Theoretical Computer Science 42 (1986) 251-368
North-Holland

PUSHDOWN MACHINES FOR THE MACRO TREE TRANSDUCER

Joost ENGELFRIET* and Heiko VOGLER**

Department of Computer Sciences, Twente University of Technology, 7500 AE Enschede, The Netherlands

Communicated by M. Nivat
Received February 1985

Abstract. The macro tree transducer can be considered as a system of recursive function procedures with parameters, where the recursion is on a tree (e.g., the syntax tree of a program). We investigate characterizations of the class of tree (tree-to-string) translations which is induced by macro tree transducers (macro tree-to-string transducers, respectively). For this purpose we define several pushdown machines of which the control is recursive without parameters, or even iterative, and which work on a generalized pushdown as storage. Because of the relevance for semantics of programming languages, we stress (besides the nondeterministic case) the study of machines for the total deterministic macro tree(-to-string) transducer, which translates every input tree into exactly one output tree (string, respectively). Finally, we characterize the n-fold composition of total deterministic macro tree transducers by recursive pushdown machines with an iterated pushdown as storage, which is a pushdown of pushdowns of ... of pushdowns.

Contents

* Present affiliation: Department of Mathematics and Computer Science, University of Leiden, 2300 RA Leiden, The Netherlands.

** Present affiliation: Lehrstuhl für Informatik II, RWTH Aachen, D-5100 Aachen, Fed. Rep. Germany.

The work of the second author has been supported by the Dutch Organization for the Advancement of Pure Research (Z.W.O.).

1. Introduction

It has become a custom to describe the semantics of programming languages in a syntax-directed way. There are more or less formalized metalanguages for the description of syntax-directed semantics such as generalized syntax-directed translation schemes [3, 4], top-down tree transducers [37, 39, 11], and attribute grammars with synthesized attributes only [33]. These concepts have in common that the meaning of a syntactic construct is determined entirely by the meaning of its constituents. However, for the accomplishment of some tasks, it is necessary to take also the context of a syntactic construct into account, e.g., when the access to the environment specified by the declaration part of a program is needed. This additional requirement on metalanguages is captured, e.g., by attribute grammars [33] and denotational semantics [41]. More recently, macro tree transducers [14, 7, 22] were studied as another concept of syntax-directed semantics in which context can be handled. Actually, (deterministic) macro tree transducers model at least the features of denotational semantics needed to simulate attribute grammars, and thus can be viewed as a first modest approach to model denotational semantics of programming languages. We refer the reader to [15] for a detailed discussion of these metalanguages of syntax-directed semantics.

In the framework of program (scheme) theory the deterministic macro tree transducer can be viewed as a system of recursive function procedures with parameters, one function procedure corresponding to each state of the transducer (cf. [7, 15]). Every function procedure is of type $T_\Sigma \times T_\Delta^n \to T_\Delta$ for some $n \geq 0$, where T_Σ and T_Δ are the sets of trees over the ranked alphabets Σ and Δ, respectively. Thus, the arguments of a function procedure are divided into two sorts: there is exactly one 'syntactic' parameter of type T_Σ and there may be some 'context' parameters of type T_Δ (top-down tree transducers are macro tree transducers without context parameters). The actual value of the syntactic parameter is a subtree of the input tree which is initially given to the main function procedure: this is a designated function procedure without context parameters, i.e., of type $T_\Sigma \to T_\Delta$ (it determines the type of the translation which is induced by the macro tree transducer). During the evaluation of procedure calls, subtrees of the input tree are selected and in this

way the actual syntactic parameters decrease in height (this is the meaning of 'syntax-directed'). The context parameters keep context information for the current value of the syntactic parameter. Context information consists of a tree in T_Δ, to be viewed as an expression over the set Δ of basic operation symbols, denoting a value in some semantic domain (the fact that this domain is not specified, means that the macro tree transducer is a program scheme). Since the context parameters are not necessarily called by value, an actual context parameter may still contain procedure calls (note that the type of every function result is the same as that of the context parameters). As opposed to the syntactic parameters, the actual values of the context parameters usually increase in size: during the evaluation of a function procedure A, a new context value (i.e., the value of a context parameter of another call) is constructed out of function procedures, basic operations, and old context values of A. During evaluation of the macro tree transducer, there may be (many) parallel recursive calls of function procedures, and, moreover, these calls may be nested to any depth. This situation is closely related to the nesting of nonterminals in level-2 schemes [8], context-free tree grammars [20], and macro grammars [24]. Thus, as usual, we can distinguish between three different evaluation strategies for such a system of recursive function procedures: unrestricted, outside-in (OI), and inside-out (IO). In the unrestricted strategy, every function procedure call may be evaluated. In the OI-strategy, only those function procedure calls may be evaluated that do not occur in the context information of other calls. In the IO-strategy, a procedure call may only be evaluated if its context parameters do not contain other calls.

Of course, for modelling semantics of programming languages, those translating devices should be considered that associate to every syntactic object (i.e., a syntax tree of a program) exactly one meaning. Consequently, in this paper our main interest is in the total deterministic macro tree transducer: for every input tree (of type T_Σ) it computes exactly one output tree (of type T_Δ). The total deterministic macro tree transducer has the nice property that the three different evaluation strategies are equivalent [22]. However, besides the total deterministic case, we also study the nondeterministic case (and this, in fact, before the total deterministic case), because it is mostly easier to treat and it already provides some insight in the main features of the considered translation device.

The aim of the present paper is to find machines which characterize the class of translations induced by macro tree transducers. Generally speaking, a machine consists of a 'storage' and a 'control' or 'program', where the program can work on the storage by testing and transforming the storage 'configurations'. In fact, the macro tree transducer itself can be considered as such a machine by taking (input) trees as configurations; the label of the root of a tree can be tested and subtrees can be selected. We denote this storage by TR. As already discussed above, the control of the macro tree transducer consists of a system of recursive function procedures in which calls may occur nestedly and in parallel. We are looking for machines which have a simpler control and (consequently) a more complicated

storage than the macro tree transducer. More precisely, instead of having nested and parallel calls of function procedures, we are aiming at machines without nested calls, and, preferably, even without parallel calls, i.e., with an iterative, sequential control. The appropriate storage types have the form of pushdowns, in which every square contains besides the usual pushdown symbol a configuration of another storage, e.g., a (pointer to the node of a) tree of the storage TR (cf. [28, 17]). For this reason, the characterizing devices are collectively called *pushdown machines.*

To facilitate our search for equivalent machines we will also consider the macro tree-to-string transducer, obtained by taking the set Δ of basic symbols to be an ordinary alphabet rather than a ranked alphabet: it translates a tree over Σ into a string over Δ, using context parameters of type Δ^*. Since trees are special strings (viz., expressions), every macro tree transducer may be viewed as a special macro tree-to-string transducer. The advantage of this is that the output tree may be written (as an expression) on the output tape of an equivalent machine.

The main guideline in our approach of finding equivalent machines for the macro tree(-to-string) transducer is the well-known formula "macro = indexed = nested stack" or, precisely, OI macro grammars and indexed grammars are equivalent [24] and the class of indexed languages is equal to the class of nested stack languages, i.e., domains of nested stack automata [1]. The reason for using this guideline is that the control of a macro tree transducer (i.e., nested recursive procedure calls), and, even more, that of a macro tree-to-string transducer, is very similar to a macro grammar; thus, a macro tree(-to-string) transducer may be viewed as a macro grammar control working on the storage TR. (However, we wish to stress that in our opinion the macro tree transducer is far more relevant to semantics than the macro grammar is to the syntax of programming languages.) This formal-language theoretic guideline is the reason why our constructions and results for the nondeterministic case use the OI-evaluation strategy; recall however that, for the total deterministic case, all the discussed strategies define the same translation.

Now we give an overview of the studied pushdown machines for the total deterministic macro tree(-to-string) transducer. In order to improve readability, we will provide the references to related results in a later section.

(a) The macro tree transducer is equivalent to the indexed tree transducer. The control of this machine consists of recursive function procedures which do not have context parameters and thus, no nested calls. The control of the indexed tree transducer works on pushdowns of pointers to the input tree and the corresponding storage type is denoted by P(TR). More precisely, a configuration of P(TR) is a pushdown in which every square contains a pushdown symbol (as in a usual pushdown) and, moreover, a pointer to a node of the input tree, which is initially given to the transducer.

(b) An equivalent pushdown machine for the macro tree-to-string transducer is the pushdown^2 tree-to-string transducer. It has an iterative, sequential control and hence, it can be considered as a transducer in the classical sense with a finite control, an input device (containing the input tree), an additional storage (which is a

pushdown of pushdowns of pointers to nodes of the input tree, denoted by $P^2(TR)$), and an output tape (containing basic symbols).

(c) Another iterative, sequential machine which characterizes the macro tree-to-string transducer is the nested stack tree-to-string transducer. It uses a variation of a nested stack [1] as additional storage, denoted by NS(TR). Every stack square contains besides a stack symbol (as in the usual nested stack) also a pointer to a node of the input tree.

(d) The results under (a) and (b) can be generalized to characterizations of compositions of macro tree(-to-string) transducers (relevant to syntax-directed semantics in phases). For every $n \geq 1$, the pushdown^n tree transducer is equivalent to the n-fold composition of the macro tree transducer. Since we use 'indexed tree transducer' and 'pushdown tree transducer' just as different names for the same transducer, this repeats for $n = 1$ the result under (a). Hence, the control of a pushdown^n tree transducer is still recursive, but the involved function procedures do not have context parameters. A configuration of its storage type is a pushdown of pushdowns of ... of pushdowns (n times) of pointers to the input tree. We denote that storage type by $P^n(TR)$. The class of tree-to-string translations corresponding to the n-fold composition of macro tree transducers is characterized by pushdown^{n+1} tree-to-string transducers. This is again an iterative, sequential machine of the usual type with $P^{n+1}(TR)$ as additional storage. For $n = 1$ this repeats the result under (b). These results show that if semantics is specified in several phases, each phase by a macro tree transducer, it can be realized in one phase by a tree(-to-string) transducer with a simple control and an iterated pushdown as storage.

These are the pushdown machines for the total deterministic macro tree transducer which we are going to present in this paper. For (a)-(c) we will also prove similar results for the nondeterministic case. However, they are not as nice as for the total deterministic case: we either have to enrich the power of the macro tree transducer slightly or restrict the storage type $P(S)$ in order to obtain characterizations. In the rest of the introduction we will describe by means of which tools the characterizations are proved. Moreover, the connections of our results to the literature are discussed.

In order to describe both our machine characterizations and the connections to already existing results in an understandable way, we set up the mentioned machine concepts in one unifying framework. This is based on the general philosophy of Scott (cf. [38]) of separating program and storage (type) of translating devices. Scott considers iterative, sequential programs, which can be drawn as a flowchart. However, we work with a generalization of Scott's approach which is suggested in [16]: any kind of program is allowed rather than just flowcharts. Here we consider grammars (from formal language theory) as programs, in particular, regular (or, right-linear) grammars, context-free grammars, macro grammars, regular tree grammars, and context-free tree grammars. For convenience we abbreviate the types of the mentioned grammars by REG, CF, MAC, RT, and CFT, respectively. These grammars are used to model the following controls; REG: flowcharts; RT, CF: recursive function procedures without context parameters (and results of type T_Δ

and Δ^*, respectively); CFT, MAC: recursive function procedures with context parameters (and, again, results of type T_Δ and Δ^*, respectively). Recall that, roughly speaking, a storage type is specified by a set of configurations which can be tested by predicates and transformed by instructions. We enable a grammar to act on a storage type by associating with each occurrence of a nonterminal A a configuration c. A rule of a grammar can be applied to the tuple $A(c)$, provided a predicate, which is specified in the rule, holds for c. The new configurations for the nonterminals of the right-hand side of the rule are obtained by transforming c according to specified instructions. Then, the execution of a program on a storage type corresponds to a derivation of a 'grammar with storage', where the terminals of the grammars are the output symbols of this device. Clearly, by considering regular grammars, the usual iterative, sequential programs of Scott are reobtained. Since a grammar with storage type induces a translation (from configurations to trees or strings), we call such a device also an $X(S)$-transducer, where X and S denote the type of grammar and storage, respectively. The class of translations induced by $X(S)$-transducers is denoted by $X(S)$; for the total deterministic case the denotation of the translation class is preceded by D_t. In fact, as discussed before, the macro tree transducer is a CFT(TR)-transducer and the macro tree-to-string transducer is a MAC(TR)-transducer.

However, in the present paper we provide machine characterizations for CFT(S) and MAC(S), where S is an arbitrary storage, rather than for CFT(TR) and MAC(TR). This is necessary in order to obtain the results for the compositions of total deterministic macro tree transducers, cf. (d). Moreover, by taking the trivial storage type (denoted by S_0), which contains only one configuration, no predicate, and the identity as instruction, we reobtain the related results in the literature for grammars: obviously, the class of ranges of translations in $X(S_0)$ is the class of languages generated by X grammars. Thus, rather than transforming the known proofs for grammars (of "macro = indexed = nested stack") into proofs for tree transducers, we give uniform proofs for $X(S_0)$-transducers, which can be specialized to the grammar case ($S = S_0$) and to the tree transducer case ($S = \mathrm{TR}$). Finally, these uniform results give more insight in specific controls and storage types, and thus, they may be of more general use.

Now we will list the results which correspond to (a)-(d) for total deterministic $X(S)$-transducers with arbitrary storage type S and provide some references to related results and concepts in the literature.

(a') $D_tCFT(S) = D_tRT(P(S))$ (cf. Theorem 5.16). For $S = \mathrm{TR}$, this says that macro tree transducers and indexed tree transducers are equivalent. Note that regular tree grammars are context-free tree grammars without (context) parameters. In [25] it is shown that every (noncircular) attribute grammar, viewed as a tree translating device, can be simulated by a total deterministic macro tree transducer (cf. also [7, 15]). Since an attribute grammar can obviously be considered as a $D_tRT(P(TR))$-transducer, where the attributes are the states of the transducer and the semantic rules are its rules, (a') includes the result of [25]. In the nondeterministic case, by

specifying S to S_0 and considering ranges of translation classes, we reobtain the result of [29] that the class of OI context-free tree languages and the class of languages accepted by restricted pushdown tree automata are the same. Actually, it is an easy observation that a restricted pushdown tree automaton $\mathfrak{M}$ can be viewed as an RT(P(TR))-transducer of which $\mathfrak{M}$ accepts the range.

In fact, result (a') also holds for strings instead of trees, i.e., $D_tMAC(S) = D_tCF(P(S))$ (cf. Theorem 5.16). By replacing S again by S_0, this reproves (in the nondeterministic case) the result of [24] that OI macro grammars and indexed grammars are equivalent: since the sequence of flags attached to every nonterminal in an indexed grammar behaves like a pushdown, indexed grammars are the same as ranges of $CF(P(S_0))$-transducers.

(b') If $P(S_{LA}) \equiv P(S)$, then $D_tMAC(S) = D_tREG(P^2(S))$ (cf. Theorem 6.15). The storage type S_{LA} is obtained from S by adding special predicates, so called look-ahead tests (cf. Definition 6.5). A look-ahead test applied to a configuration c can test properties (i.e., predicates) of several $\phi(c)$, where ϕ is a sequence of instructions (or even a program), without transforming the storage configuration c. If $S = TR$, this look-ahead concept corresponds to the usual regular look-ahead (cf. [12, 22]). Then, the condition $P(S_{LA}) \equiv P(S)$ reads: the storage types $P(S_{LA})$ and $P(S)$ are equivalent, which means that the pushdown $P(S)$ can be used to decide look-ahead tests on S-configurations. Since $P(TR_{LA}) \equiv P(TR)$ (cf. Theorem 8.1), result (b') turns, for $S = TR$, into characterization (b): macro tree-to-string transducers and pushdown^2 tree-to-string transducers are equivalent or, in the $X(S)$-transducer formalism, $D_tMAC(TR) = D_tREG(P^2(TR))$ (cf. Theorem 8.2). The tree-walking pushdown transducer of [32] is a special case of the $D_tREG(P^2(TR))$-transducer; it simulates the translation of attribute grammars viewed as tree-to-string transducers. Since tree-to-string attribute grammars can be simulated by $D_tMAC(TR)$-transducers (cf. (a')), we have reobtained the result of [32]. We note that the storage type $P^2(S_0)$, i.e., the pushdown of pushdowns, is already considered in [28, 35, 36, 9, 17]. In fact, for $S = S_0$, (b') reproves (in the nondeterministic case) the result of [36] that macro grammars (i.e., indexed grammars) are equivalent to pushdown^2 automata (called indexed pushdown automata in [36]).

We also note that, in the total deterministic case, the top-down tree-to-string transducer with regular look-ahead is characterized by the checking-tree pushdown transducer (cf. [19, Theorem 4.7]). Since the top-down tree-to-string transducer is a CF(TR)-transducer and the checking-tree pushdown transducer would here be called a REG(P(TR))-transducer, the result of [19] reads $D_tCF(TR_{LA}) = D_tREG(P(TR))$. From this equation LA cannot be dropped.

(c') If $P(S_{LA}) \equiv P(S)$, then $D_tMAC(S) = D_tREG(NS(S))$ (cf. Theorem 7.6). For $S = TR$, the condition holds, and hence, macro tree-to-string transducers and nested stack tree-to-string transducers are equivalent (cf. Theorem 8.2). This result was claimed in [15]. For $S = S_0$, this result repeats (for the nondeterministic case) the equality of OI macro languages and nested stack languages (cf. [24] for "macro = indexed" and [1] for "indexed = nested stack").

(d') In this case we do not prove a general result for $X(S)$-transducers (although that would be possible), but rather use the previous general results to obtain: $D_tCFT(TR)^n = D_tRT(P^n(TR))$ and $D_tCFT(TR)^{n-1} \circ D_tMAC(TR) = D_tREG(P^{n+1}(TR))$ (cf. Theorem 8.12). Iterated pushdown automata, i.e., automata with storage type $P^n(S_0)$, were considered in [35, 9] to characterize higher-level macro grammars, and in [17] to characterize super exponential time complexity classes. If AG denotes the class of tree translations realized by attribute grammars, then $AG^n \subseteq D_tCFT(TR)^n \subseteq AG^{n+1}$ [15]. Hence, composition of attribute grammars is characterized by iterated pushdown tree transducers.

These are the main results of this paper concerning pushdown machines for the total deterministic macro tree(-to-string) transducer. A main tool for proving the machine characterizations is the concept of storage type simulation, of which the idea goes back to [30]. We elaborate this idea and say that a storage type S_1 is simulated by the storage type S_2 (for short: $S_1 \leqslant S_2$), if there is a partial function from C_2 to C_1 (where C_i is the set of configurations of S_i) indicating which configurations of S_2 are 'representations' of configurations of S_1. Moreover, for every predicate and instruction ϕ of S_1 there must be a deterministic flowchart ω_ϕ using predicates and instructions of S_2, which imitates the meaning of ϕ on the elements of C_2 that are representations of elements of C_1. If $S_1 \leqslant S_2$ and $S_2 \leqslant S_1$, then the storage types S_1 and S_2 are equivalent (denoted by $S_1 \equiv S_2$). The main reason why the concept of storage type simulation can be used as a tool for proving the characterization results, is the fact that (*) *if* $S_1 \leqslant S_2$, *then for every* $X(S_1)$-*transducer* $\mathfrak{M}_1$ *there is an equivalent* $X(S_2)$-*transducer* $\mathfrak{M}_2$. The validity of (*) is intuitively clear, because, first, the predicates and instructions in $\mathfrak{M}_1$ can be replaced by the simulating flowcharts and second, the flowcharts can be incorporated into the main control. Since there is a clear separation between control and storage type in the machines which we consider, we can prove most of our results by first showing the equivalence of storage types and then applying (*). Because of the indicated importance of (*) for proving connections between classes of translations, we also call it the justification theorem (it justifies our use of the concept of simulation). We note that results about simulation of storage types are directly applicable to devices with other input/output conventions than $X(S)$-transducers, such as multi-head S-automata [17].

This paper is divided into eight sections, where the next one will fresh up some notation and also introduce a few new notions. The concept of $X(S)$-transducer will be explained in detail in Section 3. Section 4 contains a complete formal development of the concept of storage type simulation as far as we need it in this paper. Actually, this section is rather technical and, on first reading, it is sufficient to go through its introduction. After this careful construction of the necessary framework, we will proceed in Section 5 with the characterization of $CFT(S)$ and $MAC(S)$ by means of $RT(P(S))$- and $CF(P(S))$-transducers. In Section 6 we will turn to the characterization of $MAC(S)$ by $REG(P^2(S))$-transducers. Section 7 may be regarded as a pure application of the concept of storage type simulation. There,

the equivalence of the storage types NS and P^2 will be proved. Finally, in Section 8 we will apply the general results, obtained in the previous sections, to the storage type TR, thereby obtaining the pushdown machines for the macro tree transducer (including those for compositions of macro tree transducers).

This paper might have been shorter, but the proofs would then be considerably more complicated. We have done our best to cut the proofs into small, understandable pieces. For this, the '$X(S)$-approach' was helpful. We wish the reader good luck.

2. Preliminaries

Here, we recall some notations and notions of formal (tree) language theory which will be used in this paper.

2.1. General notation

In proofs which use induction on an argument, we abbreviate the induction hypothesis by I.H. For the denotation of a function $\theta: U \to V$ we use the usual lambda notation, i.e., $\lambda u \in U.\theta(u)$.

The empty set is denoted by $\emptyset$. For every $n \geq 1$, $[n] = \{1, \ldots, n\}$ and $[0] = \emptyset$. If ν is a sequence of length $n \geq 0$, then, for every $j \in [n]$, $\nu(j)$ denotes the jth component of ν. The infinite set $Y = \{y_1, y_2, y_3, \ldots\}$ is called the *set of parameters* and, for every $n \geq 1$, $Y_n = \{y_1, \ldots, y_n\}$ and $Y_0 = \emptyset$. For an object t, par(t) denotes the set of parameters occurring in t.

Let $A, A_1, \ldots, A_n$ be sets. $\mathcal{P}(A)$ is the *set of subsets* of A. For every $m \geq 0$, A^m is the set of *strings* over A of length m. Then, $A^* = \bigcup \{A^j \mid j \geq 0\}$ and $A^+ = \bigcup \{A^j \mid j \geq 1\}$. The empty string is denoted by λ.

For two sets A and B, a *relation R from A into B* is any subset of $A \times B$. The *inverse of R* is denoted by R^{-1}. The *domain* and the *range* of R, denoted by dom(R) and range(R), respectively, are the sets $\{u \mid$ there is a $v \in B$ such that $(u, v) \in R\}$ and $\{v \mid$ there is a $u \in A$ such that $(u, v) \in R\}$, respectively. The set of *images of A under R*, denoted by $R(A)$, is the set $R(A) = \{v \mid$ there is a $u \in A$ such that $(u, v) \in R\}$.

For two relations R_1 and R_2 the *composition* of R_1 and R_2 is the set $\{(u, w) \mid (u, v) \in R_1$ and $(v, w) \in R_2$ for some $v\}$ which is denoted by $R_1 \circ R_2$. Note that $\text{dom}(R_1 \circ R_2) = R_1^{-1}(\text{dom}(R_2))$. The *reflexive, transitive closure* and the *transitive closure* of R are denoted by R^* and R^+, respectively. The above-mentioned concepts can be defined in a similar way for classes of relations. Then we use the same denotations.

2.2. Trees and substitution

A *ranked set* Σ is a (possibly infinite) set in which every symbol has a unique rank. For every $n \geq 0$, Σ_n denotes the set of symbols of Σ which have rank n. The rank of a symbol is sometimes indicated as a superscript, e.g., $\sigma^{(2)}$ means that σ is of rank 2. If Σ is finite, then we obtain the usual concept of a *ranked alphabet.*

Throughout this paper, Ω denotes a countably infinite ranked set such that every subset Ω_n is infinite.

Let Σ be a ranked set and C be an arbitrary set. Then $\Sigma(C)$ is the ranked set $\{\sigma(c)^{(n)} \mid \sigma \in \Sigma_n \text{ with } n \geq 0 \text{ and } c \in C\}$. The set of (*labeled*) *trees* over Σ is denoted by T_Σ. A tree t in T_Σ is denoted by $\sigma(t_1, \ldots, t_n)$, where the root is labeled by $\sigma \in \Sigma_n$ and $t_1, \ldots, t_n$ are the (immediate) subtrees of t (if $n = 0$, then we write σ rather than $\sigma(\ )$). $T_\Sigma(C)$ denotes $T_{\Sigma \cup C}$, where the elements of C are viewed as symbols of rank 0. Let Int be the set of nonnegative integers. The *height* (and *size*) of a tree $t \in T_\Sigma$, denoted by height(t) (and size(t)), is provided by the function height: $T_\Sigma \to$ Int (and size: $T_\Sigma \to$ Int) which is defined inductively on the structure of the tree: (i) for $\sigma \in \Sigma_0$, height(σ) = size(σ) = 1; (ii) for $t = \sigma(t_1, \ldots, t_n)$, for some $n \geq 1$, $\sigma \in \Sigma_n$, and $t_1, \ldots, t_n \in T_\Sigma$, height($t$) = $1 + \max\{\text{height}(t_i) \mid i \in [n]\}$ and size(t) = $1 + \text{size}(t_1) + \cdots + \text{size}(t_n)$. The *yield* of a tree t, denoted by yield(t), is the concatenation of its leaf labels. The yield of a relation $R \subseteq U \times T_\Sigma$, where U is an arbitrary set and Σ is a ranked set, is the set yield(R) = $\{(u, w) \mid (u, t) \in R \text{ and } \text{yield}(t) = w\}$. This notion can be extended to classes of relations in an obvious way. Any subset of T_Σ is a *tree language* and the class of *recognizable* (*or regular*) *tree languages* is denoted by RECOG.

Quite often we use an abbreviation for the substitution of objects in strings or trees. Let v be a string (or a tree), let U and U' be arbitrary sets, and let θ be a mapping from U into $\mathcal{P}(U')$. Then $v[u \leftarrow \theta(u); u \in U]$ denotes the set of strings (or trees) obtained from v by replacing every occurrence of $u \in U$ in v by an element of $\theta(u)$, where different occurrences of u may be replaced by different elements of $\theta(u)$. For instance, if $v = abbc$, $U = \{b, c, d\}$, and $\theta(b) = \{\#, A(f)\}$, $\theta(c) = \{b\}$, and $\theta(d) = \{\$, y\}$, then $v[u \leftarrow \theta(u); u \in U] = \{a \# \# b,\ a \# A(f)b,\ aA(f) \# b,\ aA(f)A(f)b\}$. Mostly, $\theta(u)$ is a singleton; then we do not specify $\theta(u)$, but only provide its element. If U is understood from the context, then the part behind the semicolon is dropped. Furthermore, if U is a finite set like $\{u_1, \ldots, u_k\}$, then the object obtained from the replacement process (as explained above) is also abbreviated by $v[u_1 \leftarrow \theta(u_1), \ldots, u_k \leftarrow \theta(u_k)]$. Note that U may be a set of strings or trees; we only use the notation $v[u \leftarrow \theta(u); u \in U]$ if there are no overlapping occurrences of elements of U in v.

2.3. *X-grammars*

Let MOD be the set {regular, context-free, macro, regular tree, context-free tree} of modifiers, which are also abbreviated by REG, CF, MAC, RT, CFT, respectively. From now on X ranges over MOD. We assume the reader to be familiar with X-grammars (for REG, CF see [31], for MAC see [24], for RT see [26], for CFT see [20]). Before we recall some notation concerning X-grammars, we introduce the notion of X-form which provides a useful tool for the definition of right-hand sides of rules of X-grammars and for the definition of sentential forms of X-grammars. For two sets Ψ and Φ (intuitively, of nonterminals and terminals,

respectively), the set of *X-forms over* Ψ *and* Φ, denoted by $F_X(\Psi, \Phi)$, is defined as follows, where $F_X(\Psi, \Phi)$ is abbreviated by F.

(a) $X = \text{REG}$: $F = \Phi^* \cup \Phi^* \Psi$.

(b) $X = \text{CF}$: $F = (\Psi \cup \Phi)^*$.

(c) $X = \text{MAC}$ and Ψ is a ranked set: F is the smallest set for which (i)-(iii) holds: (i) $\lambda \in F$ and $\Phi \subseteq F$; (ii) if $\zeta_1, \zeta_2 \in F$, then $\zeta_1 \zeta_2 \in F$; (iii) if $A \in \Psi_k$ with $k \geq 0$ and $\zeta_1, \ldots, \zeta_k \in F$, then $A(\zeta_1, \ldots, \zeta_k) \in F$.

(d) $X = \text{RT}$ and Φ is a ranked set: $F = T_\Phi(\Psi)$.

(e) $X = \text{CFT}$ and Ψ and Φ are ranked sets: $F = T_{\Psi \cup \Phi}$.

For technical convenience we let MAC- and CFT-grammars start with an 'initial term' (consisting of nonterminals only) rather than with a single nonterminal. It is obvious that this is no essential enrichment.

An *X-grammar* G is a tuple $(N, \Delta, A_{\text{in}}, R)$, where N is the alphabet of nonterminals, Δ is the alphabet of terminals ($N \cap \Delta = \emptyset$), A_{in} is the initial term, and R is the finite set of rules such that

- if $X \in \{\text{MAC}, \text{CFT}\}$, then N is a ranked alphabet;
- if $X \in \{\text{RT}, \text{CFT}\}$, then Δ is a ranked alphabet;
- if $X = \text{MAC}$ or $X = \text{CFT}$, then $A_{\text{in}} \in F_{\text{MAC}}(N, \emptyset)$ or $A_{\text{in}} \in T_N$, respectively, otherwise $A_{\text{in}} \in N$;
- if $X \in \{\text{REG}, \text{CF}, \text{RT}\}$, then every rule has the form $A \to \zeta$ for some $A \in N$ and $\zeta \in F_X(N, \Delta)$; and
- if $X \in \{\text{MAC}, \text{CFT}\}$, then every rule has the form $A(y_1, \ldots, y_n) \to \zeta$ for some $A \in N_n$ with $n \geq 0$ and $\zeta \in F_X(N, \Delta \cup Y_n)$ (for $X = \text{CFT}$ the parameters are viewed as symbols of rank 0).

Those MAC- and CFT-grammars, in which the initial term is just one nonterminal of rank 0, are called MAC_1- and CFT_1-grammars, respectively.

The set of *sentential forms* of G, denoted by $\text{SF}(G)$, is $F_X(N, \Delta)$. The derivation relation of G is denoted by $\Rightarrow_G$; note that $\Rightarrow_G \subseteq \text{SF}(G) \times \text{SF}(G)$.

For a CFT-grammar $G = (N, \Delta, A_{\text{in}}, R)$, we use the outside-in (OI) derivation relation of G (cf. also [20]). It is denoted by $\Rightarrow_G$ and defined as follows: For $\xi_1, \xi_2 \in \text{SF}(G)$, $\xi_1 \Rightarrow_G \xi_2$ iff (i) there is a rule $A(y_1, \ldots, y_n) \to \zeta$ in R for some $n \geq 0$, $A \in N_n$, and $\zeta \in F_{\text{CFT}}(N, \Delta \cup Y_n)$, (ii) there is a $\xi \in F_{\text{CFT}}(N, \Delta \cup \{z\})$ where z is a symbol of rank 0 which occurs exactly once in ξ, and z does not occur in a subtree of the form $B(\zeta_1, \ldots, \zeta_k)$ of ξ (with $B \in N_k$), and (iii) there are $\xi'_1, \ldots, \xi'_n \in \text{SF}(G)$ such that

$$\xi_1 = \xi[z \leftarrow A(\xi'_1, \ldots, \xi'_n)] \quad \text{and} \quad \xi_2 = \xi[z \leftarrow \zeta'],$$

where $\zeta' = \zeta[y_i \leftarrow \xi'_i; i \in [n]]$.

For a MAC-grammar the OI-derivation relation is defined quite similarly (cf. also [24]).

The *language* generated by an X-grammar $G = (N, \Delta, A_{\text{in}}, R)$, denoted by $L(G)$, is the set $\{v \in F_X(\emptyset, \Delta) \mid A_{\text{in}} \Rightarrow_G^* v\}$. If no confusion arises, then we denote the class of languages generated by X-grammars also by X. Note that $\text{RECOG} = \text{RT}$. Note

also that yield(RT) = CF and yield(CFT) = MAC (where we assume that there is a specific symbol of rank 0 that is viewed as the empty string λ when the yield is taken).

We will also consider infinite X-grammars that may have infinite sets of nonterminals and rules (where the set of nonterminals is possibly ranked). The notions $\mathrm{SF}(G)$ and $\Rightarrow_G$ are defined in exactly the same way as for ordinary, finite X-grammars.

2.4. *Tree transducers*

In this section we will recall the definition of the macro tree transducer from [22]. The top-down tree transducer and the finite state tree automaton are special cases.

A *macro tree transducer* $\mathfrak{M}$ is a tuple $(Q, \Sigma, \Delta, q^{\mathrm{in}}, R)$, where Q is the finite set of states, each state has a rank ≥ 1, Σ and Δ are the ranked input and output alphabets, respectively, q^{in} is the initial state of rank 1, and R is the finite set of rules of the form

$$q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \to \zeta,$$

where $q \in Q_{n+1}$ for some $n \geq 0$, $\sigma \in \Sigma_m$ for some $m \geq 0$, $x_1, \ldots, x_m$ are the input variables, and $\zeta \in \mathrm{RHS}(Q, \Delta, m, n)$. The set $\mathrm{RHS}(Q, \Delta, m, n)$ is the smallest set RHS for which (i)–(iii) holds: (i) $\Delta \cup Y_n \subseteq \mathrm{RHS}$; (ii) for $\delta \in \Delta_k$ with $k \geq 0$ and $\zeta_1, \ldots, \zeta_k \in \mathrm{RHS}$, $\delta(\zeta_1, \ldots, \zeta_k) \in \mathrm{RHS}$; (iii) for $q \in Q_{n+1}$ with $n \geq 0$, x_i with $i \in [m]$, and $\zeta_1, \ldots, \zeta_n \in \mathrm{RHS}$, $q(x_i, \zeta_1, \ldots, \zeta_n) \in \mathrm{RHS}$.

