi}_1$, if there is an $i\in[m]$ such that $\tilde{\xi}_i$ is a sequence of applicative terms of level 1 with $\tilde{\xi}_i=(\xi_{i,1}, \ldots, \xi_{i,k})$ for some $k\geqq 0$,

$$\text{then } \phi(\xi)=\bar{A}\tilde{\phi}(\tilde{\xi}_m)\ldots\tilde{\phi}(\tilde{\xi}_{i+1})\,\tilde{\xi}\,\tilde{\phi}(\tilde{\xi}_{i-1})\ldots\tilde{\phi}(\tilde{\xi}_1),$$
$$\text{where } \tilde{\xi}=(\bar{\delta}_1, \ldots, \bar{\delta}_r, \phi(\xi_{i,1}), \ldots, \phi(\xi_{i,k})),$$

otherwise $\phi(\xi)=\bar{A}\tilde{\phi}(\tilde{\xi}_m)\ldots\tilde{\phi}(\tilde{\xi}_1)$, where $\tilde{\phi}$ is the extension of ϕ to sequences over AT(N_1) defined componentwise. Note that, if there is an $i\in[m]$ with the mentioned property, then $i\in\{1, 2\}$.

- The set of rules R_2 is determined by (i) and (ii).

(i) For every $\delta\in\Delta$ with δ: (q^k, q) for some $k\geqq 0$, $\bar{\delta}^\wedge\to\delta(y_{1,q}, \ldots, y_{k,q})$ is in R_2.

(ii) If $A^\wedge\to$ **if** b **then** ζ is in R_1 and $\zeta=t[x_i\leftarrow\zeta_i;\ i\in[k]]$, where $k\geqq 0$, $t\in\text{AT}(\Delta\cup X_k)^q$, and for every $i\in[k]$, $\zeta_i\in\text{AT}(N_1\langle F\rangle\cup\Delta\cup Y)^q$ with top$(\zeta_i)\in N_i\langle F\rangle\cup Y$, then $\bar{A}^\wedge\to$ **if** b **then** ζ' is in R_2, where $\zeta'=t[x_i\leftarrow\phi'(\zeta_i);\ i\in[k]]$. The mapping ϕ': AT$(N_1\langle F\rangle\cup\Delta\cup Y)\to$ AT$(N_2\langle F\rangle\cup Y)$ is defined as follows.

For $\zeta\in\text{AT}(N_1\langle F\rangle\cup\Delta\cup Y)$ with the decomposition $\gamma\tilde{\zeta}_m\ldots\tilde{\zeta}_1$, if there is an $i\in[m]$ such that $\tilde{\zeta}$ is a sequence of applicative terms of level 1 and $\tilde{\zeta}_i=(\zeta_{i,1}, \ldots, \zeta_{i,k})$ for some $k\geqq 0$,

$$\text{then } \phi'(\zeta)=\theta\tilde{\phi}'(\tilde{\zeta}_m)\ldots\tilde{\phi}'(\tilde{\zeta}_{i+1})\,\tilde{\zeta}\,\tilde{\phi}'(\tilde{\zeta}_{i-1})\ldots\tilde{\phi}'(\tilde{\zeta}_1),$$
$$\text{where } \tilde{\zeta}=(y_{1,\tau(1)}, \ldots, y_{r,\tau(r)}, \phi'(\zeta_{i,1}), \ldots, \phi'(\zeta_{i,k})),$$

otherwise $\phi'(\zeta)=\theta\tilde{\phi}'(\zeta_m)\ldots\tilde{\phi}'(\zeta_1)$, where $\tilde{\phi}'$ is the extension of ϕ' to sequences over $\mathrm{AT}(N_1\langle F\rangle\cup\Delta\cup Y)$ and the value of θ depends on γ: if $\gamma=A\langle f\rangle$, then $\theta=\bar{A}\langle f\rangle$; if $\gamma=\delta_i$, then $\theta=y_{i,\tau(i)}$; if $\gamma=y_{j,\tau}$ with $\tau\in D(Q)$, then $\theta=y_{j+r,\tau}$; otherwise $\theta=\gamma$.

This completes the construction of M_2. From the definition of N_2 and R_2 it follows immediately that M_2 is terminal restricted. Moreover, determinism is preserved. It is straightforward to prove that $\tau(M_1)=\tau(M_2)$. Since M_1 and M_2 share the encoding e, this implies that totality is preserved. □

The subsequent example illustrates that the idea behind the construction of the previous lemma is really simple, but perhaps buried a bit under the technicalities.

5.6. *Example.* Let $M=(N, e, \Delta, A_{\mathrm{in}}, R)$ be the 2-level count-down transducer of Example 4.8(i). We apply the construction of Lemma 5.5 to M and obtain the following terminal restricted 2-level count-down transducer $M_2=(N_2, e, \Delta, A_{\mathrm{in},2}, R_2)$.

$N_2=\{\bar{A}_{\mathrm{in}}, \bar{A}, \bar{B}, \bar{+}, \bar{\sigma}, \bar{\delta}, \bar{\$}, \bar{a}, \bar{b}\}$,

where $\bar{A}_{\mathrm{in}}$: $(\alpha_{\mathrm{term}}, (\lambda, q))$,
$\bar{A}$: $(\alpha_{\mathrm{term}}(q, q), (q, q))$,
$\bar{B}$: $(\alpha_{\mathrm{term}}, (q, q))$,
$\bar{+}$: $(\alpha_{\mathrm{term}}(q, q)(q, q), (q, q))$,
$\bar{\sigma}$: (qqq, q),
$\bar{\delta}$: (qq, q),
$\bar{\$}$: (λ, q),
$\bar{a}$: (λ, q), and
$\bar{b}$: (λ, q),

where $\alpha_{\mathrm{term}}=(qqq, q)(qq, q)(\lambda, q)(\lambda, q)(\lambda, q)$.

$$A_{\mathrm{in},2}=\bar{A}_{\mathrm{in}}(\bar{\sigma}, \bar{\delta}, \bar{\$}, \bar{a}, \bar{b})().$$

The rules of R_2 are determined by (i) and (ii).

(i) $\bar{\sigma}^{\wedge}\to\sigma(y_{1,q}, y_{2,q}, y_{3,q})$,
$\bar{\delta}^{\wedge}\to\delta(y_{1,q}, y_{2,q})$,
$\bar{\$}^{\wedge}\to\$()$,
$\bar{a}^{\wedge}\to a()$, and
$\bar{b}^{\wedge}\to b()$.

(ii) We use some abbreviations: y_t for "$y_{1,(qqq,q)}, y_{2,(qq,q)}, y_{3,(\lambda,q)}, y_{4,(\lambda,q)}, y_{5,(\lambda,q)}$", and y_σ, y_δ, $y_\$$, y_a, y_b for "$y_{1,(qqq,q)}$", "$y_{2,(qq,q)}$", "$y_{3,(\lambda,q)}$", "$y_{4,(\lambda,q)}$", "$y_{5,(\lambda,q)}$", respectively. Then the following rules are also in R_2.

$$\bar{A}_{\mathrm{in}}(y_t)()\to\bar{A}\langle\mathrm{dec}\rangle(y_t, \bar{B}\langle\mathrm{dec}\rangle(y_t))(y_a()),$$

and the same rule with y_b instead of y_a,

$$\bar{A}(y_t, y_{6,(q,q)})(y_{1,q}) \to \textbf{if not null then}$$
$$\bar{A}\langle\text{dec}\rangle(y_t, \bar{+}\langle\text{dec}\rangle(y_t, \bar{B}\langle\text{dec}\rangle(y_t), y_{6,(q,q)}))(y_\delta(y_a(), y_{1,q})),$$

and the same rule with y_b instead of y_a,

$$\bar{A}(y_t, y_{6,(q,q)})(y_{1,q}) \to \textbf{if } \text{null } \textbf{then } \delta(\$(), y_{6,(q,q)}(y_{1,q})),$$
$$\bar{+}(y_t, y_{6,(q,q)}, y_{7,(q,q)})(y_{1,q}) \to \sigma(y_{6,(q,q)}(y_{1,q}), \$(), y_{7,(q,q)}(y_{1,q})), \text{ and}$$
$$\bar{B}(y_t)(y_{1,q}) \to y_{1,q}.$$

This completes the construction of M_2. We leave it to the reader to check how, e.g., the computation of M_2 which corresponds to the one shown in Example 4.8 (i) would look like. □

Since every n-level S transducer can be transformed into a terminal restricted transducer, we can now construct the related 0-level $n-\text{AT}(S)$ transducer (just by following the requirements of Definition 5.3) which is equivalent to the original transducer (by Lemma 5.4). The formal construction and the proof are left to the reader.

5.7. Lemma. *For every* $n \geq 2$, $n\text{-}T(S) \subseteq 0\text{-}T(n\text{-AT}(S))$ *and* $D_t\, n\text{-}T(S) \subseteq D_t\, 0\text{-}T(n\text{-AT}(S))$. □

Now we turn to the converse direction, namely the simulation of $0\text{-}T(n\text{-AT}(S))$-transducers by n-level S transducers. Starting from a 0-level n-AT(S) transducer M, we want to construct a terminal restricted n-level S transducer M' such that M' and M are related. But for this purpose, M may only have one nonterminal. Hence, we first code the nonterminals of M into the storage type n-AT(S). In order to show that this is possible, we introduce an operator on storage types which adds finite information to the storage type configurations. By showing that n-AT(S) is closed under this operator, we have proved that the nonterminals of M can be coded into n-AT(S), because they form finite information.

5.8. Definition. S *with finite information*, for short S_{fin}, is the storage type (C', P', F', m', I, E'), where $C' = C \times J$ and J is an arbitrary, but fixed infinite set, $P' = P \cup \{\text{info} = j \mid j \in J\}$, $F' = \{(f, \text{info} := j) \mid f \in F \text{ and } j \in J\}$, and for every $c' = (c, k) \in C'$ and $p \in P$, $m'(p)(c') = m(p)(c)$, $m'(\text{info} = j)(c') = (k = j)$, $m'((f, \text{info} := j))(c') = (m(f)(c), j)$ if $m(f)$ is defined on c, and undefined otherwise; finally, $E' = \{\lambda u \in I \cdot (e(u), j) \mid e \in E \text{ and } j \in J\}$. □

Although the set J in the previous definition is infinite, it is justified to talk about finite information: every $X(S_{\text{fin}})$-transducer can only use finitely many elements of J.

Before proving that n-AT(S) is closed under the operator fin, we convince the reader that, for a 0-level S transducer with an arbitrary storage type S, there is an equivalent 0-level S_{fin} transducer with only one nonterminal.

5.9. Lemma. $0\text{-}T(S) \subseteq 0\text{-}T_{\text{one}}(S_{\text{fin}})$ *and* $D_t\, 0\text{-}T(S) \subseteq D_t\, 0\text{-}T_{\text{one}}(S_{\text{fin}})$.

Proof. Let $M=(N, e, \Delta, A_{\text{in}}, R)$ be a 0-level S transducer. Note that $A_{\text{in}} \in N$. We construct the equivalent 0-level S_{fin} transducer $M'=(\{*\}, e', \Delta, *, R')$ by defining $e'=\lambda u \in I \cdot (e(u), A_{\text{in}})$ (note that we can assume that $N \subseteq J$), and if $A \to$ **if** b **then** ζ is in R with $\zeta = t[x_i \leftarrow A_i\langle f_i \rangle;\ i \in [k]]$ for some $t \in \text{AT}(\Delta \cup X_k)^q$, $A_1, \ldots, A_k \in N$, and $f_1, \ldots, f_k \in F$, then $* \to$ **if** info$=A$ **and** b **then** ζ' is in R', where $\zeta' = t[x_i \leftarrow *\langle (f_i, \text{info} := A_i) \rangle;\ i \in [k]]$.

Obviously, M' and M are equivalent and if M is total deterministic, then M' is total deterministic. □

The proof of $n\text{-AT}(S)_{\text{fin}} \leqq_c n\text{-AT}(S)$ involves a "Rounds-like" construction (Theorem 7 of [Rou]) similar to the one for high-level grammars on p. 175 of [Dam] (cf. also Lemma 5.4 of [EV] and Construction 1 of [DamGue1]). Actually, we believe that the next lemma really captures the essence of Rounds' construction in terms of storage types. Every finite restriction U of $n\text{-AT}(S)_{\text{fin}}$ induces a finite subset J_f of J. A current piece of finite information j is coded into a configuration $\zeta = \gamma\langle c \rangle\, \xi_k \ldots \xi_1$ of $n\text{-AT}(S)$ by adding j to γ and by preparing for every sub-applicative term of ζ as many copies as there are elements in J_f. Then a sub-applicative term ξ with finite information j is coded by the correct copy of ξ, namely the one with j associated to the top symbol of ξ.

5.10. Lemma. *For every* $n \geqq 1$, $n\text{-AT}(S)_{\text{fin}} \leqq_c n\text{-AT}(S)$.

Proof. Let U be a finite restriction of $n\text{-AT}(S)_{\text{fin}}$ and let $J_f = \{j_1, \ldots, j_r\}$ be the finite subset of J induced by U. Let $e_1 = \lambda u \in I \cdot (e(u), j_1)$ be the encoding of U where e is an encoding of $n\text{-AT}(S)$.

We define the mapping copy: $D^*(Q) \to D^*(Q)$ as follows. For $\tau = (\alpha m, \ldots (\alpha 1, q) \ldots) \in D^*(Q)$ of level $m \geqq 0$, $\text{copy}(\tau) = (\text{copy}[\alpha m], \ldots (\text{copy}[\alpha 1], q) \ldots)$ of level m, where for every $\alpha = \alpha(1) \ldots \alpha(k) \in D^i(Q)$ with $i, k \geqq 0$, $\text{copy}[\alpha] = \text{copy}(\alpha(1))^r \ldots \text{copy}(\alpha(k))^r$.

Let Ξ_f be the finite set of symbols of $\Xi^{\leqq n}$ involved in U. Define the $D^*(Q)$-set $\langle \Xi', \text{type}' \rangle$ with $\Xi' = \{[\gamma, j] \mid \gamma \in \Xi_f \text{ and } j \in J_f\}$ and for every $[\gamma, j] \in \Xi'$, $\text{type}'([\gamma, j]) = \text{copy}(\text{type}_\Xi(\gamma))$ (recall that type_Ξ is associated to Ξ). We can assume that Ξ' is a subset of $\Xi^{\leqq n}$.

Define the mapping g: $\text{AT}(\Xi^{\leqq n}\langle C \rangle) \times J_f \to \text{AT}(\Xi^{\leqq n}\langle C \rangle)$ inductively on the structure of the applicative terms. For $\xi \in \text{AT}(\Xi^{\leqq n}\langle C \rangle)$ with the decomposition $\gamma\langle c \rangle\, \bar{\xi}_s \ldots \bar{\xi}_1$,

$$g(\xi, j) = [\gamma, j]\langle c \rangle\, \bar{\xi}'_s \ldots \bar{\xi}'_1, \text{ where for every } \bar{\xi} = (\xi_1, \ldots, \xi_k),$$
$$\bar{\xi}' = (g(\xi_1, j_1), \ldots, g(\xi_1, j_r), \ldots, g(\xi_k, j_1), \ldots, g(\xi_k, j_r)).$$

Note that g is injective. Hence, we can define the representation function h: $\text{AT}(\Xi^{\leqq n}\langle C \rangle) \to \text{AT}(\Xi^{\leqq n}\langle C \rangle) \times J$ to be the inverse of g in the usual way: $\text{dom}(h) = \text{range}(g)$ and if $\xi \in \text{dom}(h)$, then $h(\xi) = g^{-1}(\xi)$. We prove that h satisfies the requirements 1–3 of Definition 3.17.

Requirement 1. The encoding $e_2 = \lambda u \in I \cdot g(e(u), j_1)$ of $n\text{-AT}(S)$ satisfies the requirements.

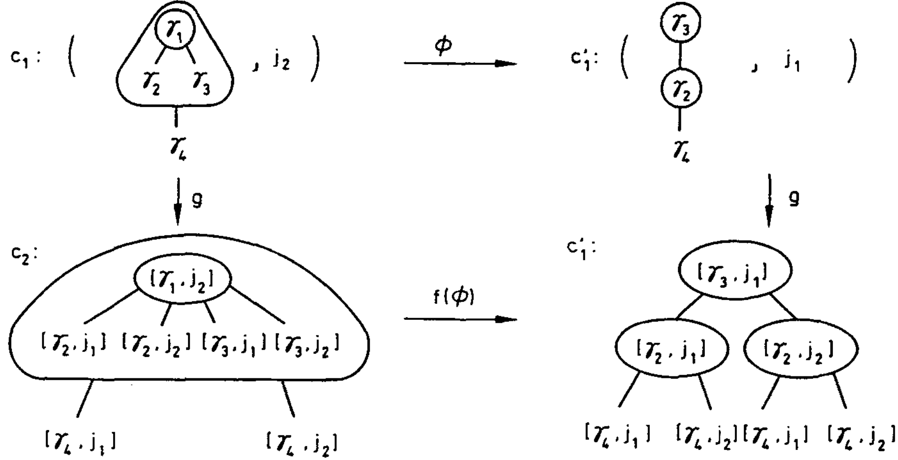

Fig. 7. Coding of finite information in 2-AT(S_0)

Requirement 2. Every predicate ϕ of U is coded by a boolean expression $b(\phi)$.
$\phi=(\text{info}=j)$: $b(\phi)$ is the disjunction of all predicates top$=[\gamma, j]$ where $\gamma\in\Xi_f$.
$\phi=(\text{top}=\gamma)$: $b(\phi)$ is the disjunction of all predicates top$=[\gamma, j]$ where $j\in J_f$.
$\phi=\text{test}(p)$: $b(\phi)=\text{test}(p)$.