$\mathfrak{M}$ is *deterministic* (*total*) if, for every pair $q \in Q_{n+1}$ and $\sigma \in \Sigma_m$, there is at most (at least) one rule of the form

$$q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \to \zeta$$

in R for some ζ.

If every state has rank 1, then $\mathfrak{M}$ is a *top-down tree transducer* and the rules have the form $q(\sigma(x_1, \ldots, x_m)) \to \zeta$, where $\zeta \in T_\Delta(Q(\{x_1, \ldots, x_m\}))$.

For a macro tree transducer $\mathfrak{M} = (Q, \Sigma, \Delta, q^{\mathrm{in}}, R)$ the OI-*derivation relation* $\Rightarrow_{\mathfrak{M}} \subseteq T^2_{Q \cup \Sigma \cup \Delta}$ is defined as follows: for $\xi_1, \xi_2 \in T_{Q \cup \Sigma \cup \Delta}$, $\xi_1 \Rightarrow_{\mathfrak{M}} \xi_2$ iff (i) there is a rule

$$q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \to \zeta \text{ in } R,$$

(ii) there is a $\xi \in T_{Q \cup \Sigma \cup \Delta}(\{z\})$ where z is a symbol of rank 0 which occurs exactly once in ξ and does not occur in a subtree of the form $p(s, \zeta_1, \ldots, \zeta_k)$ of ξ (with $p \in Q_k$, $s \in T_\Sigma$), (iii) there are $s_1, \ldots, s_m \in T_\Sigma$, and $\xi'_1, \ldots, \xi'_n \in T_{Q \cup \Sigma \cup \Delta}$ such that

$$\xi_1 = \xi[z \leftarrow q(\sigma(s_1, \ldots, s_m), \xi'_1, \ldots, \xi'_n)] \quad \text{and} \quad \xi_2 = \xi[z \leftarrow \zeta']$$

where $\zeta' = \zeta[x_i \leftarrow s_i; i \in [m]][y_j \leftarrow \xi'_j; j \in [n]]$.

The OI-*translation* induced by $\mathfrak{M}$, denoted by $\tau_{\mathrm{OI}}(\mathfrak{M})$, is the set

$$\{(s, t) \mid s \in T_\Sigma, t \in T_\Delta \text{ and } q^{\mathrm{in}}(s) \Rightarrow^*_{\mathfrak{M}} t\}.$$

The class of translations induced by (deterministic, total deterministic) macro tree transducers is denoted by (DMT_{OI}, D_tMT) MT_{OI}. In fact, for the total deterministic case we do not have to indicate the derivation mode, because the derivation modes outside-in, inside-out, and unrestricted [24, 20] provide the same class of translations (cf. [22]). For the close relationship between macro tree transducers and context-free tree grammars we refer the reader to the discussion in [22, Section 3.2]. For top-down tree transducers we denote the class of induced translations by T; in the deterministic and the total deterministic case the denotation of the class of translations is preceded by D and D_t, respectively.

A *macro tree-to-string transducer* $\mathfrak{M}$ is defined as an easy variation of the macro tree transducer in the following sense. The output alphabet of $\mathfrak{M}$ is a usual alphabet, the empty string may be a right-hand side of a rule, and, in the definition of RHS(Q, Δ, m, n), (ii) is replaced by (ii'): for $\zeta_1, \zeta_2 \in$ RHS, $\zeta_1\zeta_2 \in$ RHS. For macro tree-to-string transducers determinism and total determinism are defined in the same way as for macro tree transducers. The definition of the OI-derivation relation and of the induced OI-translation are also carried over. The class of translations induced by (deterministic, total deterministic) macro tree-to-string transducers is denoted by ($yDMT_{OI}$, yD_tMT) yMT_{OI}. It is easy to see that $yMT_{OI} = \text{yield}(MT_{OI})$ and similar for the deterministic and total deterministic case.

If every state of a macro tree-to-string transducer $\mathfrak{M}$ has rank 1, then $\mathfrak{M}$ is a *top-down tree-to-string transducer* (actually, this coincides with [19, Definition 3.1.5]). The class of translations of (deterministic, total deterministic) top-down tree-to-string transducers is denoted by (yDT, yD_tT) yT.

A top-down tree transducer in which every rule has the form

$$q(\sigma(x_1, \ldots, x_m)) \to \sigma(q_1(x_1), \ldots, q_m(x_m))$$

for some states $q_1, \ldots, q_m$, is called a *finite state tree automaton.* The class of translations induced by finite state tree automata is denoted by FTA. Actually, the translation induced by a finite state tree automaton is a partial identity on the set of trees over the input alphabet. Recall that dom(FTA) = RECOG.

For technical reasons we also consider *macro tree transducers with initial term.* Such a transducer $\mathfrak{M}$ is a macro tree transducer in which the initial nonterminal is replaced by an initial term $\alpha \in \text{RHS}(Q, \emptyset, 1, 0)$ (e.g.,

$$\alpha = q(x_1, q'(x_1), q''(x_1, q'(x_1))),$$

where q, q', q'' are states of rank 3, 1, and 2, respectively). The OI-translation induced by $\mathfrak{M}$, also denoted by $\tau_{OI}(\mathfrak{M})$, is the set $\{(s, t) \mid s \in T_\Sigma, t \in T_\Delta, \text{ and } \alpha[x_1 \leftarrow s] \Rightarrow^*_{\mathfrak{M}} t\}$. The class of translations induced by (deterministic, total deterministic) macro tree transducers with initial term is denoted by ($DMT_{OI,init}$, $D_tMT_{OI,init}$) $MT_{OI,init}$. In Theorem 3.22, we will show that allowing initial terms in macro tree transducers does not enlarge the induced class of translations, i.e., $MT_{OI,init} = MT_{OI}$, which also holds for the total deterministic case.

3. Grammars with storage

As explained in the introduction we want to set up all our main results about the implementation of nested recursion (on pushdown and nested stack devices) in a general framework. For this purpose we adopt the idea of Scott suggested in [38] where he strongly advocates the separation of the concepts of program and storage type (called machine by Scott). He considers the usual iterative, sequential programs which can be drawn as a flowchart. A storage type consists in the Scottian sense of an input function, mapping the input set into the set of configurations (of the storage type), a collection of tests and instructions on configurations, and an output function, mapping the set of configurations into the output set. Since the program uses the tests and instructions of the storage type, it can be executed on the storage type in the usual way. In [16] the idea of separating program and storage type is picked up and a generalized concept is suggested which we want to outline briefly.

Instead of the sequential flow of control, any control structure inherent in well-known generating formalisms of (tree) language theory (such as regular tree grammars, context-free grammars, macro grammars) may be used as 'programming language'. The sequential, iterative programs are reobtained by programming with regular (string) grammars. Now, how is a grammar G connected to a storage type S? Considering a sentential form ξ of G, the interaction with S is realized by attaching to every occurrence of a nonterminal A in ξ a configuration c of S. Hence, in a sentential form of such a 'grammar with storage type' (G, S), objects of the form $A(c)$ occur. Now, a rule of (G, S) specifies besides the usual components of a rule of G, how a configuration is tested and, if the test is positive, how it is transformed. For example, if we consider context-free grammars as control structure, then a rule r could look like $A \to$ **if** b **then** $B(f_1)C(f_2)$, where b is a test on configurations and f_1 and f_2 are instructions transforming them. The rule r is applicable to $A(c)$ (occurring in a sentential form ξ), if the test b proves positive for c. The result of the application is obtained by replacing $A(c)$ in ξ by $B(f_1(c))C(f_2(c))$ where $f_1(c)$ and $f_2(c)$ are the configurations resulting from the transformation of c by the instructions f_1 and f_2, respectively.

Hence, the sequence of steps of a program on a storage type (in the sense of Scott) now turns into a derivation of a 'grammar with storage type': the grammar (viewed as a, generalized, program) can be executed on the storage type. But how is the communication of the storage with the outer world reflected in such grammars? As in the old concept the input is realized by a partial function from the input set into the set of configurations. (In contrast to Scottian storage types we will allow a set of such partial input functions; then, every grammar can pick its own from this set.) The managing of output is taken over by the terminal symbols of the grammar. Hence, the output is not computed from the storage at the end of the computation, but in every derivation step of the grammar a piece of output (string or tree) is produced. Since a grammar with storage induces a translation from the input set to the output set by means of a derivation relation as explained above,

we call such a device an $X(S)$-transducer, where X is the used 'programming language', i.e., the type of grammar, and S indicates the used storage type.

In this section we will provide the basic definitions of storage type and $X(S)$-transducer (Section 3.1). Then, in Section 3.2, macro tree transducers are expressed in the $X(S)$-transducer formalism by using context-free tree grammars as programming language and trees as storage type (denoted by TR). Vice versa, we also show that any $X(S)$-transducer with a context-free tree grammar as program can be understood as a macro tree transducer which works on special trees. Finally, Section 3.3 provides the definition of the pushdown operator on storage types (denoted by P), which is needed for our simulation of nested recursion. In particular, two special storage types (P(TR) and P^2(TR)) are discussed which are involved in our pushdown machines for the macro tree transducer. Throughout this section we will prove several elementary properties of the involved formalisms.

3.1. Storage types and $X(S)$-transducers

Before defining a storage type we describe its 'operational features', i.e., the configurations and the predicate and instruction symbols together with their meaning (where the predicate symbols are used to build up the tests). We call this list of operational features a datatype (only needed for the definition of storage type).

3.1. Definition. A *datatype* is a tuple (C, P, F, m), where C is the set of configurations, P is the set of predicate symbols, F is the set of instruction symbols ($P \cap F = \emptyset$), and m is the meaning function which interprets every $p \in P$ and $f \in F$ as a total function $m(p): C \to \{\textbf{true}, \textbf{false}\}$ and as a partial function $m(f): C \to C$, respectively.

In the usual way m is extended to the set BE(P) of boolean expressions over P, where **and**, **or**, and **not** denote conjunction, disjunction, and negation, respectively, **true** and **false** are constants, and P is the set of boolean variables. Thus, for every $b \in \mathrm{BE}(P)$, $m(b)$ is a total function $C \to \{\textbf{true}, \textbf{false}\}$. Elements of BE($P$) are also called tests.

Remarks. (i) We will often say 'predicate' and 'instruction' rather than 'predicate symbol' and 'instruction symbol', respectively.

(ii) As usual, p_1 **and** $p_2 \ldots$ **and** p_n may be abbreviated by $p_1 p_2 \ldots p_n$.

(iii) If $n = 0$, then the meta-expression p_1 **and** $\ldots$ **and** p_n denotes the expression **true**; dually, p_1 **or** $\ldots$ **or** p_n denotes **false** for $n = 0$.

(iv) We extend m to F^* as usual by defining

$$m(f_1 \ldots f_n)(c) = m(f_n)(\ldots m(f_1)(c) \ldots) \quad \text{and} \quad m(\lambda)(c) = c.$$

Thus, for $g \in F^*$, $m(g)$ is a partial function $C \to C$.

(v) If a predicate is a composite symbol such as "top $= \gamma$", then we write "top $\neq \gamma$" rather than "**not**(top $= \gamma$)".

From a datatype we obtain a storage type by adding an input device that specifies a set of input elements and a set of partial functions from input elements to configurations. Each such partial function indicates a way in which the input elements can be coded as configurations.

3.2. Definition. A *storage type* S is a tuple (C, P, F, m, I, E), where (C, P, F, m) is a datatype, I is a set called the input set of S, and E is a set of partial functions $e: I \to C$, every $e \in E$ is called an (input) encoding of S.

In the rest of this paper S denotes the storage type (C, P, F, m, I, E), if not specified otherwise.

Now we define the concept of $X(S)$-transducer. Let MOD be the set {regular, context-free, macro, regular tree, context-free tree} of modifiers, which are also abbreviated by {REG, CF, MAC, RT, CFT}. We use X to range over MOD. Thus, if X is not specified, then any modifier can be substituted for X. Recall from the preliminaries that an X-grammar is denoted by $(N, \Delta, A_{\text{in}}, R)$, where N is the (possibly ranked) alphabet of nonterminals, Δ is the (possibly ranked) alphabet of terminals, A_{in} is the initial term, and R is the set of rules.

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

(i) N, Δ, and A_{in} are the alphabet of nonterminals, the alphabet of terminals, and the initial term, respectively, as for a usual X-grammar; depending on X, the alphabets N and Δ may be ranked,

(ii) $e \in E$ is the encoding, and

(iii) R is the finite set of rules of the form $\Theta \to$ **if** b **then** ζ, where $\Theta \to w$ is a rule of a usual X-grammar, $b \in \text{BE}(P)$ is a test, and ζ is obtained from w by replacing every occurrence of a nonterminal B by $B(f)$ for some $f \in F$ (i.e., $\zeta \in w[B \leftarrow B(F); B \in N]$). ζ is called the right-hand side tree (or string) of the rule. If A is the nonterminal in Θ, then $\Theta \to$ **if** b **then** ζ is called an (A, b)-rule.

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

Note that by $B(f)$ is meant a string of length 4 over the set containing N, F, and the left and right parentheses. We also note that, for a rule $\Theta \to$ **if** b **then** ζ of a CFT(S)- or MAC(S)-transducer, Θ has the form $A(y_1, \ldots, y_k)$ with $A \in N_k$. In the other cases Θ is just a nonterminal.

For convenience, we abbreviate a rule like $\Theta \to$ **if true then** ζ by $\Theta \to \zeta$. Sometimes it is also convenient to have rules of the structure **if_then_else** available. Formally, the construct $\Theta \to$ **if** b **then** ζ_1 **else** ζ_2 abbreviates the two rules $\Theta \to$ **if** b **then** ζ_1 and $\Theta \to$ **if not** b **then** ζ_2.

For $X \in \{\text{REG}, \text{RT}\}$, an $X(S)$-transducer is called regular and, in particular, REG(S)-transducers are also called iterative and sequential.

From the transducer point of view the nonterminals and the terminals of an $X(S)$-transducer are called states and output symbols, respectively. In fact, a REG(S)-transducer is very close to a usual sequential, iterative machine which consists of a finite control, an input device, a storage of type S and an output tape. A rule $A \to$ **if** b **then** $wB(f)$ of a REG(S)-transducer (with $A, B \in N$, $w \in \Delta^*$) should be read as: if the transducer is in state A and b holds for its storage configuration, then the transducer outputs w, goes into state B, and applies f to its storage configuration. Note that we will characterize macro tree-to-string transducers by two types of such machines using a pushdown of pushdowns of trees and a nested stack of trees, respectively, as storage. Also 1-way S-automata can be expressed in terms of REG(S)-transducers. Such automata use a one way input tape and a storage of type S. Clearly, the language class accepted by nondeterministic 1-way S-automata and the class of ranges of REG(S)-transducers are the same (cf. [16] and Lemma 3.9 of this paper). We note that deterministic 1-way S-automata do not correspond to deterministic REG(S)-transducers. For this reason in [16] two notions of determinism are defined: transducer and acceptor determinism. (For the connections between transducers and (checking) acceptors in general, see [13].)

The close connection between $X(S)$-transducers and X-grammars becomes more apparent when defining the translation induced by an $X(S)$-transducer $\mathfrak{M}$. Namely, an infinite X-grammar $G(\mathfrak{M})$ is associated with $\mathfrak{M}$, in which an S-configuration is part of each nonterminal. A rule $\Theta \to$ **if** b **then** ζ is represented in $G(\mathfrak{M})$ by a (possibly infinite) set of rules $\Theta_c \to \zeta_c$, where c is an S-configuration for which $m(b)(c) =$ **true**. The formal details are provided in the following definition.

3.4. Definition. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be an $X(S)$-transducer. The *X-grammar* $G(\mathfrak{M}) = (N', \Delta, A'_{\text{in}}, R')$ *associated with* $\mathfrak{M}$ is defined by $N' = N(C)$, A'_{in} is any element of N' (A'_{in} is irrelevant), and R' is obtained as follows: if $\Theta \to$ **if** b **then** ζ is a rule in R, then, for every $c \in C$ such that $m(b)(c) =$ **true** and every instruction occurring in ζ is defined on c, the rule $\Theta_c \to \zeta_c$ is in R', where $\Theta_c = \Theta[A \leftarrow A(c); A \in N]$ and $\zeta_c = \zeta[B(f) \leftarrow B(m(f)(c)); B(f) \in N(F)]$.

Note that $B(m(f)(c))$ means $B(c')$ with $c' = m(f)(c)$. Furthermore, we note that $N(C) = \{A(c) \mid A \in N, c \in C\}$, where $A(c)$ denotes a string of length 4 over the set containing N, C, and the left and right parentheses.

By means of the associated grammar we define the translation induced by an $X(S)$-transducer. Recall the definition of X-form from the preliminaries.

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

(i) The *set of sentential forms of* $\mathfrak{M}$, denoted by $\text{SF}(\mathfrak{M})$, is the set of X-forms over $N(C)$ and Δ, i.e., $\text{SF}(\mathfrak{M}) = F_X(N(C), \Delta)$.

(ii) The *derivation relation of* $\mathfrak{M}$ $\Rightarrow_{\mathfrak{M}} \subseteq \text{SF}(\mathfrak{M}) \times \text{SF}(\mathfrak{M})$ is defined by $\Rightarrow_{\mathfrak{M}} = \Rightarrow_{G(\mathfrak{M})}$. For $X \in \{\text{CFT}, \text{MAC}\}$ we choose the OI-derivation relation of the associated grammar.

(iii) The *translation induced by* $\mathfrak{M}$, denoted by $\tau(\mathfrak{M})$, is the set

$$\{(u, v) \mid u \in I,\ v \in F_X(\emptyset, \Delta), \text{ and } A_{\text{in}}(e(u)) \Rightarrow_{\mathfrak{M}}^{*} v\},$$

where, for every $c \in C$, $A_{\text{in}}(c) = A_{\text{in}}[A \leftarrow A(c);\ A \in N]$. (Recall that I is the input set of S and that A_{in} is a term in $F_X(N, \emptyset)$.)

Note that $F_X(\emptyset, \Delta) = T_\Delta$ if $X \in \{\text{CFT}, \text{RT}\}$, and Δ^* otherwise. Furthermore, note that $\tau(\mathfrak{M}) \subseteq I \times F_X(\emptyset, \Delta)$; in particular, the domain of the encoding e delimits the set of successful input elements for $\mathfrak{M}$, i.e., $\text{dom}(\tau(\mathfrak{M})) \subseteq \text{dom}(e)$. Intuitively, range($e$) is also called the set of *initial configurations* of $\mathfrak{M}$. As usual, two transducers $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are *equivalent* if $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$.

The class of translations induced by $X(S)$-transducers is denoted by $X(S)$. For deterministic transducers the corresponding translation class is denoted by $DX(S)$. Obviously, for every storage type S, $\text{REG}(S) \subseteq \text{CF}(S) \subseteq \text{MAC}(S)$ and $\text{RT}(S) \subseteq \text{CFT}(S)$. Moreover, $\text{yield}(\text{RT}(S)) = \text{CF}(S)$ and $\text{yield}(\text{CFT}(S)) = \text{MAC}(S)$ (recall the definition of 'yield' on classes of relations from the preliminaries; recall also that we allow every ranked alphabet to contain a special symbol of rank 0, which is interpreted by 'yield' as the empty string).

At this point we think that an easy example is helpful for the reader's understanding of the definitions which appeared so far in this chapter. The storage type count-down is taken from [16].

3.6. Example. The storage type *count-down* is determined by the tuple

$$(C, P, F, m, I, E) = (\text{Int}, \{\text{null}\}, \{\text{decr}\}, m, \text{Int}, \{\lambda n \in \text{Int}.n\}),$$

where Int is the set of nonnegative integers $\{0, 1, 2, \ldots\}$ and, for every $n \in \text{Int}$,

$$m(\text{null})(n) = \textbf{true} \text{ iff } n = 0,$$

$$m(\text{decr})(n) = \begin{cases} n-1 & \text{if } n \geq 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The CF(count-down)-transducer $\mathfrak{M} = (\{A_{\text{in}}\}, e, \{a\}, A_{\text{in}}, R)$ contains the rule

$$A_{\text{in}} \rightarrow \textbf{if null then } a \textbf{ else } A_{\text{in}}(\text{decr})A_{\text{in}}(\text{decr}).$$

Note that this is, in fact, an abbreviation of the two rules

$$A_{\text{in}} \rightarrow \textbf{if null then } a,$$

$$A_{\text{in}} \rightarrow \textbf{if notnull then } A_{\text{in}}(\text{decr})A_{\text{in}}(\text{decr}).$$

Obviously, $\mathfrak{M}$ is deterministic.

The encoding e of $\mathfrak{M}$ is the identity function $\lambda n \in \text{Int}.n$.

The CF-grammar $G(\mathfrak{M})$ which is associated with $\mathfrak{M}$ is specified by $(N', \{a\}, A, R')$, where $N' = \{A_{\text{in}}(n) \mid n \in \text{Int}\}$, and R' contains, for every $n \geq 1$, the rule $A_{\text{in}}(n) \rightarrow A_{\text{in}}(n-1)A_{\text{in}}(n-1)$ and the rule $A_{\text{in}}(0) \rightarrow a$.

Note that R' is an infinite set.

By considering the input element 2, which is encoded by e as the configuration 2, $\mathfrak{M}$ can compute as follows.

$$A_{\text{in}}(2) \Rightarrow A_{\text{in}}(1)A_{\text{in}}(1) \Rightarrow A_{\text{in}}(0)A_{\text{in}}(0)A_{\text{in}}(1)$$
$$\Rightarrow aA_{\text{in}}(0)A_{\text{in}}(1) \Rightarrow aaA_{\text{in}}(1) \Rightarrow^3 aaaa.$$

Obviously, $\tau(\mathfrak{M}) = \{(n, a^m) \mid n \geqslant 0 \text{ and } m = 2^n\} \subseteq \text{Int} \times \{a\}^*$.

This concludes our basic definitions concerning storage type and $X(S)$-transducer. In the rest of this section we discuss some elementary properties of $X(S)$-transducers.

As can be seen from our definition of $X(S)$-transducer, the application of a rule to a nonterminal and a configuration c automatically implies that several instructions are applied to (copies of) c, namely those instructions which occur in the right-hand side of the applied rule. However, very often we wish to consider transducers that, during a derivation step, may transform some copies of the configuration, but also may leave some others unchanged. This effect can be obtained by an instruction symbol in the storage type that denotes the identity on C. Such an instruction will also be called an *identity*. If a storage type has no identity, it can, of course, be added as follows.

3.7. Definition. Let id be a symbol not in $P \cup F$. Then, S_{id} is the storage type (C, P, F', m', I, E), where $F' = F \cup \{\text{id}\}$ and, for every $c \in C$, $m'(\text{id})(c) = c$, and $m' = m$ on P and F.

In particular, we may consider the trivial storage type S_0, which only has an identity as instruction and no predicate. Clearly, $X(S_0)$-transducers do not use the storage, i.e., they are just X-grammars. In this way the $X(S)$-transducer formalism covers also the X-grammars.

3.8. Definition. The *trivial storage type* $S_0 = (C, P, F, m, I, E)$ is defined by $C = \{c\}$, where c is an arbitrary object, $P = \emptyset$, $F = \{\text{id}\}$, and $m(\text{id})(c) = c$, $I = \{c\}$, and $E = \{m(\text{id})\}$.

Recall that X denotes the class of languages which is generated by X-grammars.

3.9. Lemma. $\text{range}(X(S_0)) = X$.

Proof. ($\text{range}(X(S_0)) \subseteq X$): Each test of a rule of an $X(S_0)$-transducer is either **true** or **false** on the only configuration. From the rules with always **true** tests, the rules for the X-grammar are obtained by dropping everything which refers to S_0.

($X \subseteq \text{range}(X(S_0))$): From a rule like $\Theta \to w$ of an X-grammar, the rule $\Theta \to$ **if true then** ζ is constructed for the corresponding $X(S_0)$-transducer, where $\zeta = w[A \leftarrow A(\text{id}); A \in N]$. □

Thus, in particular, range(CFT(S_0)) is the class of context-free tree languages (with OI-derivation mode) and range(MAC(S_0)) is the class of OI-macro languages.

There is a special form of tests which will be frequently used in the sequel.

3.10. Definition. Let $\mathfrak{M}$ be an $X(S)$-transducer and let $P_{\mathfrak{M}}$ be the set of predicate symbols used in the tests of rules of $\mathfrak{M}$. $\mathfrak{M}$ is in *$P_{\mathfrak{M}}$-standard test form* if there is a sequence $\langle p_1, \ldots, p_n \rangle$ of different predicate symbols with $n \geq 0$ and $P_{\mathfrak{M}} = \{p_1, \ldots, p_n\}$ such that every test of $\mathfrak{M}$ has the form p_1' **and** ... **and** p_n', where $p_i' \in \{p_i, \textbf{not}p_i\}$. Such a test is called a *$P_{\mathfrak{M}}$-standard test.* If $P_{\mathfrak{M}}$ is not important, then we just say 'standard' rather than '$P_{\mathfrak{M}}$-standard'.

Before we prove that every $X(S)$-transducer can be transformed into an equivalent $X(S)$-transducer in standard test form, we recall the disjunctive normal form of boolean expressions over a finite set U of boolean variables. The disjunctive normal form is a disjunction of terms; each term consists of a conjunction of negated or nonnegated variables such that every variable occurs exactly once (in either negated or nonnegated form). For instance, if $U = \{p_1, p_2, p_3\}$, then the boolean expression (p_1 **and** p_2 **and not**p_3) **or** (p_3 **and not**p_2 **and** p_1) is in disjunctive normal form. Note that the disjunctive normal form of a boolean expression without boolean variables is either **true** or **false**. Since disjunction and conjunction are commutative, the disjunctive normal form is unique up to reordering. (The notion of totality will be defined immediately after the lemma.)

3.11. Lemma. *Let $\mathfrak{M}$ be an $X(S)$-transducer and let $P_{\mathfrak{M}}$ be the set of predicate symbols used in tests of $\mathfrak{M}$. For every finite subset P' of the predicate set of S such that $P_{\mathfrak{M}} \subseteq P'$ there is an equivalent $X(S)$-transducer in P'-standard test form. Determinism and totality are preserved.*

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ and let $P' = \{p_1, \ldots, p_n\}$ for some $n \geq 0$ such that $P_{\mathfrak{M}} \subseteq P'$. Let $z = \langle p_1, \ldots, p_n \rangle$.

Construct the $X(S)$-transducer $\mathfrak{M}' = (N, e, \Delta, A_{\text{in}}, R')$ as follows. If $\Theta \to$ **if** b **then** ζ is in R, then

(1) transform b into disjunctive normal form b' over P' such that $b' = b_1$ **or** ... **or** b_r for some $r \geq 0$; order the negated and nonnegated predicates in every b_i with $i \in [r]$ according to their order in z;

(2) for every $i \in [r]$, add the rule $\Theta \to$ **if** b_i **then** ζ to R'. (Note that if $b' =$ **false**, then no rule is added.)

Obviously, $\tau(\mathfrak{M}') = \tau(\mathfrak{M})$ and determinism and totality are preserved. □

As discussed in the introduction, our main interest is focussed on the total deterministic version of the macro tree transducer. It produces for every input tree (e.g., syntax tree of a program) exactly one output tree (meaning of the syntax tree). Since we want to model the macro tree transducer as an $X(S)$-transducer (see Theorem 3.19), we define totality also in general for $X(S)$-transducers.

3.12. Definition. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be a (deterministic) $X(S)$-transducer. $\mathfrak{M}$ is *total* if $\text{dom}(\tau(\mathfrak{M})) = \text{dom}(e)$.

A total deterministic $X(S)$-transducer is referred to as a $D_tX(S)$-transducer. The class of translations induced by $D_tX(S)$-transducers is denoted by $D_tX(S)$.

Naturally, we have to convince the reader that a $D_tX(S)$-transducer does not only compute an output for every initial configuration, but in fact just one output. Hence, we have to prove that a deterministic $X(S)$-transducer computes a partial function. For $X \in \text{MOD} - \{\text{CFT}, \text{MAC}\}$ this fact is intuitively clear. But what about the other cases? To show that DCFT(S)-transducers (and DMAC(S)-transducers) compute partial functions, we define the notion of confluency for relations (cf. [42]) and show that the derivation relations of DCFT(S)-transducers are confluent (this approach is suggested in [7] for the macro tree transducer).

3.13. Definition. Let $\Rightarrow$ be a binary relation on a set A. $\Rightarrow$ is *confluent* if, for every $u, v_1, v_2 \in A$, it holds that if $u \Rightarrow^* v_1$ and $u \Rightarrow^* v_2$, then there is a $v' \in A$ such that $v_1 \Rightarrow^* v'$ and $v_2 \Rightarrow^* v'$.

3.14. Lemma. *For every* $DX(S)$*-transducer* $\mathfrak{M}$ *with* $X \in \{\text{CFT}, \text{MAC}\}$, $\Rightarrow_{\mathfrak{M}}$ *is confluent.*

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be a $DX(S)$-transducer. First we prove the following claim.

Claim 1. *For every* $\xi, \xi_1, \xi_2 \in \text{SF}(\mathfrak{M})$ *with* $\xi_1 \neq \xi_2$, *if* $\xi \Rightarrow_{\mathfrak{M}} \xi_1$ *and* $\xi \Rightarrow_{\mathfrak{M}} \xi_2$, *then there is a* $\xi' \in \text{SF}(\mathfrak{M})$ *such that* $\xi_1 \Rightarrow_{\mathfrak{M}} \xi'$ *and* $\xi_2 \Rightarrow_{\mathfrak{M}} \xi'$.

Proof of Claim 1. Since, by definition of $\Rightarrow_{\mathfrak{M}}$, $\mathfrak{M}$ derives in an outside-in fashion and since $\mathfrak{M}$ is deterministic, the only way to derive from ξ two different sentential forms ξ_1 and ξ_2 is by applying rules to two independent nonterminals, i.e., there is a $\zeta \in F_X(N(C), \Delta \cup \{z_1, z_2\})$ in which z_1 and z_2 occur exactly once, there are $\zeta_1, \zeta_2 \in \{A(c)(t_1, \ldots, t_k) \mid A \in N_k,\ k \geq 0,\ c \in C,\ t_1, \ldots, t_k \in F_X(N(C), \Delta)\}$, and there are $\zeta_1', \zeta_2' \in F_X(N(C), \Delta)$ such that

$$\xi = \zeta[z_1 \leftarrow \zeta_1, z_2 \leftarrow \zeta_2],$$

$$\zeta_1 \Rightarrow_{\mathfrak{M}} \zeta_1' \quad \text{and} \quad \zeta_2 \Rightarrow_{\mathfrak{M}} \zeta_2',$$

$$\xi_1 = \zeta[z_1 \leftarrow \zeta_1', z_2 \leftarrow \zeta_2], \qquad \xi_2 = \zeta[z_1 \leftarrow \zeta_1, z_2 \leftarrow \zeta_2'].$$

Then define $\xi' = \xi[z_1 \leftarrow \zeta_1', z_2 \leftarrow \zeta_2']$.

Obviously, $\xi_1 \Rightarrow_{\mathfrak{M}} \xi'$ and $\xi_2 \Rightarrow_{\mathfrak{M}} \xi'$. This proves Claim 1. It is easy to show that if the claim holds for $\Rightarrow_{\mathfrak{M}}$, then it also holds for $\Rightarrow_{\mathfrak{M}}^*$ (proof by induction on the length of the derivations). □

3.15. Theorem. $\text{DCFT}(S) \cup \text{DMAC}(S) \subseteq \text{PF}$, *where* PF *denotes the class of partial functions.*

Proof. This immediately follows from the confluency of the derivation relation of DCFT(S)- and DMAC(S)-transducers. □

In fact, since, for every storage type S, $\mathrm{DRT}(S) \subseteq \mathrm{DCFT}(S)$ and $\mathrm{DREG}(S) \subseteq \mathrm{DCF}(S) \subseteq \mathrm{DMAC}(S)$, we have actually proved now, for every $X \in \mathrm{MOD}$, that $DX(S) \subseteq \mathrm{PF}$.

3.16. Corollary. *For every* $\mathrm{D_tX}(S)$*-transducer* $\mathfrak{M}$ *with encoding* e *and terminal alphabet* Δ, $\tau(\mathfrak{M})$ *is a total function from* $\mathrm{dom}(e)$ *to* $F_X(\emptyset, \Delta)$.

Until now, this section contains only definitions of general concepts and the reader may already long for something more concrete. So let us hurry to fulfil our promise and describe macro tree transducers in the formalism of $X(S)$-transducers.

3.2. Macro tree transducers and CFT(S)*-transducers*

In this section we will show how the macro tree transducer can be expressed in the CFT(S)-transducer formalism (cf. Theorem 3.19) and, furthermore, that any such CFT(S)-transducer can be considered as a macro tree transducer which works on approximations of S-configurations (cf. Theorem 3.26).

Intuitively, macro tree transducers combine the features of CFT-grammars and of top-down tree transducers. The latter device serves the purpose of controlling the derivation in a syntax-directed manner by means of an input (sub-)tree. Viewing such (input-)trees together with the ability of testing the label of the root of a tree and of selecting subtrees of them, as a storage type, denoted by TR, it is intuitively clear that a top-down tree transducer is an RT(TR)-transducer and a macro tree transducer is a CFT(TR)-transducer. We want to make this precise.