Requirement 3. Every instruction $\phi=(\text{push}(\zeta), j)$ of U is coded by an instruction $f(\phi)$. For the construction of $f(\phi)$, we define a mapping g': AT$(\Xi^{\leq n}\langle F\rangle\cup Y)\times J_f\to$ AT$(\Xi^{\leq n}\langle F\rangle\cup Y)$ in a similar way as g is defined.

If $\zeta\in$ AT$(\Xi^{\leq n}\langle F\rangle\cup Y)$ with the decomposition $\psi\tilde{\zeta}_s\ldots\tilde{\zeta}_1$ and

if $\psi=\gamma\langle f\rangle$, then $g'(\zeta, j)=[\gamma, j]\langle f\rangle\tilde{\zeta}'_s\ldots\tilde{\zeta}'_1$, and

if $\psi=y_{i,\tau}$, then $g'(\zeta, j)=y_{\rho,\text{copy}(\tau)}\tilde{\zeta}'_s\ldots\tilde{\zeta}'_1$,

where $\rho=(i-1)r+j$ and for every $\tilde{\zeta}=(\zeta_1, \ldots, \zeta_k)$,

$$\tilde{\zeta}'=(g'(\zeta_1, j_1), \ldots, g'(\zeta_1, j_r), \ldots, g'(\zeta_k, j_1), \ldots, g'(\zeta_k, j_r)).$$

Then $f(\phi)=\text{push}(g'(\zeta, j))$. □

To support the readers understanding we code finite information into the storage type 2-AT(S_0) following the constructions of the previous lemma.

5.11. Example. Consider a finite restriction U of 2-AT$(S_0)_{\text{fin}}$ and let $\Xi_f\subseteq\Xi_2$ be the set of symbols involved in U. Let $J_f=\{j_1, j_2\}$. Hence, $r=2$.

Consider the U-configuration $c_1=(\xi_1, j_2)$ with the applicative term $\xi_1=\gamma_1(\gamma_2, \gamma_3)(\gamma_4)$. For convenience we have dropped the configuration c of S_0. The types of the symbols are determined as follows.

γ_1: $((q, q)(q, q), (q, q))$,
γ_2, γ_3: (q, q), and
γ_4: q.

We apply the instruction $\phi=(\text{push}(\zeta), j_1)$ with $\zeta=\gamma_3(y_{1,(q,q)}(\gamma_4))$ to c_1 and show how this application is coded by 2-AT(S_0). The corresponding commutative diagram is presented in Fig. 7.

The result of the application of ϕ to c_1 is the 2-AT$(S_0)_{\text{fin}}$-configuration $c_1'=(\xi_1', j_1)$ with $\xi_1'=\gamma_3(\gamma_2(\gamma_4))$. Before we apply g to c_1, we compute, as an example, the type of $[\gamma_1, j_1]$ in detail.

$$\begin{aligned}\text{type}([\gamma_1, j_1]) &= \text{copy}(\text{type}_\Xi(\gamma_1))\\ &= \text{copy}(((q,q)(q,q),(q,q)))\\ &= (\text{copy}[(q,q)(q,q)], (\text{copy}[q], q))\\ &= (\text{copy}((q,q))^2\,\text{copy}((q,q))^2, (qq,q)),\\ &= ((\text{copy}[q], q)^2(\text{copy}[q],q)^2, (qq,q))\\ &= ((qq,q)^2(qq,q)^2,(qq,q)) = ((qq,q)^4,(qq,q)).\end{aligned}$$

Hence, for every $j\in\{j_1, j_2\}$,

$$\begin{aligned}[\gamma_1, j]&: ((qq,q)^4,(qq,q)),\\ [\gamma_2, j], [\gamma_3, j]&: (qq,q), \text{ and}\\ [\gamma_4, j]&: q.\end{aligned}$$

The result of the application of g to c_1 is the 2-AT(S_0)-configuration

$$c_2=[\gamma_1, j_2]([\gamma_2, j_1], [\gamma_2, j_2], [\gamma_3, j_1], [\gamma_3, j_2])(t_1, t_2)$$

where $t_i=[\gamma_4, j_i]$ for $i\in\{1,2\}$.

The instruction ϕ is coded by $f(\phi)=\text{push}(g'(\zeta, j_1))=\text{push}(\zeta')$ with $\zeta'=[\gamma_3, j_1](y_{1,(qq,q)}(t_1,t_2), y_{2,(qq,q)}(t_1,t_2))$ and t_i as above.

Applying $f(\phi)$ to c_2 yields the 2-AT(S_0)-configuration

$$c_2'=[\gamma_3, j_1]([\gamma_2, j_1](t_1,t_2), [\gamma_2, j_2](t_1,t_2)).$$

In fact, $g(c_1')=c_2'$. □

5.12. Lemma. *For every* $n\geqq 1$, 0-$T(n$-AT$(S))\subseteq n$-$T(S)$ *and* D_t0-$T(n$-AT$(S))\subseteq D_t n$-$T(S)$.

Proof. By Lemma 5.9, for every 0-level n-AT(S) transducer, there is an equivalent 0-level n-AT$(S)_{\text{fin}}$ transducer with only one nonterminal. Since n-AT$(S)_{\text{fin}}$ can be coded by n-AT(S) (Lemma 5.10), this means, together with Theorem 3.19 (and Fact 4.6), that there is an equivalent 0-level n-AT(S) transducer with one nonterminal. Note that Theorem 3.19 can be applied only if the involved storage types are in the coding relation.

Now let $M=(\{*\}, e, \Delta, *, R)$ be a 0-level n-AT(S) transducer with $e=\lambda u\in I\cdot \eta\langle g(u)\rangle$ for some $\eta\in$ AT$(\Xi^{\leqq n})$ and some encoding g of S. By an easy transformation (cf., e.g., Lemma 3.30 of [EV] for a similar transformation of $X(P(S))$-transducers), we can achieve that the tests of rules of M have the form top$=\gamma$ **and** test(b) where b is a boolean expression over P.

Next we construct the terminal restricted n-level S transducer M' $=(N, e', \Delta, A_{\text{in}}, R')$ which is related to M as follows: $N=\{\gamma\in\Xi^{\leqq n}\mid \gamma$ occurs in $M\}$ where the types are preserved (note that N' is finite), $e'=g$, $A_{\text{in}}=\eta$, and if $*\to$ **if** top$=\gamma$ **and** b **then** ζ is in R with $\zeta=t[x_i\leftarrow *\langle\text{push}(\zeta_i)\rangle;\ i\in[k]]$ for

some $t \in \mathrm{AT}(\Delta \cup X_k)^q$, $k \geqq 0$, and $\zeta_1, \ldots, \zeta_k \in \mathrm{AT}(\Xi^{\leqq n}\langle F \rangle \cup Y)^q$, then $\gamma^\wedge \to$ **if** b **then** ζ' is in R', where $\zeta' = t[x_i \leftarrow \zeta_i;\ i \in [k]]$.

It is an easy observation that M and M' are related. Hence, by Lemma 5.4, M and M' are equivalent. If M is total deterministic, then M' is total deterministic. □

Now, for every $n \geqq 2$, we have obtained the characterization of n-level S transducers by means of 0-level n-AT(S) transducers and thereby verified the claim that the substitution power inherent in the control of n-level S transducers is captured by the storage type n-AT(S). The corresponding characterization for $n=1$ is shown in Corollary 6.6.

5.13. Theorem. *For every* $n \geqq 2$, n-$T(S) = 0$-$T(n$-$\mathrm{AT}(S))$ *and* $D_t n$-$T(S) = D_t 0$-$T(n$-$\mathrm{AT}(S))$.

Proof. Lemma 5.7 and Lemma 5.12. □

6. Applicative Terms and Iterated Pushdowns

In Sect. 5 the rewriting mechanism of n-level S transducers was captured by the storage type n-AT(S), and thereby the characterization of n-$T(S)$ by 0-$T(n$-AT$(S))$ (cf. Theorem 5.13) was obtained. Here we show the equivalence of the storage types n-level applicative term of S (n-AT(S)) and n-iterated bounded excursion pushdown of S ($P^n_{\mathrm{bex}}(S)$). Then it follows from the Justification Theorem 3.16 that 0-$T(n$-AT$(S))$-transducers and 0-$T(P^n_{\mathrm{bex}}(S))$-transducers are equivalent. Hence, in total, the equivalence of n-level S transducers and n-iterated pushdown S transducers is induced: n-$T(S) = 0$-$T(P^n_{\mathrm{bex}}(S))$ (cf. Theorem 6.15).

The equivalence of n-AT(S) and $P^n_{\mathrm{bex}}(S)$ is proved via an intermediate storage type. Informally, n-AT(S) can be viewed as the n-fold application of the storage type operator "tree-pushdown" (TP); formally, we show that, for every $n \geqq 1$, n-AT$(S) \equiv \mathrm{TP}^n(S)$ (cf. Corollary 6.12). The essential idea behind this equivalence is the (iterated) tree-form of applicative terms (cf. Sect. 2.2). Since TP and P_{bex} are equivalent storage type operators, i.e., for every storage type S, $\mathrm{TP}(S) \equiv P_{\mathrm{bex}}(S)$ (Theorem 5.13 of [EV]), the equivalence of $\mathrm{TP}^n(S)$ and $P^n_{\mathrm{bex}}(S)$ (cf. Corollary 6.14) then follows from the monotonicity of P_{bex} and TP (Lemma 3.23).

The concept of tree-pushdown was introduced in [Gue] and formalized as a storage type in [DamGue2]. Before we explain how the equivalence of n-AT(S) and $\mathrm{TP}^n(S)$ is obtained, we recall the formal definition of the tree-pushdown of S from [EV]. A special set of substitution variables $Z = \{z_1, z_2, z_3, \ldots\}$ is used. For every $k \geqq 0$, $Z_k = \{z_1, \ldots, z_k\}$. Recall from Sect. 2.1 that Ω is an infinite ranked set.

6.1. Definition. Let $S = (C, P, F, m, I, E)$ be a storage type. The *tree-pushdown of* S, denoted by TP(S), is the storage type (C', P', F', m', I', E'), where

- $C' = T_{\Omega\langle C \rangle}$
- $P' = \{\mathrm{call} = \delta \mid \delta \in \Omega\} \cup \{\mathrm{test}(p) \mid p \in P\}$
- $F' = \{\mathrm{expand}(\zeta) \mid \zeta \in T_{\Omega\langle F \rangle}(Z)\}$

- for every $c'=\sigma\langle c\rangle(t_1, \ldots, t_k)\in C$,
  $m'(\text{call}=\delta)(c')=\text{true}$ iff $\delta=\sigma$,
  $m'(\text{test}(p))(c')=m(p)(c)$,
  for $c''=\zeta[z_i\leftarrow t_i;\ i\in[k]][f\leftarrow m(f)(c);f\in F]$,
  $m'(\text{expand}(\zeta))(c')=c''$ if $c''\in C'$, and undefined otherwise
- $I'=I$
- $E'=\{\lambda u\in I\cdot\alpha\langle e(u)\rangle\mid\alpha\in T_\Omega, e\in E\}$ with $\alpha\langle e(u)\rangle=\alpha[\gamma\leftarrow\gamma\langle e(u)\rangle;\ \gamma\in\Omega]$. □

Note that, by convention, if $\gamma\in\Omega$ is of rank k, then for every $f\in F$ and every $c\in C$, $\gamma\langle f\rangle$ and $\gamma\langle c\rangle$ are also of rank k. We abbreviate $\text{TP}(S_0)$ by TP. In the usual way we define $\text{TP}^0(S)=S$ and for every $n\geqq 0$, $\text{TP}^{n+1}(S)=\text{TP}(\text{TP}^n(S))$.

6.2. Lemma. TP *is monotonic with respect to the simulation relation, i.e., for two storage types* S_1 *and* S_2, *if* $S_1\leqq S_2$, *then* $\text{TP}(S_1)\leqq\text{TP}(S_2)$.

Proof. This follows immediately from $\text{TP}(S)\equiv P_{\text{bex}}(S)$, where S is an arbitrary storage type S (Theorem 5.13 of [EV]) and from the monotonicity of P_{bex} (Lemma 3.23). □

The equivalence of n-level applicative term of S and n-iterated tree-pushdown of S (i.e., $\text{TP}^n(S)$) is proved by using an inductive statement in which, informally speaking, one level of n-AT(S) is replaced by one application of the operator TP. Formally, for every $n\geqq 1$, $(n+1)$-AT$(S)\equiv\text{TP}(n\text{-AT}(S))$. By considering only the involved sets of configurations, this statement says that applicative terms over $\Xi^{\leqq n+1}\langle C\rangle$ of type q can be represented by trees over $\Omega\langle\text{AT}(\Xi^{\leqq n}\langle C\rangle)^q\rangle$, and vice versa. The first direction of representation is closely related to the concept of tree-form of applicative terms over a $D^*(Q)$-set V (cf. Sect. 2.2). However, the tree-form of an applicative term in $\text{AT}(\Xi^{\leqq n+1}\langle C\rangle)^q$ is a tree indexed by elements of $\Xi^{\leqq n+1}\langle C\rangle^q$ (recall that $\text{AT}(\Xi^{\leqq n+1}\langle C\rangle)^q=T_\Psi(\Xi^{\leqq n+1}\langle C\rangle^q)$ with the ranked set $\Psi=\text{AT}(\Xi^{\leqq n+1}\langle C\rangle)^{=1}$). Hence, in order to take advantage of the tree-form concept in the present situation, we first have to show that symbols of level 0 (i.e., symbols of $\Xi^{\leqq n+1}\langle C\rangle^q$) are not essential in $(n+1)$-AT(S). Moreover, the tree-form of an applicative term is a tree over $\text{AT}(\Xi^{\leqq n+1}\langle C\rangle)^{=1}$ rather than a configuration of TP(n-AT(S)), i.e., a tree over $\Omega\langle\text{AT}(\Xi^{\leqq n}\langle C\rangle)^q\rangle$. The necessary type decrementation and the addition of a "dummy" tree-pushdown symbol of appropriate rank to every "label" in $\text{AT}(\Xi^{\leqq n}\langle C\rangle)^q$ is taken care of in the proof of $(n+1)$-AT$(S)\leqq\text{TP}(n\text{-AT}(S))$ (cf. Lemma 6.7). In the following definition and lemma the first problem is handled.

6.3. Definition. Let $n\geqq 1$. The *n-level applicative term of S without constants*, denoted by n-AT$_+(S)$, is the storage type defined exactly as n-AT(S) except that $\Xi^{\leqq n}$ is replaced by $\Xi^{1,n}=\{\gamma\in\Xi\mid 1\leqq\text{level}(\gamma)\leqq n\}$. □

6.4. Lemma. *For every* $n\geqq 1$, n-AT$(S)\equiv_c n$-AT$_+(S)$.

Proof. Obviously it suffices to prove that n-AT$(S)\leqq_c n$-AT$_+(S)$. Every symbol γ of a finite restriction U of n-AT(S), which is of level 0, is represented by the applicative term $\gamma'()$, where γ': (λ, q). Then the encoding of U and every predicate and instruction of U is coded by itself modulo the replacement of γ: q by $\gamma'()$. We leave the formal proof to the reader. □

Note that the technique of lifting the level of an object of a derived type is applied in Lemma 5.5 to nonterminals.

Since the inductive statement does not cover the proof of n-AT$(S) \equiv \text{TP}^n(S)$ for $n=1$, we prove this case separately.

6.5. Lemma. 1-AT$(S) \equiv_c$ TP(S).

Proof. Since 1-AT$(S) \equiv_c$ 1-AT$_+(S)$, it suffices to show that 1-AT$_+(S) \equiv_c$ TP(S). These two storage types are the same, apart from small differences in terminology. Note that their sets of configurations AT$(\Xi^{1,1}\langle C\rangle)^q$ and $T_{\Omega\langle C\rangle}$ are equal. Clearly, top$=\gamma$ and push(ζ) correspond to call$=\gamma$ and expand(ζ) (where $y_{i,q}$ corresponds to z_i), respectively. □

As a corollary to the previous lemma we can now fill a small gap of Sect. 5: the equivalence of 1-level S transducers and 0-level 1-AT(S) transducers.

6.6. Corollary. *1-$T(S)$ = 0-T(1-AT(S)) and D_t1-$T(S)$ = D_t0-T(1-AT(S)).*

Proof. In Theorem 5.5 of [EV] it is proved that CFT(S) = RT(TP(S)). Hence, by Lemma 6.5 and the Justification Theorem, CFT(S) = RT(1-AT(S)). Then the statement of this corollary follows from Fact 4.6. Since all the mentioned results preserve totality and determinism, also the second equation follows. □

Now let us turn to the simulation of $(n+1)$-AT(S) by TP(n-AT(S)), which forms the first part of the inductive statement. Since $(n+1)$-AT$(S) \equiv_c$ $(n+1)$-AT$_+(S)$, let us consider a configuration ξ of $(n+1)$-AT$_+(S)$. Note that $\xi \in \text{AT}(\Xi^{1,n+1}\langle C\rangle)^q$ and $\Xi^{1,n+1}\langle C\rangle$ does not contain symbols of level 0. Hence, the tree-form of ξ is a (non-indexed) tree over AT$(\Xi^{1,n+1}\langle C\rangle)^{=1}$, cf. Sect. 2.2. Then, the representation rep(ξ) of ξ is defined inductively on the structure of the tree-form of ξ. If $\xi_0(t_1, \ldots, t_k)$ is the tree-form of ξ for some $\xi_0 \in \text{AT}(\Xi^{1,n+1}\langle C\rangle)^{=1}$ and $k \geqq 0$, then rep$(\xi) = *_k\langle\tilde{\xi}_0\rangle(\text{rep}(t_1), \ldots, \text{rep}(t_k))$, where $*_k$ is a fixed (dummy) symbol in Ω of rank k. The applicative term $\tilde{\xi}_0$ over $\Xi^{\leqq n}\langle C\rangle$ of type q is obtained from ξ_0 by replacing every symbol $\gamma \in \Xi^{1,n+1}$ by the symbol $\tilde{\gamma} \in \Xi^{\leqq n}$ and the type of $\tilde{\gamma}$ is computed from the type of γ by an appropriate type decrementation. Thereby, $\tilde{\xi}_0$ becomes an applicative term of level 0 and hence an n-AT(S)-configuration. Since rep is injective, the representation function h of the simulation can be defined as the inverse of rep.