3.17. Definition. The *tree storage type*, denoted by TR, is the storage type (C, P, F, m, I, E), where

(i) $C = T_\Omega$, where Ω is the infinite ranked set defined in the preliminaries (note that, for every $k \geq 0$, Ω_k is infinite);

(ii) $P = \{\mathrm{root} = \sigma \mid \sigma \in \Omega\}$;

(iii) $F = \{\mathrm{sel}_i \mid i \geq 1\}$;

(iv) and, for every $t \in T_\Omega$,

$$m(\mathrm{root} = \sigma)(t) = \textbf{true} \text{ iff } t = \sigma(t_1, \ldots, t_k) \text{ for some } t_1, \ldots, t_k \in T_\Omega \text{ where } \sigma \text{ is of rank } k,$$

$$m(\mathrm{sel}_i)(t) = \begin{cases} t_i & \text{if } t = \sigma(t_1, \ldots, t_k) \text{ for some } \sigma \text{ of rank } k \text{ and some } t_1, \ldots, t_k \in T_\Omega, \text{ with } i \leq k, \\ \text{undefined} & \text{otherwise;} \end{cases}$$

(v) $I = T_\Omega$; and

(vi) $E = \{e \mid e : I \to C \text{ is the identity on } T_\Sigma \text{ for some finite subset } \Sigma \text{ of } \Omega\}$.

Note that the purpose of the encoding e of an X(TR)-transducer $\mathfrak{M}$ is to specify the ranked input alphabet Σ of $\mathfrak{M}$. We also note that 'in practice' an X(TR)-transducer will not actually attach trees to its nonterminals, but rather keep a global input tree and attach to its nonterminals pointers to subtrees of that tree (cf. the Introduction).

Without loss of generality we can assume that the tests occurring in an X(TR)-transducer are of the form root $= \sigma$ (i.e., boolean combinations are not needed) and that every instruction in the right-hand side of a rule is always applicable. This is proved in the next lemma.

3.18. Lemma. *Let $\mathfrak{M}$ be an X(TR)-transducer with encoding e and let* $\mathrm{dom}(e) = T_\Sigma$ *for some ranked alphabet Σ. Then there is an equivalent X(TR)-transducer $\mathfrak{M}'$ such that for every rule of $\mathfrak{M}$*

- *the test is of the form* root $= \sigma$ *for some $\sigma \in \Sigma$,*
- *if the test has the form* root $= \sigma$ *and σ has rank k, then only instructions* sel_i *with $i \leqslant k$ occur in the right-hand side of the rule.*

Determinism and totality are preserved.

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$. Since $\mathrm{range}(e) = T_\Sigma$ and the $m(\mathrm{sel}_i)$ transform T_Σ into itself, we can replace every predicate root $= \delta$ with $\delta \notin \Sigma$ occurring in R by **false**. Hence, we can assume that in the rules of $\mathfrak{M}$ only predicates of P_Σ occur, where $P_\Sigma = \{\mathrm{root} = \sigma \mid \sigma \in \Sigma\}$. Then, by Lemma 3.11, we can assume that $\mathfrak{M}$ is in P_Σ-standard test form. Of course, the rules with tests in which every predicate is negated are never applicable and can be thrown out. (They are never applicable, because in the tests every symbol of Σ is tested.) Moreover, rules with contradictory tests, i.e., tests in which root $= \sigma_1$ and root $= \sigma_2$ occur, for some $\sigma_1, \sigma_2 \in \Sigma$ with $\sigma_1 \neq \sigma_2$, are never applicable and can also be deleted. In the tests of the remaining rules we just delete the negated predicates and obtain an equivalent X(TR)-transducer $\mathfrak{M}' = (N, e, \Delta, A_{\mathrm{in}}, R')$ such that in every rule of $\mathfrak{M}'$ the test has the form root $= \sigma$ for some $\sigma \in \Sigma$.

To accomplish the second condition, we just delete those rules of $\mathfrak{M}'$ in which the test is of the form root $= \sigma$ with σ of rank k and in which the right-hand side contains an instruction sel_i with $i > k$. By definition of the derivation relation of an $X(S)$-transducer, these rules are never applicable. Hence, we obtain an equivalent X(TR)-transducer with the desired restrictions. It is obvious that both constructions of this proof preserve determinism and totality. □

The next theorem establishes the relationship between macro tree transducers and CFT(TR)-transducers. However, whereas the first transducer class starts with a single state, viz. q^{in}, the latter class can start with an initial term, viz. A_{in}. Thus, before proving that this does not provide a difference in the transformational power (cf. Theorem 3.22), we first present the 'straightforward connections'.

3.19. Theorem

$$MT_{OI} = CFT_1(TR) \quad \textit{and} \quad MT_{OI,init} = CFT(TR).$$

Determinism and total determinism are preserved and yield *can be applied to every equation, i.e.,* $yMT_{OI} = MAC_1(TR)$, *etc.*

Proof. ($MT_{OI} \subseteq CFT_1(TR)$): Let $\mathfrak{M} = (Q, \Sigma, \Delta, q^{in}, R)$ be a macro tree transducer. Define the $CFT_1(TR)$-transducer $\mathfrak{M}' = (N, e, \Delta, A_{in}, R')$ by

(i) $N = \{q^{(n-1)} \mid q \in Q_n \text{ with } n \geq 1\}$;

(ii) $e: T_\Omega \to T_\Sigma$ is the identity on T_Σ;

(iii) $A_{in} = q^{in}$; and

(iv) if $q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_n) \to \zeta$ is in R, then $q(y_1, \ldots, y_n) \to$ **if** root $=$ σ **then** ζ' is in R', where ζ' is obtained from ζ by replacing every construct like $q'(x_i, \ldots)$ by $q'(\text{sel}_i)(\ldots)$.

Then, obviously, there is a one-to-one correspondence between the derivations of $\mathfrak{M}$ and $\mathfrak{M}'$: in each sentential form of a derivation of $\mathfrak{M}$, every construct $q(s, \ldots)$ with $q \in Q$ and $s \in T_\Sigma$ has to be replaced by $q(s)(\ldots)$ to obtain a derivation of $\mathfrak{M}'$, and vice versa. Thus $\tau(\mathfrak{M}') = \tau_{OI}(\mathfrak{M})$. Determinism is preserved by the construction and if $\mathfrak{M}$ is total deterministic, then $\text{dom}(\tau_{OI}(\mathfrak{M})) = T_\Sigma$ (see [22]) and hence, $\mathfrak{M}'$ is also total. It should be clear that if $\mathfrak{M}$ is a macro tree-to-string transducer, then $\mathfrak{M}'$ is an equivalent $MAC_1(TR)$-transducer, and determinism and total determinism are also preserved in this case.

($CFT_1(TR) \subseteq MT_{OI}$): Let $\mathfrak{M}' = (N, e, \Delta, A_{in}, R')$ be a $CFT_1(TR)$-transducer and let $\text{dom}(e) = T_\Sigma$ for some ranked alphabet Σ. We can assume that the rules in R fulfil the conditions of Lemma 3.18. It is then easy to construct a macro tree transducer $\mathfrak{M}$ such that $\mathfrak{M}'$ and $\mathfrak{M}$ correspond to each other in the same way as in the first part of this proof. Again determinism is preserved.

If $\mathfrak{M}'$ is total deterministic and, for a nonterminal $A \in N_k$ with $k \geq 0$ and a test root $= \sigma$ with $\sigma \in \Sigma$, there is no $(A, \text{root} = \sigma)$-rule in R', then we add the rule

$$A(y_1, \ldots, y_k) \to \textbf{if root} = \sigma \textbf{ then } d$$

for some $d \in \Delta_0$ to R'. This additional rule does not change the translation of $\mathfrak{M}$, but now $\mathfrak{M}$ is a total deterministic macro tree transducer. Clearly, for a $MAC_1(TR)$-transducer this construction delivers a macro tree-to-string transducer, and determinism and total determinism are preserved as above.

($MT_{OI,init} = CFT(TR)$): The one-to-one correspondence between the initial terms of a macro tree transducer (with initial term) and of a CFT(TR)-transducer is very similar to the correspondence between sentential forms in the proof of $MT_{OI} \subseteq CFT_1(TR)$. After recognizing this, the proof of both directions of $MT_{OI,init} = CFT(TR)$ are the same as those in which no initial term is involved. Again determinism and total determinism are preserved. □

Since top-down tree transducers and RT(TR)-transducers are subclasses of macro tree transducers and CFT(TR)-transducers, respectively, we immediately obtain a

description of top-down tree transducers in terms of $X(S)$-transducers. Note that the initial term of an RT(TR)-transducer is a single nonterminal.

3.20. Corollary

$$T = \mathrm{RT(TR)}, \qquad \mathrm{DT} = \mathrm{DRT(TR)}, \qquad \mathrm{D_tT} = \mathrm{D_tRT(TR)},$$

and yield *can be applied to every equation, i.e.,* $\mathrm{yT} = \mathrm{CF(TR)}$, *etc.*

Proof. The first two parts of the proof of Theorem 3.19 can be taken over literally. □

Now we show that allowing initial terms rather than only one nonterminal does not increase the power of CFT(TR)- and of macro tree transducers, i.e., $\mathrm{CFT(TR)} = \mathrm{CFT_1(TR)}$ and $\mathrm{MT_{OI,init}} = \mathrm{MT_{OI}}$. First we show this property for the total deterministic case and this, for later use, immediately for a more general class of transducers.

3.21. Lemma. *Let* $\mathfrak{M}$ *be a* $\mathrm{D_tCFT}(S)$*-transducer* ($\mathrm{D_tMAC}(S)$*-transducer*) *such that, for every rule*

$$A(y_1, \ldots, y_n) \to \textbf{if } b \textbf{ then } \zeta$$

of $\mathfrak{M}$ *and every* $c \in C$ *for which* $m(b)(c) = \textbf{true}$, *every instruction occurring in* ζ *is defined on* c. *Then there is a* $\mathrm{D_tCFT_1}(S)$*-transducer* ($\mathrm{D_tMAC_1}(S)$*-transducer, respectively*) *which is equivalent to* $\mathfrak{M}$.

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$ be a $\mathrm{D_tCFT}(S)$-transducer with $A_{\mathrm{in}} \in T_N$. We can assume that the tests of the rules in R are P_{f}-standard tests, for some finite subset P_{f} of P (note that the 'definedness property' in the statement of this lemma is preserved by the construction in Lemma 3.11).

The main idea in the construction of the $\mathrm{D_tCFT_1}(S)$-transducer $\mathfrak{M}'$ is to substitute right-hand sides of appropriate rules into A_{in}. Construct $\mathfrak{M}' = (N', e, \Delta, A'_{\mathrm{in}}, R')$ with $N' = N \cup \{A'_{\mathrm{in}}\}$ where A'_{in} is a new nonterminal of rank 0, and

$$R' = R \cup \{A'_{\mathrm{in}} \to \textbf{if } b \textbf{ then } A_{\mathrm{in}}(b) \mid b \text{ is a } P_{\mathrm{f}}\text{-standard test}\}.$$

For every $\alpha \in T_N$ and every P_{f}-standard test b, $\alpha(b) \in F_{\mathrm{CFT}}(N(F), \Delta)$ is inductively defined on the structure of α as follows: if $\alpha = A(\alpha_1, \ldots, \alpha_k)$ with $k \geqslant 0$, then

$$\alpha(b) = \begin{cases} \zeta[y_j \leftarrow \alpha_j(b); j \in [k]] & \text{if } A(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ then } \zeta \text{ is in } R, \\ d & \text{otherwise,} \end{cases}$$

where d is an arbitrary symbol of Δ_0.

Note that $\mathfrak{M}'$ is deterministic. Furthermore, note that if $m(b)(c) = \textbf{true}$ for some $c \in C$, then, for every $\alpha \in T_N$,

$$\alpha(b)[f \leftarrow m(f)(c); f \in F] \in F_{\mathrm{CFT}}(N(C), \Delta),$$

i.e., every instruction occurring in $\alpha(b)$ is defined on c. This is guaranteed by the 'definedness assumption' on $\mathfrak{M}$.

Since $\tau(\mathfrak{M})$ is a total function (cf. Corollary 3.16), we only have to show that $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}')$. For this purpose, we use Claims 2 and 3.

Claim 2. *For every* $\xi \in F_{\mathrm{CFT}}(N(C), \Delta \cup Y_k)$ *and* $t \in F_{\mathrm{CFT}}(\emptyset, \Delta \cup Y_k)$, *if* $\xi \Rightarrow^*_{\mathfrak{M}} t$, *then* $\xi \Rightarrow^*_{\mathfrak{M}'} t$.

This is trivially true, because $R \subseteq R'$.

Claim 3. *For every* $\alpha \in T_N$, $t \in T_\Delta$, *and* $c \in C$ *such that* $m(b)(c) = \mathbf{true}$, *if* $\alpha[A \leftarrow A(c); A \in N] \Rightarrow^*_{\mathfrak{M}} t$, *then* $\alpha(b)[f \leftarrow m(f)(c); f \in F] \Rightarrow^*_{\mathfrak{M}'} t$.

The proof of Claim 3 is an induction on the height of α, which uses a decomposition result of derivations of CFT(S)-transducers similar to the one for OI macro grammars as discussed in [24]. Since we defined the derivation relation of a CFT(S)-transducer by means of a CFT-grammar, we clearly can use that decomposition result.

Then, in particular, if $A_{\mathrm{in}}[A \leftarrow A(c); A \in N] \Rightarrow^*_{\mathfrak{M}} t$, then $A'_{\mathrm{in}}(c) \Rightarrow_{\mathfrak{M}'} A_{\mathrm{in}}(b)[f \leftarrow m(f)(c); f \in F] \Rightarrow^*_{\mathfrak{M}'} t$. This proves $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}')$. □

3.22. Theorem

(a) $\mathrm{CFT(TR)} = \mathrm{CFT_1(TR)}$ *and* $\mathrm{MT_{OI,init}} = \mathrm{MT_{OI}}$;

(b) $\mathrm{CFT(TR)} = \mathrm{MT_{OI}}$.

In both cases total determinism is preserved.

Proof. (a) From Lemma 3.18 and Lemma 3.21, it immediately follows that $\mathrm{D_tCFT(TR)} = \mathrm{D_tCFT_1(TR)}$. Hence, by Theorem 3.19, $\mathrm{D_tMT_{OI,init}} = \mathrm{D_tMT}$. Then the equalities concerning the classes of translations of nondeterministic transducers, follow from the decomposition $\mathrm{MT_{OI}} = \mathrm{D_tMT} \circ \mathrm{SET}$ ([22, Theorem 6.10] SET is a particular class of top-down tree transducers), its obvious generalization to $\mathrm{MT_{OI,init}} = \mathrm{D_tMT_{OI,init}} \circ \mathrm{SET}$, and again Theorem 3.19. The equation in (b) is a consequence of (a) and Theorem 3.19. □

We have shown how a macro tree transducer can be expressed as a CFT(S)-transducer, namely by specializing S to be TR. But on the other hand, a CFT(S)-transducer can be viewed as a macro tree transducer, which works on approximations of S-configurations (cf. Theorem 3.26). Before we make this idea more precise, let us discuss which information concerning an S-configuration c can be relevant for the work of a CFT(S)-transducer. First, it is important to know which standard test is true for c. Then the question arises which instructions are defined on c. Let us assume that f is such an instruction (i.e., $m(f)$ is defined on c). Then we can ask the same question about relevant information on $m(f)(c)$. The information on c and those configurations that are reachable from c via application of instructions, can be stored in an infinite tree: the nodes correspond to configurations reachable from c (the root corresponding to c itself), and a node n is labeled by $\langle b; f_1, \ldots, f_k \rangle$ iff the standard test b is true for the corresponding configuration c' and $f_1, \ldots, f_k$

are precisely all instructions applicable to c'. Then the ith son of n corresponds to the configuration $m(f_i)(c)$. Clearly, this infinite tree contains all the information on c that is ever needed by any $X(S)$-transducer working on c. But here we only work with finite trees that are approximations of this infinite tree: only finitely many reachable configurations c' are considered, and hence, labels $\langle b; f_1, \ldots, f_k\rangle$ are allowed where $f_1, \ldots, f_k$ are applicable to c' (but there may be others!). We also call such a finite tree an approximation of c (note that c corresponds to the root of this tree; cf. Fig. 1 for an example of an approximation of c, where $m(f_1)$ and $m(f_2)$ are defined on c and $m(f_3)$ is defined on c_1 where $c_1 = m(f_1)(c)$; moreover,

$$m(b_1)(c) = m(b_2)(c_1) = m(b_1)(m(f_3)(c_1)) = m(b_3)(m(f_2)(c)) = \textbf{true}.$$

Now it is easy to see that a CFT(S)-transducer working on c can be reformulated as a CFT(TR)-transducer working on an approximation of c: instead of applying a test to c, the label of the root of the approximation of c is checked; and instead of applying an instruction f to c, the appropriate subtree is selected.

Now we will define the notion of approximation formally. In the sequel, let P_f be a finite subset of predicates of a storage type S and let $\tilde{\psi} = \langle f_1, \ldots, f_r\rangle$ be a sequence of different instructions of S.

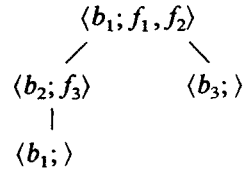


Fig. 1. An approximation of an S-configuration.

3.23. Definition. (1) The $(P_f, \tilde{\psi})$-*approximation alphabet*, denoted by $A(P_f, \tilde{\psi})$, is the ranked alphabet

$$\{\langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle^{(k)} \mid k \geq 0,\ b \text{ is a } P_f\text{-standard test, for every } i \in [k],\ \nu(i) \in [r], \text{ and, for every } i \in [k-1],\ \nu(i) < \nu(i+1)\}.$$

(2) Let $c \in C$ and let t be a tree over $A(P_f, \tilde{\psi})$. Then, t is a $(P_f, \tilde{\psi})$-*approximation of* c if, for every node n of t,

- $m(\text{inst}(t, n))(c)$ is defined;
- if the label of n is $\langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle$, then $m(b)(m(\text{inst}(t, n))(c)) = \textbf{true}$.

Here, $\text{inst}(t, n) \in \{f_1, \ldots, f_r\}^*$ is the sequence of instructions of S defined as follows: (i) if n is the root of t, then $\text{inst}(t, n) = \lambda$; (ii) if n is a node of t labeled by $\langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle$ and if n' is the ith son of n, then $\text{inst}(t, n') = \text{inst}(t, n) f_{\nu(i)}$. (Note that $\lambda n.\text{inst}(t, n)$ is injective.)

It is an easy observation that the $(P_f, \tilde{\psi})$-approximations of a configuration can be computed by a special RT(S)-transducer, which we call $(P_f, \tilde{\psi})$-approximator.

3.24. Definition. Let $\mathfrak{M}=(\{*\}, e, \Delta, *, R)$ be an RT(S)-transducer. $\mathfrak{M}$ is an $(P_f, \tilde{\psi})$-*approximator* of S if

(i) Δ is the $(P_f, \tilde{\psi})$-approximation alphabet;

(ii) $R=\{* \to \textbf{if } b \textbf{ then } \langle b; \sigma\rangle(*(f_{\nu(1)}), \ldots, *(f_{\nu(k)}))\mid$

$\langle b; \sigma\rangle = \langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle \in \Delta_k$ for some $k \geq 0\}$.

The class of translations induced by approximators of S is denoted by AP(S). Thus, AP(S) $\subseteq$ RT(S).

3.25. Lemma. *Let $\mathfrak{A}$ be a $(P_f, \tilde{\psi})$-approximator of S. Let $c \in C$ and let t be a tree over $A(P_f, \tilde{\psi})$. $*(c) \Rightarrow_{\mathfrak{A}}^{*} t$ iff t is a $(P_f, \tilde{\psi})$-approximation of c.*

Proof. In each direction, the proof uses an inductive characterization of the notion of approximation, namely: for every $c \in C$ and every tree

$$t=\langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle(t_1, \ldots, t_k)$$

over the approximation alphabet, t is an approximation of c iff $m(b)(c)=$ **true** and, for every $i \in [k]$, $m(f_{\nu(i)})$ is defined on c and t_i is an approximation of $m(f_{\nu(i)})(c)$.

Since the proofs are easy inductions on the length of derivations of $\mathfrak{A}$ and on the height of t, respectively, they are omitted here. □

In the next theorem we will show that the translation class CFT(S) can be characterized by the composition of approximators of S and CFT(TR)-transducers.

3.26. Theorem. CFT(S) = AP(S) $\circ$ CFT(TR).

Proof. (1) (CFT(S) $\subseteq$ AP(S) $\circ$ CFT(TR)): Let $\mathfrak{M}=(N, e, \Delta, A_{in}, R)$ be a CFT(S)-transducer in P_f-standard test form for some finite subset P_f of P such that all instructions which occur in the right-hand sides of rules of R are in $\{f_1, \ldots, f_r\}$. Define $\tilde{\psi}=\langle f_1, \ldots, f_r\rangle$ and let $\mathfrak{A}$ be the $(P_f, \tilde{\psi})$-approximator of S with encoding e. Construct the CFT(TR)-transducer $\mathfrak{M}'=(N, e', \Delta, A_{in}, R')$ as follows:

(i) $\mathrm{dom}(e')=\{t \mid t$ is a tree over $A(P_f, \tilde{\psi})\}$;

(ii) if $B(y_1, \ldots, y_n) \to$ **if** b **then** ζ is in R, and if $\{f_{\rho(1)}, \ldots, f_{\rho(m)}\}$ is the set of instructions occurring in ζ, then, for every subset $V=\{\nu(1), \ldots, \nu(k)\} \subseteq [r]$ with $k \geq 0$ and $\nu(i) < \nu(i+1)$ (for every $i \in [k-1]$) and $\{\rho(1), \ldots, \rho(m)\} \subseteq V$,

$$B(y_1, \ldots, y_n) \to \textbf{if root} = \langle b; f_{\nu(1)}, \ldots, f_{\nu(k)}\rangle \textbf{ then } \zeta[f_{\nu(i)} \leftarrow \mathrm{sel}_i; i \in [k]]$$

is in R'.

For the proof of $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{A}) \circ \tau(\mathfrak{M}')$, we consider a derivation $d=(\xi_0 \Rightarrow \xi_1 \Rightarrow \cdots \Rightarrow \xi_m)$ of $\mathfrak{M}$ with $\xi_0 = A_{in}(c)$ and $c \in \mathrm{range}(e)$. In order to enable $\mathfrak{M}'$ to simulate this derivation (with a one-to-one correspondence between the sentential forms), the approximation of c which is computed by $\mathfrak{A}$ must contain enough

information about c. This is achieved by taking the approximation $\mathrm{ap}(d)$ of c such that c' occurs in d iff there is a corresponding node in $\mathrm{ap}(d)$ (note that every c' occurring in d is of the form $m(\phi)(c)$ for some sequence of instructions $\phi \in \{f_1, \ldots, f_r\}^*$; the corresponding node n in $\mathrm{ap}(d)$ then has $\mathrm{inst}(\mathrm{ap}(d), n) = \phi$). Then it can be proved by an easy induction that, for every j with $0 \leqslant j \leqslant m$,

$$A_{\mathrm{in}}(\mathrm{ap}(d)) \Rightarrow^j_{\mathfrak{M}'} \xi_j[c' \leftarrow \mathrm{sub}(\mathrm{ap}(d), c');\ c' \in C],$$

where $\mathrm{sub}(\mathrm{ap}(d), c')$ is the subtree of $\mathrm{ap}(d)$ of which the root corresponds to c'. The formal details are left out.

For the proof of the other direction, i.e., $\tau(\mathfrak{A}) \circ \tau(\mathfrak{M}') \subseteq \tau(\mathfrak{M})$, we consider a configuration c and an approximation t of c, which is computed by $\mathfrak{A}$. Then it is intuitively clear that every successful derivation of $\mathfrak{M}'$ that starts with $A_{\mathrm{in}}(t)$ can be simulated by a derivation of $\mathfrak{M}$ that starts with $A_{\mathrm{in}}(c)$, with a one-to-one correspondence of the sentential forms.

(2) $(\mathrm{AP}(S) \circ \mathrm{CFT}(\mathrm{TR}) \subseteq \mathrm{CFT}(S))$: Let $\mathfrak{A}$ be the $(P_{\mathrm{f}}, \tilde{\psi})$-approximator of S for some P_{f} and some $\tilde{\psi}$, with encoding e. Let $\mathfrak{M}'$ be a CFT(TR)-transducer. First, we transform $\mathfrak{M}'$ according to Lemma 3.18. Hence, the tests of rules have the form $\mathrm{root} = \sigma$ and every select instruction is applicable to the actual tree. Then, we throw out all those rules of $\mathfrak{M}'$ in which the test contains a symbol which is not in the $(P_{\mathrm{f}}, \tilde{\psi})$-approximation alphabet. Clearly, they do not contribute to $\tau(\mathfrak{A}) \circ \tau(\mathfrak{M}')$. Let $(N, e', \Delta, A_{\mathrm{in}}, R')$ be the so transformed CFT(TR)-transducer $\mathfrak{M}'$ where $\mathrm{dom}(e')$ is the set of trees over $A(P_{\mathrm{f}}, \tilde{\psi})$.

Then, the CFT(S)-transducer $\mathfrak{M} = (N \cup \{q\}, e, \Delta, A_{\mathrm{in}}, R)$ is determined by R, as follows (where q is a new nonterminal of rank 1): if

$$B(y_1, \ldots, y_n) \rightarrow \textbf{if } \mathrm{root} = \langle b; f_{\nu(1)}, \ldots, f_{\nu(k)} \rangle \textbf{ then } \zeta$$

is in R', then

$$B(y_1, \ldots, y_n) \rightarrow \textbf{if } b \textbf{ then } q(f_{\nu(1)})(\ldots q(f_{\nu(k)})(\zeta') \ldots)$$

is in R, where $\zeta' = \zeta[\mathrm{sel}_i \leftarrow f_{\nu(i)}]$. Furthermore, the rule $q(y_1) \rightarrow y_1$ is in R.

The only purpose of q is to check whether the $f_{\nu(i)}$'s are defined or not. The definedness of $f_{\nu(i)}$ for which sel_i occurs in ζ is implicitly checked by the definition of the derivation relation. But not necessarily all $f_{\nu(i)}$'s occur in ζ'. If $\mathfrak{M}$ would not check their definedness, then it could successfully apply a rule to a configuration c, whereas the original rule in $\mathfrak{M}'$ is not applicable: the approximator cannot provide an appropriate input tree, because one of the $f_{\nu(i)}$'s is undefined on c.

Again, the proof of $\tau(\mathfrak{A}) \circ \tau(\mathfrak{M}') = \tau(\mathfrak{M})$ is intuitively clear from the construction, and therefore, we do not provide a formal one. □

An immediate consequence of this theorem is a decomposition for CFT(S).

3.27. Corollary. $\mathrm{CFT}(S) \subseteq \mathrm{RT}(S) \circ \mathrm{CFT}(\mathrm{TR})$.

Proof. Since $\mathrm{AP}(S) \subseteq \mathrm{RT}(S)$, the statement follows from Theorem 3.26. □

3.3. Pushdown storage type

The main topic of the present paper is to implement the nested recursion inherent in macro tree transducers on pushdown machines. Since we want to set up all results concerning this implementation in general, i.e., for $\mathrm{CFT}(S)$-transducers, we have to deal with parameters the values of which are configurations of an arbitrary storage type S. This motivates the definition of an operator P on an arbitrary storage type S, which organizes pairs, consisting of a usual pushdown symbol and an S-configuration, as pushdown (cf. [28, 16, 17]). Hopefully it will always be clear from the context whether P refers to the pushdown operator or to the set P of predicate symbols of S.

3.28. Definition. Let S be the storage type (C, P, F, m, I, E). The *pushdown of* S, denoted by $\mathrm{P}(S)$, is the storage type (C', P', F', m', I', E'), where

(i) $C' = (\Gamma \times C)^+$ and Γ is a fixed infinite set of pushdown symbols (intuitively, the top of the pushdown is at the left);

(ii) $P' = \{\mathrm{top} = \gamma \mid \gamma \in \Gamma\} \cup \{\mathrm{test}(p) \mid p \in P\}$;

(iii) $F' = \{\mathrm{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{pop}\}$
$\cup \{\mathrm{stay}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{stay}(\gamma) \mid \gamma \in \Gamma\}$
$\cup \{\mathrm{id}\}$;

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

$m'(\mathrm{top} = \gamma)(c') = \mathbf{true}$ iff $\delta = \gamma$,

$m'(\mathrm{test}(p))(c') = m(p)(c)$,

$$m'(\mathrm{push}(\gamma, f))(c') = \begin{cases} (\gamma, c_1)(\delta, c)\beta & \text{if } m(f) \text{ is defined on } c \text{ and } c_1 = m(f)(c), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$m'(\mathrm{pop})(c') = \begin{cases} \beta & \text{if } \beta \neq \lambda, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$m'(\mathrm{stay}(\gamma, f))(c') = \begin{cases} (\gamma, c_1)\beta & \text{if } m(f) \text{ is defined on } c \text{ and } c_1 = m(f)(c), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$m'(\mathrm{stay}(\gamma))(c') = (\gamma, c)\beta$,

$m'(\mathrm{id})(c') = (\delta, c)\beta$;

(v) $I' = I$; and

(vi) $E' = \{\lambda u \in I.(\gamma_0, e(u)) \mid \gamma_0 \in \Gamma, e \in E\}$.

Remarks. (i) By definition of E', each $X(\mathrm{P}(S))$-transducer has its own bottom symbol. For instance, the encoding e' of the $X(\mathrm{P}(\mathrm{TR}))$-transducer specifies both the ranked input alphabet Σ ($\mathrm{dom}(e') = T_\Sigma$) and the initial pushdown symbol γ_0 ($\mathrm{range}(e') = \{(\gamma_0, t) \mid t \in T_\Sigma\}$).

(ii) There is no empty pushdown in $\mathrm{P}(S)$.

(iii) Naturally, we can iterate the pushdown operator: $\mathrm{P}^0(S) = S$ and, for every $n \geqslant 0$, $\mathrm{P}^{n+1}(S) = \mathrm{P}(\mathrm{P}^n(S))$.

(iv) The pushdown of S in which no stay(γ, f) instructions are allowed is denoted by $\mathrm{P}_1(S)$, and the pushdown of S which contains no kind of stay instructions is denoted by $\mathrm{P}_0(S)$.

(v) The pushdown of the trivial storage type S_0, which is clearly the usual pushdown, is abbreviated by P, i.e., $\mathrm{P}(S_0) = \mathrm{P}$. Considering an indexed grammar G [2], the reader will agree that the sequence of flags attached to each nonterminal in a sentential form of G behaves just as a pushdown (see [29, 16]). Thus, an indexed grammar is close to a CF(P)-transducer. Hence, it is not surprising that IND = range(CF(P)) where IND is the class of indexed languages. Moreover, the restricted pushdown tree automaton, defined in [29] as an acceptor of the OI context-free tree languages, can be viewed as a RT(P)-transducer. Since

$$\mathrm{yield}(\mathrm{range}(\mathrm{RT}(\mathrm{P}))) = \mathrm{range}(\mathrm{CF}(\mathrm{P})) = \mathrm{IND},$$

RT(P)-transducer may also be called indexed tree grammars.

(vi) For $X(\mathrm{P}(\mathrm{TR}))$-transducers the same remark holds as for $X(\mathrm{TR})$-transducers (see after Definition 3.17): thus they may be viewed as having pointers to the input tree in their pushdown squares rather than subtrees. The same will hold for more complicated storage types involving TR (such as $\mathrm{P}^2(\mathrm{TR})$, and nested stack of TR (cf. Section 7)).

In Sections 5 and 6, the regular $X(\mathrm{P}(S))$-transducers play a central role in the characterization of $\mathrm{CFT}(S)$ and $\mathrm{MAC}(S)$. We want to give names to them.

3.29. Definition. An *indexed S-transducer* is an $\mathrm{RT}(\mathrm{P}(S))$-transducer and a *pushdown*^{2} *S-to-string transducer* is a $\mathrm{REG}(\mathrm{P}^2(S))$-transducer.

We call an RT(P(TR))-transducer an *indexed tree transducer* rather than an indexed TR-transducer, and similarly, $\mathrm{REG}(\mathrm{P}^2(\mathrm{TR}))$-transducers will be called *pushdown*^{2} *tree-to-string transducers.*

Now we will prove two elementary properties of $X(\mathrm{P}(S))$-transducers. In the next lemma we will show that we can assume the tests in rules of an $X(\mathrm{P}(S))$-transducer to be of a simple form. For this purpose we uniquely extend the 'mapping' $\mathrm{test}: P \to \{\mathrm{test}(p) \mid p \in P\}$ to the mapping

$$\mathrm{test}: \mathrm{BE}(P) \to \mathrm{BE}(\{\mathrm{test}(p) \mid p \in P\})$$

such that it is a boolean homomorphism. For instance,

$$\text{test}(p_1 \textbf{ and } (\textbf{not} p_2 \textbf{ or true})) = \text{test}(p_1) \textbf{ and } (\textbf{not}\text{test}(p_2) \textbf{ or true}).$$

Note that this mapping 'test' is surjective.

3.30. Lemma. *For every $X(\mathrm{P}(S))$-transducer there is an equivalent $X(\mathrm{P}(S))$-transducer in which every test has the form* $\text{top} = \gamma$ **and** $\text{test}(b)$, *where $\gamma \in \Gamma$ and $b \in \mathrm{BE}(P)$ and P is the set of predicates of S. Determinism and totality are preserved.*

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be an $X(\mathrm{P}(S))$-transducer and let $\gamma_1, \ldots, \gamma_r$ with $r \geq 0$ be the pushdown symbols occurring in R. Let $e = \lambda u \in I.(\gamma_0, g(u))$ for some encoding g of S. Let $\{\text{test}(p_1), \ldots, \text{test}(p_n)\}$ with $n \geq 0$ be the set of predicates of the form $\text{test}(p)$ which occur in R.