It is easy to see that the predicates top$=\gamma$ and test(p) of n-AT(S) are simulated by test(top$=\tilde{\gamma}$) and by test(test(p)), respectively. Since the substitution of applicative terms for parameters of level 0 is simulated by the substitution of tree-pushdowns for variables in Z, it is also intuitively clear, that an instruction of $(n+1)$-AT(S) can be simulated by a TP(n-AT(S))-instruction. For an example of the representation consider Fig. 8 (the types of the involved symbols are defined in Example 6.8; c_1 and c_2 are configurations of the storage type S).

This discussion suggests that $(n+1)$-AT(S) can even be coded by TP(n-AT(S)). But, if we compare the encodings of both storage types, then we recognize that this is not true. Let $\lambda u \in I \cdot \alpha\langle e(u)\rangle$ be an encoding of TP(n-AT(S)), where $\alpha \in T_\Omega$ and e is an encoding of n-AT(S). Hence, for $u \in I$, to every node of α the same applicative term $e(u)$ is associated. But obviously, the encodings of $(n+1)$-AT(S) do not have a similar property. Hence we have to use a small

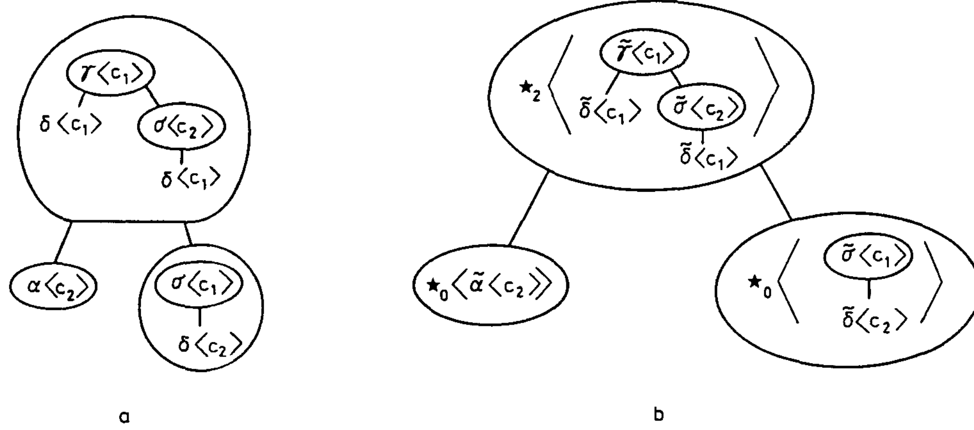

Fig. 8. **a** applicative term in $\mathrm{AT}(\Xi^{1,2}\langle C\rangle)^q$, **b** tree over $\Omega\langle \mathrm{AT}(\Xi^{1,1}\langle C\rangle)^q\rangle$

trick to obtain, for an encoding $e_1=\lambda u\in I\cdot\eta\langle e(u)\rangle$ of $(n+1)$-AT(S), an appropriate encoding of TP(n-AT(S)). Assume that, e.g., $\mathrm{rep}(\eta)=*_2\langle\tilde{\eta}_1\rangle(*_0\langle\tilde{\eta}_2\rangle(), *_0\langle\tilde{\eta}_3\rangle())$ for some $\tilde{\eta}_i\in\mathrm{AT}(\Xi^{\leqq n}\langle C\rangle)^q$. Then we combine the different applicative terms $\tilde{\eta}_i$ into one term $\psi=\$(\tilde{\eta}_1,\tilde{\eta}_2,\tilde{\eta}_3)$, where $\$:(qqq,q)$ is a new symbol in $\Xi^{\leqq n}$, and the encoding e_2 of TP(n-AT(S)) is the function $\lambda u\in I\cdot\mathrm{sel}_1\langle\psi'\rangle(\mathrm{sel}_2\langle\psi'\rangle, \mathrm{sel}_3\langle\psi'\rangle)$ with $\psi'=\psi\langle e(u)\rangle$ and sel_1, sel_2, and sel_3 are new symbols of Ω indicating which subterm of t is the correct one (where sel_1 has rank 2, and sel_2, sel_3 have rank 0). If an $(n+1)$-AT(S)-instruction ϕ is simulated on $e_2(u)$ for some u, then first, the subtree $\tilde{\eta}_1$ is selected by the instruction $\mathrm{expand}(*_2\langle\mathrm{push}(y_{1,q})\rangle(z_1,z_2))$ and second, ϕ is simulated in the usual way. The need for this simulation in two steps destroys the possibility of coding $(n+1)$-AT(S) by TP(n-AT(S)). In Example 6.8 the constructions of the next lemma are illustrated.

6.7. Lemma. *For every* $n\geqq 1$, $(n+1)\text{-AT}(S)\leqq\text{TP}(n\text{-AT}(S))$.

Proof. By Lemma 6.4 and the transitivity of $\leqq_d$ (Lemma 4.19 of [EV]) it suffices to prove that, for every finite restriction U of $(n+1)$-AT$_+(S)$, $U\leqq_d\mathrm{TP}(n\text{-AT}(S))$. Let $\Xi_f\subseteq\Xi^{1,n+1}$ be the finite set of symbols used in U. Furthermore, let $e_1=\lambda u\in I\cdot\eta\langle e(u)\rangle$ be the encoding of U with $\eta\in\mathrm{AT}(\Xi_f)^q$ and $e\in E$. Define the mapping $\mathrm{dec}: D^+(Q)\to D^*(Q)$, which decrements the level of a type, as follows. For every $k\geqq 0$, $\mathrm{dec}((q^k,q))=q$, and for every $(\alpha,v)\in D^+(Q)$ with $\alpha\in D^+(Q)^k$ for some $k\geqq 0$, $\mathrm{dec}((\alpha,v))=(\mathrm{dec}(\alpha(1))\ldots\mathrm{dec}(\alpha(k)),\mathrm{dec}(v))$.

By means of this type decrementation we define the finite $D^*(Q)$-set $\langle\tilde{\Xi},\mathrm{type}\rangle$, where $\tilde{\Xi}=\{\tilde{\gamma}\mid\gamma\in\Xi_f\}$ and $\mathrm{type}(\tilde{\gamma})=\mathrm{dec}(\mathrm{type}_\Xi(\gamma))$. (Note that, since every symbol γ in Ξ_f has at least level 1, dec is defined on $\mathrm{type}_\Xi(\gamma)$.) W.l.o.g. we can assume that $\tilde{\Xi}\subseteq\Xi^{\leqq n}$. The isomorphism $\tilde{}:\Xi_f\to\tilde{\Xi}$ is extended to the isomorphism $\tilde{}:\mathrm{AT}(\Xi_f)^{=1}\to\mathrm{AT}(\tilde{\Xi})^q$ in the obvious way. The extension to the isomorphism $\mathrm{AT}(\Xi_f\langle C\rangle)^{=1}\to\mathrm{AT}(\tilde{\Xi}\langle C\rangle)^q$ is also denoted by $\tilde{}$.

For the definition of the representation function, we construct the set of subterms of η of level 1, i.e., the set of labels of the tree-form of η. Formally, define the mapping set: $\mathrm{AT}(\Xi_f)^q \rightarrow \mathrm{PS}$, where PS is the powerset of $\mathrm{AT}(\tilde{\Xi})^q$, as follows. Let $\xi \in \mathrm{AT}(\Xi_f)^q$ and $\xi = \xi_0(\xi_1, \ldots, \xi_k)$ for some $\xi_0 \in \mathrm{AT}(\Xi_f)^v$ with $v = (q^k, q)$ and $k \geqq 0$, and $\xi_1, \ldots, \xi_k \in \mathrm{AT}(\Xi_f)^q$; then $\mathrm{set}(\xi) = \{\tilde{\xi}_0\} \cup \mathrm{set}(\xi_1) \cup \ldots \cup \mathrm{set}(\xi_k)$. Let $\mathrm{set}(\eta) = \{\tilde{\eta}_1, \ldots, \tilde{\eta}_r\}$ for some $r \geqq 1$. The number r determines the type of the additional symbol $\$ \in \Xi^{\leqq n}$ at which we want to hang the $\tilde{\eta}_i$'s:

$$\$: (q^r, q).$$

Define the ranked set $\Sigma = \{\mathrm{sel}_j \mid j \in [r]\}$ with $\mathrm{sel}_j \in \Sigma_k$ if $\eta_j : (q^k, q)$. We can assume that $\Sigma \subseteq \Omega$.

The function $g: I \rightarrow \mathrm{AT}(\Xi^{\leqq n}\langle C \rangle)^q$ is defined by $g(u) = \psi\langle e(u) \rangle$ with $\psi = \$(\tilde{\eta}_1, \ldots, \tilde{\eta}_r)$.

Now we define the representation function

$$h: T_{\Omega\langle\Psi\rangle} \rightarrow \mathrm{AT}(\Xi^{\leqq n+1}\langle C \rangle)^q \text{ with } \Psi = \mathrm{AT}(\Xi^{\leqq n}\langle C \rangle)^q$$

and prove that h satisfies the requirements of Definition 3.13.

$$\mathrm{dom}(h) = T_\Phi(T_{\Phi'}) \text{ with } \Phi = \{*_k\langle \tilde{\xi} \rangle \mid k \geqq 0 \text{ and } \xi \in \mathrm{AT}(\Xi_f\langle C \rangle) \text{ of type } (q^k, q)\}$$

where $*_k \in \Omega$ of rank k, not in Σ, and $\Phi' = \Sigma\langle \mathrm{range}(g) \rangle$.

Let $t \in \mathrm{dom}(h)$.

(i) If $t = *_k\langle \tilde{\xi} \rangle(t_1, \ldots, t_k)$ with $k \geqq 0$, $\xi \in \mathrm{AT}(\Xi_f\langle C \rangle)$ of type (q^k, q), and $t_1, \ldots, t_k \in \mathrm{dom}(h)$, then $h(t) = \xi(h(t_1), \ldots, h(t_k))$.

(ii) If $t = \mathrm{sel}_j\langle g(u) \rangle(t_1, \ldots, t_k)$, $k \geqq 0$, $u \in \mathrm{dom}(g)$, and $t_1, \ldots, t_k \in \mathrm{dom}(h)$, then $h(t) = (\eta_j)\langle e(u) \rangle(h(t_1), \ldots, h(t_k))$, where $(\eta_j)\langle e(u) \rangle$ is obtained from η_j in the same way as $\eta\langle e(u) \rangle$ is obtained from η.

Requirement 1. Before we define an encoding e_2 which simulates e_1, we define a partial function tree: $\mathrm{AT}(\Xi_f)^q \rightarrow T_\Omega$ which provides the "initial" tree-push-down (corresponding to η). Let $\xi \in \mathrm{AT}(\Xi_f)^q$. Then, as usual, $\xi = \xi_0(\xi_1, \ldots, \xi_k)$ for some $k \geqq 0$ and some $\xi_0, \xi_1, \ldots, \xi_k$. If $\tilde{\xi}_0 = \tilde{\eta}_j$ for some $j \in [r]$, then $\mathrm{tree}(\xi) = \mathrm{sel}_j(\mathrm{tree}(\xi_1), \ldots, \mathrm{tree}(\xi_k))$.

Now it is an easy observation that the encoding $e_2 = \lambda u \in I \cdot (\mathrm{tree}(\eta))\langle g(u) \rangle$ satisfies the requirement 1 of Definition 3.13.

Requirement 2. Every predicate ϕ of U is simulated by a TP(n-AT(S))-flowchart ω for predicates.

$\phi = (\mathrm{top} = \gamma)$: For every $\mathrm{sel}_j \in \Sigma$ with $\eta_j : (q^k, q)$ for some $k \geqq 0$,

$A_{\mathrm{in}} \rightarrow$ **if** call $= \mathrm{sel}_j$ **then** $A\langle \mathrm{expand}(*_k\langle \mathrm{push}(y_{j,q}) \rangle(z_1, \ldots, z_k)) \rangle$

$A_{\mathrm{in}} \rightarrow$ **if** b **then** $A\langle \mathrm{id} \rangle$, and

$A \rightarrow$ **if** test$(\mathrm{top} = \tilde{\gamma})$ **then** true$\langle \mathrm{id} \rangle$ **else** false$\langle \mathrm{id} \rangle$

are rules of ω, where b is the conjunction of all boolean expressions of the form **not**(call $= \mathrm{sel}_j$), where $\mathrm{sel}_j \in \Sigma$.

$\phi = \mathrm{test}(p)$: ϕ is simulated by the boolean expression test(test(p)).

Requirement 3. Every instruction push(ζ) is simulated by a TP(n-AT(S))-flowchart ω for instructions. For every $\mathrm{sel}_j \in \Sigma$ with η_j: (q^k, q) for some $k \geq 0$,

$$A_{\mathrm{in}} \to \textbf{if } \mathrm{call} = \mathrm{sel}_j \textbf{ then } A_{\mathrm{in}}\langle \mathrm{expand}(*_k\langle \mathrm{push}(y_{j,q})\rangle(z_1, \ldots, z_k))\rangle \text{ and}$$

$$A_{\mathrm{in}} \to \textbf{if } b \textbf{ then } \mathrm{stop}\langle \mathrm{expand}(\phi(\zeta))\rangle$$

are rules of ω, where b is defined as in requirement 2. The mapping ϕ: $\mathrm{AT}(\Xi_f\langle F\rangle \cup Y)^q \to T_{\Omega\langle F'\rangle}(Z)$, where F' denotes the set of instructions of n-AT(S), is defined inductively as follows.

(i) For $y_{i,q} \in Y$, $\phi(y_{i,q}) = z_i$.
(ii) For $\zeta = \zeta_0(\zeta_1, \ldots, \zeta_k) \in \mathrm{AT}(\Xi_f\langle F\rangle \cup Y)^q$,
$\phi(\zeta) = *_k\langle \mathrm{push}(\tilde{\zeta}_0)\rangle(\phi(\zeta_1), \ldots, \phi(\zeta_k))$ with
$\tilde{\zeta}_0 = \zeta_0[\gamma \leftarrow \tilde{\gamma};\ \gamma \in \Xi_f][y_{i,\tau} \leftarrow y_{i,\mathrm{dec}(\tau)};\ y_{i,\tau} \in Y]$.

Here we only provide a proof of requirement 3.2 and that only for trees in $T_\Phi \subseteq \mathrm{dom}(h)$ (see above). Actually, the correctness of this case follows immediately from the next statement $(*)$ for which we assume that 3.3.1 and 3.1.2 are true. The meaning functions of $(n+1)$-AT(S), TP(n-AT(S)), n-AT(S), and S are denoted by m_1, m_2, m', and m, respectively. The sets of instructions of the latter two storage types are denoted by F' and by F, respectively.

For every $\zeta \in \mathrm{AT}(\Xi_f\langle F\rangle \cup Y)^q$ and every $c' \in T_\Phi$,
$(*)$ if $m_1(\mathrm{push}(\zeta))(h(c'))$ is defined, then
$h(m_2(\mathrm{expand}(\phi(\zeta)))(c') = m_1(\mathrm{push}(\zeta))(h(c'))$.

The proof is by induction on the structure of ζ. Let $c' = *_k\langle \tilde{\xi}\rangle(t_1, \ldots, t_k) \in T_\Phi$.

If $\zeta \in \Xi_f\langle F\rangle \cup Y$, then there are two cases.

Case 1. $\zeta \in \Xi_f\langle F\rangle$: Actually, this case cannot occur, because ζ: q and every symbol of $\Xi_f\langle F\rangle$ has at least level 1.

Case 2. $\zeta = y_{i,q}$: $h(m_2(\mathrm{expand}(\phi(y_{i,q})))(c')) = h(t_i)$
$= m_1(\mathrm{push}(y_{i,q}))(\xi(h(t_1), \ldots, h(t_k))) = m_1(\mathrm{push}(y_{i,q}))(h(c'))$.

Now let $\zeta = \zeta_0(\zeta_1, \ldots, \zeta_\rho)$ with $\rho \geq 0$ and assume that the statement holds for $\zeta_1, \ldots, \zeta_\rho$. Let $\xi = \gamma\langle c\rangle \xi_\mu \ldots \xi_1$ with $\mu \geq 0$, γ: $(\alpha\mu, \ldots(\alpha 1, (q^k, q))\ldots)$, and $\xi_i \in \mathrm{AT}(\Xi_f\langle C\rangle)^{\alpha i}$. Hence, $h(c') = \xi(h(t_1), \ldots, h(t_k)) = \gamma\langle c\rangle \xi_\mu \ldots \xi_1(h(t_1), \ldots, h(t_k))$. For every $\alpha \in D^*(Q)^s$ with $s \geq 0$, we denote $(\mathrm{dec}(\alpha(1)), \ldots, \mathrm{dec}(\alpha(s)))$ by $\mathrm{dec}(\alpha)$.