Then, by Lemma 3.11, we can construct an equivalent $X(\mathrm{P}(S))$-transducer $\mathfrak{M}'$ which is in P'-standard test form, where $P' = \{\text{top} = \gamma_0, \text{top} = \gamma_1, \ldots, \text{top} = \gamma_r, \text{test}(p_1), \ldots, \text{test}(p_n)\}$. As discussed in Lemma 3.18 for $(\text{root} = \sigma)$-predicates in $X(\mathrm{TR})$-transducers, we can throw out rules in which every $(\text{top} = \gamma_i)$-predicate is negated and also those rules in which more than one $(\text{top} = \gamma_i)$-predicate is not negated. In the tests of the remaining rules the negated $(\text{top} = \gamma_i)$-predicates are deleted. Now each test looks like

$$\text{top} = \gamma \textbf{ and } t(p_1) \textbf{ and } \ldots \textbf{ and } t(p_n),$$

where $t(p_i) \in \{\text{test}(p_i), \textbf{not}\text{test}(p_i)\}$ for every $i \in [n]$. Since

$$t(p_1) \textbf{ and } \ldots \textbf{ and } t(p_n) = \text{test}(p_1' \textbf{ and } \ldots \textbf{ and } p_n'),$$

where $p_i' = p_i$ if $t(p_i) = \text{test}(p_i)$, and $p_i' = \textbf{not} p_i$ otherwise, we have obtained the desired form. □

For $X \in \{\mathrm{RT}, \mathrm{CF}\}$, the presence of stay instructions in an $X(\mathrm{P}(S))$-transducer actually does not contribute to its translation power, i.e., there is an equivalent $X(\mathrm{P}(S))$-transducer.

3.31. Lemma. *Let $X \in \{\mathrm{RT}, \mathrm{CF}\}$. Then, $X(\mathrm{P}(S)) = X(\mathrm{P}_1(S)) = X(\mathrm{P}_0(S))$. Determinism and totality are preserved.*

Proof. Of course, we only have to prove that $X(\mathrm{P}(S)) \subseteq X(\mathrm{P}_1(S)) \subseteq X(\mathrm{P}_0(S))$. We only consider the case that $X = \mathrm{RT}$, the other case is similar. Furthermore, we do not give a formal proof, because we are sure that the gentle reader can deduce from our informal description of the construction a formal one.

$(\mathrm{RT}(\mathrm{P}(S)) \subseteq \mathrm{RT}(\mathrm{P}_1(S)))$: A stay$(\gamma, f)$ instruction is simulated by a stay$(\#)$, where $\#$ is a new symbol, followed by a push(γ, f). Then, obviously, a pop is replaced by a pop followed by a sequence of pops, which deletes all underlying squares that contain $\#$ (this sequence may be empty). Clearly, determinism is preserved.

$(\mathrm{RT}(\mathrm{P}_1(S)) \subseteq \mathrm{RT}(\mathrm{P}_0(S)))$: Consider an $\mathrm{RT}(\mathrm{P}_1(S))$-transducer $\mathfrak{M}$. The construction of an equivalent $\mathrm{RT}(\mathrm{P}_0(S))$-transducer $\mathfrak{M}'$ proceeds in two simple steps.

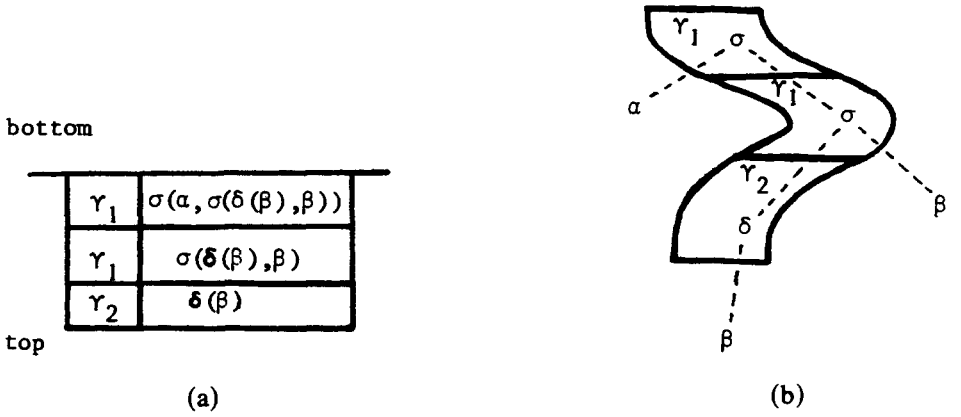
Step 1. Construct an $\mathrm{RT}(\mathrm{P}_1(S))$-transducer $\mathfrak{M}_1$ which is equivalent to $\mathfrak{M}$, by replacing every push(γ, f) instruction in rules of $\mathfrak{M}$ by push$(\langle \gamma, f\rangle, f)$.

Step 2. Construct $\mathfrak{M}'$ from $\mathfrak{M}_1$ as follows:

- add to every nonterminal A in a derivation of $\mathfrak{M}_1$ a finite amount of information which contains the symbol of the bottom square of the pushdown that A is presently scanning,
- if $\mathfrak{M}_1$ applies a stay(γ) instruction to a pushdown in which the topmost square looks like $(\langle \sigma, f\rangle, c)$, then replace this instruction in $\mathfrak{M}'$ by a pop followed by push$(\langle \gamma, f\rangle, f)$; note that the considered pushdown has more than one square. If the pushdown has only one square, then replace stay(γ) by the identity and change the finite information in the nonterminal into γ. All described transformations preserve determinism and totality. □

Since the storage types P(TR) and P^2(TR) are involved in the pushdown machines which characterize the macro tree transducer, we want to explain them here a bit more in detail.

A consequence of Theorem 4.21 is the equivalence of P(TR) and P_1(TR), and, since there is a very intuitive way of looking at P_1(TR), let us discuss the latter version. The configurations of P(TR) (or P_1(TR)) are pushdowns that contain in the second part of each square a tree. Whenever a push$(\ldots, \mathrm{sel}_i)$ is applied to such a configuration, say $c' = (\gamma, t)\beta$, the new square contains the ith subtree of t. Therefore, if pop is applied to c', the 'father-tree' of t appears in the topmost square. Since all the trees stored in c' are subtrees of one tree, say $\tilde{t}$ ($\tilde{t}$ is the tree which is stored in the bottom square of c'), it is obvious that c' describes a path from the root of $\tilde{t}$ to the root of t. Because of the correspondence between path and pushdown, we can also lay the pushdown on top of $\tilde{t}$ and view it as a layer of cells: every cell corresponds to a square of the pushdown and it covers the root of the subtree of $\tilde{t}$ which is contained in that square (cf. Fig. 2). Note that the pushdown lies upside down on the tree: its bottom is at the root.



(a) (b)

Fig. 2. (a) Configuration of P(TR). (b) Corresponding layer on tree.

From the discussion above, it follows that a sequence of $P_1(TR)$-instructions corresponds to a tree-walk on $\tilde{t}$; hence, we may call $P_1(TR)$ a tree-walking storage type. Actually, this storage type is used by the checking-tree pushdown transducer [19], which is equivalent to the $REG(P_1(TR))$-transducer.

One pushdown machine for the macro tree-to-string transducer (namely, the pushdown^2 tree transducer) uses the storage type $P^2(TR)$, which is equivalent to $P_1^2(TR)$ (this is due to the monotonicity of P_1, cf. Theorem 4.22, and the equivalence of $P(TR)$ and $P_1(TR)$). Again we want to shed some more light on the behavior of this storage type ($P_1^2(TR)$) by means of our layer model. Now, a configuration c' of $P_1^2(TR)$ can be described as a tree $\tilde{t}$ on which a whole collection of layers is put, and that on top of each other (cf. Fig. 3; for the sake of better readability the inscriptions of the cells of the layers are left out; also, the pushdown cells of the main pushdown are not shown). The configuration of Fig. 3 can be obtained from an initial configuration $(\gamma, (\gamma, \tilde{t}))$, where

$$\tilde{t} = \sigma(\sigma(\alpha, \delta(\alpha, \sigma(\alpha, \beta))), \alpha),$$

by the following sequence of instructions (in which all pushdown symbols are dropped):

$$\text{push}(\text{push}(\text{sel}_1)); \text{push}(\text{push}(\text{sel}_2)); \text{push}(\text{push}(\text{sel}_1));$$

$$\text{push}(\text{pop}); \text{push}(\text{push}(\text{sel}_2)).$$

If a $\text{push}(\ldots, \phi)$ is applied to a configuration c' of $P_1^2(TR)$, then a new layer s' is put on top of the other ones. s' is obtained by applying ϕ to the topmost layer s of c'. Note that ϕ is either a $\text{push}(\ldots, \text{sel}_i)$ or a pop or a $\text{stay}(\ldots)$ instruction. Now, it is clear that the new layer s' lies precisely on top of s, except for the cases $\phi = \text{push}(\ldots, \text{sel}_i)$ and $\phi = \text{pop}$: then, s' is one cell longer or shorter than s, respectively. Obviously, again the layers correspond to paths of the underlying tree, which means that a sequence of $\text{push}(\ldots, \phi)$ instructions describes a tree-walk on $\tilde{t}$. If now a pop is applied to c', then the topmost layer is taken away from the collection. But this means that the path on $\tilde{t}$ described by the sequence of

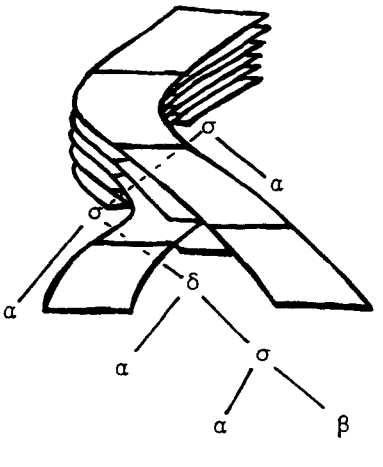


Fig. 3. A configuration c' of $P_1^2(TR)$ viewed as a tree $\tilde{t}$ with a collection of layers.

push(..., ϕ) instructions is traced back. Hence, $P_1^2(TR)$ is a tree-walking storage type with backtracking (see [32] for an automaton which uses a slightly weaker version of $P_1^2(TR)$).

From the arrangement of the layers and the way they are created, it should be obvious that a lot of redundant information is contained in a $P_1^2(TR)$-configuration. A more efficient storage type is the nested stack of trees. In fact, in Section 7 we will generalize the nested stack of [1] to an operator NS on storage types and prove that $NS(S)$ and $P^2(S)$ are equivalent storage types.

4. Simulation of storage types

In [30], a method is suggested which provides for an abstract program an equivalent concrete one. (In this context 'abstract' and 'concrete' denote consecutive stages of stepwise refinement of programs.) The technique which is involved in this method, is called data representation: with every abstract variable t (i.e., with every variable t occurring in the abstract program) a number of concrete variables $z_1, \ldots, z_n$ is associated which represent t, and, for every operation f on t, a procedure proc_f is available working on $z_1, \ldots, z_n$. The main tool for proving the correctness of a concrete program with respect to a given abstract program is the representation function h. It maps the values of the concrete variables $z_1, \ldots, z_n$ into the value domain of the abstract variable t. Correctness holds if the diagram in Fig. 4(a) commutes. (V_c and V_a denote the value domains of the concrete and the abstract variables, respectively.)

Fig. 4. Representation diagrams: (a) abstract and concrete variables; (b) abstract and concrete storage types.

We want to take advantage of the idea of data representation and try to prove the main topic of our paper, namely, implementation of nested recursion in CFT(S)-transducers, mainly by means of manipulation and comparison of storage types.

The first step in our attempt is the definition of a simulation relation $\leqslant$ on storage types (cf. [17]) such that, for two storage types S_1 and S_2, $S_1 \leqslant S_2$ means (the abstract storage type) S_1 is simulated by (the concrete storage type) S_2. A representation function h is inherent in $\leqslant$, mapping the set C_2 of S_2-configurations into the set C_1 of S_1-configurations. If $h(c_2) = c_1$ with $c_1 \in C_1$ and $c_2 \in C_2$, then c_2 is called a representation of c_1. With every predicate and instruction ϕ of S_1, a procedure

$\omega(\phi)$ is associated. For our purposes it suffices that $\omega(\phi)$ is a deterministic flowchart (i.e., an iterative program) which uses predicates and instructions of S_2; hence, $\omega(\phi)$ is also called an S_2-flowchart. The meaning of an S_2-flowchart ω, denoted by oper(ω), is a partial function from C_2 to C_2, intuitively defined as follows. If $c_2 \in C_2$ is passed as an initial value to the initial node of ω and if a computation leads into a final node of ω and transforms c_2 into c_2', then $\text{oper}(\omega)(c_2) = c_2'$. (Note that we only consider deterministic flowcharts.) Now, the notion 'S_1 is simulated by S_2' is justified by requiring that the diagram in Fig. 4(b) commutes. Considering a predicate p of S_1, the lower line of Fig. 4(b) collapses, because we only consider predicates without side-effects. (But note that we allow side-effects in $\omega(p)$, cf. [30].) For the correctness of the simulation of p by the flowchart $\omega(p)$, we have to require additionally the following. Informally, the truth value of p on a configuration c_1 of C_1 has to be obtained by the result of the execution of ω on c_2, where c_2 is a representation of c_1; more precisely, $\omega(p)$ has two final nodes, called **true** and **false**, and if the computation of $\omega(p)$ with c_2 as initial configuration leads to the node $Z \in \{\textbf{true}, \textbf{false}\}$, then $Z = \textbf{true}$ iff p is true on c_1, i.e., the diagram in Fig. 4(b) (continued) commutes.

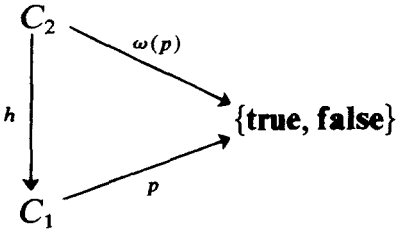


Fig. 4(b)(*continued*). Comparison of truth values.

From the correctness of the data representation Hoare concludes the correctness of the concrete program with respect to the abstract one. In the context of $X(S)$-transducers this consequence reads as follow: whenever the storage type of an $X(S)$-transducer $\mathfrak{M}$ can be simulated by another storage type S', then there is an $X(S')$-transducer which is equivalent to $\mathfrak{M}$. In fact, we will prove in Theorem 4.18 that (*) *if* $S_1 \leqslant S_2$, *then* $X(S_1) \subseteq X(S_2)$. This is also intuitively clear: if $\omega(\phi)$ is an S_2-flowchart simulating a predicate or instruction ϕ of S_1, and ϕ occurs in a rule of an $X(S_1)$-transducer $\mathfrak{M}$, then, figuratively speaking, first, ϕ can be replaced by $\omega(\phi)$ and second, $\omega(\phi)$ can be embedded in the main control of $\mathfrak{M}$. Thus $\mathfrak{M}$ changes into an $X(S_2)$-transducer which simulates the derivations of $\mathfrak{M}$ in a stepwise fashion (cf. [6]); clearly, the translation of $\mathfrak{M}$ is not changed by this construction. Since $\omega(\phi)$ is deterministic, this embedding preserves determinism.

The conceptual separation of control and storage type (as discussed in Section 3) together with the implication (*) justifies to attack the desired characterizations by mainly considering and comparing storage types. For this reason (*) is also referred to as the 'justification theorem'.

Before we profit from this environment, we have to give the precise definitions of the used tools. Although some of the definitions and lemmata are a bit complex,

the reader is asked to let himself be guided by the easy intuition behind the concept of simulation (as explained here). Throughout this section, S_i denotes the storage type $(C_i, P_i, F_i, m_i, I_i, E_i)$ with $i \in \{1, 2, 3\}$.

In Section 4.1 the flowcharts together with their meanings will be defined. There also the simulation relation will be formalized. Since the transitivity of $\leqslant$ and the justification theorem are based on the same construction ('embedding of a flowchart into an $X(S)$-transducer', cf. Lemma 4.16), they will both be proved in Section 4.2. In Section 4.3 finally, we will show that the pushdown operator P is monotonic with respect to $\leqslant$ (cf. Theorem 4.22). This monotonicity allows to prove properties of iterated pushdown storage types by induction on the level of iteration.

4.1. Flowcharts and simulation relation

Since the simulation relation is based on the notion of flowcharts, we will give a precise definition of this intuitively clear concept. In fact, our notion of flowchart is the *usual* one, but we formalize it by taking advantage of the notion of $X(S)$-transducer. (For S_{id}, see Definition 3.7.)

4.1. Definition. An *S-flowchart* is a $\mathrm{DREG}(S_{\mathrm{id}})$-transducer such that all its rules have the form $A \to$ **if** b **then** $B(\phi)$, where A and B are nonterminals, $b \in \mathrm{BE}(P)$, and $\phi \in F \cup \{\mathrm{id}\}$.

The class of S-flowcharts is denoted by $\mathrm{FC}(S)$. Note that the initial term of an S-flowchart is, by definition, a single nonterminal. Furthermore, note that we allow the identity instruction in every rule. Hence, if every instruction in an S-flowchart ω is the identity, then ω would 'compute' a partial identity on C (recall the meaning of a flowchart from the informal discussion in the introduction of this section). This behavior is quite natural for the simulation of a predicate, because it has no side-effects, i.e., does not change the tested configuration. Against that, it is in general impossible to simulate an instruction by the identity (because there may not be an identity for use in the simulating $X(S)$-transducer). For this reason we will distinguish between flowcharts for predicates and flowcharts for instructions. To prepare the definitions we will need the notion of a path of a flowchart which contains an instruction. Since the encoding and the terminal alphabet of a flowchart ω are not relevant, we denote the corresponding components in ω by $\_$.

4.2. Definition. Let $\omega = (N, \_, \_, A_{\mathrm{in}}, R) \in \mathrm{FC}(S)$ and let $n \geqslant 1$. If, for every $i \in [n]$, $A_{i-1} \to$ **if** b_i **then** $A_i(\phi_i)$ 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 $\mathrm{PATH}(\omega, A, B)$, where A and B are nonterminals of ω.

4.3. Definition. Let $\omega = (N, \_, \_, A_{in}, R) \in \mathrm{FC}(S)$.

(i) ω is an *S-flowchart for predicates* if **true** and **false** are nonterminals and the left-hand side of every rule is different from **true** and **false**.

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

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

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

(i) The *operation induced by* ω, denoted by $\mathrm{oper}(\omega)$, is the relation

$$\{(c_1, c_2) \in C \times C \mid A_{in}(c_1) \Rightarrow_\omega^* x(c_2) \text{ with } x \in D\},$$

where $D = \{\mathbf{true}, \mathbf{false}\}$ if $\omega \in P\text{-FC}(S)$, and $D = \{\mathbf{stop}\}$ otherwise.

(ii) If $\omega \in P\text{-FC}(S)$, then the *predicate induced by* ω, denoted by $\mathrm{pred}(\omega)$, is the relation

$$\{(c_1, x) \in C \times \{\mathbf{true}, \mathbf{false}\} \mid A_{in}(c_1) \Rightarrow_\omega^* x(c_2) \text{ for some } c_2 \in C\}.$$

4.5. Lemma. *For every* $\omega \in P\text{-FC}(S) \cup F\text{-FC}(S)$, $\mathrm{oper}(\omega)$ *is a partial function from* C *to* C, *and, for every* $\omega \in P\text{-FC}(S)$, $\mathrm{pred}(\omega)$ *is a partial function from* C *to* $\{\mathbf{true}, \mathbf{false}\}$.

Proof. Since S-flowcharts are particular $\mathrm{DREG}(S_{id})$-transducers, the statement of this lemma follows immediately from the confluency of the derivation relation of $\mathrm{DREG}(S_{id})$-transducers proved in Lemma 3.14. □

Note that contrary to the meaning of a predicate symbol which is a total function, the predicate induced by a flowchart is a partial function. However, in the definition of the simulation relation we require that $\mathrm{pred}(\omega)$ is total at least on the domain of the representation function, i.e., for every configuration c of the concrete storage type such that c actually represents a configuration of the abstract storage type, $\mathrm{pred}(\omega)$ is defined on c.

Now we will provide the definition of the simulation relation; since we will slightly generalize it later, we now call it *direct* simulation. The heart of the direct simulation relation is the representation function. For every encoding, predicate, and instruction of the abstract storage type there is (1) an encoding of the concrete storage type, (2) a flowchart for predicates and (3) a flowchart for instructions over the concrete storage type, respectively, such that certain requirements hold. The requirements are organized as follows: for every i with $i \in \{1, 2, 3\}$, $i.1.1$ and $i.1.2$ describe the situation in which the simulation is valid, whereas $i.2$ describes the validity of the simulation itself.

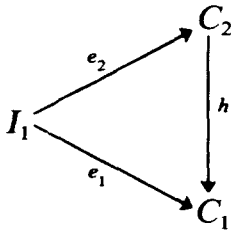
4.6. Definition. S_1 is *directly simulated by* S_2, for short $S_1 \leqslant_d S_2$, if $I_1 \subseteq I_2$ and there is a partial function $h: C_2 \to C_1$ called the representation function such that

(1) for every $e_1 \in E_1$ there is an $e_2 \in E_2$ such that

(1.1.1) $\mathrm{dom}(e_1) = \mathrm{dom}(e_2)$,

(1.1.2) $e_2(I_2) \subseteq \mathrm{dom}(h)$,

(1.2) for every $u \in \mathrm{dom}(e_2)$, $h(e_2(u)) = e_1(u)$, i.e., the following diagram commutes:

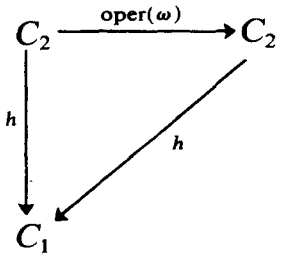


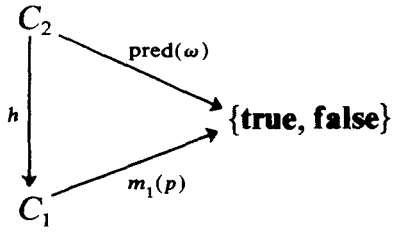
(2) for every $p \in P_1$ there is an $\omega \in P\text{-FC}(S_2)$ such that

(2.1.1) for every $c_2 \in \mathrm{dom}(h)$: $\mathrm{oper}(\omega)(c_2)$ is defined,

(2.1.2) $\mathrm{oper}(\omega)(\mathrm{dom}(h)) \subseteq \mathrm{dom}(h)$,

(2.2) for every $c_2 \in \mathrm{dom}(h)$: $h(\mathrm{oper}(\omega)(c_2)) = h(c_2)$ and $\mathrm{pred}(\omega)(c_2) = m_1(p)(h(c_2))$, i.e., the following diagrams commute:



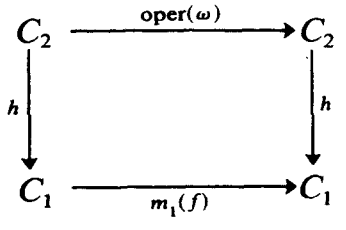


(3) for every $f \in F_1$ there is an $\omega \in F\text{-FC}(S_2)$ such that

(3.1.1) for every $c_2 \in \mathrm{dom}(h)$: $m_1(f)$ is defined on $h(c_2)$ iff $\mathrm{oper}(\omega)(c_2)$ is defined,

(3.1.2) $\mathrm{oper}(\omega)(\mathrm{dom}(h)) \subseteq \mathrm{dom}(h)$,

(3.2) for every $c_2 \in \mathrm{dom}(h)$ such that $m_1(f)(h(c_2))$ is defined, $h(\mathrm{oper}(\omega)(c_2)) = m_1(f)(h(c_2))$, i.e., the following diagram commutes:



We note that condition (*i*.1.2), which concerns the 'preservation' of $\mathrm{dom}(h)$ by the involved flowcharts, is related to the invariant condition I, which Hoare postulates for the correctness proof of a data representation (cf. [30]).

If $S_1 \leqslant_d S_2$ and $S_2 \leqslant_d S_1$, then S_1 and S_2 are called *directly equivalent*, denoted by $S_1 \equiv_d S_2$.

Naturally, we expect from a simulation relation of storage types that it is reflexive and transitive. Reflexivity of $\leqslant_d$ is easy to prove.

4.7. Lemma. $\leqslant_d$ *is reflexive.*

Proof. Let S be a storage type. Then we show that $S \leqslant_d S$. Clearly, $I \subseteq I$. Let $h: C \to C$

be the identity on C which serves as representation function. In Definition 4.6(1) we can take $e_2 = e_1$.

For every $p \in P$, define the S-flowchart for predicates $(\{A_{in}, \textbf{true}, \textbf{false}\}, \_, \_, A_{in}, R)$, where R contains the rule

$$A_{in} \to \textbf{if } p \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id}).$$

For every $f \in F$, define the S-flowchart for instructions $(\{A_{in}, \textbf{stop}\}, \_, \_, A_{in}, R)$ where R contains the rule $A_{in} \to \textbf{stop}(f)$.

Obviously, the requirements (2) and (3) of Definition 4.6 are fulfilled for these flowcharts. □

Since the proof of the transitivity of $\leqslant_d$ and the proof of the justification theorem are based on the same idea (formalized in Definition 4.11 and Lemma 4.16), both proofs will be provided in the next section.

For most of the simulations of storage types treated in this paper, it is essential that the simulated storage type has only a finite number of predicates and instructions, and one encoding: based on this finiteness property, whenever a case analysis is needed, it can be realized by a flowchart. The consideration of storage types with the discussed finiteness property is not a severe restriction, because every transducer uses only a finite number of predicates and instructions and exactly one encoding; hence, the transducer induces a finite restriction on the involved storage type. Consequently, in the assumption of the justification theorem we require the weaker fact that every finite restriction of the one storage type can be (directly) simulated by the other storage type.

4.8. Definition. Let S be a storage type. 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 $e \in E$.

4.9. Definition. S_1 is *simulated by* S_2 (for short $S_1 \leqslant S_2$) if, for every finite restriction U of S_1, $U \leqslant_d S_2$.

If $S_1 \leqslant S_2$ and $S_2 \leqslant S_1$, then S_1 and S_2 are *equivalent*, denoted by $S_1 \equiv S_2$. Obviously, if $S_1 \leqslant_d S_2$, then $S_1 \leqslant S_2$. Hence, the reflexivity of $\leqslant$ is an immediate consequence of the reflexivity of $\leqslant_d$.

4.10. Theorem. *$\leqslant$ is reflexive.*

4.2. Justification theorem and transitivity of $\leqslant$

The common kernel in the proofs of the justification theorem and the transitivity of $\leqslant_d$ and $\leqslant$ is the notion of simulation of an $X(S_1)$-transducer $\mathfrak{M}_1$ by an $X(S_2)$-transducer $\mathfrak{M}_2$. Intuitively, the concept of storage type simulation is transferred to the $X(S)$-transducer formalism in the following sense: to every rule r of $\mathfrak{M}_1$ a set

of rules $R_{2,r}$ of $\mathfrak{M}_2$ corresponds such that the application of the rule r can be simulated by applying rules of $R_{2,r}$ and vice versa, i.e., every maximal derivation caused by the application of rules in $R_{2,r}$ simulates the application of r. More precisely, there is a partial function $\tilde{h}$ from sentential forms of $\mathfrak{M}_2$ to sentential forms of $\mathfrak{M}_1$ which is the identity on Δ, and for every rule r of $\mathfrak{M}_1$ there is a set $R_{2,r}$ of rules of $\mathfrak{M}_2$ such that the diagram in Fig. 5 commutes.

Of course, the mapping $\tilde{h}$ connects derivations of the two involved transducers. In order to connect also the beginning of these derivations, we furthermore require that $\tilde{h}(A^1_{\text{in}}(e_2(u))) = A^2_{\text{in}}(e_1(u))$ where e_i and A^i_{in} are the encoding and the initial term of $\mathfrak{M}_i$ respectively, and u is an element of the input set of $\mathfrak{M}_1$ and $\mathfrak{M}_2$. In outline this is the notion of simulation of transducers, which is formalized in the next definition (in a slightly weaker way).

4.11. Definition. Let $X \in \{\text{RT}, \text{CF}, \text{REG}\}$. For $i \in \{1, 2\}$ let $\mathfrak{M}_i = (N_i, e_i, \Delta, A^i_{\text{in}}, R_i)$ be an $X(S_i)$-transducer. Let $h: C_2 \to C_1$ be a partial function, and let $\tilde{h}: F_X(N_1(C_2), \Delta) \to F_X(N_1(C_1), \Delta)$ be the natural extension of h (it just replaces every c_2 by $h(c_2)$). $\mathfrak{M}_1$ is *h-simulated by* $\mathfrak{M}_2$ (for short, $\mathfrak{M}_1 \leqslant^{(h)} \mathfrak{M}_2$) if $N_1 \subseteq N_2$ and $A^1_{\text{in}} = A^2_{\text{in}}$, denoted by A_{in} in the sequel, and

$$(1)\quad \begin{cases} \text{dom}(e_1) = \text{dom}(e_2), \\ e_2(I_2) \subseteq \text{dom}(h), \\ \text{and, for every } u \in \text{dom}(e_2),\ \tilde{h}(A_{\text{in}}(e_2(u))) = A_{\text{in}}(e_1(u)); \end{cases}$$

and

$$(2)\quad \begin{cases} \text{for every } A \in N_1,\ c_2 \in \text{dom}(h), \text{ and } \xi_2 \in F_X(N_1(C_2), \Delta), \\ \text{if } A(c_2) \Rightarrow^*_{\mathfrak{M}_2} \xi_2, \text{ then } \xi_2 \in F_X(N_1(\text{dom}(h)), \Delta) \\ \quad \text{and } A(h(c_2)) \Rightarrow^*_{\mathfrak{M}_1} \tilde{h}(\xi_2); \end{cases}$$

and

$$(3)\quad \begin{cases} \text{for every } A \in N_1,\ c_2 \in \text{dom}(h), \text{ and } \xi_1 \in F_X(N_1(C_1), \Delta), \\ \text{if } A(h(c_2)) \Rightarrow^*_{\mathfrak{M}_1} \xi_1, \text{ then there is a } \xi_2 \in F_X(N_1(\text{dom}(h)), \Delta) \\ \quad \text{such that } A(c_2) \Rightarrow^*_{\mathfrak{M}_2} \xi_2 \text{ and } \tilde{h}(\xi_2) = \xi_1. \end{cases}$$

h is also called the representation function.

Note that, for the simulation of the sentential forms of $\mathfrak{M}_1$, $\mathfrak{M}_2$ uses its sentential forms that belong to $F_X(N_1(\text{dom}(h)), \Delta)$.

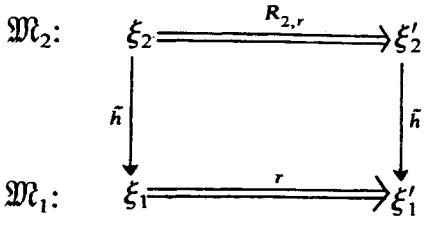


Fig. 5. Simulation of $\mathfrak{M}_1$ by $\mathfrak{M}_2$, where ξ_i and ξ'_i are sentential forms of $\mathfrak{M}_i$ for $i = 1, 2$.

For convenience we abbreviate in this section the class of $X(S)$-transducers by $X(S)_t$. Then, for two classes of transducers $X(S_1)_t$ and $X(S_2)_t$ and a partial function $h: C_2 \to C_1$, we write $X(S_1)_t \leqslant^{(h)} X(S_2)_t$ if, for every $X(S_1)$-transducer $\mathfrak{M}$, there is an $X(S_2)$-transducer $\mathfrak{M}'$ such that $\mathfrak{M} \leqslant^{(h)} \mathfrak{M}'$.

It is intuitively clear that the notion of h-simulation connects two transducers in a stronger way than the requirement that they are equivalent, or, in other words, if $\mathfrak{M}_1$ is h-simulated by $\mathfrak{M}_2$, then $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$ (but not necessarily vice versa). For the interested reader, we provide a formal proof.

4.12. Lemma. *For $i \in \{1, 2\}$ let $\mathfrak{M}_i = (N_i, e_i, \Delta, A^i_{\text{in}}, R_i)$ be an $X(S_i)$-transducer. Let $h: C_2 \to C_1$ be a partial function.*

If $\mathfrak{M}_1 \leqslant^{(h)} \mathfrak{M}_2$, then $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$.

Proof. Let $\mathfrak{M}_i$ and h be as above and assume that $\mathfrak{M}_1 \leqslant^{(h)} \mathfrak{M}_2$. We prove that $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$.

($\tau(\mathfrak{M}_1) \subseteq \tau(\mathfrak{M}_2)$): Let $(u, v) \in \tau(\mathfrak{M}_1)$. Then $u \in \text{dom}(e_1)$. By (1) of Definition 4.11, $u \in \text{dom}(e_2)$, $e_2(u) \in \text{dom}(h)$, and $h(e_2(u)) = e_1(u)$. Since $(u, v) \in \tau(\mathfrak{M}_1)$, $A^1_{\text{in}}(e_1(u)) \Rightarrow^*_{\mathfrak{M}_1} v$, i.e., $A^1_{\text{in}}(h(e_2(u))) \Rightarrow^*_{\mathfrak{M}_1} v$. Then, by (3) of Definition 4.11, there is a $\xi \in F_X(N_1(\text{dom}(h)), \Delta)$: $A^2_{\text{in}}(e_2(u)) \Rightarrow^*_{\mathfrak{M}_2} \xi$ and $\tilde{h}(\xi) = v$. Then, $\xi = v$, because $v \in F_X(\emptyset, \Delta)$. Hence, $(u, v) \in \tau(\mathfrak{M}_2)$.