$h(m_2(\mathrm{expand}(\phi(\zeta)))(c'))$
$= h(m_2(\mathrm{expand}(*_\rho\langle \mathrm{push}(\tilde{\zeta}_0)\rangle(\phi(\zeta_1), \ldots, \phi(\zeta_\rho))))(*_k\langle \tilde{\xi}\rangle(t_1, \ldots, t_k)))$
$= h(*_\rho\langle \mathrm{push}(\tilde{\zeta}_0)\rangle(\phi(\zeta_1), \ldots, \phi(\zeta_\rho))[\,]_1)$,
where $[\,]_1 = [z_i \leftarrow t_i;\ i \in [k]][\mathrm{push}(\psi) \leftarrow m'(\mathrm{push}(\psi))(\tilde{\xi});\ \mathrm{push}(\psi) \in F']$
$= h(*_\rho\langle m'(\mathrm{push}(\tilde{\zeta}_0))(\tilde{\xi})\rangle(\phi(\zeta_1)[\,]_1, \ldots, \phi(\zeta_\rho)[\,]_1))$
$= h(*_\rho\langle \tilde{\zeta}_0[\,]_2\rangle(\phi(\zeta_1)[\,]_1, \ldots, \phi(\zeta_\rho)[\,]_1))$,
where $[\,]_2 = [y_{\mathrm{dec}(\alpha i)} \leftarrow \tilde{\xi}_i;\ i \in [\mu]][f \leftarrow m(f)(c); f \in F]$
$= \zeta_0[\,]_3(h(m_2(\mathrm{expand}(\phi(\zeta_1)))(c')), \ldots, h(m_2(\mathrm{expand}(\phi(\zeta_\rho)))(c')))$,
where $[\,]_3 = [y_{\alpha i} \leftarrow \xi_i;\ i \in [\mu]][f \leftarrow m(f)(c); f \in F]$
$= \zeta_0[\,]_3(m_1(\mathrm{push}(\zeta_1))(h(c')), \ldots, m_1(\mathrm{push}(\zeta_\rho))(h(c')))$
(by induction hypothesis)

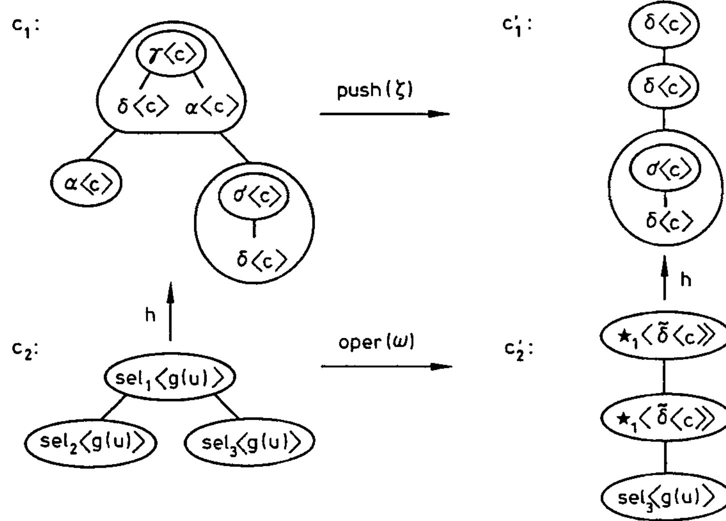

Fig. 9. Simulation of $2-\mathrm{AT}(S_0)$ by $\mathrm{TP}(1-\mathrm{AT}(S_0))$

$=\zeta_0[\,]_3(\zeta_1[\,]_4, \ldots, \zeta_r[\,]_4)$,
  where $[\,]_4=[y_{\alpha i}\leftarrow \xi_i;\, i\in[\mu]]\,[y_{j,q}\leftarrow h(t_j);\, j\in[k]]\,[f\leftarrow m(f)(c);\, f\in F]$
$=\zeta_0(\zeta_1, \ldots, \zeta_\rho)[\,]_4$
$=m_1(\mathrm{push}(\zeta))(h(c'))$. □

6.8. Example. We illustrate the simulation of 2-AT(S_0) by TP(1-AT(S_0)). Let γ: $((q, q)(\lambda, q), (q, q))$, σ: $((q, q), (\lambda, q))$, δ: (q, q), and α: (λ, q) be symbols of $\Xi^{1,2}$. The corresponding elements of $\tilde{\Xi}$ are $\tilde{\gamma}$: (qq, q), $\tilde{\sigma}$: (q, q), $\tilde{\delta}$: q, and $\tilde{\alpha}$: q. Let U be a finite restriction of 2-AT$_+(S_0)$ with encoding $e_1=\lambda u\in I\cdot\eta\langle e(u)\rangle$. Since $e_1(u)$ has a fixed value, viz. $\eta\langle c\rangle$ where c is the configuration of S_0, we denote $e_1(u)$ by c_1. Assume that $\eta=\gamma(\delta, \alpha)(\alpha(), \sigma(\delta)())$. Then $\mathrm{set}(\eta)=\{\tilde{\eta}_1, \tilde{\eta}_2, \tilde{\eta}_3\}$ with $\tilde{\eta}_1=\tilde{\gamma}(\tilde{\delta}, \tilde{\alpha})$, $\tilde{\eta}_2=\tilde{\alpha}$, and $\tilde{\eta}_3=\tilde{\sigma}(\tilde{\delta})$. Hence, $r=3$ and $\$$: (q^3, q). Since for every $u\in I$, $e(u)=c$, $g(u)=\$(\tilde{\gamma}\langle c\rangle(\tilde{\delta}\langle c\rangle, \tilde{\alpha}\langle c\rangle), \tilde{\alpha}\langle c\rangle, \tilde{\sigma}\langle c\rangle(\tilde{\delta}\langle c\rangle))$. Before determining the encoding e_2 by which e_1 is simulated, we compute $\mathrm{tree}(\eta)$: $\mathrm{tree}(\eta)=\mathrm{sel}_1(\mathrm{sel}_2(), \mathrm{sel}_3())$. Hence, $e_2=\lambda u\in I\cdot \mathrm{sel}_1\langle g(u)\rangle(\mathrm{sel}_2\langle g(u)\rangle(), \mathrm{sel}_3\langle g(u)\rangle())$. Since $g(u)$ has a fixed value, we can denote $e_2(u)$ by c_2. It is easy to see that $h(c_2)=c_1$.

Now we apply the instruction push(ζ) with $\zeta=\delta\langle \mathrm{id}\rangle(y_{1,(q,q)}(y_{2,q}))$ to c_1. The result is the applicative term $c'_1=\delta\langle c\rangle(\delta\langle c\rangle(\sigma\langle c\rangle(\delta\langle c\rangle)()))$. ζ is transformed by ϕ into $*_1\langle \mathrm{push}(\tilde{\delta}\langle \mathrm{id}\rangle)\rangle(*_1\langle \mathrm{push}(y_{1,q})\rangle(z_2))$. Let ω be the flowchart that simulates push(ζ), and let $c'_2=\mathrm{oper}(\omega)(c_2)$. Then, $c'_2=*_1\langle\tilde{\delta}\langle c\rangle\rangle(*_1\langle\tilde{\delta}\langle c\rangle\rangle(\mathrm{sel}_3\langle g(u)\rangle()))$. It is easy to check that $h(c'_2)=c'_1$. Figure 9 shows the corresponding commutative diagram. □

The following lemma proves the other direction of our inductive statement, viz. TP(n-AT(S))$\leqq$($n+1$)-AT(S). Now we can even show that TP(n-AT(S)) is coded by the ($n+1$)-level applicative term of S.

The first idea one has in mind, is to apply the construction of the previous lemma (without the problem with the encoding), but in the other direction. Additionally, the label of every node of a TP(n-AT(S))-configuration has to be incorporated into the associated n-level applicative term ξ. However, this construction does not work as we want to point out at the following incomplete example. For $i \in \{0, 1\}$, let $-_i \in \Omega_i$. Let $\delta \in \Xi^{\leqq n}$ of type (q, q), let $\xi \in \mathrm{AT}(\Xi^{\leqq n}\langle C\rangle)^q$, and let t be a tree over $\Omega\langle \mathrm{AT}(\Xi^{\leqq n}\langle C\rangle)^q\rangle$. Hence, $c_1 = -_1\langle \delta\langle c\rangle(\xi)\rangle(t)$ is a possible configuration of TP(n-AT(S)). The application of the instruction $\phi = \mathrm{expand}(-_1\langle \mathrm{push}(y_{1,q})\rangle(-_0\langle \mathrm{push}(y_{1,q})\rangle()))$ to c_1 yields the configuration $c_1' = -_1\langle\xi\rangle(-_0\langle\xi\rangle())$. Roughly speaking, ϕ copies ξ such that each copy is associated to symbols of different rank, i.e., to symbols $-_1$ and $-_0$ of rank 1 and of rank 0, respectively. Using the straightforward construction, c_1' is represented by $c_2' = \bar{\xi}_1(\bar{\xi}_0)$, where $\bar{\xi}_i$ is the applicative term of type (q^i, q) "representing" $-_i\langle\xi\rangle$. Note that $\bar{\xi}_0$ and $\bar{\xi}_1$ have different types. Since ξ occurs only once in the representation of c_1, it is now easy to see that there is no instruction ϕ' of $(n+1)$-AT(S) which simulates ϕ, i.e., which provides two copies of ξ with different types.

To overcome this problem we use a "Rounds-like" construction. For every sub-term ξ' of an applicative term ξ, which is associated to a tree-pushdown symbol in a configuration of TP(n-AT(S)), as many copies are prepared as there are tree-pushdown symbols. Then, according to the rank of that tree-pushdown symbol, to which ξ' is associated after the application of an instruction, the copy with the correct type can be chosen. An example which illustrates the formal construction, can be found after the next lemma.

6.9. Lemma. *For every* $n \geqq 1$, TP(n-AT(S)) $\leqq_c (n+1)$-AT(S).

Proof. Let U be a finite restriction of TP(n-AT(S)) with encoding e_1. Let $\Omega_f = \{\gamma_1, \ldots, \gamma_r\}$ and $\Xi_f = \{\delta_1, \ldots, \delta_s\}$ be the finite subsets of symbols of Ω and $\Xi^{\leqq n}$, respectively, which occur in U. We have to show that $U \leqq_{dc} (n+1)$-AT(S), i.e., there is a representation function $h: \mathrm{AT}(\Xi^{\leqq n+1}\langle C\rangle)^q \to T_\Psi$ with $\Psi = \Omega\langle \mathrm{AT}(\Xi^{\leqq n}\langle C\rangle)^q\rangle$ for which the requirements 1–3 of Definition 3.17 hold.

We define the mapping $\mathrm{inc}(\gamma)$: $D^*(Q) \to D^+(Q)$, for every $\gamma \in \Omega_f$, as follows. If $\tau = (\alpha m, \ldots (\alpha 1, q) \ldots) \in D^*(Q)$ for some $m \geqq 0$, then $\mathrm{inc}(\gamma)(\tau) = (\mathrm{inc}[\alpha m], \ldots (\mathrm{inc}[\alpha 1], (q^j, q)) \ldots)$ with $j = \mathrm{rank}(\gamma)$, and for every $\alpha = \alpha(1) \ldots \alpha(k) \in D^i(Q)^*$ and $k \geqq 0$, $\mathrm{inc}[\alpha]$ abbreviates the sequence $\mathrm{inc}(\gamma_1)(\alpha(1)) \ldots \mathrm{inc}(\gamma_r)(\alpha(1)) \ldots \mathrm{inc}(\gamma_1)(\alpha(k)) \ldots \mathrm{inc}(\gamma_r)(\alpha(k))$ (note that, if $k = 0$, then $\mathrm{inc}[\alpha] = \lambda$).

We use this incrementation of the level of types to introduce a finite $D^*(Q)$-set, namely $\langle \bar{\Xi}, \mathrm{type}\rangle$ with $\bar{\Xi} = \{[\gamma, \delta] \mid \gamma \in \Omega_f,\ \delta \in \Xi_f\}$ and $\mathrm{type}([\gamma, \delta]) = \mathrm{inc}(\gamma)(\mathrm{type}_\Xi(\delta))$. We can assume that $\bar{\Xi} \subseteq \Xi^{\leqq n+1}$.

We define for every $\gamma \in \Omega_f$ the mapping $\tilde{\gamma}$ on $\mathrm{AT}(\Xi_f\langle C\rangle)$, which copies every sub(-applicative-)term r times. For every $\tau \in D^*(Q)$, the one-to-one mapping $\tilde{\gamma}$: $\mathrm{AT}(\Xi_f\langle C\rangle)^\tau \to \mathrm{AT}(\Xi^{\leqq n+1}\langle C\rangle)^\nu$ with $\nu = \mathrm{inc}(\gamma)(\tau)$ is defined inductively as follows. If $\xi \in \mathrm{AT}(\Xi_f\langle C\rangle)^\tau$ has the decomposition $\delta\langle c\rangle\, \xi_\mu \ldots \xi_1$, then $\tilde{\gamma}(\xi) = [\gamma, \delta]\langle c\rangle\, \bar{\xi}_\mu \ldots \bar{\xi}_1$ and for every $i \in [\mu]$, $\bar{\xi}_i$ abbreviates the sequence $(\tilde{\gamma}_1(\xi_i(1)), \ldots, \tilde{\gamma}_r(\xi_i(1)), \ldots, \tilde{\gamma}_1(\xi_i(k_i)), \ldots, \tilde{\gamma}_r(\xi_i(k_i)))$, where k_i is the length of ξ_i and $\xi_i(j)$ is the j-th component of ξ_i.

We use the mappings $\tilde{\gamma}$ to define the one-to-one mapping g:

$T_{\Psi'} \to \mathrm{AT}(\Xi^{\leqq n+1}\langle C\rangle)^q$ with $\Psi' = \Omega_f\langle \mathrm{AT}(\Xi_f\langle C\rangle)^q\rangle$ inductively. For $\gamma\in\Omega_f$ of rank $k\geqq 0$, $\xi\in\mathrm{AT}(\Xi_f\langle C\rangle)^q$ and $t_1, \ldots, t_k\in T_{\Psi'}$, $g(\gamma\langle\xi\rangle(t_1, \ldots, t_k)) = \tilde{\gamma}(\xi)(g(t_1), \ldots, g(t_k))$.

Finally, we define the representation function h: $\mathrm{AT}(\Xi^{\leqq n+1}\langle C\rangle)^q \to T_\Psi$ with $\Psi = \Omega\langle \mathrm{AT}(\Xi^{\leqq n}\langle C\rangle)^q\rangle$ to be the inverse of g. We show that h satisfies the requirements of Definition 3.17.

Requirement 1. Let $e_1 = \lambda u\in I\cdot\alpha\langle e(u)\rangle$ for some $e\in E$. It is an easy observation that $e_2 = \lambda u\in I\cdot g(\alpha\langle e(u)\rangle)$ is an encoding of $(n+1)$-AT(S) and that it satisfies the requirements.

Requirement 2. The predicates call$=\gamma$ and test (top$=\delta$) of U are coded by the boolean expressions top$=[\gamma, \delta_1]$ **or**...**or** top$=[\gamma, \delta_s]$ and top$=[\gamma_1, \delta]$ **or**...**or** top$=[\gamma_r, \delta]$, respectively, and the predicate test(test(p)) of U is coded by test(p).

Requirement 3. Let expand(ζ) be an instruction of U where $\zeta\in T_\Phi(Z)$ with $\Phi = \{\gamma\langle \mathrm{push}(\phi)\rangle \mid \gamma\in\Omega_f,\ \phi\in\mathrm{AT}(\Xi_f\langle F\rangle\cup Y)^q\}$ and $\mathrm{rank}(\gamma\langle\mathrm{push}(\phi)\rangle) = \mathrm{rank}(\gamma)$. Define the one-to-one mapping g': $T_\Phi(Z)\to\mathrm{AT}(\Xi^{\leqq n+1}\langle F\rangle\cup Y)^q$ (similar to g) for γ of rank $k\geqq 0$, by $g'(\gamma\langle\mathrm{push}(\phi)\rangle(t_1, \ldots, t_k)) = \tilde{\gamma}'(\phi)(g'(t_1), \ldots, g'(t_k))$ and $g'(z_j) = y_{j,q}$, and for every $\gamma_j\in\Omega_f$, the one-to-one mapping $\tilde{\gamma}'_j$: $\mathrm{AT}(\Xi_f\langle F\rangle\cup Y)^\tau \to \mathrm{AT}(\Xi^{\leqq n+1}\langle F\rangle\cup Y)^v$ with $v = \mathrm{inc}(\gamma_j)(\tau)$ is defined inductively (similar to $\tilde{\gamma}_j$) by $\tilde{\gamma}'_j(\delta\langle f\rangle\zeta_\mu\ldots\zeta_1) = [\gamma_j, \delta]\langle f\rangle\tilde{\zeta}_\mu\ldots\tilde{\zeta}_1$, and $\tilde{\gamma}'_j(y_{i,k}\zeta_\mu\ldots\zeta_1) = y_{\sigma,\rho}\tilde{\zeta}_\mu\ldots\tilde{\zeta}_1$ where $\sigma = (i-1)\cdot r+j$ and $\rho = \mathrm{inc}(\gamma_j)(\kappa)$; $\tilde{\zeta}$ is obtained from ζ in the same way as $\tilde{\xi}$ is obtained from ξ. Then push$(g'(\zeta))$ codes expand(ζ).

Here we provide a formal proof of Requirement 3.2 under the assumption that 3.1.1 and 3.1.2 hold. Let m_1, m_2, m', and m denote the meaning functions of TP(n-AT(S)), of $(n+1)$-AT(S), of n-AT(S), and of S, respectively. The sets of instructions of the latter two storage types are denoted by F' and by F, respectively. Before starting the main proof of 3.2, we formalize the intuitively clear fact that, for every $\gamma\in\Omega_f$, $\tilde{\gamma}$ and $\tilde{\gamma}'$ correspond to each other.

(*) For every $\phi\in\mathrm{AT}(\Xi_f\langle F\rangle\cup Y)$, $c\in C$,
$(\alpha\mu, \ldots(\alpha 1, q)\ldots)\in D^*(Q)$ with $\mu\geqq 0$, $\xi_1, \ldots, \xi_\mu\in\mathrm{AT}(\Xi_f\langle C\rangle)$
with ξ_i: αi for every $i\in[\mu]$, and $\gamma_j\in\Omega_f$,
$\tilde{\gamma}'_j(\phi)[y_{\mathrm{inc}[\alpha i]}\leftarrow\tilde{\xi}_i;\ i\in[\mu]][c]_f = \tilde{\gamma}_j(\phi[y_{\alpha i}\leftarrow\xi_i;\ i\in[\mu]][c]_f)$,
where $[c]_f$ abbreviates $[f\leftarrow m(f)(c);\ f\in F]$.

The formal proof (by induction on the structure of ϕ) is left to the reader. Requirement 3.2 follows immediately from the next statement.

(**) For every $\zeta\in T_\Phi(Z)$ and $t\in T_{\Psi'}$,
$m_2(\mathrm{push}(g'(\zeta)))(g(t)) = g(m_1(\mathrm{expand}(\zeta))(t))$.

We prove this statement by induction on the structure of ζ. Let $t = \gamma\langle\xi\rangle(t_1, \ldots, t_k)$, $k\geqq 0$, $\xi = \sigma\langle c\rangle\xi_\mu\ldots\xi_1$, and σ: $(\alpha\mu, \ldots(\alpha 1, q)\ldots)$.

$\zeta = z_j$: $m_2(\mathrm{push}(g'(\zeta)))(g(t))$
$= m_2(\mathrm{push}(y_{j,q}))(\tilde{\gamma}(\xi)(g(t_1), \ldots, g(t_k)))$
$= g(t_j)$
$= g(m_1(\mathrm{expand}(z_j))(t))$.

$\zeta = \delta\langle\mathrm{push}(\phi)\rangle(\zeta_1, \ldots, \zeta_v)$ with $v\geqq 0$ and assume that (**) holds for $\zeta_1, \ldots, \zeta_v$.

First we compute $g(t)$.

$$\begin{aligned} g(t) &= g(\gamma\langle\xi\rangle(t_1, \ldots, t_k)) \\ &= \tilde{\gamma}(\xi)(g(t_1), \ldots, g(t_k)) \\ &= [\gamma, \sigma]\langle c\rangle\, \tilde{\xi}_\mu \ldots \tilde{\xi}_1(g(t_1), \ldots, g(t_k)) \end{aligned}$$

with $[\gamma, \sigma]: (\mathrm{inc}[\alpha\mu], \ldots(\mathrm{inc}[\alpha 1], (q^k, q))\ldots)$.

Then $m_2(\mathrm{push}(g'(\zeta)))(g(t))$

$= m_2(\mathrm{push}(\tilde{\delta}'(\phi)(g'(\zeta_1), \ldots, g'(\zeta_\nu))))(g(t))$

$= \tilde{\delta}'(\phi)(g'(\zeta_1), \ldots, g'(\zeta_\nu))[\,]_1[c]_f$,

where $[\,]_1 = [y_{\mathrm{inc}[\alpha i]} \leftarrow \tilde{\xi}_i;\ i\in[\mu]][y_{i,q} \leftarrow g(t_i);\ i\in[k]]$

and $[c]_f = [f \leftarrow m(f)(c);\ f\in F]$

$= (\tilde{\delta}'(\phi)[\,]_2)(g'(\zeta_1)[\,]_1[c]_f, \ldots, g'(\zeta_\nu)[\,]_1[c]_f)$,

where $[\,]_2 = [y_{\mathrm{inc}[\alpha i]} \leftarrow \tilde{\xi}_i;\ i\in[\mu]][c]_f$

(this equality holds, because $\tilde{\delta}'(\phi)$ contains no $y_{i,q}$)

$= \tilde{\delta}(\phi[y_{\alpha i} \leftarrow \xi_i;\ i\in[\mu]][c]_f)(m_2(\mathrm{push}(g'(\zeta_1)))(g(t)), \ldots, m_2(\mathrm{push}(g'(\zeta_\nu)))(g(t)))$

(this equation holds by $(*)$)

$= \tilde{\delta}(m'(\mathrm{push}(\phi))(\xi))(g(m_1(\mathrm{expand}(\zeta_1))(t)), \ldots, g(m_1(\mathrm{expand}(\zeta_\nu))(\mathrm{t})))$

(by induction hypothesis)

$= \tilde{\delta}(m'(\mathrm{push}(\phi))(\xi))(g(\zeta_1[\,]_3), \ldots, g(\zeta_\nu[\,]_3))$,

where $[\,]_3 = [z_j \leftarrow t_j;\ j\in[k]][\mathrm{push}(\psi) \leftarrow m'(\mathrm{push}(\psi))(\xi);\ \mathrm{push}(\psi)\in F']$

$= g(\delta\langle m'(\mathrm{push}(\phi))(\xi)\rangle(\zeta_1[\,]_3, \ldots, \zeta_\nu[\,]_3))$

$= g(m_1(\mathrm{expand}(\zeta))(t))$. □

6.10. Example. Here we illustrate the coding of TP(2-AT(S_0)) by 3-AT(S_0). Let $\delta_1: ((q, q)(q, q), (q, q))$, $\delta_2, \delta_3: (q, q)$, and $\delta_4: q$ be elements of Ξ_f and let $\Omega_f = \{\gamma_2, \gamma_0\}$ where γ_2 has rank 2, and γ_0 has rank 0. Consider the configuration $c_1 = \gamma_2\langle t_1\rangle(\gamma_0\langle t_2\rangle(), \gamma_0\langle t_3\rangle())$ of TP(2-AT(S_0)) with $t_1 = \delta_1(\delta_2, \delta_3)(\delta_4)$ and $t_2 = t_3 = \delta_4$. For the sake of convenience we have left out the configuration c of S_0. An an exercise we compute type $([\gamma_2, \delta_1]) = \mathrm{inc}(\gamma_2)\,(\mathrm{type}_\Xi(\delta_1))$ in detail:

$\mathrm{inc}(\gamma_2)(((q, q)(q, q), (q, q))) = (\mathrm{inc}[(q, q)(q, q)], (\mathrm{inc}[q], (qq, q)))$

$= (\mathrm{inc}_2\, \mathrm{inc}_0\, \mathrm{inc}_2\, \mathrm{inc}_0, (\mathrm{inc}(\gamma_2)(q)\ \mathrm{inc}(\gamma_0)(q), (qq, q)))$

with $\mathrm{inc}_i = \mathrm{inc}(\gamma_i)((q, q))$ for $i\in\{0, 2\}$

$= (\tau_2\tau_0\tau_2\tau_0, ((qq, q)(\lambda, q), (qq, q)))$

with $\tau_i = ((qq, q)(\lambda, q), (q^i, q))$ for $i\in\{0, 2\}$.

For the computation of $g(c_1)$ we also precompute, for $i\in\{0, 2\}$, $\mathrm{inc}(\gamma_i)((q, q)) = (\mathrm{inc}[q], (q^i, q)) = ((qq, q)(\lambda, q), (q^i, q))$ and $\mathrm{inc}(\gamma_i)(q) = (q^i, q)$.

Hence, for $i\in\{0, 2\}$, $[\gamma_i, \delta_2]$, $[\gamma_i, \delta_3]: ((qq, q)(\lambda, q), (q^i, q))$ and $[\gamma_i, \delta_4]: (q^i, q)$.

Then $g(c_1) = \tilde{\gamma}_2(t_1)(g(\gamma_0\langle t_2\rangle())$,
$g(\gamma_0\langle t_3\rangle())) = [\gamma_2, \delta_1](t_{2,2}, t_{0,2}, t_{2,3}, t_{0,3})\ (t_{2,4}, t_{0,4})\ (t_{0,4}(), t_{0,4}())$ with $t_{i,j} = [\gamma_i, \delta_j]$ for $i\in\{0, 2\}$ and $j\in\{2, 3, 4\}$.

Now we apply the instruction $\phi = \mathrm{expand}(\gamma_2\langle\phi_1\rangle(\gamma_0\langle\phi_1\rangle(), z_2)$ to c_1, where $\phi_1 = \mathrm{push}(y_{2,(q,q)}(y_{1,q}))$. (The identity of S_0 is dropped.) This yields the configuration $c'_1 = \gamma_2\langle\delta_3(\delta_4)\rangle(\gamma_0\langle\delta_3(\delta_4)\rangle(), \gamma_0\langle\delta_4\rangle())$. The instruction ϕ is coded by $\phi' = \mathrm{push}(y_{3,\tau}(y_{1,(qq,q)}, y_{2,(\lambda,q)})(y_{4,\tau'}(y_{1,(qq,q)}, y_{2,(\lambda,q)})(), y_{2,q}))$ with $\tau = \mathrm{inc}(\gamma_2)((q, q)) = ((qq, q)(\lambda, q), (\lambda, q))$ and $\tau' = \mathrm{inc}(\gamma_0)((q, q)) = ((qq, q)(\lambda, q), (\lambda, q))$. If we apply ϕ' to $g(c_1)$, then we obtain the configuration $c'_2 = t'_2(t'_0, [\gamma_0, \delta_4]())$, where for

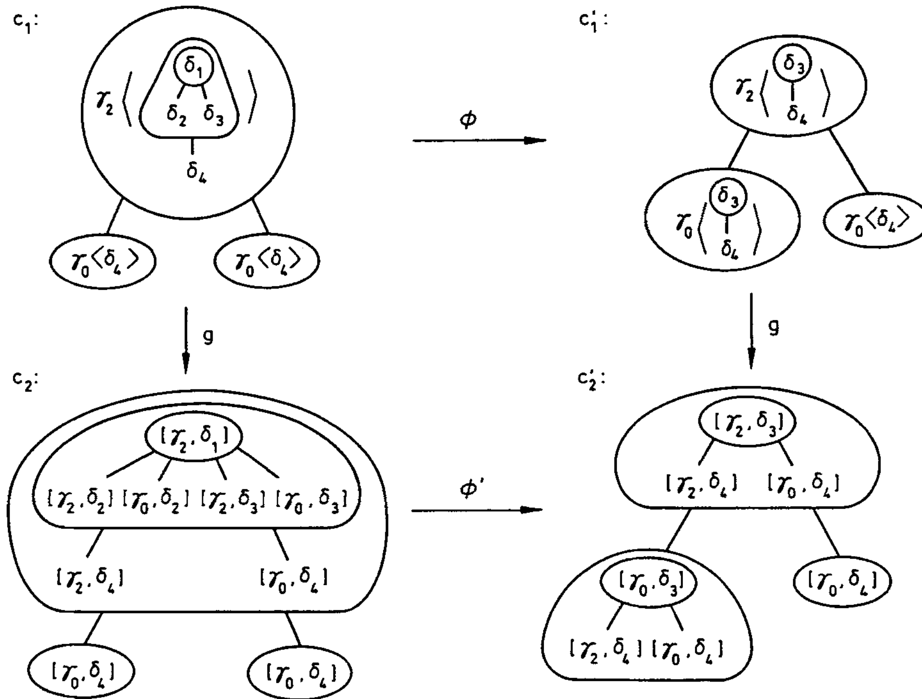

Fig. 10. Coding of TP(2-AT(S_0)) by 3-AT(S_0)

$i \in \{0, 2\}$, $t_i' = [\gamma_i, \delta_3]([\gamma_2, \delta_4](), [\gamma_0, \delta_4]())$. It is easy to compute that $g(c_1') = c_2'$. In Fig. 10 the corresponding commutative diagram is shown. □

The previous two lemmata prove the inductive statement.

6.11. Lemma. *For every* $n \geqq 1$, $(n+1)$-AT$(S) \equiv$ TP$(n$-AT$(S))$.

From this statement the equivalence of n-level applicative terms of S and n-iterated tree-pushdown of S follows immediately.

6.12. Corollary. *For every* $n \geqq 1$, n-AT$(S) \equiv$ TP$^n(S)$.

Proof. The proof is by induction on n. For $n = 1$, the equivalence follows from Lemma 6.5. Assume that n-AT$(S) \equiv$ TP$^n(S)$ holds. By monotonicity of TP (Lemma 6.2) it follows that TP$(n$-AT$(S)) \equiv$ TP$^{n+1}(S)$. Since $(n+1)$-AT$(S) \equiv$ TP$(n$-AT$(S))$ (Lemma 6.11), it follows that $(n+1)$-AT$(S) \equiv$ TP$^{n+1}(S)$. □

As a consequence of this storage type equivalence and the Justification Theorem, we obtain the characterization of n-$T(S)$ in terms of 0-$T(n$-AT$(S))$-transducers.

6.13. Theorem. *For every* $n \geqq 1$, n-$T(S) = 0$-$T($TP$^n(S))$ *and* $D_t n$-$T(S) = D_t 0$-$T($TP$^n(S))$.

Proof. By Theorem 5.13 and Corollary 6.6, for every $n \geqq 1$, n-$T(S) = 0$-$T(n$-AT$(S))$. By Corollary 6.12 and Theorem 3.16, for every $n \geqq 1$, n-$T(S) = 0$-$T($TP$^n(S))$ is proved. Since total determinism is preserved in the mentioned theorems, also the second equation is obtained. □

In Theorem 5.13 of [EV] the equivalence of TP(S) and $P_{\mathrm{bex}}(S)$ for arbitrary S is proved. From this equivalence and the monotonicity of P_{bex} (Lemma 3.23), the equivalence of n-iterated tree-pushdown of S and n-iterated bounded excursion pushdown of S easily follows. The second statement of the next corollary follows similarly from the monotonicity of P (Theorem 4.22 of [EV]).

6.14. Corollary. *For every* $n \geqq 1$, $\mathrm{TP}^n(S) \equiv P^n_{\mathrm{bex}}(S)$. *If S contains an identity, then* $\mathrm{TP}^n(S) \equiv P^n(S)$.

Finally, we obtain our main general characterization result: the equivalence of n-level S transducers and n-iterated pushdown S transducers.

6.15. Theorem. *For every* $n \geqq 1$, $n\text{-}T(S) = 0\text{-}T(P^n_{\mathrm{bex}}(S))$ *and* $D_t n\text{-}T(S) = D_t 0\text{-}T(P^n_{\mathrm{bex}}(S))$. *If S contains an identity, then* "bex" *can be dropped.*

Proof. Follows immediately from Theorem 6.13, Corollary 6.14, and Theorem 3.16. □

Note that S_0 has an identity. Thus, for $S = S_0$, we reobtain the result of [DamGue2] that $n\text{-}T = 0\text{-}T(P^n)$, formulated in a different way (cf. the introduction). For the monadic case, we also reobtain the characterization of n-level string grammars by 1-way P^n automata [DamGoe]. This is explained further in Sect. 8.

The present section is closed with an easy consequence of the previous theorem: the Justification Theorem for n-level S transducers.

6.16. Theorem. *Let S_1 and S_2 be storage types. For every $n \geqq 0$, if $S_1 \leqq S_2$, then* $n\text{-}T(S_1) \subseteq n\text{-}T(S_2)$ *and* $D_t\, n\text{-}T(S_1) \subseteq D_t\, n\text{-}T(S_2)$.

Proof. Assume that $S_1 \leqq S_2$. Then, by monotonicity of P_{bex} (Lemma 3.23), for every $n \geqq 0$, $P^n_{\mathrm{bex}}(S_1) \leqq P^n_{\mathrm{bex}}(S_2)$ is proved by induction on n. By Theorem 3.16, $0\text{-}T(P^n_{\mathrm{bex}}(S_1)) \subseteq 0\text{-}T(P^n_{\mathrm{bex}}(S_2))$ follows, and $n\text{-}T(S_1) \subseteq n\text{-}T(S_2)$ is proved by Theorem 6.15. Since the mentioned theorems preserve determinism and totality, the second statement of the present theorem is obtained. □

7. Characterization of High Level Tree Transducers

After having developed in the previous two sections the characterization of $n\text{-}T(S)$ by $0\text{-}T(P^n_{\mathrm{bex}}(S))$ for arbitrary storage S, we can now state the main result of this paper concerning the nondeterministic case: the characterization of n-level tree transducers by means of n-iterated pushdown tree transducers.

7.1. Theorem. *For every* $n \geqq 1$, $n\text{-}T(\mathrm{TR}) = 0\text{-}T(P^n_{\mathrm{bex}}(\mathrm{TR}))$.

Proof. Immediately from Theorem 6.15. □

Since we are mostly interested in the total deterministic case, we spend in the rest of this section some more effort at obtaining a "cleaner" characterization of $D_t\, n\text{-}T(\mathrm{TR})$ in the sense that the restriction on the iterated pushdown (namely to be bounded excursion) can be dropped (cf. Theorem 7.12). Note that this does not follow from Theorem 6.15, because TR does not have an identity.

The proof of this characterization involves a number of results of [EV], but also some new ones. In the following we derive (by way of discussion) the proof-tree of Theorem 7.12 in a "top-down" manner. This motivates the

new results and thus helps the reader to understand the subsequent formal "bottom-up" treatment, in which the new results are proved and combined with the needed results of [EV]. In particular, this procedure illucidates the reason why we have to study the restriction of $P(S)$ to be bounded excursion in order to get rid of it.

For the proof of Theorem 7.12, we show the equality of $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR}))$ and $D_t\,0\text{-}T(P^n(\text{TR}))$ (cf. Lemma 7.11; for $n=1$, cf. Lemma 5.15 of [EV]). To obtain this equality, we first characterize $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR}))$ by the n-fold composition of the total deterministic macro tree transducer (cf. Lemma 7.10), i.e., by $D_t1\text{-}T(\text{TR})^n$, and then use the fact that the latter class coincides with $D_t\,0\text{-}T(P^n(\text{TR}))$, which is proved in Theorem 8.12 of [EV].