($\tau(\mathfrak{M}_2) \subseteq \tau(\mathfrak{M}_1)$): Let $(u, v) \in \tau(\mathfrak{M}_2)$. Then $u \in \text{dom}(e_2)$ and, by (1) of Definition 4.11, $e_2(u) \in \text{dom}(h)$. Moreover, $A^2_{\text{in}}(e_2(u)) \Rightarrow^*_{\mathfrak{M}_2} v$. Since $v \in F_X(N_1(C_2), \Delta)$, it now follows from (2) of Definition 4.11 that $A^1_{\text{in}}(h(e_2(u))) \Rightarrow^*_{\mathfrak{M}_1} \tilde{h}(v) = v$; by (1),

$$e_2(u) \in \text{dom}(h) \quad \text{and} \quad h(e_2(u)) = e_1(u).$$

Hence, $A^1_{\text{in}}(e_1(u)) \Rightarrow^*_{\mathfrak{M}_1} v$, and thus, $(u, v) \in \tau(\mathfrak{M}_1)$.

This proves that $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$. □

In the central lemma (Lemma 4.16) it will be proved that if $S_1 \leqslant_d S_2$ and h is the involved representation function, then $X(S_1)_t \leqslant^{(h)} X(S_2)_t$. Of course, by Lemma 4.12, the justification theorem is an immediate consequence of this lemma. But what about the transitivity of $\leqslant_d$? Assume that $S_1 \leqslant_d S_2$ and that $S_2 \leqslant_d S_3$ and let ϕ be either a predicate or an instruction of S_1. Since $S_1 \leqslant_d S_2$, there is an S_2-flowchart ω for which the requirements of Definition 4.6 hold. Since ω is a particular DREG(S_{id})-transducer and Lemma 4.16 preserves the property of being a flowchart (i.e., for a flowchart, the simulating transducer is also a flowchart), it immediately follows from $S_2 \leqslant_d S_3$ that there is an S_3-flowchart ω' such that ω is simulated by ω'. By using the properties of the simulation of transducers and the fact that the requirements of Definition 4.6 are accomplished by ϕ and ω, we can in fact prove that these requirements are also fulfilled by ϕ and ω'. This then proves the transitivity of $\leqslant_d$.

In order to facilitate the construction in Lemma 4.16, we use some technical definitions and lemmata. By allowing chain rules, the tests of rules of $X(S)$-transducers can be reduced to a simple form, namely, either **true** or p or **not**p for some predicate p (cf. Lemma 4.15).

4.13. Definition. Let $X \in \{\mathrm{RT}, \mathrm{CF}, \mathrm{REG}\}$.

(i) A rule of an $X(S_{\mathrm{id}})$-transducer is a *chain rule* if it has the form $A \to$ **if** b **then** $B(\mathrm{id})$.

(ii) An $X(S)$*-transducer with chain rules* is an $X(S_{\mathrm{id}})$-transducer in which the instruction symbol id only occurs in chain rules.

Note that every $X(S)$-transducer with chain rules is a particular $X(S_{\mathrm{id}})$-transducer, and that every S-flowchart is a $\mathrm{DREG}(S)$-transducer with chain rules.

Next, we have to make precise how the tests of simple $X(S)$-transducers look like.

4.14. Definition. Let $X \in \{\mathrm{RT}, \mathrm{CF}, \mathrm{REG}\}$ and let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$ be an $X(S)$-transducer with chain rules. $\mathfrak{M}$ is *simple* if, for every nonterminal $A \in N$, either all rules of $\mathfrak{M}$ with left-hand side A have the form $A \to \zeta$ with $\zeta \notin N(\{\mathrm{id}\})$, or there is a predicate $p \in P$ such that all rules of $\mathfrak{M}$ with left-hand side A have either of the two following forms:

$$A \to \textbf{if } p \textbf{ then } B(\mathrm{id}) \quad \text{or} \quad A \to \textbf{if not}p \textbf{ then } B'(\mathrm{id})$$

for some $B, B' \in N$.

By using chain rules we can construct, for every transducer $\mathfrak{M}$, a simple transducer which simulates $\mathfrak{M}$.

4.15. Lemma. (1) *Let $X \in \{\mathrm{RT}, \mathrm{CF}, \mathrm{REG}\}$. Then, for every $X(S)$-transducer $\mathfrak{M}$, there is a simple $X(S)$-transducer $\mathfrak{M}'$ such that $\mathfrak{M} \leqslant^{(\mathrm{id})} \mathfrak{M}'$, where* id *is the identity on C. Determinism and totality are preserved.*

(2) *Let $Y \in \{P, F\}$. Then, for every flowchart ω in Y-$\mathrm{FC}(S)$, there is a simple flowchart ω' in Y-$\mathrm{FC}(S)$ such that* $\mathrm{oper}(\omega) = \mathrm{oper}(\omega')$ *and, if $Y = P$, then also* $\mathrm{pred}(\omega) = \mathrm{pred}(\omega')$.

Proof. (1) Let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$ be an $X(S)$-transducer. By Lemma 3.11, we can assume that $\mathfrak{M}$ is in standard test form. Assume that every test has the form $p_1' \dots p_n'$ with $p_i' \in \{p_i, \textbf{not}p_i\}$ for some $n \geqslant 0$.

Construct the $X(S_{\mathrm{id}})$-transducer $\mathfrak{M}' = (N', e, \Delta, A_{\mathrm{in}}, R')$ as follows:

$$N' = N \cup \{\langle A, p_1' p_2' \dots p_i' \rangle \mid A \in N,\ i \in [n], \text{ and } p_j' \in \{p_j, \textbf{not}p_j\} \text{ for every } j \in [i]\}$$

and R' is determined by the following two cases.

Case 1: $n = 0$. Then the tests of the rules of $\mathfrak{M}$ are either **true** or **false**. If we throw out the rules the tests of which are **false**, then $\mathfrak{M}'$ is in the desired form.

Case 2: $n \geq 1$.

(i) for every $A \in N$ and $p_1' \in \{p_1, \mathbf{not}p_1\}$,

$$A \to \mathbf{if}\ p_1'\ \mathbf{then}\ \langle A, p_1' \rangle(\mathrm{id})$$

is in R';

(ii) for every $\langle A, p_1' \dots p_i' \rangle \in N$ and $p_{i+1}' \in \{p_{i+1}, \mathbf{not}p_{i+1}\}$ such that $i+1 \leq n$,

$$\langle A, p_1' \dots p_i' \rangle \to \mathbf{if}\ p_{i+1}'\ \mathbf{then}\ \langle A, p_1' \dots p_i' p_{i+1}' \rangle(\mathrm{id})$$

is in R';

(iii) if $A \to \mathbf{if}\ p_1' \dots p_n'\ \mathbf{then}\ \zeta$ is in R, then $\langle A, p_1' \dots p_n' \rangle \to \zeta$ is in R'.

Note that determinism is preserved by the construction and that $\mathfrak{M}'$ is a simple $X(S)$-transducer. Obviously, the following claim holds, which proves requirements (2) and (3) of Definition 4.11.

Claim 4. *For every* $A \in N_1$, $c \in C$, *and* $\xi \in F_X(N_1(C), \Delta)$, $A(c) \Rightarrow_{\mathfrak{M}}^* \xi$ *iff* $A(c) \Rightarrow_{\mathfrak{M}'}^* \xi$.

Since requirement (1) of Definition 4.11 trivially holds, it follows that $\mathfrak{M} \leqslant^{(\mathrm{id})} \mathfrak{M}'$. If $\mathfrak{M}$ is total, then $\mathrm{dom}(e) = \mathrm{dom}(\tau(\mathfrak{M}))$. From Lemma 4.12 it follows that $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$; hence, $\mathrm{dom}(\tau(\mathfrak{M}')) = \mathrm{dom}(e)$, i.e., $\mathfrak{M}'$ is also total. This means that totality is preserved.

(2) It is obvious that the special requirements on $\mathrm{REG}(S_{\mathrm{id}})$-transducers for being flowcharts for predicates or instructions are preserved by the construction. From Claim 4 it immediately follows that $\mathrm{oper}(\mathfrak{M}) = \mathrm{oper}(\mathfrak{M}')$ and, for $\mathfrak{M} \in P\text{-}\mathrm{FC}(S)$, that also $\mathrm{pred}(\mathfrak{M}) = \mathrm{pred}(\mathfrak{M}')$. □

Now we prove the central lemma from which both the justification theorem and the transitivity of $\leqslant_{\mathrm{d}}$ and $\leqslant$ straightforwardly follow. For convenience the classes of simple $X(S)$-transducers and of $X(S)$-transducers with chain rules are abbreviated by $X_{\mathrm{sp}}(S)_{\mathrm{t}}$ and $X_{\mathrm{c}}(S)_{\mathrm{t}}$, respectively. The class of simple S-flowcharts for predicates and for instructions is denoted by $P\text{-}\mathrm{FC}_{\mathrm{sp}}(S)$ and $F\text{-}\mathrm{FC}_{\mathrm{sp}}(S)$, respectively.

4.16. Lemma. *Let* $X \in \{\mathrm{RT}, \mathrm{CF}, \mathrm{REG}\}$. *If* $S_1 \leqslant_{\mathrm{d}} S_2$ *and* $h: C_2 \to C_1$ *is the involved representation function, then*

(1) $X_{\mathrm{sp}}(S_1)_{\mathrm{t}} \leqslant^{(h)} X_{\mathrm{c}}(S_2)_{\mathrm{t}}$, *determinism and totality are preserved*;

(2) $Y\text{-}\mathrm{FC}_{\mathrm{sp}}(S_1) \leqslant^{(h)} Y\text{-}\mathrm{FC}(S_2)$ *for* $Y \in \{P, F\}$.

Proof. (1) Let $S_1 \leqslant_{\mathrm{d}} S_2$. Let $\mathfrak{M}_1 = (N_1, e_1, \Delta, A_{\mathrm{in}}^1, R_1)$ be a simple $X(S_1)$-transducer and let $(P_1)_{\mathrm{f}}$ and $(F_1)_{\mathrm{f}}$ be the finite sets of predicates and instructions respectively, which occur in R_1.

W.l.o.g., we can assume that, apart from **true**, **false**, and **stop**, the sets of nonterminals of the flowcharts in the simulation $S_1 \leqslant_{\mathrm{d}} S_2$ are pairwise disjoint. For every $\phi \in P_1 \cup F_1$ we denote the S_2-flowchart for which requirement (2) or (3) of Definition 4.6 respectively holds by

$$\omega(\phi) = (N(\phi), \_, \_, A_{\mathrm{in}}(\phi), R(\phi)).$$

For every $\phi \in P_1 \cup F_1$ and for every symbol z we define the *z-copy of* $\omega(\phi)$ to be the S_2-flowchart

$$(\langle N(\phi), z\rangle, \_, \_, \langle A_{\text{in}}(\phi), z\rangle\langle R(\phi), z\rangle),$$

where (i) $\langle N(\phi), z\rangle$ denotes $\{\langle B, z\rangle \mid B \in N(\phi)\}$, and (ii) if $B \to$ **if** b **then** ζ is in $R(\phi)$, then

$$\langle B, z\rangle \to \textbf{if } b \textbf{ then } \zeta[B(\psi) \leftarrow \langle B, z\rangle(\psi);\ B(\psi) \in N(\phi)(F \cup \{\text{id}\})]$$

is in $\langle R(\phi), z\rangle$. In words, the symbol z is attached to every nonterminal of $\omega(\phi)$.

Now we construct the $X(S_2)$-transducer $\mathfrak{M}_2 = (N_2, e_2, \Delta, A^2_{\text{in}}, R_2)$ with chain rules and prove that $\mathfrak{M}_1 \leqslant^{(h)} \mathfrak{M}_2$.

(i) $N_2 = N_1 \cup \bigcup\{\langle N(\phi), A\rangle \mid \phi \in (F_1)_f \cup (P_1)_f, A \in N_1\}$;

(ii) the encoding e_2 is the one which satisfies requirement (1) of Definition 4.6 with respect to h and e_1;

(iii) $A^2_{\text{in}} = A^1_{\text{in}}$;

(iv) $R_2 = \bigcup\{R_{2,r} \mid r \in R_1\}$, where $R_{2,r}$ is defined as follows: Let r be the rule $A \to$ **if** b **then** ζ of $\mathfrak{M}_1$.

Case 1. $b =$ **true**: Then, by definition of simplicity, $\zeta \notin N_1(\{\text{id}\})$. Hence, $\zeta = \zeta'[z_i \leftarrow B_i(f_i);\ i \in [n]]$ for some $\zeta' \in F_X(\emptyset, \Delta \cup Z_n)$ with $Z_n = \{z_1, \ldots, z_n\}$ and $B_1(f_1), \ldots, B_n(f_n) \in N_1((F_1)_f)$ with $n \geqslant 0$.

Intuitively, the set $R_{2,r}$ is obtained from r by starting at every nonterminal $B_i(f_i)$ the simulation of f_i with the B_i-copy of $\omega(f_i)$, and by adding the B_i-copies of the rules of $\omega(f_i)$ to $R_{2,r}$. Unfortunately, we cannot replace $B_i(f_i)$ in ζ just by $\langle A_{\text{in}}(f_i), B_i\rangle(\text{id})$, because, in general, the identity is only allowed in chain rules of $\mathfrak{M}_2$. However, by the definition of flowcharts for instructions, every $\pi \in \text{PATH}(\omega(f_i), A_{\text{in}}(f_i), \textbf{stop})$ contains an instruction. Then we can use the first occurrence of an instruction of F_2 to modify the right-hand side of the rule r.

Formally, let $\pi = \langle A_0, b_1, A_1, \phi_1, \ldots, b_m, A_m, \phi_m\rangle$ be a path of some S-flowchart ω, let π contain an instruction, and let π be loop-free, i.e., all $A_0, A_1, \ldots, A_m$ are different. If ϕ_j is the first instruction of S in π (i.e., $\phi_1 = \cdots = \phi_{j-1} = \text{id}$ and $\phi_j \in F$), then we define

$$\text{run}(\pi) = \langle b_1 \textbf{ and } \ldots \textbf{ and } b_j, A_j(\phi_j)\rangle.$$

Intuitively, if $\pi \in \text{PATH}(\omega, A_{\text{in}}, \textbf{stop})$ and $\text{run}(\pi) = \langle b, A(f)\rangle$, this means that, for every S-configuration c for which b holds, ω will follow path π up to nonterminal A, without transforming c, and then apply f to it.

For every $i \in [n]$, let π_i be a loop-free path in $\text{PATH}(\omega(f_i), A_{\text{in}}(f_i), \textbf{stop})$ and let $\text{run}(\pi_i) = \langle b_i, A^i(g_i)\rangle$. Then, for every such choice of $\pi_1, \ldots, \pi_n$,

$$A \to \textbf{if } b_1 \textbf{ and } \ldots \textbf{ and } b_n \textbf{ then } \zeta''$$

is in $R_{2,r}$, where $\zeta'' = \zeta'[z_i \leftarrow \langle A^i, B_i\rangle(g_i);\ i \in [n]]$;

for every $B_i(f_i)$ with $i \in [n]$, $\langle R(f_i), B_i\rangle \subseteq R_{2,r}$ (i.e., the rules of the B_i-copy of $\omega(f_i)$ are in $R_{2,r}$), and $\langle \textbf{stop}, B_i\rangle \to B_i(\text{id})$ is in $R_{2,r}$.

Case 2. $b = p$ or $b = \mathbf{not}p$ for some $p \in (P_1)_f$: Then, $\zeta = B(\mathrm{id})$ for some $B \in N_1$. Then,

- $A \to \langle A_{in}(p), B\rangle(\mathrm{id})$ is in $R_{2,r}$;
- $\langle R(p), B\rangle \subseteq R_{2,r}$ (i.e., the rules of the B-copy of $\omega(p)$ are in $R_{2,r}$);
- if $b = p$, then $\langle \mathbf{true}, B\rangle \to B(\mathrm{id})$ is in $R_{2,r}$, else $\langle \mathbf{false}, B\rangle \to B(\mathrm{id})$ is in $R_{2,r}$.

This ends the construction of $\mathfrak{M}_2$. Note that, if $\mathfrak{M}_1$ is deterministic, then so is $\mathfrak{M}_2$. The crucial observation which justifies this statement, is the fact that $\mathfrak{M}_1$ is simple and that the flowcharts are deterministic.

Obviously, Definition 4.11(1) is accomplished (it coincides with requirement (1) of Definition 4.6). The proofs of (2) and (3) straightforwardly follow from the Claims 5 and 6 respectively, which closely correspond to the notion of stepwise simulation of storage types.

Claim 5. *For every* $r \in R_1$, $\xi_2 \in F_X(N_1(\mathrm{dom}(h)), \Delta)$, *and* $\xi_2' \in F_X(N_1(C_2), \Delta)$, *if* $\xi_2 \Rightarrow^+_{\mathfrak{M}_2(r)} \xi_2'$ *and if only in the first step of this derivation a rule is applied to a nonterminal in* N_1, *then* $\xi_2' \in F_X(N_1(\mathrm{dom}(h)), \Delta)$ *and* $\tilde{h}(\xi_2) \Rightarrow_{\mathfrak{M}_1} \tilde{h}(\xi_2')$.

Claim 6. *For every* $\xi_1' \in F_X(N_1(C_1), \Delta)$, $\xi_2 \in F_X(N_1(\mathrm{dom}(h)), \Delta)$, *and* $r \in R_1$, *if* $\tilde{h}(\xi_2) \Rightarrow_{\mathfrak{M}_1} \xi_1'$ *via the application of the rule* r, *then there is a* $\xi_2' \in F_X(N_1(\mathrm{dom}(h)), \Delta)$ *such that* $\xi_2 \Rightarrow^+_{\mathfrak{M}_2(r)} \xi_2'$ *and* $\tilde{h}(\xi_2') = \xi_1'$.

Here $\tilde{h}$ is defined as in Definition 4.11 and $\mathfrak{M}_2(r)$ is the $X(S_2)$-transducer $(N_2, e_2, \Delta, A_{in}^2, R_{2,r})$ with chain rules. Note that $\mathfrak{M}_2(r)$ is deterministic.

The proofs of Claims 5 and 6 are rather technical and skipped here. They are based on properties (2) and (3) of Definition 4.6. If $\mathfrak{M}_1$ is total, then $\mathrm{dom}(\tau(\mathfrak{M}_1)) = \mathrm{dom}(e_1)$. Since $\mathfrak{M}_1 \leqslant^{(h)} \mathfrak{M}_2$, it follows that $\mathrm{dom}(e_1) = \mathrm{dom}(e_2)$ (from Definition 4.11(1)) and that $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_2)$ (from Lemma 4.12). Hence, $\mathrm{dom}(\tau(\mathfrak{M}_2)) = \mathrm{dom}(e_2)$, i.e., $\mathfrak{M}_2$ is total. This means that totality is preserved.

(2) If $\mathfrak{M}_1 \in Y\text{-FC}_{sp}(S_1)$ for some $Y \in \{P, F\}$, then $\mathfrak{M}_2 \in Y\text{-FC}(S_2)$. This is true, because, in particular, an instruction in $\mathfrak{M}_1$ is 'replaced' by a flowchart for instructions which contains, by definition, an instruction on each path from the initial nonterminal to **stop**. □

In order to get rid of the small corporal defect in Lemma 4.16(1), namely that we end up with a transducer with chain rules, we will prove in the next lemma that adding chain rules does actually not increase the power of a transducer.

4.17. Lemma. *Let* $X \in \{\mathrm{RT}, \mathrm{CF}, \mathrm{REG}\}$. *For every* $X(S)$*-transducer* $\mathfrak{M} = (N, e, \Delta, A_{in}, R)$ *with chain rules there is an* $X(S)$*-transducer* $\mathfrak{M}' = (N, e, \Delta, A_{in}, R')$ *(without chain rules) such that for every* $A \in N$, $c \in C$, *and every* $w \in F_X(\emptyset, \Delta)$,

$$A(c) \Rightarrow^*_{\mathfrak{M}} w \text{ iff } A(c) \Rightarrow^*_{\mathfrak{M}'} w.$$

In particular, $\mathfrak{M}$ *and* $\mathfrak{M}'$ *are equivalent. Determinism and totality are preserved.*

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be an $X(S)$-transducer with chain rules. By Lemma 3.11, we can assume that $\mathfrak{M}$ is in standard test form.

Construct the set of rules of the $X(S)$-transducer $\mathfrak{M}' = (N, e, \Delta, A_{\text{in}}, R')$ as follows:

(i) if $r \in R$ is no chain rule, then $r \in R'$;

(ii) if, for some standard test b,

$$A \to \textbf{if } b \textbf{ then } B_1(\text{id})$$

is in R, and, for every $i \in [n]$ with $n \geq 0$,

$$B_i \to \textbf{if } b \textbf{ then } B_{i+1}(\text{id})$$

is in R, and $B_{n+1} \to \textbf{if } b \textbf{ then } \zeta$ is in R, but is no chain rule, then $A \to \textbf{if } b \textbf{ then } \zeta$ is in R'.

Obviously, $\mathfrak{M}'$ is an $X(S)$-transducer without chain rules and it should be clear that for every $A \in N$, $c \in C$, and $w \in F_X(\emptyset, \Delta)$, $A(c) \Rightarrow^*_{\mathfrak{M}} w$ iff $A(c) \Rightarrow^*_{\mathfrak{M}'} w$. Since e is the encoding for both $\mathfrak{M}$ and $\mathfrak{M}'$, the transducers are equivalent. If $\mathfrak{M}$ is deterministic, then so is $\mathfrak{M}'$; obviously, totality is preserved. □

Now we can prove the justification theorem.

4.18. Theorem (Justification theorem). *Let* $X \in \{\text{RT}, \text{CF}, \text{REG}\}$. *If* $S_1 \leqslant S_2$, *then* $X(S_1) \subseteq X(S_2)$. *Determinism and totality are preserved.*

Proof. Let $\mathfrak{M}_1$ be an $X(S_1)$-transducer. By Lemma 4.15 there is a simple $X(S_1)$-transducer $\mathfrak{M}_1'$ such that $\mathfrak{M}_1 \leqslant^{(\text{id})} \mathfrak{M}_1'$ and hence, $\tau(\mathfrak{M}_1) = \tau(\mathfrak{M}_1')$. $\mathfrak{M}_1'$ uses only a finite number of predicates and instructions. Hence, there is a finite restriction U of S_1 such that $\mathfrak{M}_1'$ is an $X(U)$-transducer. Since $S_1 \leqslant S_2$, it follows that $U \leqslant_{\text{d}} S_2$. Let h be the involved representation function.

Then, by Lemma 4.16, there is an $X(S_2)$-transducer $\mathfrak{M}_2$ with chain rules such that $\mathfrak{M}_1' \leqslant^{(h)} \mathfrak{M}_2$. By Lemma 4.12 it follows that $\tau(\mathfrak{M}_1') = \tau(\mathfrak{M}_2)$. Finally, by Lemma 4.17, there is an $X(S_2)$-transducer $\mathfrak{M}_2'$ such that $\tau(\mathfrak{M}_2) = \tau(\mathfrak{M}_2')$. This proves $X(S_1) \subseteq X(S_2)$. All the involved constructions preserve determinism and totality. □

Now we turn to the transitivity of $\leqslant_{\text{d}}$.

4.19. Lemma. $\leqslant_{\text{d}}$ *is transitive.*

Proof. Let $S_1 \leqslant_{\text{d}} S_2$ and $S_2 \leqslant_{\text{d}} S_3$ and let $h_{2,1}: C_2 \to C_1$ and $h_{3,2}: C_3 \to C_2$ be the involved representation functions, respectively. We prove that $S_1 \leqslant_{\text{d}} S_3$.

W.l.o.g., we can assume that all flowcharts used in the two simulations are pairwise disjoint except that they share the final nonterminals **true** and **false** or **stop**, respectively.

Since we now deal with three simulations, we refer, for each of them in a different way, to requirements (1), (2), and (3) of Definition 4.6, e.g., if (2.1.1) holds in the simulation $S_1 \leqslant_{\text{d}} S_2$ for the predicate p and the flowchart $\omega(p)$, then we say that (2.1.1; $h_{2,1}$, p, $\omega(p)$) holds.

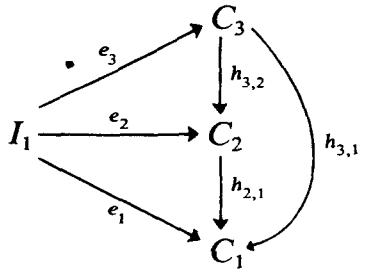


Fig. 6.

Similarly, we refer to requirements (2) and (3) of Definition 4.11; e.g., if requirement (2) holds for $\mathfrak{M}_1$, $\mathfrak{M}_2$, and the representation function $h_{2,3}$, then we say that (4.11(2); $h_{3,2}, \mathfrak{M}_1, \mathfrak{M}_2$) holds.

Define $h_{3,1}: C_3 \to C_1$ by $h_{3,1}(c_3) = h_{2,1}(h_{3,2}(c_3))$ for every $c_3 \in C_3$, whenever this is defined. Since $I_1 \subseteq I_2$ and $I_2 \subseteq I_3$, $I_1 \subseteq I_3$.

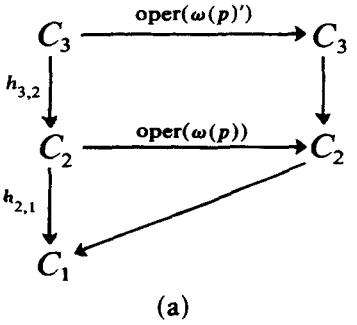
Requirement (1) *of Definition* 4.6 (cf. also Fig. 6): Let $e_1 \in E_1$. Then, since $S_1 \leqslant_d S_2$, there is an $e_2 \in E_2$ such that (1; $h_{2,1}, e_1, e_2$) holds. Since $S_2 \leqslant_d S_3$, there is an $e_3 \in E_3$ such that (1; $h_{3,2}, e_2, e_3$) holds. We show that (1; $h_{3,1}, e_1, e_3$) holds. Obviously, (1.1.1; $h_{3,1}, e_1, e_3$) holds, i.e., $\mathrm{dom}(e_1) = \mathrm{dom}(e_2) = \mathrm{dom}(e_3)$. Let $u \in \mathrm{dom}(e_3)$. Then, by (1.1.2; $h_{3,2}, e_2, e_3$), $e_3(u) \in \mathrm{dom}(h_{3,2})$. By (1.2; $h_{3,2}, e_2, e_3$), $h_{3,2}(e_3(u)) = e_2(u)$ and by (1.1.2; $h_{2,1}, e_1, e_2$),

$$e_2(u) = h_{3,2}(e_3(u)) \in \mathrm{dom}(h_{2,1}).$$

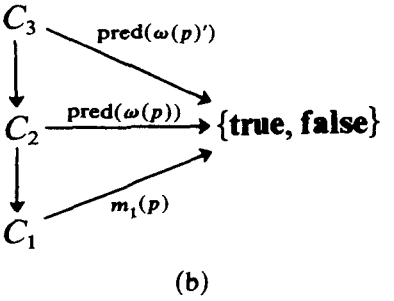
Hence, $e_3(u) \in \mathrm{dom}(h_{3,1})$ and (1.1.2; $h_{3,1}, e_1, e_3$) holds. Since $h_{2,1}(e_2(u)) = e_1(u)$ (by (1.2; $h_{2,1}, e_1, e_2$)), it follows that $h_{2,1}(h_{3,2}(e_3(u))) = h_{2,1}(e_2(u)) = e_1(u)$. Hence, (1.2; $h_{3,1}, e_1, e_3$) holds.

Requirement (2) *of Definition* 4.6 (cf. also Fig. 7): Let $p \in P_1$ and let $\omega(p) \in P\text{-FC}(S_2)$ such that (2; $h_{2,1}, p, \omega(p)$) holds. By Lemma 4.15(2) we may assume that $\omega(p)$ is simple.

Since $S_2 \leqslant_d S_3$ and $h_{3,2}$ is the involved representation function, it follows from Lemma 4.16(2) that there is an S_3-flowchart $\omega(p)'$ for predicates such that $\omega(p) \leqslant^{(h_{3,2})} \omega(p)'$, i.e., (4.11($i$); $h_{3,2}, \omega(p), \omega(p)'$) for $i \in \{1, 2, 3\}$ holds. We prove that (2; $h_{3,1}, p, \omega(p)'$) holds. Let $c_3 \in \mathrm{dom}(h_{3,1})$. Then there is a $c_2 \in C_2$ and a $c_1 \in C_1$ such that $h_{3,2}(c_3) = c_2$ and $h_{2,1}(c_2) = c_1$. By (2.1.1; $h_{2,1}, p, \omega(p)$), $\mathrm{oper}(\omega(p))(c_2)$ is defined. Let $\mathrm{oper}(\omega(p))(c_2) = c_2'$, Since $c_2 = h_{3,2}(c_2)$, it follows from (4.11(3); $h_{3,2}, \omega(p), \omega(p)'$) that there is a $c_3' \in \mathrm{dom}(h_{3,2})$ such that



(a)



(b)

Fig. 7.

$(c_3, c_3') \in \text{oper}(\omega(p)')$ and $h_{3,2}(c_3') = c_2'$. Hence, $\text{oper}(\omega(p)')(c_3)$ is defined which proves $(2.1.1; h_{3,1}, p, \omega(p)')$. By $(2.1.2; h_{2,1}, p, \omega(p))$, $c_2' \in \text{dom}(h_{2,1})$. Hence, $c_3' \in \text{dom}(h_{3,2} \circ h_{2,1}) = \text{dom}(h_{3,1})$ which proves $(2.1.2; h_{3,1}, p, \omega(p)')$.

Let $x \in \{\textbf{true}, \textbf{false}\}$ such that $A_{\text{in}}(c_3) \Rightarrow^*_{\omega(p)'} x(c_3')$, where A_{in} is the initial nonterminal of $\omega(p)$ and $\omega(p)'$. Then, by $(4.11(2); h_{3,2}, \omega(p), \omega(p)')$, $A_{\text{in}}(c_2) \Rightarrow^*_{\omega(p)} x(c_2')$, and hence, $\text{pred}(\omega(p))(c_2) = x$.

By $(2.2; h_{2,1}, p, \omega(p))$,

$$h_{2,1}(\text{oper}(\omega(p))(c_2)) = c_1 \quad \text{and} \quad \text{pred}(\omega(p))(c_2) = m_1(p)(c_1).$$

Hence,

$$h_{3,1}(\text{oper}(\omega(p)')(c_3)) = h_{3,1}(c_3') = h_{2,1}(c_2') = c_1$$

and

$$\text{pred}(\omega(p)')(c_3) = x = \text{pred}(\omega(p))(c_2) = m_1(p)(c_1).$$

Thus, $(2.2; h_{3,1}, p, \omega(p)')$ holds. Requirement (3) of Definition 4.6 is proved similarly to (2). The proof is left to the reader. □

4.20. Theorem. $\leqslant$ *is transitive.*

Proof. Let $S_1 \leqslant S_2$ and $S_2 \leqslant S_3$. Then, for every finite restriction U_1 of S_1, $U_1 \leqslant_d S_2$. Since there are only finitely many predicates and instructions and only one encoding in U_1, and since the flowcharts simulating these predicates and instructions are also finite, $U_1 \leqslant_d S_2$ induces a finite restriction on S_2, i.e., there is a finite restriction U_2 of S_2 such that $U_1 \leqslant_d U_2$. Then, since $S_2 \leqslant S_3$, $U_2 \leqslant_d S_3$. Hence, by transitivity of $\leqslant_d$, $U_1 \leqslant_d S_3$. This proves the transitivity of $\leqslant$. □

Since, obviously, $\equiv$ is symmetric, it follows from Theorems 4.10 and 4.20 that $\equiv$ is an equivalence relation.

4.3. Monotonicity of the pushdown operator

In this section we will show that the application of the pushdown operator preserves the simulation of storage types, i.e., P is monotonic with respect to $\leqslant$; more precisely, (*) *if* $S_1 \leqslant S_2$, *then* $\text{P}(S_1) \leqslant \text{P}(S_2)$. Although this theorem is intuitively clear, it considerably simplifies some proofs of simulations of storage types in which the pushdown operator occurs iteratively: first, the desired simulation can be shown for $\text{P}(S)$, and then this connection established by the simulation relation can be lifted by (*) to any number of applications of the pushdown operator. This technique is used, e.g., in Theorem 8.12.

In order to gain some technical convenience for the proof of the monotonicity of P, but also for later proofs in which the pushdown operator is involved, we will first show that the operator P can renounce $\text{stay}(\gamma, f)$ instructions without losing power, cf. Lemma 3.31.

4.21. Theorem. $P(S) \equiv P_1(S)$.

Proof. Obviously, we only have to prove that $P(S) \leqslant P_1(S)$. Let $U = (C', P'_f, F'_f, m'_f, I', \{e'\})$ be a finite restriction of $P(S)$, where $e' = \lambda u \in I.(\gamma_0, e(u))$ for some $\gamma_0 \in \Gamma$ and $e \in E$ (cf. Definition 3.28).

Let # be a symbol in Γ which does not occur in U. Define $h: C' \to C'$ as follows: if

$$c' = (\gamma_n, c_n)(\gamma_{n-1}, c_{n-1}) \ldots (\gamma_1, c_1) \in C'$$

for some $n \geqslant 1$, $\gamma_1, \ldots, \gamma_n \in \Gamma$, and $c_1, \ldots, c_n \in C$, then, for all $\beta_1, \ldots, \beta_n \in (\{\#\} \times C)^*$,

$$c'' = (\gamma_n, c_n)\beta_n(\gamma_{n-1}, c_{n-1})\beta_{n-1} \ldots (\gamma_1, c_1)\beta_1 \in \mathrm{dom}(h)$$

and $h(c'') = c'$.

Requirement (1) *of Definition* 4.6: For $e_1 = e'$ we can take $e_2 = e'$, too. Let $u \in \mathrm{dom}(e')$. Obviously, $e'(u) \in \mathrm{dom}(h)$ and $h(e'(u)) = e'(u)$.

Requirement (2) *of Definition* 4.6: Let $\phi \in P'_f$. Then define the $P_1(S)$-flowchart $(\{A_{in}, \mathbf{true}, \mathbf{false}\}, \_, \_, A_{in}, R)$ for predicates, where R contains the rule

$$A_{in} \to \mathbf{if}\ \phi\ \mathbf{then}\ \mathbf{true}(\mathrm{id})\ \mathbf{else}\ \mathbf{false}(\mathrm{id}).$$

Thus, every predicate is simulated 'by itself'.

Requirement (3) *of Definition* 4.6: *Case* (a): Let $\phi \in \{\mathrm{push}(\gamma, f), \mathrm{stay}(\gamma), \mathrm{id} \mid \gamma \in \Gamma, f \in F\}$. Then define the $P_1(S)$-flowchart $\omega(\phi) = (\{A_{in}, \mathbf{stop}), \_, \_, A_{in}, R)$ for instructions, where R contains the rule $A_{in} \to \mathbf{stop}(\phi)$. Again ϕ is simulated 'by itself'.

Case (b): Let $\phi = \mathrm{pop}$. Then define the $P_1(S)$-flowchart $\omega(\phi) = (\{A_{in}, A, \mathbf{stop}\}, \_, \_, A_{in}, R)$ for instructions, where R contains the rules $A_{in} \to A(\mathrm{pop})$ and

$$A \to \mathbf{if}\ \mathrm{top} = \#\ \mathbf{then}\ A(\mathrm{pop})\ \mathbf{else}\ \mathbf{stop}(\mathrm{id}).$$

Case (c): Let $\phi = \mathrm{stay}(\gamma, f)$. Then define the $P_1(S)$-flowchart $\omega(\phi) = (\{A_{in}, A, \mathbf{stop}\}, \_, \_, A_{in}, R)$ for instructions, where R contains the rules $A_{in} \to A(\mathrm{stay}(\#))$ and $A \to \mathbf{stop}(\mathrm{push}(\gamma, f))$.

It is easy to check that the requirements of Definition 4.6 are fulfilled. Note in particular that the flowcharts satisfy the requirements of Definition 4.3. □

Now we prove the monotonicity of P.

4.22. Theorem. *If* $S_1 \leqslant S_2$, *then* $P(S_1) \leqslant P(S_2)$ *and* $P_1(S_1) \leqslant P_1(S_2)$.

Proof. Let $S_1 \leqslant S_2$. We prove that $P_1(S_1) \leqslant P(S_2)$. Then, by Theorem 4.21 and by the transitivity of $\leqslant$, the monotonicity of P and P_1 is proved.

Consider a finite restriction U_1' of $P_1(S_1)$; it clearly induces a finite restriction $U_1 = (C_1, (P_1)_f, (F_1)_f, (m_1)_f, I_1, \{e_1\})$ of S_1. Since $S_1 \leqslant S_2$, it follows that $U_1 \leqslant_d S_2$. Let $h: C_2 \to C_1$ be the involved representation function. Let Γ_f be the finite set of symbols of Γ which occur in U_1'. We prove that $U_1' \leqslant_d P(S_2)$.

We define the representation function $h': C_2' \to C_1'$, where C_i' is the set of $P(S_i)$-configurations with $i \in \{1, 2\}$ as follows: if $c_1, \ldots, c_n \in \mathrm{dom}(h)$ for some $n \geqslant 1$ and $\gamma_1, \ldots, \gamma_n \in \Gamma_f$, then,

$$c' = (\gamma_n, c_n)(\gamma_{n-1}, c_{n-1}) \ldots (\gamma_1, c_1) \in \mathrm{dom}(h')$$

and

$$h'(c') = (\gamma_n, h(c_n))(\gamma_{n-1}, h(c_{n-1})) \ldots (\gamma_1, h(c_1)).$$

Requirement (1) *of Definition* 4.6: Let e_1' be the encoding of U_1'. Then there is a $\gamma_0 \in \Gamma$ such that $e_1' = \lambda u \in I.(\gamma_0, e_1(u))$. Since $U_1 \leqslant_d S_2$, there is an $e_2 \in E_2$ such that $\mathrm{dom}(e_1) = \mathrm{dom}(e_2) = \mathrm{dom}(e_2 \circ h)$ and, for every $u \in \mathrm{dom}(e_2)$, $h(e_2(u)) = e_1(u)$. Define $e_2' = \lambda u \in I.(\gamma_0, e_2(u))$. This e_2' clearly satisfies the requirements.

Requirement (2) *of Definition* 4.6: For every $p' \in (P_1')_f$ the required $P(S_2)$-flowchart for predicates is defined by providing the set of nonterminals N and the set of rules R. If no confusion arises, we drop the definition of the set of nonterminals and assume that A_{in} is the initial nonterminal.

$p' \in (P_1')_f$	$\omega(p') \in P\text{-FC}(P(S_2))$
$\mathrm{top} = \gamma$	$A_{in} \to$ **if** $\mathrm{top} = \gamma$ **then true**(id) **else false**(id)
$\mathrm{test}(p)$	*see* (☆)

(☆) Let $\omega(p) = (N(p), \_, \_, A_{in}, R(p))$ be the S_2-flowchart for predicates such that requirement (2) of Definition 4.6 holds with respect to p. Define $\omega(\mathrm{test}(p)) = (N(p), \_, \_, A_{in}, R)$ with R as follows:

(i) if $A \to$ **if** b **then** $B(g)$ in $\omega(p)$ with $g \in F_2$, then, for every $\gamma \in \Gamma_f$, $A \to$ **if** $\mathrm{top} = \gamma$ **and** $\mathrm{test}(b)$ **then** $B(\mathrm{stay}(\gamma, g))$ is in R;

(ii) if $A \to$ **if** b **then** $B(\mathrm{id})$ is in $\omega(p)$, then $A \to$ **if** $\mathrm{test}(b)$ **then** $B(\mathrm{id})$ is in R.

Intuitively, $\omega(p)$ is executed on the S_2-configuration in the topmost square of a given pushdown.

The proof of requirement (2) for $\mathrm{top} = \gamma$ and $\omega(\mathrm{top} = \gamma)$ is left to the reader. Here we consider $\mathrm{test}(p)$ and $\omega(\mathrm{test}(p))$ as an exercise.

(2.1.1): Let $c_1' \in \mathrm{dom}(h')$. Then $c_1' = (\gamma, c_1)\beta$ for some $\gamma \in \Gamma$, $c_1 \in \mathrm{dom}(h)$, and $\beta \in \mathrm{dom}(h')$. Since $c_1 \in \mathrm{dom}(h)$, $\mathrm{oper}(\omega(p))(c_1)$ is defined. Hence, there is an $x \in$ {**true**, **false**} and a $c_2 \in \mathrm{dom}(h)$ such that

$$A_{in}(c_1) \Rightarrow^*_{\omega(p)} x(c_2) \quad \text{and} \quad h(c_2) = h(c_1) \quad \text{and} \quad x = m_1(p)(h(c_1)).$$

Then $\omega(\mathrm{test}(p))$ can compute as follows.

$$A_{in}(c_1') = A_{in}((\gamma, c_1)\beta) \Rightarrow^* x((\gamma, c_2)\beta).$$

Hence, $\mathrm{oper}(\omega(\mathrm{test}(p)))(c_1')$ is defined.

(2.1.2): Since $c_2 \in \mathrm{dom}(h)$ and $\beta \in \mathrm{dom}(h')$, $\mathrm{oper}(\omega(\mathrm{test}(p)))(c_1') = (\gamma, c_2)\beta \in \mathrm{dom}(h')$.

(2.2): $h'(\mathrm{oper}(\omega(\mathrm{test}(p)))(c_1')) = h'((\gamma, c_2)\beta) = (\gamma, h(c_2))h'(\beta) = (\gamma, h(c_1))h'(\beta) = h'(c_1')$ and $\mathrm{pred}(\omega(\mathrm{test}(p)))(c_1') = x = m_1(p)(h(c_1)) = m_1'(\mathrm{test}(p))((\gamma, h(c_1))h'(\beta)) = m_1'(\mathrm{test}(p))(h'(c_1'))$.

Requirement (3) *of Definition* 4.6: Again, we only provide the nonterminals and the rules of the required flowcharts.

$f' \in (F_1')_f$	$\omega(f') \in F\text{-FC}(\mathrm{P}(S_2))$
pop	$A_{\mathrm{in}} \to \mathbf{stop}(\mathrm{pop})$
id	$A_{\mathrm{in}} \to \mathbf{stop}(\mathrm{id})$
stay(γ)	$A_{\mathrm{in}} \to \mathbf{stop}(\mathrm{stay}(\gamma))$
push(γ, f)	*see* (★)

(★) Let $\omega(f) = (N(f), \_, \_, A_{\mathrm{in}}, R(f))$ be the S_2-flowchart such that requirement (3) of Definition 4.6 holds. Then define the $\mathrm{P}(S_2)$-flowchart $\omega(\mathrm{push}(\gamma, f))$ for instructions as follows:

- $N = \{A_{\mathrm{in}}, \mathbf{stop}\} \cup \{\langle A, i\rangle \mid A \in N(f), i \in \{1, 2\}\}$;
- R is defined by (i)–(iv):
  - (i) $A_{\mathrm{in}} \to \langle A_{\mathrm{in}}, 1\rangle(\mathrm{id})$ is in R;
  - (ii) if $A \to \mathbf{if}\ b\ \mathbf{then}\ B(g)$ with $g \in F_2$ is a rule in $\omega(f)$, then

$$\langle A, 1\rangle \to \mathbf{if}\ \mathrm{test}(b)\ \mathbf{then}\ \langle B, 2\rangle(\mathrm{push}(\gamma, g))$$

and

$$\langle A, 2\rangle \to \mathbf{if}\ \mathrm{top} = \delta\ \mathbf{and}\ \mathrm{test}(b)\ \mathbf{then}\ \langle B, 2\rangle(\mathrm{stay}(\delta, g))$$

for every $\delta \in \Gamma_f$ are in R;

  - (iii) if $A \to \mathbf{if}\ b\ \mathbf{then}\ B(\mathrm{id})$ is in $\omega(f)$, then $\langle A, i\rangle \to \mathbf{if}\ \mathrm{test}(b)\ \mathbf{then}\ \langle B, i\rangle(\mathrm{id})$ with $i \in \{1, 2\}$ is in R;
  - (iv) $\langle \mathbf{stop}, 2\rangle \to \mathbf{stop}(\mathrm{id})$ is in R.

Intuitively, since it is possible that F_2 does not contain an identity, $\omega(\mathrm{push}(\gamma, f))$ cannot start its simulation by the application of push(γ, id), but has to 'wait' with the push until it meets an instruction of F_2 (it then goes from 'state 1' into 'state 2').

It is an easy observation that requirement (3) holds for the presented flowcharts. □

5. Characterization of CFT(S) by indexed S-transducers

In this section we are going to connect the classes CFT(S) and RT(P(S)). This is achieved in three subsections. In Section 5.1 we will implement a CFT(S)-transducer on an indexed S-transducer (cf. Theorem 5.8). But, in general, an indexed S-transducer is more powerful than a CFT(S)-transducer. For instance, a macro tree transducer (which is a CFT(TR)-transducer) can produce, for one input tree,

only a finite number of output trees, whereas this number may clearly be infinite for indexed tree transducers (cf. Theorem 5.9). In order to establish a stronger connection between the classes CFT(S) and RT(P(S)), there are two possibilities: restrict the class RT(P(S)) or extend the class CFT(S). In Section 5.2 the first solution will be realized. By restricting the pushdown operator in such a way that the number of excursions from a given pushdown square (initiated by a push instruction) is bounded, a characterization of CFT(S) in terms of modified indexed S-transducers is accomplished (cf. Theorem 5.14). The last subsection of this section will cover the second solution. The concept of CFT(S)-transducer is extended by allowing identity instructions in particular places. Thereby we will obtain a characterization of indexed S-transducers in terms of extended CFT(S)-transducers (cf. Theorem 5.24). However, for the total deterministic case, the two modifications fall together (cf. Theorem 5.16 and Corollary 5.25).

5.1. Implementation of CFT(S) *on indexed S-transducers*

For the implementation of CFT(S) on indexed S-transducers, the main idea is to get rid of the context parameters of nonterminals of the CFT(S)-transducer. Instead of providing a direct construction, as it is done in [24] for the simulation of an OI macro grammar by an indexed grammar, we first define an operator on storage types, which is called tree-pushdown (denoted by TP). The rewriting process of a CFT(S)-transducer is imitated by the instructions of the storage type TP(S). This will be expressed in Theorem 5.5, where CFT(S) is characterized by RT(TP(S))-transducers. Finally, since TP(S) can be simulated by P(S) (cf. Lemma 5.6), the application of the justification theorem provides the desired simulation result (cf. Theorem 5.8).

The concept of the tree-pushdown was introduced in [29] and formulated as a storage type in [10]. The idea goes back to Rounds (cf. [37]). He considered index-erasing and index-creating productions, where the first type of production corresponds to the select instruction in the tree storage type and the second type corresponds to the push instruction of the pushdown storage type. Intuitively, a tree-pushdown is, as the name says, a pushdown in the form of a tree, where the top of the pushdown is the root of the tree. If a 'push' instruction is applied to a tree-pushdown t, then the root of t is replaced by a tree ζ, which is specified in the 'push' instruction. The tree ζ may contain variables $y_1, y_2, \ldots$ at its leaves. The corresponding subtree-pushdowns of the root of t are substituted for the variables in ζ. (The situation in which ζ is a variable clearly corresponds to a pop in P(S).) Obviously, the meaning of such 'push' instructions is closely related to the way in which a rule is applied in CFT(S)-transducers. To stress the connection with CFT(S)-transducers, 'push' will be called 'expand' and the predicates of TP(S) have the form 'call $= \sigma$'. Actually, 'call $= \sigma$' and 'expand' refer to the program-schematic point of view of macro tree transducers: the call of a function procedure σ is expanded (or replaced) by its body.

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

(i) $C'=T_{\Omega\times C}$ and if σ has rank k, then, for every $c\in C$, $\langle\sigma, c\rangle$ has also rank k;

(ii) $P'=\{\mathrm{call}=\delta \mid \delta\in\Omega\}\cup\{\mathrm{test}(p) \mid p\in P\}$;

(iii) $F'=\{\mathrm{expand}(\zeta) \mid \zeta\in T_\Phi(Y)\}$ with $\Phi=\Omega\times F$ and if σ has rank k, then, for every $f\in F$, $\langle\sigma, f\rangle$ has also rank k;

(iv) for every $c'=\langle\sigma, c\rangle(t_1, \ldots, t_k)\in C'$, $m'(\mathrm{call}=\delta)(c')=\mathbf{true}$ iff $\delta=\sigma$, $m'(\mathrm{test}(p))(c')=m(p)(c)$, and

$$m'(\mathrm{expand}(\zeta))(c')=\begin{cases} c''=\zeta[f\leftarrow m(f)(c); f\in F][y_1\leftarrow t_1, \ldots, y_k\leftarrow t_k] \\ \qquad\qquad \text{if } c''\in C', \\ \text{undefined} \quad \text{otherwise;}\end{cases}$$

(v) $I'=I$; and

(vi) $E'=\{\lambda u\in I.\alpha(e(u)) \mid \alpha\in T_\Omega,\ e\in E, \text{ and } \alpha(e(u))=\alpha[\gamma\leftarrow\langle\gamma, e(u)\rangle;\ \gamma\in\Omega]\}$.

Note that the condition $c''\in C'$ (in the definition of $m'(\mathrm{expand}(\zeta))$ requires that every parameter in ζ is in Y_k and that every instruction of ζ is defined on c. In order to avoid confusion with parentheses in the tree notation, we denote the ordered pairs in the tree-pushdowns by $\langle\ ,\ \rangle$ instead of $(\ ,\ )$.

As for the pushdown operator, we abbreviate $\mathrm{TP}(S_0)$ by TP. Actually, the storage type TP is used in the pushdown tree automaton of [29] which is closely related to the RT(TP)-transducer.

The form of tests occurring in rules of $X(\mathrm{TP}(S))$-transducers can be restricted to a simple form.

5.2. Lemma. *For every $X(\mathrm{TP}(S))$-transducer there is an equivalent $X(\mathrm{TP}(S))$-transducer in which every test has the form* $\mathrm{call}=\delta$ **and** $\mathrm{test}(b)$, *where $\delta\in\Omega$ and $b\in\mathrm{BE}(P)$ and P is the set of predicate symbols of S. Determinism and totality are preserved.*

Proof. The proof of Lemma 3.30 can be taken over. □

By viewing the tree-pushdown symbols σ in a configuration $c'=\langle\sigma, c\rangle(t_1, \ldots, t_k)$ of $\mathrm{TP}(S)$ as nonterminals of a $\mathrm{CFT}(S)$-transducer $\mathfrak{M}$, the meaning of the instruction $\mathrm{expand}(\zeta)$ applied to c' and the application of the rule $\sigma(y_1, \ldots, y_k)\to\zeta$ of $\mathfrak{M}$ to c' (viewed as a sentential form) correspond to each other: the nonterminal (or tree-pushdown symbol) σ is expanded to ζ, the instructions in ζ are applied to c, and, for every $i\in[k]$, t_i is substituted for y_i. Hence, it is not surprising to obtain the following characterization, where $1X(S)$ is the class of translations induced by $X(S)$-transducers with one nonterminal only.

5.3. Lemma. $\mathrm{CFT}(S)=1\mathrm{RT}(\mathrm{TP}(S))$, $\mathrm{MAC}(S)=1\mathrm{CF}(\mathrm{TP}(S))$, *and determinism and totality are preserved.*

Proof. (CFT(S) $\subseteq$ 1RT(TP(S))): Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be a CFT(S)-transducer. Note that $A_{\text{in}} \in T_n$. We will construct the RT(TP(S))-transducer $\mathfrak{M}''$ with one nonterminal, say $*$, such that, roughly, its sentential forms are obtained from those of $\mathfrak{M}$ by 'inserting' $*$ just above each outermost occurrence of a nonterminal of $\mathfrak{M}$. Thus, if a sentential form of $\mathfrak{M}$ is of the form $\tilde{t}[\xi_1, \ldots, \xi_n]$, where $\tilde{t}$ contains terminals only and the roots of $\xi_1, \ldots, \xi_n$ correspond to outermost nonterminals, then the corresponding sentential form of $\mathfrak{M}''$ is $\tilde{t}[*(\xi_1), \ldots, *(\xi_n)]$. Note that for $\mathfrak{M}''$, the ξ_i's are tree-pushdown configurations (and the nonterminals of $\mathfrak{M}$ are tree-pushdown symbols).

For convenience we first show that $\mathfrak{M}$ can be transformed equivalently into a CFT(S)-transducer $\mathfrak{M}'$ such that in the parameter positions of nonterminals in the right-hand side trees of rules in R, no terminal symbol occurs. This is achieved as usual by replacing each terminal symbol δ in parameter positions by a nonterminal $\langle\delta\rangle$ and by adding appropriate rules to R. The only problem is to find an instruction symbol which can be attached to $\langle\delta\rangle$ and which does not block the derivation. For this purpose, we define for every $f \in F$ mappings $h, h_f: T_{N(F)\cup\Delta}(Y) \to T_{N'(F)\cup\Delta}(Y)$, where

$$N' = N \cup \{\langle\delta\rangle^{(k)} \mid \delta \in \Delta_k \text{ with } k \geq 0\}$$

as follows: (i) for $y \in Y$, $h(y) = h_f(y) = y$; (ii) for $\zeta = \delta(\zeta_1, \ldots, \zeta_k) \in T_{N(F)\cup\Delta}(Y)$, where $\delta \in \Delta_k$ with $k \geq 0$, $h(\zeta) = \delta(h(\zeta_1), \ldots, h(\zeta_k))$ and $h_f(\zeta) = \langle\delta\rangle(f)(h_f(\zeta_1), \ldots, h_f(\zeta_k))$; (iii) for $\zeta = A(g)(\zeta_1, \ldots, \zeta_k) \in T_{N(F)\cup\Delta}(Y)$, where $A \in N_k$ with $k \geq 0$ and $g \in F$, $h(\zeta) = A(g)(h_g(\zeta_1), \ldots, h_g(\zeta_k))$ and $h_f(\zeta) = A(g)(h_g(\zeta_1), \ldots, h_g(\zeta_k))$.

The CFT(S)-transducer $\mathfrak{M}' = (N', e, \Delta, A_{\text{in}}, R')$ is constructed by providing the set of rules R'.

(i) If $A(y_1, \ldots, y_k) \to$ **if** b **then** ζ is in R, then $A(y_1, \ldots, y_k) \to$ **if** b **then** $h(\zeta)$ is in R'.

(ii) For every $\delta \in \Delta_k$ with $k \geq 0$ occurring in the right-hand side of a rule in R, $\langle\delta\rangle(y_1, \ldots, y_k) \to \delta(y_1, \ldots, y_k)$ is in R'. Clearly, $\tau(\mathfrak{M}') = \tau(\mathfrak{M})$ and determinism and totality are preserved by this construction.

Since Ω is infinite, we can assume that $N' \subseteq \Omega$ and ranks are preserved in this inclusion.

For every tree $\zeta \in T_{N'(F)\cup\Delta}(Y)$ of the form $\tilde{t}[\zeta_1, \ldots, \zeta_n]$, where $\tilde{t} \in T_\Delta(Z_n)$ with $Z_n = \{z_1, \ldots, z_n\}$ and $n \geq 0$, and for every $i \in [n]$, $\zeta_i \in T_{N'(F)}(Y)$, we define

$$\phi(\zeta) = \tilde{t}[*(\text{expand}(\zeta_1')), \ldots, *(\text{expand}(\zeta_n'))],$$

where $*$ is a new symbol and $\zeta_i' = \zeta_i[A(f) \leftarrow \langle A, f\rangle; A(f) \in N'(F)]$.

Then construct the 1RT(TP(S))-transducer $\mathfrak{M}'' = (\{*\}, e'', \Delta, *, R'')$ by $e'' = \lambda u \in I.A_{\text{in}}[A \leftarrow \langle A, e(u)\rangle; A \in N]$, and if $A(y_1, \ldots, y_k) \to$ **if** b **then** ζ is in R', then

$$* \to \textbf{if call} = A \textbf{ and test}(b) \textbf{ then } \phi(\zeta)$$

is in R''. The following claim connects the derivations of $\mathfrak{M}'$ and $\mathfrak{M}''$. Let $\Psi = N'(C)$.

Claim 7. *For every* $A \in N'_k$ *with* $k \geq 0$, $c \in C$, $\xi_1, \ldots, \xi_k \in T_\Psi$, *and every* $\xi \in T_\Delta(T_\Psi)$,

$$A(c)(\xi_1, \ldots, \xi_k) \Rightarrow_{\mathfrak{M}'} \xi \text{ iff } \Theta(A(c)(\xi_1, \ldots, \xi_k)) \Rightarrow_{\mathfrak{M}''} \Theta(\xi),$$

where Θ *is defined on trees* ξ *of the form* $\tilde{t}[\xi_1, \ldots, \xi_n]$ *where* $\tilde{t} \in T_\Delta(Z_n)$ *and, for every* $i \in [n]$, $\xi_i \in T_\Psi$ *by*

$$\Theta(t) = \tilde{t}[*(\xi'_1), \ldots, *(\xi'_n)] \quad \text{and} \quad \xi'_i = \xi_i[A(c) \leftarrow \langle A, c\rangle; A(c) \in N'(C)].$$

It is obvious that Claim 7 implies $\tau(\mathfrak{M}') = \tau(\mathfrak{M}'')$. Moreover, determinism is preserved by the construction. Since $\mathrm{dom}(e'') = \mathrm{dom}(e)$ and $\tau(\mathfrak{M}) = \tau(\mathfrak{M}'')$, totality is also preserved.

$(1\mathrm{RT}(\mathrm{TP}(S)) \subseteq \mathrm{CFT}(S))$: Let $\mathfrak{M} = (\{*\}, \lambda u \in I.\alpha(e(u)), \Delta, *, R)$ be a $1\mathrm{RT}(\mathrm{TP}(S))$-transducer, where $\alpha \in T_\Omega$ and e is an encoding of S. By Lemma 5.2 we can assume that the tests in rules of R have the form call $= \delta$ **and** test(b) with $\delta \in \Omega$ and $b \in \mathrm{BE}(P)$.

We construct the $\mathrm{CFT}(S)$-transducer $\mathfrak{M}' = (N', e, \Delta, \alpha, R')$ by

$$N' = \{\delta^{(k)} \mid \delta \in \Omega,\ \delta \text{ is of rank } k \text{ with } k \geq 0, \text{ and } \delta \text{ occurs in an argument of an 'expand' instruction of some rule in } R \text{ or in } \alpha\},$$

and R' is defined as follows: if

$$* \to \textbf{if call} = \sigma \textbf{ and test}(b) \textbf{ then } \zeta$$

is a rule in R, where σ has rank k with $k \geq 0$ and ζ contains only parameters of the set $\{y_1, \ldots, y_k\}$, then

$$\sigma(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ then } \phi^{-1}(\zeta)$$

is in R', where ϕ is defined in the first part of this proof. Obviously, this construction preserves determinism.

Analogous to Claim 7 we can prove Claim 8 from which the equality of the translations $\tau(\mathfrak{M})$ and $\tau(\mathfrak{M}')$ immediately follows. Let C' be the set of $\mathrm{TP}(S)$-configurations.

Claim 8. *For every* $c' \in C'$, $\xi' \in T_\Delta(\{*\}(C'))$, *and* $t \in T_\Delta$,

$$*(c') \Rightarrow_{\mathfrak{M}} \xi' \Rightarrow^*_{\mathfrak{M}} t \text{ iff } \Theta^{-1}(*(c')) \Rightarrow_{\mathfrak{M}'} \Theta^{-1}(\xi') \Rightarrow^*_{\mathfrak{M}'} t.$$

Since $\mathrm{dom}(\lambda u \in I.\alpha(e(u))) = \mathrm{dom}(e)$ and $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$, totality is preserved.

Finally, since $\mathrm{MAC}(S) = \mathrm{yield}(\mathrm{CFT}(S))$ and $\mathrm{yield}(1\mathrm{RT}(S)) = 1\mathrm{CF}(S)$, we also obtain $\mathrm{MAC}(S) = 1\mathrm{CF}(\mathrm{TP}(S))$ and again determinism and totality are preserved. □

In [37] it is stated that a context-free tree grammar is exactly a one-state creative dendrogrammar (creative dendrogrammars are tree grammars which use index-creating productions). It is an easy observation that creative dendrolanguages are precisely the ranges of RT(TP)-transducers (cf. [29, 10]). Hence, from Lemma 5.3 (with $S = S_0$) we reobtain the statement of Rounds in the form $\mathrm{CFT} = \mathrm{range}(\mathrm{CFT}(S_0)) = \mathrm{range}(1\mathrm{RT}(\mathrm{TP}))$ (or, applying yield, $\mathrm{MAC} = \mathrm{range}(1\mathrm{CF}(\mathrm{TP}))$).

Rounds also proved that every creative dendrolanguage can be generated by a one-state creative dendrogrammar (cf. Theorem 7 in [37]). We take over his construction and generalize the result.

5.4. Lemma

$$\mathrm{RT}(\mathrm{TP}(S)) = 1\mathrm{RT}(\mathrm{TP}(S)), \qquad \mathrm{CF}(\mathrm{TP}(S)) = 1\mathrm{CF}(\mathrm{TP}(S)),$$

and determinism and totality are preserved.

Proof. Obviously, we only have to show one direction. Let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$ be an $\mathrm{RT}(\mathrm{TP}(S))$-transducer. By Lemma 5.2 we can assume that the tests of the rules have the form call $= \delta$ **and** test(b). Let $N = \{A_1, \ldots, A_r\}$ for some $r \geq 1$ and let $A_{\mathrm{in}} = A_1$. Furthermore, let $e = \lambda u \in I.\alpha(g(u))$ for some $\alpha \in T_\Omega$ and some encoding g of the storage type S. Let C' denote the configuration set of $\mathrm{TP}(S)$.

The idea of the construction is to simulate an instruction expand(ζ) by expand(ζ'), where ζ' is obtained from ζ by copying every subtree of ζ as many times as there are nonterminals in $\mathfrak{M}$, i.e., r times. In the root of each copy the corresponding nonterminal is encoded. Then, if $\mathfrak{M}$ applies a rule with right-hand side $\ldots A_j(\mathrm{expand}(y_i))\ldots$ to a configuration $\langle \sigma, c\rangle(t_1, \ldots, t_k)$, the constructed transducer would continue with the jth copy of t_i and find A_j in the label of its root. Thus, corresponding to each tree-pushdown symbol $\sigma \in \Omega_k$ we need new tree-pushdown symbols $(A_1, \sigma), \ldots, (A_r, \sigma)$ of rank $r \cdot k$: each son of σ is replaced by a sequence of r consecutive sons of (A_i, σ). Without loss of generality, we may assume that these new symbols are again in Ω. Formally, we define for every $j \in [r]$ a mapping $\langle\ \rangle_j : T_{\Omega\times\Psi}(Y) \to T_{\Omega'\times\Psi}(Y)$, where Ω' is the ranked set

$$\{(A_i, \sigma)^{(r\cdot k)} \mid \sigma \in \Omega \text{ of rank } k \geq 0 \text{ and } i \in [r]\}$$

and Ψ is an arbitrary set. The argument of the mapping $\langle\ \rangle_j$ is put between the brackets.

(i) For $y_i \in Y$, $\langle y_i\rangle_j = y_\rho$ where $\rho = (i-1)\cdot r + j$.

(ii) For $\sigma \in \Omega$ of rank k with $k \geq 0$, $\psi \in \Psi$, and $t_1, \ldots, t_k \in T_{\Omega\times\Psi}(Y)$,

$$\langle\langle \sigma, \psi\rangle(t_1, \ldots, t_k)\rangle_j$$
$$= \langle (A_j, \sigma), \psi\rangle(\langle t_1\rangle_1, \ldots, \langle t_1\rangle_r, \langle t_2\rangle_1, \ldots, \langle t_2\rangle_r, \ldots, \langle t_k\rangle_1, \ldots, \langle t_k\rangle_r).$$

Now we construct the $1\mathrm{RT}(\mathrm{TP}(S))$-transducer $\mathfrak{M}' = (\{*\}, e', \Delta, *, R')$ where $*$ is a new nonterminal, as follows. $e' = \lambda u \in I.\langle \alpha(g(u))\rangle_1$, where $\Psi = \{g(u)\}$, and R' is defined as follows: if

$$A_j \to \textbf{if call} = \delta \textbf{ and test}(b) \textbf{ then } t[A_{\nu(1)}(\mathrm{expand}(t_1)), \ldots, A_{\nu(k)}(\mathrm{expand}(t_k))]$$

is in R, where $t \in T_\Delta(Z_k)$ for some $k \geq 0$ and $\nu(1), \ldots, \nu(k) \in [r]$, then

$$* \to \textbf{if call} = (A_j, \delta) \textbf{ and test}(b) \textbf{ then}$$
$$t[*(\mathrm{expand}(\langle t_1\rangle_{\nu(1)})), \ldots, *(\mathrm{expand}(\langle t_k\rangle_{\nu(k)}))]$$

is a rule in R' (where, of course, $\Psi = F$). Note that if $\mathfrak{M}$ is deterministic, then so is $\mathfrak{M}'$.

The following claim can easily be proved by induction on n (where, this time, $\Psi = C$).

Claim 9. *For every $n \geq 0$ and j, $\nu(1), \ldots, \nu(k) \in [r]$ with $k \geq 0$, every $\xi, \xi_1, \ldots, \xi_k \in C'$, and every $t \in T_\Delta(Z_k)$,*

$$A_j(\xi) \Rightarrow_{\mathfrak{M}}^n t[A_{\nu(1)}(\xi_1), \ldots, A_{\nu(k)}(\xi_k)]$$

$$\text{iff } *(\langle \xi \rangle_j) \Rightarrow_{\mathfrak{M}'}^n t[*(\langle \xi_1 \rangle_{\nu(1)}), \ldots, *(\langle \xi_k \rangle_{\nu(k)})].$$

Taking $j = 1$, $\xi = \alpha(g(u))$, and $t \in T_\Delta$, from Claim 9 it follows that $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$. Since $\mathrm{dom}(e) = \mathrm{dom}(g) = \mathrm{dom}(e')$, totality is preserved.

This proves $\mathrm{RT}(\mathrm{TP}(S)) = 1\mathrm{RT}(\mathrm{TP}(S))$. The equality $\mathrm{CF}(\mathrm{TP}(S)) = 1\mathrm{CF}(\mathrm{TP}(S))$ is obtained by applying 'yield'. □

From the previous two lemmata we obtain a (regular) characterization of $\mathrm{CFT}(S)$ in terms of $\mathrm{RT}(\mathrm{TP}(S))$-transducers. For $S = S_0$ it shows that the context-free grammars and the creative dendrogrammars are equivalent [37], and that context-free tree languages are accepted by pushdown tree automata [29]. Actually, since the tree-pushdown is a special pushdown device, we have obtained the first pushdown machine for $\mathrm{CFT}(S)$ and $\mathrm{MAC}(S)$.

5.5. Theorem

$$\mathrm{CFT}(S) = \mathrm{RT}(\mathrm{TP}(S)), \qquad \mathrm{MAC}(S) = \mathrm{CF}(\mathrm{TP}(S)),$$

and determinism and totality are preserved.

Proof. Immediate from Lemma 5.3 and Lemma 5.4. □

Now we will show how the storage type $\mathrm{TP}(S)$ can be simulated by $\mathrm{P}(S)$, i.e., we will prove that $\mathrm{TP}(S) \leq \mathrm{P}(S)$. Following Definition 4.6, we have to specify a representation function h and construct flowcharts which simulate instructions of the form expand(ζ). Let us first discuss these flowcharts a bit; then the main idea of the representation function is immediately understandable.

Assume that a tree-pushdown configuration of the form $c' = \langle \sigma, c \rangle(t_1, \ldots, t_k)$ is represented by the pushdown configuration $\beta' = (\sigma, c)\beta$. Now, the instruction expand(ζ) is simulated by the $\mathrm{P}(S)$-flowchart ω which first replaces σ in β' by ζ (viewed as a term) via a stay instruction. Then, either ζ has the form $\langle \gamma, f \rangle(\xi_1, \ldots, \xi_\nu)$ for some instruction f of the underlying storage type S, or ζ has the form y_j.