The characterization of $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR}))$ by $D_t1\text{-}T(\text{TR})^n$ follows by straightforward induction from the statement: if S is closed under look-ahead, then $D_t\,0\text{-}T(P_{\text{bex}}(S))=D_t\,0\text{-}T(S)\circ D_t1\text{-}T(\text{TR})$ (cf. Lemma 7.9; see also Corollary 8.11 of [EV]), and from the fact that, for $n\geqq 1$, $P^n_{\text{bex}}(\text{TR})$ is closed under look-ahead (cf. Lemma 7.8). Since Lemma 7.9 is immediate from results of [EV], the only thing we have to prove is Lemma 7.8.

The concept of look-ahead for storage types is introduced in [Eng5] and is formalized as an operator on storage types, i.e., if S is a storage type, then "S with look-ahead" is one too. This enriched storage type contains predicates, so called look-ahead tests, by which properties of "successors" $m(f_n)(\ldots m(f_2)(m(f_1)(c))\ldots)$ (for some instructions $f_1, f_2, \ldots, f_n$ of S) of an S-configuration c can be tested without changing c. We recall the formal definition; cf. also Definition 6.5 of [EV].

7.2. Definition. Let $S=(C, P, F, m, I, E)$ be a storage type. *S with look-ahead*, denoted by S_{LA}, is the storage type (C, P', F, m', I, E), where m' restricted to $P\cup F$ is equal to m, $P'=P\cup\{\langle A, H\rangle \mid H$ is a CF(S)-transducer and A is a nonterminal of $H\}$ and for every $c\in C$, $m'(\langle A, H\rangle)(c)=\text{true}$ iff there is a $w\in\Sigma^*$ such that $A\langle c\rangle \overset{H}{\Longrightarrow}{}^* w$, where Σ is the terminal alphabet of H. $\square$

A predicate like $\langle A, H\rangle$ of S_{LA} is called a *look-ahead test* and H is called a *look-ahead transducer.*

The notion "S is closed under look-ahead" means that S can handle its look-ahead tests, or precisely, that S_{LA} is equivalent to S, i.e., $S_{\text{LA}}\equiv S$ (note that $S\leqq S_{\text{LA}}$ is trivial). Hence, continuing to enlarge the proof-tree of Theorem 7.12, we have to show that for every $n\geqq 1$, $P^n_{\text{bex}}(\text{TR})_{\text{LA}}\equiv P^n_{\text{bex}}(\text{TR})$ (cf. Lemma 7.8). For P rather than P_{bex}, the corresponding result has been shown in [EV] and actually, the proof of Lemma 7.8 follows the same lines. It consists of two parts. First, we prove that $P_{\text{bex}}(S)_{\text{LA}}\leqq P_{\text{bex}}(S_{\text{LA}})$ (cf. Lemma 7.6) and second, that $P_{\text{bex}}(\text{TR}_{\text{LA}})$ is equivalent to $P_{\text{bex}}(\text{TR})$ (cf. Lemma 7.7). Then the "crucial direction " of Lemma 7.8 is proved by induction on n by using these two lemmata and the monotonicity of P_{bex} (cf. Lemma 3.23).

Now we have reached the leaves of the proof-tree of Theorem 7.12 (viz. Lemma 7.6, Lemma 7.7, and Lemma 7.9) and we can move back to its root, by first proving its leaves. We start with Lemma 7.6.

In the proof of $P_{\text{bex}}(S)_{\text{LA}}\leqq P_{\text{bex}}(S_{\text{LA}})$ a comparison of $\text{dom}(\text{CF}(P_{\text{bex}}(S)))$ and $\text{dom}(\text{CF}(S))$ is involved, because a look-ahead test $\langle A, H\rangle$, where H is a

$CF(P_{bex}(S))$-transducer, should be simulated by a $P_{bex}(S_{LA})$-flowchart in which look-ahead tests of the form $\langle B, H'\rangle$ are allowed, where H' is a $CF(S)$-transducer. For our purpose it suffices to show that $dom(CF(P_{bex}(S))) \subseteq dom(CF(S))$, although we could even prove equality.

7.3. Lemma. $dom(CF(P_{bex}(S))) \subseteq dom(CF(S))$.

Proof. Since $P_{bex}(S) \leqq P(S)$ (Lemma 3.22), we obtain from the Justification Theorem 3.16, $dom(CF(P_{bex}(S))) \subseteq dom(CF(P(S)))$. Then the statement of this lemma follows from Lemma 6.11 of [EV], which shows: $dom(CF(P(S))) = dom(CF(S))$. □

Rather than using the previous lemma immediately in the proof of $P_{bex}(S)_{LA} \leqq P_{bex}(S_{LA})$, we simplify this proof by allowing the look-ahead transducers of the simulating storage type to be $CF(P_{bex}(S))$-transducers (instead of only $CF(S)$-transducers). This gives rise to a modification of the look-ahead on storage types (cf. also Definition 6.12 of [EV], in which the indexed look-ahead of S is defined).

7.4. Definition. Let $S_{LA} = (C, P, F, m, I, E)$. The *bounded indexed look-ahead of* S, denoted by $S_{bi\text{-}LA}$ is the storage type (C, P', F, m', I, E), where m' restricted to $P \cup F$ is equal to m, $P' = P \cup \{\langle A, \gamma, v, mx, H\rangle \mid H$ is a $CF(P_{bex}(S))$-transducer, A is a nonterminal of H, $\gamma \in \Gamma$, $mx \geqq 0$, and $0 \leqq v \leqq mx\}$, and for every $c \in C$, $m(\langle A, \gamma, v, mx, H\rangle)(c) = \text{true}$ iff there is a $w \in \Sigma^*$ such that $A\langle(\gamma, c, v, mx)\rangle \overset{(H)}{\Longrightarrow}{}^* w$, where Σ is the terminal alphabet of H. □

Allowing bounded indexed look-ahead tests instead of usual look-ahead tests does not increase the power of the storage type. In the proof of this fact, Lemma 7.3 plays the essential role. We note that the proof is a modification of the proof of Lemma 6.13 of [EV].

7.5. Lemma. $S_{bi-LA} \equiv_c S_{LA}$.

Proof. Since S_{bi-La} is defined as an "enrichment" of S_{LA}, we only have to prove $S_{bi\text{-}LA} \leqq_c S_{LA}$. Let U be a finite restriction of $S_{bi\text{-}LA}$ and let m_1 and m_2 be the meaning functions of $S_{bi\text{-}LA}$ and of S_{LA}, respectively. We show that $U \leqq_{dc} S_{LA}$ with the identity on $C' = (\Gamma \times C \times \text{nat} \times \text{nat})^+$ as representation function h. Clearly, we only have to code a bounded indexed look-ahead test $\langle A_1, \gamma, v, mx, H_1\rangle$ of U by a boolean expression b over predicates of S_{LA}. In the sequel we construct one look-ahead test of S_{LA} which already serves this purpose.

Let $H_1 = (N, e_1, \Sigma_1, -, R_1)$ and let $C_1 = \{c \in C \mid m_1(\langle A_1, \gamma, v, mx, H_1\rangle)(c) = \text{true}\}$. We transform C_1 into the domain of a $CF(P_{bex}(S'))$-transducer H'_1 by defining $S' = (C, P, F, m, C, \{id_C\})$ and $H'_1 = (N'_1, e'_1, \Sigma_1, *_1, R'_1)$, where id_C is the identity on C, $N'_1 = N_1 \cup \{*_i \mid i \in [v]\}$ and the $*_i$ are new nonterminals, $e'_1 = \lambda u \in C \cdot (\gamma, id_C(u), 0, mx)$, and $R'_1 = R_1 \cup \{*_i \to *_{i+1}\langle \text{stay}\rangle \mid i \in [v\text{-}1]\} \cup \{*_v \to A_1\langle \text{stay}\rangle\}$. Since (by definition) H'_1 starts with the excursion counter equal to 0, the extra rules $*_i \to *_{i+1}\langle \text{stay}\rangle$ are used to increase this counter upto v, before the derivation of H_1 is started. Now it is easy to see that $C_1 = dom(\tau(H'_1))$.

Hence, by Lemma 7.3, there is a $CF(S')$-transducer H'_2 such that $dom(\tau(H'_1)) = dom(\tau(H'_2))$. Let $H'_2 = (N_2, id_C, \Sigma_2, A_2, R_2)$. But obviously,

$\text{dom}(\tau(H_2'))=\{c\in C \mid \text{there is a } w\in\Sigma_2^* \text{ such that } A_2\langle c\rangle \Rightarrow_{H_2}^* w\}$ where H_2 is the CF(S)-transducer $(N_2, -, \Sigma_2, -, R_2)$. Since $\{c\in C \mid \text{there is a } w\in\Sigma_2^* \text{ such that } A_2\langle c\rangle \Rightarrow_{H_2}^* w\}=\{c\in C \mid m_2(\langle A_2, H_2\rangle)(c)=\text{true}\}$, it is clear that $\langle A_2, H_2\rangle$ is the desired boolean expression b, or in other words, that the look-ahead test $\langle A_2, H_2\rangle$ codes the bounded indexed look-ahead test $\langle A_1, \gamma, v, mx, H_1\rangle$. □

The proof of $P_{\text{bex}}(S)_{\text{LA}} \leqq P_{\text{bex}}(S_{\text{LA}})$ is very similar to the proof of $P(S)_{\text{LA}} \leqq P(S_{\text{LA}})$ (Theorem 6.14 of [EV]). We briefly recall the idea of the construction. Actually, since $S_{\text{LA}} \equiv S_{\text{bi-LA}}$ and since P_{bex} is monotonic, we only have to show how $P_{\text{bex}}(S)_{\text{LA}}$ is simulated by $P_{\text{bex}}(S_{\text{bi-LA}})$. This amounts to simulate a look-ahead test $\langle A, H\rangle$ on $P_{\text{bex}}(S)$, where H is a CF($P_{\text{bex}}(S)$)-transducer, by a $P_{\text{bex}}(S_{\text{bi-LA}})$-flowchart ω for predicates. For this purpose a computation of H, which starts from $A\langle c'\rangle$ for some $P_{\text{bex}}(S)$-configuration $c'=(\gamma, c, v, mx)\beta$, is split into two parts. In the first part, only those steps of the computation are considered, which do not test the rest-pushdown β. Roughly speaking, this part is simulated by the bounded indexed look-ahead test $\langle A, \gamma, v, mx, H'\rangle$ on S, where H' is just a small modification of H. In the second part, the computations on the rest-pushdown β are collected; they start from $A_i\langle\beta\rangle$ for some nonterminal A_i of H. The flowchart ω has to check now, whether $A_i\langle\beta\rangle$ can derive a terminal word. This information is captured in the so-called termination behaviour of β, which is written in the topmost pushdown square of the representation of c'. The termination behaviour of a pushdown β is a sequence of bits which tells for every nonterminal A_i of H whether $\langle A_i, H\rangle$ is true on β or not. Clearly, the termination behaviour of a pushdown β can be computed simultaneously with the growth of β in an inductive way.

7.6. Lemma. $P_{\text{bex}}(S)_{\text{LA}} \leqq P_{\text{bex}}(S_{\text{LA}})$.

Proof. By Lemma 7.5 and monotonicity of P_{bex} (Lemma 3.23), it suffices to prove that $P_{\text{bex}}(S)_{\text{LA}} \leqq P_{\text{bex}}(S_{\text{bi-LA}})$. Let U be a finite restriction of $P_{\text{bex}}(S)_{\text{LA}}$ with encoding $e_1=\lambda u\in I\cdot(\gamma, e(u), 0, mx)$ for some $\gamma\in\Gamma$, encoding e of S, and $mx\geqq 0$.

Since, by Definition 7.2, the excursion bound of a CF($P_{\text{bex}}(S)$)-transducer H is irrelevant for the result of a look-ahead test $\langle A, H\rangle$, where A is a nonterminal of H, we can assume that e_1 is the encoding of every look-ahead transducer occurring in the look-ahead tests of U. Since U contains only finitely many look-ahead tests, we can take the disjoint union of the involved look-ahead transducers in the usual way, and thus we can assume that all the look-ahead tests of U share the same look-ahead transducer H.

In order to replace a look-ahead test $\langle A, H\rangle$ of U by the correct bounded indexed look-ahead test (cf. the discussion before this lemma), the value of the excursion counter of the topmost pushdown square of the currrent $P_{\text{bex}}(S)$-configuration has to be known. We store this information, for every pushdown square, in the corresponding pushdown symbol. Whenever the "real" excursion counter is incremented, also its "copy" is updated.

Now the constructions of the representation function and of the simulating flowcharts proceed in the same way as in Theorem 6.14 of [EV]. Since, roughly speaking, every instruction is simulated by an instruction of the same type,

the property of U of being "bounded excursion" is preserved by this construction. The formal definitions of the representation function and of the simulating flowcharts is left as an exercise to the reader. □

The second leaf of the above mentioned proof-tree of Theorem 7.12 is the equivalence of $P_{bex}(\mathrm{TR}_{LA})$ and $P_{bex}(\mathrm{TR})$. Of course, it suffices to show that $P_{bex}(\mathrm{TR}_{LA})$ can be simulated by $P_{bex}(\mathrm{TR})$. Hence, every predicate test$(\langle A, H\rangle)$, where H is a CF(TR)-transducer, must be simulated by a $P_{bex}(\mathrm{TR})$-flowchart ω for predicates. What does this intuitively mean? In the particular case that $S=\mathrm{TR}$, the definition of S_{LA} (Definition 7.2) is equivalent to the definition of regular look-ahead [Eng2] (cf. remark after Definition 6.5 of [EV]). Hence, a look-ahead test of TR_{LA} checks, whether a tree is in a regular tree language or not. Now recall from Definition 3.9 that ω is a deterministic REG$(P_{bex}(\mathrm{TR})_{id})$-transducer. Since REG$(P(\mathrm{TR}))$-transducers are very close to the checking-tree pushdown transducers of [EngRozSlu] (cf. the discussion of $P(\mathrm{TR})$ in Sect. 3.3 of [EV]), we could call ω a "checking-tree bounded excursion pushdown transducer". Then $P_{bex}(\mathrm{TR}_{LA}) \leqq P_{bex}(\mathrm{TR})$ can be reformulated as follows: deterministic checking-tree bounded excursion pushdown transducers are closed under regular look-ahead. Actually, for the unbounded version of these transducers, this closure property has been shown in Theorem 4.7 of [EngRozSlu], and in fact, as a first approximation, we can take over the proof. However, the bound on the number of excursions in the pushdown requires a special treatment. In the sequel we explain our construction informally.

For every predicate test$(\langle A, H\rangle)$ (with look-ahead test $\langle A, H\rangle$) of a finite restriction U of $P_{bex}(\mathrm{TR}_{LA})$, the set R of trees on which $\langle A, H\rangle$ is true is a regular tree language. Thus, we can associate with every look-ahead test $\langle A, H\rangle$ a total deterministic bottom-up finite tree automaton $B=(Q, \Sigma, \delta, F)$ such that $L(B)=R$. Then the predicate test$(\langle A, H\rangle)$ can be simulated by the $P_{bex}(\mathrm{TR})$-flowchart ω_{test}, where ω_{test} imitates the automaton B on a (sub)tree by a depth-first tree-walk on t: first ω_{test} marks the topmost pushdown square s and runs down into the tree. Then it simulates B in a bottom-up fashion until the state $q=\delta(t)$ is computed. If $q\in F$, then $t\in R$, i.e., $\langle A, H\rangle$ is true on t. Up to now, this is a repetition of the construction used in [EngRozSlu]. But now we have to deal with the problem that the excursion counter of the pushdown square s is increased by ω_{test}, whereas the application of test$(\langle A, H\rangle)$ does not increase the excursion counter. We solve this problem as follows (cf. the proof of the monotonicity of P_{bex} in Lemma 3.23): whenever a new square s with some (sub)tree t is pushed, the result of every look-ahead test on t is computed and put into square s. (Note that, since we consider a finite restriction U of $P_{bex}(\mathrm{TR}_{LA})$, there are only finitely many predicates of the form test$(\langle A, H\rangle)$.) Then the simulation of test$(\langle A, H\rangle)$ becomes even simpler: the result of this predicate can be tested from the pushdown symbol of s.

7.7. Lemma. $P_{bex}(\mathrm{TR}_{LA}) \equiv P_{bex}(\mathrm{TR})$.

Proof. It suffices to prove $P_{bex}(\mathrm{TR}_{LA}) \leqq P_{bex}(\mathrm{TR})$. Let U be a finite restriction of $P_{bex}(\mathrm{TR}_{LA})$ with encoding $e_1 = \lambda t \in T_\Omega \cdot (\gamma, e(t), 0, mx)$ and $e\colon T_\Omega \to T_\Sigma$ is the identity on Σ for some ranked alphabet Σ. Let Γ_f be the finite set of pushdown

symbols occurring in U. Let $\langle A_1, H_1\rangle, \ldots, \langle A_r, H_r\rangle$ be the look-ahead tests involved in predicates of U with $r \geqq 0$. If $r=0$, then the simulation is trivial. Assume that $r \geqq 1$. For every $i \in [r]$, the set $R_i = \{t \in T_\Sigma \mid m(\langle A_i, H_i\rangle)(t) = \text{true}\}$ equals $\text{dom}(\tau(H_i'))$, where m is the meaning function of TR_{LA} and the CF(TR)-transducer H_i' is the same as H_i, except that A_i is the initial nonterminal. Since the domain of a CF(TR)-transducer, which is the domain of a top-down tree transducer (cf. Fact 4.6 and Corollary 3.20 of [EV]), is a regular tree language [Rou, Tha, Engl], there is a total deterministic bottom-up tree automaton $B_i = (Q_i, \Sigma, \delta_i, F_i)$ such that $L(B_i) = R_i$.