In the first case, the subtrees of ζ are put as a list of terms on the pushdown β' via the instruction stay$((\xi_1, \ldots, \xi_\nu))$ followed by the instruction push(γ, f).

In the second case, ω first applies a pop to β'. Actually, the first component of the topmost pushdown square of β now contains a list of trees, because this is the

way in which the pushdown is built up. Then ω considers the jth component of this list which is possibly just another y, say y_ν. In this case, ω pops again and examines the νth component of the list now appearing on top of the pushdown. ω continues this process of popping when encountering a y, until it finds a tree which has the form $\langle\gamma, f\rangle(\xi_1, \ldots, \xi_\nu)$ for some instruction f of S. Then ω finishes its simulation by applying the instructions stay$((\xi_1, \ldots, \xi_\nu))$ and push(γ, f) to the pushdown.

As already indicated above, a tree-pushdown configuration is represented by a pushdown in which the first component of every square consists of a list of trees over $\Omega \times F$ and Y with one exception: the topmost square always has the form (σ, c). Now, the representation function h just substitutes for every y_j in a tree of such a list the jth component of the list which is contained in one square below (cf. also Example 5.7). By repeatedly substituting, a sequence $(t_1, \ldots, t_k)$ of tree-pushdown configurations is obtained which is turned into a configuration of TP(S) by left-concatenation with $\langle\sigma, c\rangle$.

Unfortunately, this construction does not yet work correctly at the bottom of the pushdown: additionally, we have to store the initial tree α of the tree-pushdown into one bottom square, whereas it would need two squares according to the above description. However, it is not possible to reserve an extra square at the bottom for α, because whenever a push is applied to a one-square pushdown configuration (γ, c), an instruction is applied to c. In TP(S) however, the whole initial tree is 'applied' to the same configuration of S. But this is only a technical detail by which the reader should not be bothered. Intuitively, it is correct to view the bottom pushdown square as two squares containing the same configuration of S.

There is still another detail which we have to take care of in the simulation. It is certainly possible that ζ contains an instruction f, e.g., in its second subtree ξ_2, which is not defined on c and hence, by definition, expand(ζ) is not defined on c'. However, the flowchart which we have constructed so far, works very well. At least until the point at which the list of subtrees of ζ, namely $(\xi_1, \ldots, \xi_\nu)$, occurs again on the top of the pushdown. Now it is possible that the second component of this list is not considered anymore. Thus, our flowchart would not have recognized that there is an undefined instruction in ζ, although it should be also undefined on the representation of c'. To capture also this detail, we let the flowchart test the definedness of every instruction $f_1, \ldots, f_k$ occurring in ζ before starting the above explained simulation. This is done by the simple sequence push$(\#, f_1)$; pop; ...; push$(\#, f_k)$; pop.

5.6. Lemma. TP(S) $\leqslant$ P(S).

Proof. Let $S = (C, P, F, m, I, E)$ be a storage type. Let $U = (C', P'_f, F'_f, m'_f, I', \{e'\})$ be a finite restriction of the tree-pushdown of S. Let $F'_f = \{\text{expand}(\zeta_1), \ldots, \text{expand}(\zeta_r)\}$ for some $r \geqslant 0$ and $\zeta_i \in T_{\Omega \times F}(Y)$. Let F_f be the finite set of instructions of S which occur in the ζ_i's. Let $e' = \lambda u \in I.\alpha(e(u))$ for some

$\alpha \in T_\Omega$ and $e \in E$. Denote by Σ the finite subset of Ω such that $\Sigma = \{\sigma \mid \sigma$ occurs in $\zeta_1, \ldots, \zeta_r$, or in $\alpha\}$. Denote by n the maximal index of a parameter occurring in one of the ζ_i's, i.e., $n = \max\{i \mid y_i$ occurs in $\zeta_1, \ldots, \zeta_r\}$. We prove that $U \leqslant_d \mathrm{P}(S)$.

The input sets of U and of $\mathrm{P}(S)$ are both equal to I.

Since Γ is infinite, we can assume that NOT-BOTTOM $\cup$ BOTTOM $\subseteq \Gamma$, where NOT-BOTTOM and BOTTOM are two finite sets defined as follows:

$$\text{NOT-BOTTOM} = \Sigma \cup \text{SUB}(\{\zeta_1, \ldots, \zeta_r\}) \cup \text{seq-SUB}(\{\zeta_1, \ldots, \zeta_r\}),$$

$$\text{BOTTOM} = \text{NOT-BOTTOM} \times \text{seq-SUB}(\{\alpha\}) \cup \text{SUB}(\{\alpha\}) \cup \text{seq-SUB}(\{\alpha\}),$$

where, for every set T of trees, $\text{SUB}(T) = \{s \mid s$ is a subtree of t for some $t \in T\}$ and $\text{seq-SUB}(T) = \{(s_1, \ldots, s_k) \mid \sigma(s_1, \ldots, s_k)$ is a subtree of t for some $t \in T$ and some $\sigma \in \Sigma\}$ (if σ is of rank 0, then the empty sequence $(\ )$ is in $\text{seq-SUB}(T)$). NOT-BOTTOM and BOTTOM are sets of symbols that will occur not at the bottom of, respectively at the bottom of the pushdown during the simulation of the tree pushdown.

We define the representation function $h : (\Gamma \times C)^+ \to C'$ as follows. Only pushdowns of a special form are representing tree-pushdown configurations, namely, $\text{dom}(h) \subseteq$ PUSHDOWN, where

$$\text{PUSHDOWN} = (\Sigma \times C)(\text{seq-SUB} \times C)^*((\text{seq-SUB} \times \text{seq-SUB}(\{\alpha\})) \times C) \cup (\Sigma \times \text{seq-SUB}(\{\alpha\})) \times C,$$

where seq-SUB abbreviates $\text{seq-SUB}(\{\zeta_1, \ldots, \zeta_r\})$. Note that PUSHDOWN $\subseteq$ (NOT-BOTTOM $\times C$)*(BOTTOM $\times C$).

The values of h are defined as follows:

(i) Let $c' = ((\sigma, (\alpha_1, \ldots, \alpha_n)), c) \in (\Sigma \times \text{seq-SUB}(\{\alpha\})) \times C$ and let σ be of rank η with $\eta \geqslant 0$. Then $c' \in \text{dom}(h)$ and $h(c') = \langle \sigma, c \rangle(\alpha_1(c), \ldots, \alpha_\eta(c))$, where $\alpha_i(c) = \alpha[\gamma \leftarrow \langle \gamma, c \rangle; \gamma \in \Sigma]$ for every $i \in [\eta]$.

(ii) Let $c' = (\sigma, c)\beta$ with

$$\beta = (\tilde{\xi}_1, c_1)(\tilde{\xi}_2, c_2) \ldots (\tilde{\xi}_\rho, c_\rho)((\tilde{\xi}_{\rho+1}, \tilde{\alpha}), c_{\rho+1}),$$

where $\rho \geqslant 0$ and $c' \in$ PUSHDOWN. If σ is of rank k_1 with $k_1 \geqslant 0$ and $\tilde{\xi}_1$ is a k_1-tuple, and if, for every $i \in [\rho]$, $\tilde{\xi}_{i+1}$ is a k_{i+1}-tuple for some $k_{i+1} \geqslant \max\{j \mid y_j \in \text{par}(\tilde{\xi}_i)\}$, and if $\tilde{\alpha}$ is an η-tuple for some $\eta \geqslant \max\{j \mid y_j \in \text{par}(\tilde{\xi}_{\rho+1})\}$, then $c' \in \text{dom}(h)$ and $h(c') = \langle \sigma, c \rangle \text{subst}(\beta)$, where

$$\text{subst} : (\text{seq-SUB} \times C)^*((\text{seq-SUB} \times \text{seq-SUB}(\{\alpha\})) \times C) \to \bigcup_{\nu \geqslant 0} \{(\xi_1, \ldots, \xi_\nu) \mid \xi_i \in C'\}$$

is a partial function with $\text{dom}(\text{subst}) = \{\beta \mid (\sigma, c)\beta \in \text{dom}(h)$ for some $\sigma \in \Sigma$ and $c \in C\}$, inductively defined on the length of the pushdown as follows:

(i) if $\beta = (((\xi_1, \ldots, \xi_\nu), (\alpha_1, \ldots, \alpha_\eta)), c) \in \text{dom}(\text{subst})$, then $\text{subst}(\beta) = (\xi'_1, \ldots, \xi'_\nu)$, where

$$\xi'_i = \xi_i[f \leftarrow m(f)(c); f \in F][y_j \leftarrow \alpha_j(c); y_j \in \text{par}(\xi_i)]$$

for every $i \in [\nu]$, where $\alpha_j(c) = \alpha_j[\gamma \leftarrow \langle \gamma, c \rangle; \gamma \in \Sigma]$;

(ii) if $\beta = ((\xi_1, \ldots, \xi_\nu), c)\beta' \in \mathrm{dom}(\mathrm{subst})$, then $\mathrm{subst}(\beta) = (\xi'_1, \ldots, \xi'_\nu)$, where

$$\xi'_i = \xi_i[f \leftarrow m(f)(c); f \in F][y_j \leftarrow \mathrm{pr}_j(\mathrm{subst}(\beta')); y_j \in \mathrm{par}(\xi_i)]$$

for every $i \in [\nu]$. (pr_j projects the jth component from a tuple.) This completes the definition of subst, and thus of h.

Requirement (1) *of Definition* 4.6: Recall that $e' = \lambda u \in I.\alpha(e(u))$ is the encoding of U. Let $\alpha = \gamma(t_1, \ldots, t_k)$. Define $e'' = \lambda u \in I'.((\gamma, (t_1, \ldots, t_k)), e(u))$. Obviously, $\mathrm{dom}(e') = \mathrm{dom}(e'')$. For $u \in I'$, $e''(u) \in \mathrm{dom}(h)$. Moreover,

$$h(e''(u)) = h(((\gamma, (t_1, \ldots, t_k)), e(u))) = \langle \gamma, e(u)\rangle(t_1(e(u)), \ldots, t_k(e(u)))$$

(for every $i \in [k]$, $t_i(e(u)) = t_i[\gamma \leftarrow \langle \gamma, e(u)\rangle; \gamma \in \Sigma] = e'(u)$).

Requirement (2) *of Definition* 4.6: (i) Consider $\pi = (\mathrm{call} = \sigma)$, $\pi \in P'_f$, with σ of rank $k \geqslant 0$. Since seq-SUB($\{\alpha\}$) is a finite set, we can use the abbreviation $\mathrm{top} \in \{\sigma\} \times \text{seq-SUB}(\{\alpha\})$ as predicate with its obvious meaning. Define $\omega_\pi = (\{A_{\mathrm{in}}, \textbf{true}, \textbf{false}\}, \_, \_, A_{\mathrm{in}}, R) \in P\text{-FC}(\mathrm{P}(S))$, where R contains the rule

$$A_{\mathrm{in}} \rightarrow \textbf{if } \mathrm{top} = \sigma \textbf{ or } \mathrm{top} \in \{\sigma\} \times \text{seq-SUB}(\{\alpha\}) \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id}).$$

(ii) Consider $\pi = \mathrm{test}(p)$ with $\pi \in P'_f$. Define $\omega_\pi = (\{A_{\mathrm{in}}, \textbf{true}, \textbf{false}\}, \_, \_, A_{\mathrm{in}}, R) \in P\text{-FC}(\mathrm{P}(S))$, where R contains the rule

$$A_{\mathrm{in}} \rightarrow \textbf{if } \mathrm{test}(p) \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id}).$$

The correctness of requirement (2.2) is based on the fact that the topmost square of a pushdown configuration c'_2 which represents a tree-pushdown configuration c'_1 (i.e., $h(c'_2) = c'_1$), is exactly the label of the root of c'_1 (except when c'_2 contains only one square).

Requirement (3) *of Definition* 4.6: Consider $\phi = \mathrm{expand}(\zeta)$ with $\phi \in F'_f$. Let $f_1, \ldots, f_\kappa$ with $\kappa \geqslant 0$ be the instructions of S occurring in ζ. Define the P(S)-flowchart $\omega_\phi = (N, \_, \_, A_{\mathrm{in}}, R)$ for instructions as follows:

- $N = \{A_{\mathrm{in}}, A, \textbf{'B}, \textbf{stop}\} \cup \{p_j \mid j \in [n]\} \cup \{q_{\gamma,f} \mid \gamma \in \Sigma, f \in F_f\}$
  $\cup \{q_\gamma \mid \gamma \in \Sigma\} \cup \{A_j \mid j \in [\kappa - 1]\} \cup \{B_j \mid j \in [\kappa]\}.$

(Recall that n is the maximal index of a parameter which occurs in one of the ζ_i's.)

- R is defined by (i)–(v), as follows:

(i) If $\kappa \geqslant 1$, then $A_{\mathrm{in}} \rightarrow B_1(\mathrm{push}(\#, f_1))$ and, for every $j \in [\kappa - 1]$, $B_j \rightarrow A_j(\mathrm{pop})$, $A_j \rightarrow B_{j+1}(\mathrm{push}(\#, f_{j+1}))$, and $B_\kappa \rightarrow A(\mathrm{pop})$ are in R; if $\kappa = 0$, then $A_{\mathrm{in}} \rightarrow A(\mathrm{stay})$ is in R.

(ii) For every $\sigma \in \Sigma_\eta$ with $\mathrm{par}(\zeta) \subseteq Y_\eta$ and every $(\alpha_1, \ldots, \alpha_\eta) \in \text{seq-SUB}(\{\alpha\})$,

$$A \rightarrow \textbf{if } \mathrm{top} = \sigma \textbf{ then } B(\mathrm{stay}(\zeta)),$$

$$A \rightarrow \textbf{if } \mathrm{top} = (\sigma, (\alpha_1, \ldots, \alpha_\eta)) \textbf{ then } B(\mathrm{stay}((\zeta, (\alpha_1, \ldots, \alpha_\eta))))$$

are in R.

Note that, before writing ζ on the pushdown, we have to check that ζ does not contain too many parameters, i.e., that the rank of σ is greater than the number of

parameters in ζ. If we did not check this, then ω_ϕ might be defined, whereas ϕ would not be defined.

(iii) For every $j \in [n]$ and every $(\alpha_1, \ldots, \alpha_\eta) \in \text{seq-SUB}(\{\alpha\})$ for some $\eta \geq 0$,

$$B \to \textbf{if top} = y_j \textbf{ then } p_j(\text{pop}),$$

$$B \to \textbf{if top} = (y_j, (\alpha_1, \ldots, \alpha_\eta)) \textbf{ then } p_j(\text{stay}((\alpha_1, \ldots, \alpha_\eta)))$$

are in R.

(iv) For every $j \in [n]$ and every $(\xi_1, \ldots, \xi_\nu) \in \text{seq-SUB}(\{\zeta_1, \ldots, \zeta_r\})$ for some $\nu \geq j$ and every $(\alpha_1, \ldots, \alpha_\eta) \in \text{seq-SUB}(\{\alpha\})$ for some $\eta \geq j$, the rules

$$p_j \to \textbf{if top} = (\xi_1, \ldots, \xi_\nu) \textbf{ then } B(\text{stay}(\xi_j)),$$

$$p_j \to \textbf{if top} = (\alpha_1, \ldots, \alpha_\eta) \textbf{ then } B(\text{stay}(\alpha_j)),$$

$$p_j \to \textbf{if top} = ((\xi_1, \ldots, \xi_\nu), (\alpha_1, \ldots, \alpha_\eta)) \textbf{ then } B(\text{stay}((\xi_j, (\alpha_1, \ldots, \alpha_\eta))))$$

are in R.

(v) For every $\langle \gamma, f \rangle(\xi_1, \ldots, \xi_\nu) \in \text{SUB}(\{\zeta_1, \ldots, \zeta_r\})$ for some $\nu \geq 0$, and every $\delta \in \Sigma_\eta$ and $(\alpha_1, \ldots, \alpha_\eta) \in \text{seq-SUB}(\{\alpha\})$ for some $\eta \geq 0$, the rules

$$B \to \textbf{if top} = \langle \gamma, f \rangle(\xi_1, \ldots, \xi_\nu) \textbf{ then } q_{\gamma,f}(\text{stay}((\xi_1, \ldots, \xi_\nu))),$$

$$B \to \textbf{if top} = (\langle \gamma, f \rangle(\xi_1, \ldots, \xi_\nu), (\alpha_1, \ldots, \alpha_\eta))$$

$$\textbf{then } q_{\gamma,f}(\text{stay}((\xi_1, \ldots, \xi_\nu), (\alpha_1, \ldots, \alpha_\eta))),$$

$$q_{\gamma,f} \to \textbf{stop}(\text{push}(\gamma, f)),$$

$$B \to \textbf{if top} = \delta(\alpha_1, \ldots, \alpha_\eta) \textbf{ then } q_\delta(\text{stay}((\alpha_1, \ldots, \alpha_\eta))),$$

$$q_\delta \to \textbf{if top} = (\alpha_1, \ldots, \alpha_\eta) \textbf{ then stop}(\text{stay}((\delta, (\alpha_1, \ldots, \alpha_\eta))))$$

are in R.

The proof of requirement (3) consists of a case analysis: either ζ has the form $\langle \gamma, f \rangle(\xi_1, \ldots, \xi_\nu)$ or it is just a y_j. In the first case, the requirement is easy to prove. To prove its correctness in the second case, we recall that only those pushdowns are in $\text{dom}(h)$ for which the substitution process can be finished successfully, i.e., if y_ρ occurs as a component of a list in a pushdown square, then the list of trees contained in the square one below has at least ρ components. □

We want to convince the reader now that actually a rather easy simulation idea is buried under the technicalities of the previous proof. We hope to accomplish this by giving an example.

5.7. Example. Let σ, δ, γ, κ, and ν be symbols of Ω of rank 2, 1, 1, 0, and 0, respectively. We consider a finite restriction of TP(S) with the encoding $e' = \lambda u \in I.\alpha(e(u))$, where $\alpha = \delta(\sigma(\kappa, \nu))$ and e is some encoding of S. Let c be a configuration of S which is in range(e).

Let $c_{1,0} = \langle \delta, c\rangle(\langle \sigma, c\rangle(\langle \kappa, c\rangle, \langle \nu, c\rangle))$ be the corresponding initial configuration of TP(S). It is represented by the pushdown configuration $c_{2,0} = ((\delta, (\sigma(\kappa, \nu))), c)$, i.e., $h(c_{2,0}) = c_{1,0}$.

In order to avoid unreadable expressions we will denote a pushdown $(\ldots, c)(\ldots, c')\ldots(\ldots, c'')$ by $[\ldots|c][\ldots|c']\ldots[\ldots|c'']$ in the sequel. For instance, $c_{2,0} = [\delta, (\sigma(\kappa, \nu))|c]$. (Note that we also omit the outermost parenthesis in the first component of the bottom pushdown square.)

Now, by means of three examples, we want to show how instructions of the form expand(ζ) are simulated.

(1) Apply $\phi = \text{expand}(\langle \sigma, g\rangle(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1))$ to $c_{1,0}$. The result is

$$c_{1,1} = \langle \sigma, \tilde{g}(c)\rangle(\langle \kappa, \tilde{f}(c)\rangle, \langle \gamma, \tilde{f}(c)\rangle(\langle \sigma, c\rangle(\langle \kappa, c\rangle, \langle \nu, c\rangle))),$$

where $\tilde{f}(c)$ and $\tilde{g}(c)$ abbreviates $m(f)(c)$ and $m(g)(c)$, respectively.

Simulation of ϕ by ω_ϕ on $c_{2,0}$:

$$\begin{aligned} A_{\text{in}}(c_{2,0}) &\Rightarrow B_1([\#|\tilde{f}(c)]c_{2,0}) \Rightarrow A_1(c_{2,0}) \\ &\Rightarrow B_2([\#|\tilde{g}(c)]c_{2,0}) \Rightarrow A(c_{2,0}) \\ &\Rightarrow B([\langle \sigma, g\rangle(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1)), (\sigma(\kappa, \nu))|c]) \\ &\Rightarrow q_{\sigma,g}([(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1)), (\sigma(\kappa, \nu))|c]) \Rightarrow \mathbf{stop}(c_{2,1}) \end{aligned}$$

with $c_{2,1} = [\sigma|\tilde{g}(c)][(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1)), (\sigma(\kappa, \nu))|c]$. In fact, $h(c_{2,1}) = c_{1,1}$.

(2) Apply $\phi' = \text{expand}(y_2)$ to $c_{1,1}$. The result is $c_{1,2} = \langle \gamma, \tilde{f}(c)\rangle(\langle \sigma, c\rangle(\langle \kappa, c\rangle, \langle \nu, c\rangle))$.

Simulation of ϕ' by $\omega_{\phi'}$ on $c_{2,1}$:

$$\begin{aligned} A_{\text{in}}(c_{2,1}) &\Rightarrow A(c_{2,1}) \\ &\Rightarrow B([y_2|\tilde{g}(c)][(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1)), (\sigma(\kappa, \nu))|c]) \\ &\Rightarrow p_2([(\langle \kappa, f\rangle, \langle \gamma, f\rangle(y_1)), (\sigma(\kappa, \nu))|c]) \\ &\Rightarrow B([\langle \gamma, f\rangle(y_1), (\sigma(\kappa, \nu))|c] \\ &\Rightarrow q_{\gamma,f}([(y_1), (\sigma(\kappa, \nu))|c]) \Rightarrow \mathbf{stop}(c_{2,2}) \end{aligned}$$

with $c_{2,2} = [\gamma|\tilde{f}(c)][(y_1), (\sigma(\kappa, \nu))|c]$. In fact, $h(c_{2,2}) = c_{1,2}$.

(3) Apply $\phi'' = \text{expand}(y_1)$ to $c_{1,2}$. The result is $c_{1,3} = \langle \sigma, c\rangle(\langle \kappa, c\rangle, \langle \beta, c\rangle)$.

Simulation of ϕ'' by $\omega_{\phi''}$ on $c_{2,2}$:

$$\begin{aligned} A_{\text{in}}(c_{2,2}) &\Rightarrow A(c_{2,2}) \Rightarrow B([y_1|\tilde{f}(c)][(y_1), (\sigma(\kappa, \nu))|c] \\ &\Rightarrow p_1([(y_1), (\sigma(\kappa, \nu))|c]) \Rightarrow B([y_1, (\sigma(\kappa, \nu))|c]) \\ &\Rightarrow p_1([(\sigma(\kappa, \nu))|c]) \Rightarrow B([\sigma(\kappa, \nu)|c]) \Rightarrow q_\sigma([(\kappa, \nu)|c]) \\ &\Rightarrow \mathbf{stop}(c_{2,3}) \quad \text{with } c_{2,3} = [\sigma, (\kappa, \nu)|c]. \end{aligned}$$

And, in fact, $h(c_{2,3}) = c_{1,3}$.

From the characterization of CFT(S) by RT(TP(S))-transducers, the simulation of TP(S) by P(S), and the justification theorem we obtain the desired simulation result.

5.8. Theorem

$$\mathrm{CFT}(S) \subseteq \mathrm{RT}(\mathrm{P}(S)), \qquad \mathrm{MAC}(S) \subseteq \mathrm{CF}(\mathrm{P}(S)),$$

and determinism and totality are preserved.

Proof. Immediate from Theorem 5.5, Lemma 5.6, and Theorem 4.18. □

As already indicated above, we cannot prove the converse result.

5.9. Theorem. *There is a storage type S and an* $\mathrm{RT}(\mathrm{P}(S))$*-transducer* $\mathfrak{M}$ *such that* $\tau(\mathfrak{M}) \notin \mathrm{CFT}(S)$.

Proof. Let $S = \mathrm{TR}$ and let $\mathfrak{M} = (N, e, \Delta, A_{\mathrm{in}}, R)$ be an $\mathrm{RT}(\mathrm{P}(S))$-transducer, where

(i) $N = \{A_{\mathrm{in}}, A\}$;

(ii) $e = \lambda t \in T_\Omega.(\gamma, g(t))$, where $\gamma \in \Gamma$ and g is the identity on trees over $\{\sigma^{(1)}, \alpha^{(0)}\}$;

(iii) $\Delta = \{f^{(1)}, a^{(0)}\}$;

(iv) R contains the three rules

$$A_{\mathrm{in}} \to A(\mathrm{push}(\gamma, \mathrm{sel}_1)), \qquad A \to f(A_{\mathrm{in}}(\mathrm{pop})), \qquad A \to a.$$

Obviously, for every $n \geq 0$, $(\sigma(\alpha), f^n a) \in \tau(\mathfrak{M})$, where $f^n a$ is a short form for the monadic tree over Δ with n f's.

We show that $\tau(\mathfrak{M})$ cannot be realized by a macro tree transducer. By [22, Theorem 3.24], the height of an output tree of a macro tree transducer is exponentially bounded in the height of the corresponding input tree. Thus, the number of output trees for one input tree is finite. But in $\tau(\mathfrak{M})$ there is no bound on the number of output trees for the input tree $\sigma(\alpha)$. Hence, $\tau(\mathfrak{M}) \notin \mathrm{MT}_{\mathrm{OI}}$. By Theorem 3.22 it follows that $\tau(\mathfrak{M}) \notin \mathrm{CFT}(\mathrm{TR})$. □

Since we want to establish characterization results, in the next subsection we will study a restricted pushdown operator and in Section 5.3 an extension of $\mathrm{CFT}(S)$-transducers.

5.2. Characterization of $\mathrm{CFT}(S)$

In this subsection we will provide a characterization of $\mathrm{CFT}(S)$ by taking the simulation result of the previous section into account. Since our philosophy is to prove as much as possible on the level of storage types, we are actually looking for a restricted pushdown operator, say P′, that is equivalent to $\mathrm{TP}(S)$. By Theorem 5.5 and by the justification theorem, this equivalence yields the characterization of $\mathrm{CFT}(S)$ by $\mathrm{RT}(\mathrm{P}'(S))$.

Since, in particular, we also want to prove then that $\mathrm{TP}(S) \leq \mathrm{P}'(S)$, let us look again at the simulation of $\mathrm{TP}(S)$ by $\mathrm{P}(S)$ in Lemma 5.6, and see whether we can find an appropriate restriction on P such that the simulation is still allright.

For this purpose, consider the configuration $c_1 = \langle \gamma, c \rangle (t_1, \ldots, t_k)$ of $\mathrm{TP}(S)$ and let $c_2 = (\gamma, c)\beta$ be the configuration of $\mathrm{P}(S)$ such that $h(c_2) = c_1$, where h is the representation function defined in the proof of Lemma 5.6. Now we will consecutively apply some expand instructions to c_1 and conclude a property which is associated with the topmost pushdown square of c_2. Let, e.g., the first applied instruction be expand(ζ), where $\zeta = \langle \delta, f_1 \rangle (\zeta_1, \zeta_2)$. In the simulation of this expand instruction (after checking the definedness of the instructions in ζ), first the topmost pushdown square is inscribed with ζ; we mark this pushdown square for the time being. Second, ζ is worked through in a top-down manner (note that ζ is a tree). In our situation, the root of ζ is considered and, as a result, the marked square is inscribed with (ζ_1, ζ_2) and a push (δ, f_1) is applied. During the simulation of the denoted expand instructions it may happen that the marked square appears again at the top of the pushdown (it then contains ζ_1 or ζ_2). But how often can this situation occur? Clearly, the number of such excursions from the marked square is bounded by the height of ζ, because whenever a new excursion is initiated, the height of the tree by which the marked square is inscribed is decreased by one. Thus, between the time the square is pushed on the pushdown and the time it is popped off again, it appears only a bounded number of times on top of the pushdown. Actually, this property, called bounded excursion, holds for every square of the pushdown.

This bounded excursion property was introduced by Van Leeuwen [40] in order to show that his preset pushdown automata, when restricted to be bounded excursion, precisely accept E0L languages. In our terminology, E0L = range(CF(count-down)) and the preset automata correspond to ranges of REG(P(count-down))-transducers, where 'count-down' is the storage type defined in Example 3.6 (cf. [16]).

Technically, the bounded excursion property is forced upon the pushdown operator by adding a third and a fourth track to every pushdown square. The third track contains a counter which counts the number of excursions from the square that have already been executed, and the fourth track contains just a constant number which delimits the excursion counter. Now, for the restricted pushdown only those configurations are allowed in which, for every square, the counter is smaller than or equal to the constant number which is the maximal number of excursions.

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

(i) $C' = (\Gamma \times C \times \mathrm{nat} \times \mathrm{nat})^+$, where nat is the set of nonnegative integers;

(ii) $P' = \{\mathrm{top} = \gamma \mid \gamma \in \Gamma\} \cup \{\mathrm{test}(p) \mid p \in P\}$;

(iii) $F' = \{\mathrm{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{pop}\} \cup \{\mathrm{stay}(\gamma) \mid \gamma \in \Gamma\} \cup \{\mathrm{stay}\}$;

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

$$m'(\mathrm{top} = \gamma)(c') = (\delta = \gamma), \qquad m'(\mathrm{test}(p))(c') = m(p)(c),$$

$$m'(\text{push}(\gamma,f))(c') = \begin{cases} (\gamma, m(f)(c), 0, k)(\delta, c, i+1, k)\beta \\ \quad \text{if } m(f) \text{ is defined on } c \text{ and if } i+1 \leqslant k, \\ \text{undefined} \quad \text{otherwise,} \end{cases}$$

$$m'(\text{pop})(c') = \begin{cases} \beta & \text{if } \beta \neq \lambda, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$m'(\text{stay}(\gamma))(c') = \begin{cases} (\gamma, c, i+1, k)\beta & \text{if } i+1 \leqslant k, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$$m'(\text{stay})(c') = \begin{cases} (\delta, c, i+1, k)\beta & \text{if } i+1 \leqslant k, \\ \text{undefined} & \text{otherwise;} \end{cases}$$

(v) $I' = I$;

(vi) $E' = \{\lambda u \in I.(\gamma_0, e(u), 0, k) \mid \gamma_0 \in \Gamma, e \in E, k \geqslant 0\}$.

Note that $\mathrm{P}_{\mathrm{bex}}(S)$ has no identity. Clearly, every $X(\mathrm{P}_{\mathrm{bex}}(S))$-transducer $\mathfrak{M}$ is an $X(\mathrm{P}(S))$-transducer with the bounded excursion property as explained above; the bound is determined by the encoding of $\mathfrak{M}$.

Actually, $\mathrm{P}_{\mathrm{bex}}(S)$ is the storage type we were looking for. We now prove that $\mathrm{TP}(S)$ and $\mathrm{P}_{\mathrm{bex}}(S)$ are equivalent storage types.

5.11. Lemma. $\mathrm{TP}(S) \leqslant \mathrm{P}_{\mathrm{bex}}(S)$.

Proof. Let U be a finite restriction of $\mathrm{TP}(S)$ and let $F_f' = \{\text{expand}(\zeta_1), \ldots, \text{expand}(\zeta_r)\}$ be the finite set of instructions of U. Let $e' = \lambda u \in I.\alpha(e(u))$ be the encoding of U and let m' be the meaning function of U. We want to prove that $U \leqslant_{\mathrm{d}} \mathrm{P}_{\mathrm{bex}}(S)$.

Informally, we knit the representation function following the same pattern as in Lemma 5.6 (with some appropriate restrictions on its domain). The simulating flowcharts are taken over literally. Then, the only work left to be done is to find a bound for the number of excursions and prove it to be correct. For this purpose let $\mathrm{mx} = (\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta + 5) \cdot \mathrm{height}(\alpha)$, where

$$\mathrm{max}\zeta = \max\{\mathrm{height}(\zeta_i) \mid \mathrm{expand}(\zeta_i) \in F_f'\},$$

$$\mathrm{maxF} = \max\{n+1 \mid \text{there are } n \text{ instructions in } \zeta_i \text{ for some } i \in [r]\}.$$

Actually, we will show that mx is the desired bound. The factors 3 and 5 are caused by the way in which the flowcharts compute. The factor maxF arises from the checking of the definedness of instructions of S. Intuitively, $\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta$ is the bound which is respected by every square of a simulating pushdown not occurring at its bottom. As discussed before Lemma 5.6, actually, the bottom square of a simulating pushdown encodes two squares. The 'upper' one also respects the bound $\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta$. From the 'lower' square, which contains the initial tree-pushdown α, at most $\mathrm{height}(\alpha)$ excursions can be started. However, every such excursion runs

through the 'upper' part from which $3 \cdot \max\zeta$ excursions can be started. This explains the product $3 \cdot \max\zeta \cdot \text{height}(\alpha)$ in mx.

Formally, let h be the representation function in the proof of Lemma 5.6. From h we obtain the representation function

$$h_{\text{exc}}: (\Gamma \times C \times \text{nat} \times \text{nat})^+ \to C',$$

where C' is the configuration set of TP(S), by restricting the domain of h.

$$\begin{aligned}
&\text{dom}(h_{\text{exc}}) \\
&= \{(\sigma, c, 0, \text{mx})(\tilde{\xi}_1, c_1, \nu_1, \text{mx}) \ldots \\
&\quad (\tilde{\xi}_\rho, c_\rho, \nu_\rho, \text{mx})((\tilde{\xi}_{\rho+1}, \tilde{\alpha}), c_{\rho+1}, \nu_{\rho+1}, \text{mx}) \mid \\
&\quad (\sigma, c)(\tilde{\xi}_1, c_1) \ldots (\tilde{\xi}_\rho, c_\rho)((\tilde{\xi}_{\rho+1}, \tilde{\alpha}), c_{\rho+1}) \in \text{dom}(h) \\
&\quad \text{and, for every } i \in [\rho], \\
&\quad 3 \cdot \text{height}(\tilde{\xi}_i) + \nu_i \leqq \text{mx} \text{ and } 3 \cdot \text{height}(\tilde{\xi}_{\rho+1}) + 5 \\
&\quad + (\text{maxF} + 3 \cdot \max\zeta + 5) \cdot \text{height}(\tilde{\alpha}) + \nu_{\rho+1} \leqq \text{mx}\} \\
&\cup \{((\sigma, (\alpha_1, \ldots, \alpha_\eta)), c, \nu, \text{mx}) \mid ((\sigma, (\alpha_1, \ldots, \alpha_\eta)), c) \in \text{dom}(h) \\
&\quad \text{and } (\text{maxF} + 3 \cdot \max\zeta + 5) \cdot \text{height}(\sigma(\alpha_1, \ldots, \alpha_\eta)) + \nu \leqq \text{mx}\},
\end{aligned}$$

where $\text{height}((\xi_{i,1}, \ldots, \xi_{i,\nu})) = \max\{\text{height}(\xi_{i,j}) \mid j \in [\nu]\}$.