Now it is easy to construct a P(TR)-flowchart ω_{test} (for instructions) such that for every $\gamma \in \Gamma$ and $t \in T_\Sigma$, $\text{oper}(\omega_{\text{test}})((\gamma, t)) = ([\gamma, \tilde{\theta}], t)$ where $\tilde{\theta} \in \{0,1\}^r$ and for every $i \in [r]$: $\tilde{\theta}(i) = 1$ iff $t \in R_i$; ω_{test} computes for every subtree t' of t the state sequence $(q_1, \ldots, q_r)$ in which for every $i \in [r]$, $q_i = \delta_i(t')$. We leave the formal construction to the reader.

It is obvious that ω_{test} can also be considered as a P_{bex}(TR)-flowchart, because the number of excursions taken from every pushdown square is bounded by $mx_{\text{test}} = 2 \cdot \max \Sigma + 1$, where $\max \Sigma$ is the maximal rank of a symbol occurring in Σ: computing the state sequence $(q_1, \ldots, q_r)$ of a direct subtree t'' of any subtree t' of t and putting it into the square at the root of t' takes one excursion each (the latter one is a trivial stay (δ) excursion). Since there are at most $\max \Sigma$ (direct) subtrees of t', this motivates the term $2 \cdot \max \Sigma$. This bound holds for every square except the one which is associated with the root of t. For entering the sequence of look-ahead test results $\tilde{\theta}$ into this square another (trivial) excursion has to be made. Hence, $\text{oper}(\omega_{\text{test}})((\gamma, t, 0, mx_{\text{test}})) = ([\gamma, \tilde{\theta}], t, v', mx_{\text{test}})$, where $v' = 2 \cdot \text{rank}(\sigma) + 1$ and σ is the root of t.

This discussion induces that we have to take $mx' = mx + mx_{\text{test}}$ as bound for the simulating storage type P_{bex}(TR). But now it is possible that the extra excursions $mx_{\text{test}} - v'$ are misused in the following sense: the simulation of, e.g., a push instruction is possible, whereas the application of the push instruction itself to the represented configuration is undefined, because the excursion counter has reached already mx. In order to avoid this misuse, we make $n = mx' - v'$ dummy excursions via a sequence of stay instructions. This is formalized in the P_{bex}(TR)-flowchart ω_+ which tests the rank of the root of the tree, that is contained in the topmost pushdown square, and then applies the appropriate number of stay instructions.

For the definition of the representation function $h: C' \to C'$ with $C' = (\Gamma \times C \times \text{nat} \times \text{nat})^+$ we define the set $\tilde{\Gamma} = \{[\gamma, \tilde{\theta}] \mid \gamma \in \Gamma_f \text{ and } \tilde{\theta} \in \{0,1\}^r\}$. We can assume w.l.o.g. that $\tilde{\Gamma} \subseteq \Gamma$. Then h is defined as follows.

(i) For every $\gamma \in \Gamma_f$, $t \in T_\Sigma$, $c' = (\gamma, t, 0, mx') \in \text{dom}(h)$ and $h(c') = (\gamma, t, 0, mx)$.

(ii) For every $[\gamma, \tilde{\theta}] \in \tilde{\Gamma}$, $t \in T_\Sigma$ such that, for every $i \in [r]$, $\tilde{\theta}(i) = 1$ iff $t \in R_i$; for every v with $0 \leqq v \leqq mx$, and $\beta \in \text{dom}(h)$, $c' = ([\gamma, \tilde{\theta}], t, v + mx_{\text{test}}, mx')\beta \in \text{dom}(h)$ and $h(c') = (\gamma, t, v, mx)\, h(\beta)$.

Actually, the situation expressed in case (i) occurs only, because the result of the encoding e_2, which simulates e_1, does not contain the appropriate look-ahead sequence, but it has to be an element of the domain of h (cf. Requirement 1.1.2 of Definition 3.13). This means that, whenever a predicate or an instruction ϕ is simulated on the result of the encoding e_2, then first, the sequence of

look-ahead tests is computed and second, ϕ is simulated. In order to recognize this situation we abbreviate the boolean expression top$=\gamma_1$ **or**...**or** top$=\gamma_k$, where $\Gamma_f=\{\gamma_1, \ldots, \gamma_k\}$, by "initial".

In the following we prove Requirements 1–3. We describe the desired flowcharts as PASCAL-like programs rather than defining them formally. Also, we consider ω_{test} and ω_+ as blocks of statements rather than as rules of a REG(P_{bex}(TR))-transducer.

Requirement 1. The encoding $e_2=\lambda t\in T_\Omega\cdot(\gamma, e(t), 0, mx')$ satisfies the requirements.

Requirement 2. Every predicate ϕ of U is simulated by the following P_{bex}(TR)-flowchart for predicates.

begin
  if initial **then begin** ω_{test}; ω_+ **end**;
  if b_ϕ **then** true⟨id⟩ **else** false⟨id⟩
end

where the boolean expression b_ϕ depends on ϕ.

$\phi=(\text{top}=\delta)$: b_ϕ is the disjunction of all predicates top$=[\delta, \vec{\theta}]$ for $\vec{\theta}\in\{0, 1\}^r$.
$\phi=\text{test}(\text{root}=\sigma)$: $b_\phi=\text{test}(\text{root}=\sigma)$.
$\phi=\text{test}(\langle A_i, H_i\rangle)$: b_ϕ is the disjunction of all predicates top$=[\gamma, \vec{\theta}]$ such that the i-th component of $\vec{\theta}$ is 1.

Requirement 3. Every instruction ϕ of U is simulated by the following P_{bex}(TR)-flowchart.

begin if initial **then begin** ω_{test}; ω_+ **end**;
  ω_ϕ
end

where the block ω_ϕ of statements depends on ϕ.

$\phi=\text{push}(\delta, \text{sel}_i)$: **begin** push($\delta$, sel$_i$); ω_{test}; ω_+ **end**.
$\phi=\text{pop}$: **begin** pop **end**
$\phi=\text{stay}(\delta)$: **begin** for every $[\gamma, \vec{\theta}]\in\tilde{\Gamma}$:
  if top$=[\gamma, \vec{\theta}]$ **then** stay($[\delta, \vec{\theta}]$)
  end
$\phi=\text{stay}$: **begin** stay **end** □

Now we can show the closure of P^n_{bex}(TR) under look-ahead.

7.8. Lemma. *For every* $n\geqq 1$, $P^n_{\text{bex}}(\text{TR})_{\text{LA}}\equiv P^n_{\text{bex}}(\text{TR})$.

Proof. Since $P^n_{\text{bex}}(\text{TR})\leqq P^n_{\text{bex}}(\text{TR})_{\text{LA}}$ is trivial, it suffices to prove $P^n_{\text{bex}}(\text{TR})_{\text{LA}}\leqq P^n_{\text{bex}}(\text{TR})$. The proof is by induction on n. For $n=1$, the statement follows immediately from Lemma 7.6, Lemma 7.7, and the transitivity of $\leqq$ (Theorem 4.20 of [EV]). Assume that the statement holds for n, i.e., (*) $P^n_{\text{bex}}(\text{TR})_{\text{LA}}\leqq P^n_{\text{bex}}(\text{TR})$. Then $P^{n+1}_{\text{bex}}(\text{TR})_{\text{LA}}=P_{\text{bex}}(P^n_{\text{bex}}(\text{TR}))_{\text{LA}}$ (by definition)

$\leqq P_{\text{bex}}(P^n_{\text{bex}}(\text{TR})_{\text{LA}})$ (by Lemma 7.6) $\leqq P_{\text{bex}}(P^n_{\text{bex}}(\text{TR}))$ (by (*) and monotonicity of P_{bex}, cf. Lemma 3.23). □

The last leaf of the proof-tree of Theorem 7.12 is a slight modification of Corollary 8.11 of [EV].

7.9. Lemma. *If* $S_{\text{LA}} \equiv S$, *then* $D_t\,0\text{-}T(P_{\text{bex}}(S)) = D_t\,0\text{-}T(S) \circ D_t\,1\text{-}T(\text{TR})$.

Proof. Assume that $S_{\text{LA}} \equiv S$. Then by Corollary 8.11 of [EV]: $D_t\,\text{RT}(P(S)) = D_t\,\text{RT}(S) \circ D_t\,\text{CFT}(\text{TR})$. Since $D_t\,\text{RT}(P(S)) = D_t\,\text{RT}(P_{\text{bex}}(S))$ (Lemma 5.15 of [EV]) and since, for every storage type S', $D_t\,\text{RT}(S')$ and $D_t\,\text{CFT}(S')$ are $D_t\,0\text{-}T(S')$ and $D_t\,1\text{-}T(S')$, respectively (Fact 4.6), the statement of the lemma follows immediately. □

After having proved the lemmas at the leaves, we now can move up in the proof-tree following the discussion at the beginning of this chapter in the opposite direction. From the previous lemma and the fact that $P^n_{\text{bex}}(\text{TR})$ is closed under look-ahead, it follows that $D_t\,0 - T(P^n_{\text{bex}}(\text{TR}))$ is characterized by the n-fold composition of total deterministic macro tree transducers.

7.10. Lemma. *For every* $n \geqq 1$, $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR})) = D_t\,1\text{-}T(\text{TR})^n$.

Proof. The case $n = 1$ is proved in Theorem 6.15 (for $S = \text{TR}$). Assume that the statement holds for n. Then $D_t\,0\text{-}T(P^{n+1}_{\text{bex}}(\text{TR})) = D_t\,0\text{-}T(P_{\text{bex}}(P^n_{\text{bex}}(\text{TR})))$ (by definition) $= D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR})) \circ D_t\,1\text{-}T(\text{TR})$ (by Lemma 7.8 and Lemma 7.9) $= D_t\,1\text{-}T(\text{TR})^n \circ D_t\,1\text{-}T(\text{TR})$ (by induction hypothesis) $= D_t\,1\text{-}T(\text{TR})^{n+1}$. □

Since also the class $D_t\,\text{RT}(P^n(\text{TR}))$ is characterized by the n-fold composition of macro tree transducers (Theorem 8.12 of [EV]), we now know that the bound on the number of excursions in $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR}))$-transducers can be dropped.

7.11. Lemma. *For every* $n \geqq 1$, $D_t\,0\text{-}T(P^n_{\text{bex}}(\text{TR})) = D_t\,0\text{-}T(P^n(\text{TR}))$.

Proof. By Theorem 8.12 of [EV], $D_t\,\text{RT}(P^n(\text{TR})) = D_t\,\text{CFT}(\text{TR})^n$. By Fact 4.6 and Lemma 7.10 we obtain the desired result. □

Finally we have reached the root of the proof-tree. Since we are mainly interested in the characterization of total deterministic n-$T(\text{TR})$-transducers, the next theorem presents the main result of this paper.

7.12. Theorem. *For every* $n \geqq 1$, $D_t\,n\text{-}T(\text{TR}) = D_t\,0\text{-}T(P^n(\text{TR}))$.

Proof. Theorem 6.15 and Lemma 7.11. □

8. Some Consequences

Here we want to point out some more properties of n-level tree transducers, derivable from results of this paper.

A rather surprising consequence is the fact that, in the total deterministic case, n-level tree transducers are equivalent to the n-fold composition of macro

tree transducers, i.e., of 1-level tree transducers. In the nondeterministic case we only obtain inclusion. Recall that for a class K of relations, K^n denotes $\{R_1 \circ R_2 \circ \ldots \circ R_n \mid R_i \in K \text{ for all } i \in [n]\}$.

8.1. Theorem. *For every* $n \geqq 1$,

(a) $D_t\, n\text{-}T(\mathrm{TR}) = D_t\, 1\text{-}T(\mathrm{TR})^n$.
(b) $n\text{-}T(\mathrm{TR}) \subseteq 1\text{-}T(\mathrm{TR})^n$.

Proof. By Theorem 7.12, $D_t\, n\text{-}T(\mathrm{TR}) = D_t\, 0\text{-}T(P^n(\mathrm{TR}))$. Then (a) follows from Theorem 8.12 of [EV], which says that $D_t\, \mathrm{RT}(P^n(\mathrm{TR})) = D_t\, \mathrm{CFT}(\mathrm{TR})^n$, and Fact 4.6. To prove (b) we first have to know that $\mathrm{RT}(P^n_{\mathrm{bex}}(\mathrm{TR})) \subseteq \mathrm{CFT}(\mathrm{TR})^n$. This can be proved inductively using Theorem 5.14 of [EV] and Corollary 3.27 of [EV]. Then (b) follows from Theorem 7.1 and Fact 4.6. □

As an immediate consequence of 8.1(a) and the fact that total deterministic macro tree transducers capture the translational power of attribute grammars AG (viewed as tree transducers, [EngFil, ChiMar, CouFra]), which was shown in [Eng4], we obtain the equivalence of the concepts of high level tree transducers and the composition closure of attribute grammars, i.e., $\bigcup\{D_t\, n\text{-}T(\mathrm{TR}) \mid n \geqq 0\} = \bigcup\{\mathrm{AG}^n \mid n \geqq 1\}$.

The characterization of n-level tree transducers by n-iterated pushdown tree transducers (Theorem 7.1 and Theorem 7.12) is an OI-like result in the sense that the involved derivation relation is defined in the outside-in mode. In the total deterministic case, there is also an IO-like decomposition result for n-level tree transducers. This, on first sight, surprising statement is based on the fact that for total deterministic macro tree transducers there is no difference between the classes of translations induced by the three derivation modes outside-in, inside-out, and unrestricted (Theorem 4.1 of [EngVogl]). Then the IO-like decomposition result for $D_t\, n\text{-}T(\mathrm{TR})$ follows from Theorem 8.1(a) and the decomposition of total deterministic macro tree transducers ($D_t\, \mathrm{CFT}(\mathrm{TR}) = D_t\, \mathrm{RT}(\mathrm{TR}) \circ \mathrm{YIELD}$, e.g., Theorem 4.8 of [EngVogl]; cf. also [Eng4, CouFra]), which is an IO-like result. Note that YIELD is a class of substitution functions, which is defined in [Mai, EngSch, Dam] in an algebraical way and in [Eng3] in a more syntactical way. Recall that for two classes of relations K_1 and K_2, $K_1 \circ K_2$ denotes $\{R_1 \circ R_2 \mid R_1 \in K_1 \text{ and } R_2 \in K_2\}$.

8.2. Theorem. *For every* $n \geqq 0$, $D_t\, n\text{-}T(\mathrm{TR}) = D_t\, 0\text{-}T(\mathrm{TR}) \circ \mathrm{YIELD}^n$.

Proof. Immediate from Theorem 8.1(a), and Corollary 4.13 of [EngVogl]. □

Thus, as the culmination of the work in [EngVogl], [EV], and this paper, we have obtained the equivalence of the following five concepts (all total deterministic):

(1) high-level tree transducers
(2) iterated pushdown tree transducers
(3) compositions of macro tree transducers
(4) top-down tree transducers composed with YIELDs
(5) compositions of attribute grammars.

The combination of OI- and IO-like properties of $D_t\, n\text{-}T(S)$, for arbitrary S, is expressed in the following result which shows the connection between the pushdown operator and YIELD.

8.3. Theorem. *For every* $n \geqq 0$, *if* $S_{\mathrm{LA}} \equiv S$, *then* $D_t(n+1)\text{-}T(S) = D_t\, n\text{-}T(P(S)) = D_t\, n\text{-}T(S) \circ \mathrm{YIELD}$.

Proof. Let $S_{\mathrm{LA}} \equiv S$. First, it can be proved by induction that $P^n_{\mathrm{bex}}(S)$ is closed under look-ahead, i.e., that $P^n_{\mathrm{bex}}(S)_{\mathrm{LA}} \equiv P^n_{\mathrm{bex}}(S)$ by using the assumption $S_{\mathrm{LA}} \equiv S$, Lemma 7.6, and the monotonicity of P_{bex} (Lemma 3.23). In a similar way the closure of $P^n(S)$ under look-ahead can be proved by using the assumption, Theorem 6.14 of [EV] and the monotonicity of P (Theorem 4.22 of [EV]). Second, it can be proved again by induction that $D_t\, 0\text{-}T(P^n_{\mathrm{bex}}(S)) = D_t\, 0\text{-}T(P^n(S))$ by using the two related decompositions of Lemma 7.9 and Corollary 8.11 of [EV]. Together with Theorem 6.15 we obtain (*) $D_t\, n\text{-}T(S) = D_t\, 0\text{-}T(P^n(S))$. Then

$D_t(n+1)\text{-}T(S) = D_t\, n\text{-}T(P(S)) = D_t\, 0\text{-}T(P^{n+1}(S))$ (by (*))
$= D_t\, 0\text{-}T(P^n(S)) \circ D_t\, 1\text{-}T(\mathrm{TR})$ (by Corollary 8.11 of [EV] and Fact 4.6)
$= D_t\, 0\text{-}T(P^n(S)) \circ D_t\, 0\text{-}T(\mathrm{TR}) \circ \mathrm{YIELD}$ (by Theorem 4.8 of [EngVogl])
$= D_t\, 0\text{-}T(P^n(S)) \circ \mathrm{YIELD}$ (because, for every S', $D_t\, 0\text{-}T(S')$ is closed under right composition with $D_t\, 0\text{-}T(\mathrm{TR})$; this fact has literally the same proof as Lemma 8.9 of [EV])
$= D_t\, n\text{-}T(S) \circ \mathrm{YIELD}$ (by (*)). □

There are two ways of defining n-level string languages: either by monadic n-level tree grammars or by the yield of $(n-1)$-level tree grammars. The fact that these two concepts are equivalent, with one level difference, was shown in Theorem 7.17 of [Dam]. Similarly, we can consider two ways of defining n-level S-to-string transductions, i.e., classes of relations in $C \times \Sigma^*$ for some alphabet Σ. We compare n-level S transducers, which have a "monadic" terminal alphabet, with the "yield" of $(n-1)$-level S transducers. In order to make this precise, we need some terminology.

8.4. Definition.

(i) A *monadic* $D(Q)$*-set* Δ is a $D(Q)$-set such that there is a designated symbol # in Δ of type (λ, q) and every other symbol in Δ has type (q, q).

(ii) Let Σ be an alphabet. The monadic $D(Q)$-set *associated* with Σ, denoted by $m(\Sigma)$, is defined by $m(\Sigma)^{(q,q)} = \Sigma$.

(iii) For an alphabet Σ, define the bijection flat: $T_{m(\Sigma)} \to \Sigma^*$ by $\mathrm{flat}(\#(\,)) = \lambda$ and for every $\sigma \in \Sigma$ and $t \in T_{m(\Sigma)}$, $\mathrm{flat}(\sigma(t)) = \sigma \cdot \mathrm{flat}(t)$. For $L \subseteq T_{m(\Sigma)}$, $\mathrm{flat}(L) = \{\mathrm{flat}(t) \mid t \in L\}$. □

The mapping flat turns monadic trees in a horizontal direction by transforming them into strings. As we did for yield (in Sect. 2.1), we extend flat to relations $R \subseteq A \times T_{m(\Sigma)}$ (and classes of relations), where A is an arbitrary set, by defining $\mathrm{flat}(R) = \{(a, \mathrm{flat}(t)) \mid (a, t) \in R\}$.

8.5. Definition. Let $n \geqq 0$. An n-level S transducer $M = (N, e, \Delta, A_{\mathrm{in}}, R)$ is *monadic* if Δ is a monadic $D(Q)$-set. □

The class of translations induced by monadic n-level S transducers is denoted by $n\text{-}T_{\text{mon}}(S)$. Now we can formalize the comparison between the two possible methods of defining n-level S-to-string transductions and show that they are equivalent. At the same time we provide a sequential machine characterization for these transductions.

8.6. Theorem. *For every* $n \geqq 0$,

(a) $\text{flat}((n+1)\text{-}T_{\text{mon}}(S)) = \text{yield}(n\text{-}T(S))$
(b) $\text{flat}(n\text{-}T_{\text{mon}}(S)) = \text{REG}(P^n_{\text{bex}}(S))$;
if S contains an identity, then "bex" can be dropped.

Proof. Restricting Theorem 6.15 to a monadic terminal alphabet, we obtain $n\text{-}T_{\text{mon}}(S) = 0\text{-}T_{\text{mon}}(P^n_{\text{bex}}(S))$. In general, for every storage type S', $\text{flat}(0\text{-}T_{\text{mon}}(S')) = \text{REG}(S')$. In fact, the right-hand sides of corresponding rules of a monadic 0-level S' transducer M and a regular S' transducer M' are related via the bijection $\gamma\colon T_{m(\Sigma)}(N\langle F\rangle) \to \Sigma^* \cup \Sigma^* N\langle F\rangle$, where $m(\Sigma)$ and Σ are the terminal alphabets of M and M', respectively, and N is the set of nonterminals of M and M'. γ is defined by $\gamma(A\langle f\rangle) = A\langle f\rangle$, $\gamma(\#()) = \lambda$, and for $\sigma \in \Sigma$ and $t \in T_{m(\Sigma)}$, $\gamma(\sigma(t)) = \sigma \cdot \gamma(t)$. This shows that $\text{flat}(n\text{-}T_{\text{mon}}(S)) = \text{REG}(P^n_{\text{bex}}(S))$, which proves (b). By Theorem 6.3 of [EV], $\text{REG}(P^{n+1}_{\text{bex}}(S)) = \text{CF}(P^n_{\text{bex}}(S))$, and clearly, $\text{CF}(P^n_{\text{bex}}(S)) = \text{yield}(\text{RT}(P^n_{\text{bex}}(S)))$ (where we assume that there is a specific terminal symbol of rank 0 that is viewed as the empty string when the yield is taken). Then, by Fact 4.6 and Theorem 6.15, $\text{REG}(P^{n+1}_{\text{bex}}(S)) = \text{yield}(n\text{-}T(S))$. From this and statement (b) of this theorem, (a) follows. □

Note that, by taking the trivial storage type and ranges of transductions, Theorem 8.6(a) reproves Theorem 7.17 of [Dam].

In [DamGoe] the class of n-level string languages is defined as range $(\text{flat}(n\text{-}T_{\text{mon}}(S_0)))$ and it is shown there that this class is characterized by n-iterated pushdown automata. Clearly, this is a special case of Theorem 8.6(b).

Of course, Theorem 8.6 holds in particular for n-level tree-to-string transducers ($S = \text{TR}$). In the next theorem we state the corresponding result for total deterministic n-level tree-to-string transducers.

8.7. Theorem. *For every* $n \geqq 0$,

(a) $\text{flat}(D_t(n+1)\text{-}T_{\text{mon}}(\text{TR})) = \text{yield}(D_t n\text{-}T(\text{TR}))$
(b) $\text{flat}(D_t n\text{-}T_{\text{mon}}(\text{TR})) = D_t\,\text{REG}(P^n(\text{TR}))$.

Proof. Taking the monadic case of Theorem 7.12, (b) follows as in the proof of the previous theorem. By Theorem 8.12 of [EV] (and Fact 4.6) $D_t\,\text{REG}(P^{n+1}(\text{TR})) = \text{yield}(D_t 1\text{-}T(\text{TR})^n)$, and (a) now follows from (b) and Theorem 8.1(a). □

In the next theorem we show that high level tree transducers form a strict hierarchy.

8.8. Theorem. *The families of translation classes* $\{n\text{-}T(\text{TR}) \mid n \geqq 0\}$ *and* $\{D_t n\text{-}T(\text{TR}) \mid n \geqq 0\}$ *are strict hierarchies. There is even a translation in* $D_t(n+1)\text{-}T(\text{TR})$ *which is not in* $n\text{-}T(\text{TR})$.

Proof. The theorem follows from Lemma 4.15 of [EngVogl], which says that there is a translation in $D_t\mathrm{CFT(TR)}^{n+1}$, which is not in $\mathrm{CFT(TR)}^n$, and from Theorem 8.1. □

Another consequence of Theorem 8.1 is the closure of the class of regular tree languages under the inverse of high level tree transductions.

8.9. Corollary. *For every* $n \geq 0$, RT *is closed under* $n\text{-}T(\mathrm{TR})^{-1}$. *The domain of every n-level tree transducer is a regular tree language.*

Proof. By Theorem 8.1(b), and by Theorem 7.4 of [EngVogl]. □

In [Eng2] the concept of top-down tree transducer with regular look-ahead is introduced to overcome the inability of these transducers to inspect subtrees before deleting them. It is shown in [EngVogl] that macro tree transducers have this ability, i.e., they are closed under regular look-ahead (cf. Theorem 6.15 of [EngVogl]). Here we show that, for every $n \geq 1$, n-level tree transducers are closed under regular look-ahead. Recall from the discussion after Definition 7.2 that $\mathrm{TR_{LA}}$ formalizes the concept of regular look-ahead in terms of storage types.

8.10. Theorem. *Let* $n \geq 1$. *Then* $n\text{-}T(\mathrm{TR_{LA}}) = n\text{-}T(\mathrm{TR})$ *and* $D_t n\text{-}T(\mathrm{TR_{LA}}) = D_t n\text{-}T(\mathrm{TR})$.

Proof. The equalities follow immediately from Theorem 6.15, Lemma 7.7, the monotonicity of P_{bex} (Lemma 3.23), and the Justification Theorem (Theorem 3.16). □

As a consequence of the closure under look-ahead, it was shown in [EngVogl] that the class of ranges of total deterministic macro tree transducers coincides with $D_t\mathrm{CFT(TR)}$ applied to RT, i.e., with the class $\bigcup\{\tau(L) \mid \tau \in D_t\mathrm{CFT(TR)} \text{ and } L \in \mathrm{RT}\}$ (Theorem 7.1 of [EngVogl]). Now we can show the corresponding result for total deterministic n-level tree transducers.

8.11. Corollary. *For every* $n \geq 0$, $\mathrm{range}(D_t n\text{-}T(\mathrm{TR})) = (D_t n\text{-}T(\mathrm{TR}))(\mathrm{RT})$.

Proof. The direction $\mathrm{range}(D_t n\text{-}T(\mathrm{TR})) \subseteq (D_t n\text{-}T(\mathrm{RT}))(\mathrm{RT})$ is obvious. To prove the other direction, one only has to observe that a total deterministic n-level tree transducer can check by look-ahead whether the input tree is in the specified regular tree language or not. Then the statement of this corollary follows from the closure of $D_t n\text{-}T(\mathrm{TR})$ under regular look-ahead (Theorem 8.10), cf. the proof of Theorem 7.1 of [EngVogl]. □

We note that in Theorem 7.10 of [Dam] it was proved that $(D_t n\text{-}T(\mathrm{TR}))(\mathrm{RT})$ is an infinite hierarchy.

Finally, we want to consider the class $\bigcup_n 1\text{-}T(\mathrm{TR})^n$ of compositions of macro tree transducers, and in particular the corresponding class $\bigcup_n \mathrm{yield}(\mathrm{range}(1\text{-}T(\mathrm{TR})^n))$ of "tree transformation languages". This class deserves further investigation: it is a large class, containing many well-known hierarchies of classes of languages. In fact, by Theorem 8.1, it contains the high-level tree transformation languages, i.e., $\bigcup_n \mathrm{yield}(\mathrm{range}(n\text{-}T(\mathrm{TR})))$. It contains the OI-hierarchy languages, i.e., $\bigcup_n \mathrm{yield}(\mathrm{range}(n\text{-}T(S_0)))$, because $\mathrm{range}(n\text{-}T(S_0)) \subseteq$

range$(n\text{-}T(\mathrm{TR}))$ as can be easily shown (see Corollary 3.12 of [EngVog1] for $n=1$). It also contains the IO-hierarchy languages $\bigcup_n \mathrm{yield}(\mathrm{YIELD}^n(\mathrm{RT}))$ [EngSch, Dam]: by Theorem 8.2 $\mathrm{YIELD}^n(\mathrm{RT}) \subseteq (D_t n\text{-}T(\mathrm{TR}))(\mathrm{RT})$, and by Corollary 8.11 the latter class equals range$(D_t n\text{-}T(\mathrm{TR}))$. Since every top-down tree transducer is a macro tree transducer, it also contains the languages $\bigcup_n \mathrm{yield}(\mathrm{range}(0\text{-}T(\mathrm{TR})^n))$ of the top-down tree transducer hierarchy [Eng7]. And finally, it contains the ETOL-control hierarchy languages [Eng7], because, as shown in [Vog2], this hierarchy is inside the OI-hierarchy. On the other hand, the class $\bigcup_n \mathrm{yield}(\mathrm{range}(1\text{-}T(\mathrm{TR})^n))$ is a proper subclass of the class of recursive languages. Recursiveness follows from Corollary 8.9 (cf. Theorem 7.5 of [EngVog1]), and the fact that, for every ranked alphabet Σ, there is a macro tree transducer M such that, for every $t \in T_\Sigma$, flat$(\tau(M)(t)) = \mathrm{yield}(t)$, cf. Sect. C of [Eng4]. Proper inclusion follows from the obvious fact that this class is closed under arbitrary homomorphisms.

Acknowledgments. We wish to point out that our study of high-level tree transducers would hardly have been possible without the many discussions we had with Werner Damm. He influenced our understanding of the subject in a substantial way. Moreover, we thank one of the referees for many detailed comments and suggestions to the first version of this paper.

References

[AhoUll1] Aho, A.V., Ullman, J.D.: Translations on a context-free grammar. Inf. Control **19**, 439–475 (1971)

[AhoUll2] Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Vol. 1, 2. Englewood Cliffs, N.J.: Prentice Hall 1973

[Bra] Brainerd, W.S.: Tree generating regular systems. Inform. Control **14**, 217–231 (1969)

[ChiMar] Chirica, L.M., Martin, D.E.: An order algebraic definition of Knuthian semantics. Math. Syst. Theory **13**, 1–27 (1979)

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

[Dam] Damm, W.: The IO- and OI-hierarchies. Theor. Comput. Sci. **20**, 95–206 (1982)

[DamGoe] Damm, W., Goerdt, A.: An automata-theoretical characterization of the OI-hierarchy. Inf. Control **71**, 1–32 (1986)

[DamGue1] Damm, W., Guessarian, I.: Combining T and Level n. In: Proceedings of the 9th Mathematical Foundations of Computer Sciences 1981. (Lect. Notes Comput. Sci., Vol. 118, p. 262–270). Berlin Heidelberg New York: Springer 1981

[DamGue2] Damm, W., Guessarian, I.: Implementation techniques for recursive tree transducers on higher-order data types. Report 83-16, Laboratoire Informatique Theorique et Programmation, Université Paris 7 (1983)

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

[Eng2] Engelfriet, J.: Top-down tree transducers with regular look-ahead. Math. Syst. Theory **10**, 289–303 (1977)

[Eng3] Engelfriet, J.: Some open questions and recent results on tree transducers and tree languages. In: Book, R.V. (ed.) Formal language theory; perspectives and open problems. New York: Academic Press 1980

[Eng4] Engelfriet, J.: Tree transducers and syntax-directed semantics. TW-Memorandum Nr.363 (1981), Twente University of Technology; also: Proceedings of the 7th CAAP, march 1982, Lille, pp. 82–107

[Eng5] Engelfriet, J.: Recursive automata. (1982, unpublished notes)

[Eng6] Engelfriet, J.: Iterated pushdown automata and complexity classes. Proceedings of the 15th STOC, april 1983, Boston, pp. 365–373. New York: ACM 1983

[Eng7] Engelfriet, J.: Three hierarchies of transducers. Math. Syst. Theory **15**, 95–125 (1982)

[EngFil] Engelfriet, J., File G.: The formal power of one-visit attribute grammars. Acta Informatica **16**, 275–302 (1981)

[EngRozSch] Engelfriet, J., Rozenberg, G., Slutzki, G.: Tree transducers, L-systems, and two-way machines. J. Comput. Syst. Sci. **20**, 150–202 (1980)

[EngSch] Engelfriet, J., Schmidt, E.M.: IO and OI. J. Comput. Syst. Sci. **15**, 328–353 (1977); J. Comput. Sci. **16**, 67–99 (1978)

[EngVog1] Engelfriet, J., Vogler, H.: Macro tree transducers. J. Comput. Syst. Sci. **31**, 71–146 (1985)

[EngVog2] Engelfriet, J., Vogler, H.: Regular characterizations of macro tree transducers. In: Courcelle B. (ed.) 9th Colloquium on Trees in Algebra and Programming, march 1984, Bordeaux, France. Cambridge University Press, pp. 103–117

[EngVog3] = [EV] Engelfriet, J., Vogler, H.: Pushdown machines for the macro tree transducer. Theor. Comput. Sci. **42**, 251–369 (1986)

[EngVog4] Engelfriet, J., Vogler, H.: Characterization of high level tree transducers. In: Brauer, W. (ed.) Proceedings of 12th International Colloquium on Automata, Languages, and Programming 1985, Nafplion, Greece. (Lect. Notes Comput. Sci., Vol. 199, pp. 171–178) Berlin Heidelberg New York: Springer 1985

[Fis] Fischer, M.J., Grammars with macro-like productions; Ph.D. Thesis, Harvard University, USA, 1968

[GecSte] Gecseg, F., Steinby, M.: Tree automata. Budapest: Akademiai Kiado 1984

[Gre] Greibach, S.A.: Full AFLs and nested iterated substitution. Inf. Control **16**, 7–35 (1970)

[Gue] Guessarian, I.: Pushdown tree automata. Math. Syst. Theory **16**, 237–263 (1983)

[HopUll] Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Addison-Wesley Publ. Comp. 1969

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

[Mai] Maibaum, T.S.E.: A generalized approach to formal languages. J. Comput. Syst. Sci. **8**, 409–439 (1974)

[Mas] Maslov, A.N.: Multi-level stack automata. Probl. Inf. Transm. **12**, 38–43 (1976)

[MarVer] Martin, D.F., Vere, S.A.: On syntax-directed transductions and tree transducers. 2nd Annual ACM Symposium on Theory of Computation, May 1970

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

[Sco] Scott, D.: Some definitional suggestions for automata theory. J. Comput. Syst. Sci. **1**, 187–212 (1967)

[Tha] Thatcher, J.W.:: Generalized^2 sequential machine maps. J. Comput. Syst. Sci. **4**, 339–367 (1970)

[Vog1] Vogler, H.: Berechnungsmodelle syntaxgesteuerter Übersetzungen. Diplomarbeit, RWTH Aachen, April 1981

[Vog2] Vogler, H.: The OI-hierarchy is closed under control. Inf. Computat. 1988 (to appear)

[Vog3] Vogler, H.: Tree transducers and pushdown machines. Ph.D. thesis, Twente University of Technology, The Netherlands, March 1986

[vLe] Leeuwen, J. van: Notes on pre-set pushdown automata. In: Rozenberg, G., Salomaa, A. (eds.) L Systems. (Lect. Notes Comput. Sci., Vol. 15, pp. 177–188) Berlin Heidelberg New York: Springer 1974

[Wat] Watt, D.A.: Contextual constraints. In: Lorho, B. (ed.) Methods and tools for compiler construction, an advanced course. pp. 45–80. Cambridge: Cambridge University Press 1984

Received August 15, 1985 / May 10, 1988