Let $c' \in \text{dom}(h_{\text{exc}})$. Then $h_{\text{exc}}(c') = h(c'')$, where c'' is obtained from c' by deleting the third and the fourth component of each pushdown square.

Now we have to show that the requirements (1)-(3) of Definition 4.6 hold.

Requirement (1) *of Definition* 4.6: Let $\alpha = \gamma(t_1, \ldots, t_k)$. For the encoding e' of U, we define the corresponding encoding

$$e'' = \lambda u \in I.((\gamma, (t_1, \ldots, t_k)), e(u), 0, \text{mx}).$$

Obviously, requirement (1) holds.

Requirement (2) *and* (3) *of Definition* 4.6: For every predicate and instruction occurring in U, we define the simulating flowchart in exactly the same way as in the proof of Lemma 5.6. Requirement (2) is still valid, because for every flowchart ω which simulates a predicate, $\text{oper}(\omega)$ is the total identity. (Note that a flowchart for predicates may contain the identity in its rules.) For a flowchart ω_ϕ which simulates an instruction ϕ, we have to check whether ω_ϕ is defined on a pushdown configuration if and only if ϕ is defined on the corresponding tree-pushdown configuration and, moreover, whether $\text{oper}(\omega_\phi)$ preserves $\text{dom}(h_{\text{exc}})$. Obviously, requirement (3.2) still holds. (3.1.1) also holds in one direction: if ω_ϕ is defined on c, then ϕ is defined on $h_{\text{exc}}(\phi)$. Via a case analysis, we will now prove that (the other direction of) (3.1.1) and (3.1.2) are correct.

Let $\phi = \text{expand}(\zeta)$ and let $c_2 \in \text{dom}(h_{\text{exc}})$.

Case 1. $\zeta = y_j$ and $c_2 = ((\sigma, (\alpha_1, \ldots, \alpha_\eta)), c, \nu, \text{mx})$: Since $\text{height}(\sigma(\alpha_1, \ldots, \alpha_\eta)) \geqq 1$ and

$$(\text{maxF} + 3 \cdot \max\zeta + 5) \cdot \text{height}(\sigma(\alpha_1, \ldots, \alpha_\eta)) + \nu \leqq \text{mx},$$

it follows that $\nu+6 \leqslant \mathrm{mx}$. Assume that $m'(\phi)(h_{\mathrm{exc}}(c_2))$ is defined. Since $c_2 \in \mathrm{dom}(h_{\mathrm{exc}})$, $j \leqslant \eta$. Let $\alpha_j = \delta(\alpha_{j,1}, \ldots, \alpha_{j,\mu})$ for some $\alpha_{j,1}, \ldots, \alpha_{j,\mu} \in T_\Omega$, where $\mu \geqslant 0$ and $\delta \in \Omega$. ω_ϕ can compute as follows:

$$\begin{aligned} A_{\mathrm{in}}(c_2) &\Rightarrow A(((\sigma, (\alpha_1, \ldots, \alpha_\eta)), c, \nu+1, \mathrm{mx})) \\ &\Rightarrow B(((y_j, (\alpha_1, \ldots, \alpha_\eta)), c, \nu+2, \mathrm{mx})) \\ &\Rightarrow p_j(((\alpha_1, \ldots, \alpha_\eta), c, \nu+3, \mathrm{mx})) \Rightarrow B((\alpha_j, c, \nu+4, \mathrm{mx})) \\ &\Rightarrow q_\delta(((\alpha_{j,1}, \ldots, \alpha_{j,\mu}), c, \nu+5, \mathrm{mx})) \\ &\Rightarrow \mathbf{stop}((\delta, (\alpha_{j,1}, \ldots, \alpha_{j,\mu}), c, \nu+6, \mathrm{mx})). \end{aligned}$$

Hence, $\mathrm{oper}(\omega_\phi)(c_2)$ is defined. This proves (3.1.1).

Since $\mathrm{height}(\alpha_j) \leqslant \mathrm{height}(\sigma(\alpha_1, \ldots, \alpha_\eta)) - 1$, it is clear that

$$(\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta + 5) \cdot \mathrm{height}(\delta(\alpha_{j,1}, \ldots, \alpha_{j,\mu})) + \nu + 6 \leqslant \mathrm{mx}.$$

This proves (3.1.2).

Case 2. $\zeta = \langle \delta, g \rangle(\zeta'_1, \ldots, \zeta'_k)$ and $c_2 = (\sigma, c, 0, \mathrm{mx})(\tilde{\xi}_1, c_1, \nu_1, \mathrm{mx}) \ldots (\tilde{\xi}_\rho, c_\rho, \nu_\rho, \mathrm{mx})((\tilde{\xi}_{\rho+1}, \tilde{\alpha}), c_{\rho+1}, \nu_{\rho+1}, \mathrm{mx})$: Let $\{f_1, \ldots, f_\kappa\}$ with $\kappa \geqslant 1$ be the set of instruction symbols occurring in ζ. Since $\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta + 5 \leqslant \mathrm{mx}$ (note that $\mathrm{height}(\alpha) \geqslant 1$) and $\mathrm{height}((\zeta'_1, \ldots, \zeta'_k)) \leqslant \mathrm{max}\zeta - 1$, we get

$$\mathrm{maxF} + 3 \cdot \mathrm{height}((\zeta'_1, \ldots, \zeta'_k)) + 3 + 5 \leqslant \mathrm{mx}.$$

Hence,

$$3 \cdot \mathrm{height}((\zeta'_1, \ldots, \zeta'_k)) + \kappa + 3 \leqslant \mathrm{mx}.$$

Let $\tilde{\xi}_1$ be a μ-tuple for some $\mu \geqslant 0$. Assume that $m'(\phi)$ is defined on $h_{\mathrm{exc}}(c_2)$. Then, for every $i \in [\kappa]$, $m(f_i)$ is defined on c (thus, in particular, $m(g)$ is defined on c) and $\mathrm{par}(\zeta) \subseteq Y_\mu$. Hence, ω_ϕ can compute as follows:

$$\begin{aligned} A_{\mathrm{in}}(c_2) &\Rightarrow B_1((\#, m(f_1)(c), 0, \mathrm{mx})(\sigma, c, 1, \mathrm{mx}) \ldots) \\ &\Rightarrow A_1((\sigma, c, 1, \mathrm{mx}) \ldots) \\ &\quad \vdots \\ &\Rightarrow B_\kappa((\#, m(f_\kappa)(c), 0, \mathrm{mx})(\sigma, c, \kappa, \mathrm{mx}) \ldots) \\ &\Rightarrow A((\sigma, c, \kappa, \mathrm{mx}) \ldots) \\ &\Rightarrow B((\zeta, c, \kappa+1, \mathrm{mx}) \ldots) \\ &\Rightarrow q_{\delta,f}(((\zeta'_1, \ldots, \zeta'_k), c, \kappa+2, \mathrm{mx}) \ldots) \\ &\Rightarrow \mathbf{stop}((\delta, m(g)(c), 0, \mathrm{mx})((\zeta'_1, \ldots, \zeta'_k), c, \kappa+3, \mathrm{mx}) \ldots). \end{aligned}$$

Hence, $\mathrm{oper}(\omega_\phi)(c_2)$ is defined which proves (3.1.1). Since $\mathrm{par}(\zeta) \subseteq Y_\mu$, the function subst is defined on $((\zeta'_1, \ldots, \zeta'_k), c, \kappa+3, \mathrm{mx}) \ldots$. Together with the calculation above this proves (3.1.2).

Case 3. $\zeta = y_j$ and c_2 as in Case 2: Then, according to ω_ϕ, c_2 is popped and the jth element of the list is considered, which occurs in the topmost pushdown square of the current configuration. This process is repeated until the list of the current topmost square contains a tree which is not in Y in the desired component. Let c_2' be the configuration resulting from the above explained process (the nonterminal of ω_ϕ is B). This configuration may be of three different forms.

(a) $c_2' = (\langle \delta, f \rangle(\zeta_1', \ldots, \zeta_k'), c_i, \nu_i + 1, \mathrm{mx})(\tilde{\xi}_{i+1}, c_{i+1}, \nu_{i+1}, \mathrm{mx}) \ldots$ or

(b) $c_2' = ((\langle \delta, f \rangle(\zeta_1', \ldots, \zeta_k'), \tilde{\alpha}), c_{\rho+1}, \nu_{\rho+1} + 1, \mathrm{mx})$.

These situations are similar to the one in Case 2 and, actually, the proofs of (3.1.1) and (3.1.2) are quite similar.

(c) $c_2' = (\alpha_j, c_{\rho+1}, \nu_{\rho+1} + 3, \mathrm{mx})$.

This is an easy case, left to the reader.

Case 4. Let ζ and $\{f_1, \ldots, f_\kappa\}$ be as in Case 2 and c_2 as in Case 1. Since

$$(\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta + 5) \cdot \mathrm{height}(\sigma(\alpha_1, \ldots, \alpha_\eta)) + \nu \leq \mathrm{mx},$$

certainly, $\nu + \kappa + 3 \leq \mathrm{mx}$. Assume that $m'(\phi)$ is defined on $h_{\mathrm{exc}}(c_2)$. By an argumentation similar to that in Case 1, we can conclude that the computation of ω_ϕ ends up with the $\mathrm{P}_{\mathrm{bex}}(S)$-configuration

$$(\delta, m(g)(c), 0, \mathrm{mx})(((\zeta_1', \ldots, \zeta_k'), (\alpha_1, \ldots, \alpha_\eta)), c, \nu + \kappa + 3, \mathrm{mx}).$$

This proves (3.1.1). By an easy calculation it can be proved that

$$3 \cdot \mathrm{height}((\zeta_1', \ldots, \zeta_k')) + 5$$
$$+ (\mathrm{maxF} + 3 \cdot \mathrm{max}\zeta + 5) \cdot \mathrm{height}((\alpha_1, \ldots, \alpha_\eta)) + \nu + \kappa + 3 \leq \mathrm{mx}.$$

This proves (3.1.2). □

In the next lemma we will show the reverse result, namely, that $\mathrm{P}_{\mathrm{bex}}(S)$ can be simulated by $\mathrm{TP}(S)$. The main idea in this simulation is to realize the counter of a pushdown square by a monadic piece of tree-pushdown, and whenever the counter is increased, the topmost symbol of the tree-pushdown is consumed, i.e., the length of the tree-pushdown decreases. Clearly, at the beginning, the length of the piece of tree-pushdown should be equal to the bound on the number of excursions. However, since $\mathrm{P}_{\mathrm{bex}}$ has stay instructions, a monadic tree-pushdown is not sufficient for the simulation. But, in fact, for every γ which occurs in a considered finite restriction of $\mathrm{P}_{\mathrm{bex}}$, we prepare a copy of the monadic tree-pushdown. Then a stay(γ_i) instruction is simulated by taking the ith subtree of the actual tree-pushdown (note that a stay instruction is also viewed as an excursion and hence increases the excursion counter). This trick is very similar to the construction in [37], which we already applied to prove Lemma 5.4.

5.12. Lemma. $P_{bex}(S) \leq TP(S)$.

Proof. Let U be a finite restriction of $P_{bex}(S)$. Let $e_1' = \lambda u \in I.(\gamma_1, e(u), 0, mx)$ be the encoding of U. (Note that this fixes the maximal excursion number mx.) Let $\Gamma_f = \{\gamma_1, \ldots, \gamma_r\}$ be the finite set of symbols of Γ that occur in a predicate or instruction of U. Since Ω is infinite for every rank, we can assume that $\Gamma_f \subseteq \Omega_r$, $\Gamma_f' = \{\gamma_i' \mid \gamma_i \in \Gamma_f\} \subseteq \Omega_1$, and $\Gamma_f'' = \{\gamma_i'' \mid \gamma_i \in \Gamma_f\} \subseteq \Omega_0$, where γ_i' and γ_i'' are new symbols. Let C' be the set of configurations of $TP(S)$. We prove that $U \leq_d TP(S)$.

The input sets of $P_{bex}(S)$ and of $TP(S)$ are both equal to that of S.

Before we provide the representation function, we define three auxiliary mappings 'tree', 'tree'', and 'c-tree', where tree and tree' are used to define c-tree, which, in its turn, is used to define the representation function. Furthermore, tree' is also used for the construction of the arguments of expand instructions which (in appropriate flowcharts) simulate the instructions of U.

(a) Define $\text{tree}: \Lambda \to T_{\Omega \times C}$ with $\Lambda = \Gamma_f \times C \times \{j \mid 0 \leq j \leq mx\}$ inductively on the third argument as follows:

(i) for every $\gamma \in \Gamma_f$ and $c \in C$, $\text{tree}(\gamma, c, mx) = \langle \gamma'', c \rangle$;

(ii) for every $\gamma \in \Gamma_f$, $c \in C$, and j with $0 \leq j \leq mx - 1$,

$$\text{tree}(\gamma, c, j) = \langle \gamma, c \rangle(\text{tree}(\gamma_1, c, j+1), \ldots, \text{tree}(\gamma_r, c, j+1)).$$

Intuitively, $\text{tree}(\gamma, c, j)$ is the full r-ary tree of height $mx - j + 1$ where the root is labeled by $\langle \gamma, c \rangle$ and the sons of each node are labeled by $\langle \gamma_1, c \rangle, \ldots, \langle \gamma_r, c \rangle$; at the bottom the γ's are double primed (cf. Fig. 8).

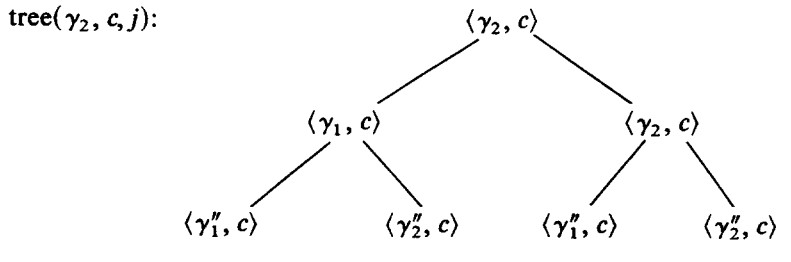


Fig. 8. $mx = 3$, $j = 1$, and $\Gamma_f = \{\gamma_1, \gamma_2\}$.

(b) Let Ψ be an arbitrary set of objects (which later will be either equal to C or equal to F). Define

$$\text{tree}': \Gamma_f \times \Psi \times \{j \mid 0 \leq j \leq mx\} \times [r] \to T_{\Omega \times \Psi}(Y)$$

inductively on the third argument as follows:

(i) for every $\gamma \in \Gamma_f$, $\psi \in \Psi$, and $\nu \in [r]$, $\text{tree}'(\gamma, \psi, mx, \nu) = \langle \gamma', \psi \rangle(y_\nu)$;

(ii) for every $\gamma \in \Gamma_f$, $\psi \in \Psi$, $\nu \in [r]$, and j with $0 \leq j \leq mx - 1$,

$$\text{tree}'(\gamma, \psi, j, \nu) = \langle \gamma, \psi \rangle(\text{tree}'(\gamma_1, \psi, j+1, \nu), \ldots, \text{tree}'(\gamma_r, \psi, j+1, \nu)).$$

Obviously, for $\Psi = C$, $\text{tree}'(\gamma, c, j, \nu)$ looks similar to $\text{tree}(\gamma, c, j)$, but at the leaves it contains objects of the form $\langle \gamma_i', \psi \rangle(y_\nu)$ rather than $\langle \gamma_i'', c \rangle$.

(c) Define $c\text{-tree}: \Lambda^+ \to C'$, with Λ as in (a), inductively on the length of the argument as follows:

(i) for every $\phi = (\gamma, c, j) \in \Lambda$, $c\text{-tree}(\phi) = \text{tree}(\gamma, c, j)$;

(ii) for every $\phi = (\gamma, c, j) \in \Lambda$ and $\beta \in \Lambda^+$,

$$c\text{-tree}(\phi\beta) = \text{tree}'(\gamma, c, j, 1)[y_1 \leftarrow c\text{-tree}(\beta)],$$

where $\Psi = C$.

Note that c-tree is injective.

The representation function $h: C' \to (\Gamma \times C)^+$ is defined by

- $\text{dom}(h) = \text{range}(c\text{-tree})$,
- let $c \in \text{dom}(h)$ such that $c\text{-tree}(\psi_1 \ldots \psi_n) = c$; then,

$$h(c) = (\gamma_{\rho(1)}, c_1, j_1, \text{mx}) \ldots (\gamma_{\rho(n)}, c_n, j_n, \text{mx}),$$

where $(\gamma_{\rho(i)}, c_i, j_i) = \psi_i$ for every $i \in [n]$.

Since c-tree is injective, every U-configuration has exactly one representation. Fig. 9 shows the representation of $(\gamma_1, c, 2, 3)(\gamma_2, c', 1, 3)$.

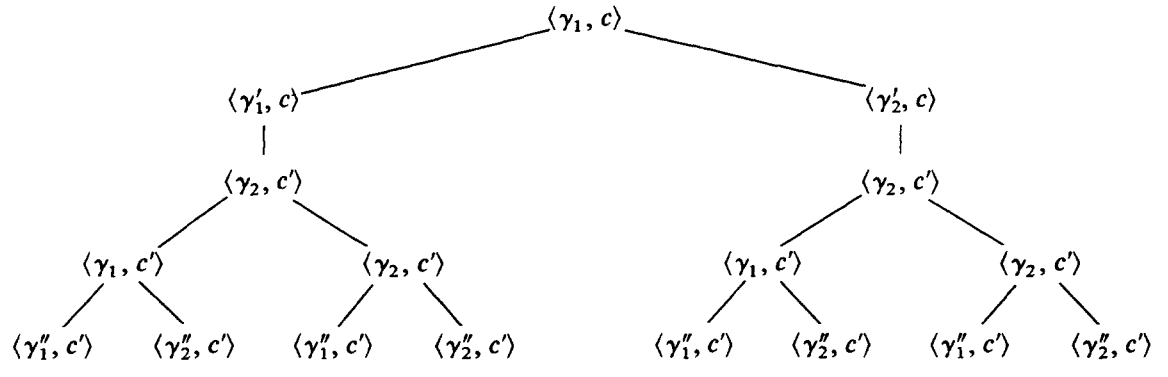


Fig. 9.

Requirement (1) *of Definition* 4.6: Define $e_2' = \lambda u \in I.c\text{-tree}((\gamma_1, e(u), 0))$. Since c-tree is total, $\text{dom}(e_2') = \text{dom}(e) = \text{dom}(e_1')$. If $u \in \text{dom}(e)$, then $e_2'(u) \in \text{dom}(h)$ and

$$e_1'(u) = (\gamma_1, e(u), 0, \text{mx}) = h(c\text{-tree}((\gamma_1, e(u), 0))) = h(e_2'(u)).$$

Requirement (2) *of Definition* 4.6: Let ϕ be a predicate of U. The TP(S)-flowchart ω_ϕ for predicates simulating ϕ is completely determined by its set of rules R:

(i) $\phi = (\text{top} = \gamma_i)$:

$$A_{\text{in}} \to \textbf{if } \text{call} = \gamma_i \textbf{ or } \text{call} = \gamma_i' \textbf{ or } \text{call} = \gamma_i'' \textbf{ then true}(\text{id}) \textbf{ else false}(\text{id})$$

is in R;

(ii) $\phi = \text{test}(p)$:

$$A_{\text{in}} \to \textbf{if } \text{test}(p) \textbf{ then true}(\text{id}) \textbf{ else false}(\text{id})$$

is in R.

Requirement (3) *of Definition* 4.6: Let ϕ be an instruction occurring in U. The TP(S)-flowchart ω_ϕ for instructions is again determined by its set of rules:

(i) $\phi = \text{pop}$: That piece of the representing tree-pushdown configuration has to be 'erased', which corresponds to the popped pushdown square. (This piece is not needed any more to simulate further excursions.) Formally, for every $\gamma \in \Gamma_f$ and every $\gamma' \in \Gamma'_f$,

$$A_{in} \to \textbf{if call} = \gamma \textbf{ then } A_{in}(\text{expand}(y_1)),$$

$$A_{in} \to \textbf{if call} = \gamma' \textbf{ then stop}(\text{expand}(y_1))$$

are in R.

(ii) $\phi = \text{push}(\gamma, f)$: Intuitively, if ϕ is applied to c' and c' is simulated by the tree-pushdown c'', then c'' has to be expanded by a piece of tree ζ, which allows to simulate mx excursions from the new pushdown square. However, if the new square is popped, then the pushdown symbol, say γ_i, should occur again on top. This situation is prepared in the simulation by substituting the ith subtree of c' at the leaves of ζ; note that the root of the ith subtree of c'' contains γ_i. Formally, for every $\gamma_i \in \Gamma_f$,

$$A_{in} \to \textbf{if call} = \gamma_i \textbf{ then stop}(\text{expand}(\text{tree}'(\gamma, f, 0, i)))$$

is in R, where $\Psi = F$.

(iii) $\phi = \text{stay}(\gamma_j)$: For every $\gamma \in \Gamma_f$,

$$A_{in} \to \textbf{if call} = \gamma \textbf{ then stop}(\text{expand}(y_j))$$

is in R.

(iv) $\phi = \text{stay}$: For every $\gamma_i \in \Gamma_f$,

$$A_{in} \to \textbf{if call} = \gamma_i \textbf{ then stop}(\text{expand}(y_i))$$

is in R. (Note that in the last three cases, there is no rule for call $= \gamma'$: the excursion number is too high.)

We prove requirement (3) only for the application of $\phi = \text{push}(\gamma_k, f)$ to a U-configuration c_1 such that $h(c_2) = c_1$ with $c_2 = c\text{-tree}((\gamma_i, c, j)\beta)$ for some i, c, j, and $\beta \neq \lambda$. Hence, $c_2 = \text{tree}'(\gamma_i, c, j, 1)[c\text{-tree}(\beta)]$ and $c_1 = (\gamma_i, c, j, \text{mx})\beta'$ for some β' such that $h(c\text{-tree}(\beta)) = \beta'$. Then, $m'(\phi)(h(c_2))$ is defined iff $m(f)(c)$ is defined and $j+1 \leqslant \text{mx}$ iff $m(f)(c)$ is defined and ω_ϕ can compute as follows.

$$\begin{aligned} A_{in}(c_2) \;=\; & A_{in}(\langle \gamma_i, c\rangle(\text{tree}(\gamma_1, c, j+1, 1)[c\text{-tree}(\beta)], \ldots, \\ & \qquad \text{tree}(\gamma_r, c, j+1, 1)[c\text{-tree}(\beta)])) \\ \Rightarrow\; & \text{stop}(\text{tree}(\gamma_k, m(f)(c), 0, i)[y_i \leftarrow \text{tree}(\gamma_i, c, j+1, 1)[c\text{-tree}(\beta)]]). \end{aligned}$$

This proves that (3.1.1) holds.

Since

$$\begin{aligned} & c\text{-tree}((\gamma_k, m(f)(c), 0)(\gamma_i, c, j+1)\beta) \\ & \quad = \text{tree}'(\gamma_k, m(f)(c), 0, 1)[c\text{-tree}((\gamma_i, c, j+1)\beta)] \\ & \quad = \text{tree}'(\gamma_k, m(f)(c), 0, i)[\text{tree}'(\gamma_i, c, j+1, 1)[c\text{-tree}(\beta)]], \end{aligned}$$

also (3.1.2) holds.

Finally,

$$h(\text{oper}(\omega_\phi))(c_2) = h(c\text{-tree}((\gamma_k, m(f)(c), 0)(\gamma_i, c, j+1)\beta))$$
$$= (\gamma_k, m(f)(c), 0, \text{mx})(\gamma_i, c, j+1, \text{mx})\beta' = m'(\phi)(h(c_2)).$$

This proves (3.2). □

The previous two lemmata prove the equivalence of $\text{TP}(S)$ and $\text{P}_{\text{bex}}(S)$.

5.13. Theorem. $\text{TP}(S) \equiv \text{P}_{\text{bex}}(S)$.

As an immediate consequence we obtain the desired characterization of $\text{CFT}(S)$ by bounded excursion pushdown machines.

5.14. Theorem

$$\text{CFT}(S) = \text{RT}(\text{P}_{\text{bex}}(S)), \qquad \text{MAC}(S) = \text{CF}(\text{P}_{\text{bex}}(S)),$$

and the determinism and totality are preserved.

Proof. This immediately follows from Theorem 5.5, Theorem 5.13, and the justification theorem. □

However, in the total deterministic case we can drop the restriction of the pushdown to be bounded excursion. Actually, in the next lemma we will prove that, for a $\text{D}_t\text{RT}(\text{P}(S))$-transducer, the restriction on the number of excursions does not decrease the power of the transducer.

5.15. Lemma. *For* $X \in \{\text{RT}, \text{CF}, \text{REG}\}$, $\text{D}_t X(\text{P}(S)) = \text{D}_t X(\text{P}_{\text{bex}}(S))$.

Proof. Since $\text{P}_{\text{bex}}(S) \leqslant \text{TP}(S)$ (by Lemma 5.12) and $\text{TP}(S) \leqslant \text{P}(S)$ (by Lemma 5.6), it follows from the transitivity of $\leqslant$ (cf. Theorem 4.20) and the justification theorem that $\text{D}_t X(\text{P}_{\text{bex}}(S)) \subseteq \text{D}_t X(\text{P}(S))$. Hence, we only have to prove that $\text{D}_t X(\text{P}(S)) \subseteq \text{D}_t X(\text{P}_{\text{bex}}(S))$. By Lemma 3.31, it is sufficient to consider $\text{D}_t X(\text{P}_1(S))$-transducers, because $\text{D}_t X(\text{P}_1(S)) = \text{D}_t X(\text{P}(S))$.

For technical convenience we add, in the scope of this proof, an excursion counter as third component to every pushdown square of $\text{P}_1(S)$-configurations which is handled just as in $\text{P}_{\text{bex}}(S)$. But, in contrast to the latter storage type, the counter is not checked against a maximal number. Denote this storage type by $\text{P}_{\text{ex}}(S)$. Of course, $\text{P}(S) \equiv \text{P}_{\text{ex}}(S)$ (and similarly for P_1).

We only give a proof for $X = \text{REG}$. The other cases are similar. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be a $\text{D}_t\text{REG}(\text{P}_{\text{ex}}(S))$-transducer with k nonterminals. Let $e = \lambda u \in I.(\gamma, g(u), 0)$ for some $\gamma \in \Gamma$ and some encoding g of S. (0 is the value of the counter.)

Construct the DREG($P_{bex}(S)$)-transducer $\mathfrak{M}' = (N, e', \Delta, A_{in}, R)$ by $e' = \lambda u \in I.(\gamma, g(u), 0, k)$. We claim that $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$. Since $\tau(\mathfrak{M})$ is a total function (cf. Corollary 3.16) and $\tau(\mathfrak{M}')$ is a partial function (cf. Theorem 3.15), and $\mathrm{dom}(e) = \mathrm{dom}(e')$, it suffices to prove that $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}')$.

For this purpose, consider a successful derivation of $\mathfrak{M}$, e.g., $A_{in}(\gamma, g(u), 0) \Rightarrow^*_{\mathfrak{M}} w$ with $u \in \mathrm{dom}(g)$ and $w \in \Delta^*$. Then we claim that the excursion counter in every pushdown square occurring in the derivation has a value which is smaller than k. Suppose that the derivation runs through the sentential form $\xi = vA(c')$, where $v \in \Delta^*$, $A \in N$, and $c' = (\delta, c, k)\beta$. This means that c' occurs at least $k+1$ times in the derivation. Since there are only k nonterminals in $\mathfrak{M}$, at least one nonterminal, say B, is repeated with the configuration c'. Hence, $\mathfrak{M}$ loops, because it is deterministic and it has to repeat the part of the derivation between the two occurrences of $B(c')$. This contradicts the successfulness of the derivation.

Since the excursion counter can only grow larger than k by a push(γ, f)-instruction applied to some configuration $(\gamma', c, k)\beta$, the above argumentation implies that it is always smaller than k. Hence, $\mathfrak{M}'$ can execute the same derivations as $\mathfrak{M}$ and $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$.

If $X = \mathrm{RT}$ or $X = \mathrm{CF}$, then, by considering paths of derivation trees, we can use the above argumentation and prove the respectation of bounds also for these cases. □

We have now obtained one of the main characterizations of CFT(S) by pushdown machines (viz., indexed S-transducers) for the important total deterministic case.

5.16. Theorem. $D_t\mathrm{CFT}(S) = D_t\mathrm{RT}(\mathrm{P}(S))$ *and* $D_t\mathrm{MAC}(S) = D_t\mathrm{CF}(\mathrm{P}(S))$.

Proof. Immediate from Theorem 5.14 and Lemma 5.15. □

The last theorem generalizes the result in [25] that all noncircular attribute grammars can be simulated by total deterministic macro tree transducers (cf. also [7, 15]): every noncircular attribute grammar (AG) can be easily simulated by a total deterministic indexed tree transducer (assuming that for every context-free grammar G there is an encoding in TR which is the identity on the set of derivation trees of G); the attributes of the AG are the nonterminals of the transducer and the semantic rules of the AG its rules (cf. [16]). Thus, using the previous result, $\mathrm{AG} \subseteq D_t\mathrm{RT}(\mathrm{P}(\mathrm{TR})) = D_t\mathrm{CFT}(\mathrm{TR})$.

5.3. Characterization of RT(P(S))

In this section we will treat the second solution to achieve a characterization result: we enrich the structure of CFT(S)-transducers such that they reach the power of RT(P(S))-transducers.

Following again our philosophy, we want to prove the desired characterization result on the level of storage types. This means that we have to look for a modification of $\mathrm{TP}(S)$, say $\mathrm{TP}'(S)$, which is equivalent to $\mathrm{P}(S)$, and then we have to find an appropriate extension of $\mathrm{CFT}(S)$, say $\mathrm{CFT}'(S)$, such that the characterization of Theorem 5.5 also holds in the form $\mathrm{CFT}'(S) = \mathrm{RT}(\mathrm{TP}'(S))$.

How does the enrichment of the tree-pushdown of S look like? We can approach the solution of the problem by comparing the meaning of a push instruction with the meaning of an expand instruction. Whereas a push does not modify the topmost square of the actual pushdown, but just puts another square on top, an 'expand' always consumes (pops) the topmost item of the present configuration. Thus, if, in an application of an instruction like expand(ζ) with $\zeta \notin Y$, we allow the tree-pushdown to preserve the configuration in the topmost item, then we can simulate the application of push instructions. This is achieved by allowing the identity on the configurations in the argument of an expand instruction, but only at leaves. Hereby, we obtain the extended tree-pushdown.

5.17. Definition. The *extended tree-pushdown of* S, denoted by $\mathrm{TP}_{\mathrm{ext}}(S)$, is the storage type (C', P', F', m', I', E'), where C', P', m' (restricted to P'), I', and E' are defined as in Definition 5.1, and

$$F' = \{\mathrm{expand}(\zeta) \mid \zeta \in T_{\Omega \times F}(Y \cup \mathrm{EXT})\},$$

where

$$\mathrm{EXT} = \{\langle \gamma, \mathrm{id}\rangle(y_1, \ldots, y_k) \mid \gamma \in \Omega \text{ of rank } k \text{ with } k \geqslant 0\}$$

and id is, as usual, the identity on C, and, for every $c' = \langle \delta, c\rangle(t_1, \ldots, t_k) \in C'$ with $k \geqslant 0$,

$$m'(\mathrm{expand}(\zeta))(c') = \begin{cases} c'' = \zeta[f \leftarrow m(f)(c); f \in F][\mathrm{id} \leftarrow c][y_j \leftarrow t_j; j \in [k]] \\ \quad \text{if } c'' \in C' \text{ and, for every } \langle \gamma, \mathrm{id}\rangle \text{ occurring in } \zeta, \ \gamma \text{ has rank } k, \\ \text{undefined} \quad \text{otherwise.} \end{cases}$$

Note that if S contains an identity, then $\mathrm{TP}_{\mathrm{ext}}(S) \equiv \mathrm{TP}(S)$.

In the next two lemmata the equivalence of $\mathrm{P}(S)$ and $\mathrm{TP}_{\mathrm{ext}}(S)$ is proved. In the simulation of $\mathrm{P}(S)$ by $\mathrm{TP}_{\mathrm{ext}}(S)$, a pushdown is just viewed as a monadic tree-pushdown.

5.18. Lemma. $\mathrm{P}(S) \leqslant \mathrm{TP}_{\mathrm{ext}}(S)$.

Proof. By Theorem 4.21, $\mathrm{P}(S) \equiv \mathrm{P}_1(S)$. Recall that $\mathrm{P}_1(S)$ is the pushdown of S without instructions of the form stay(γ, f).

Let U be a finite restriction of $\mathrm{P}_1(S)$ and let $e_1' = \lambda u \in I.(\gamma_0, e(u))$ be the encoding of U. Let

$$\Gamma_f = \{\gamma \in \Gamma \mid \gamma \text{ occurs in a predicate or an instruction of } U\} \cup \{\gamma_0\}.$$

Since Ω is infinite in every rank, we can assume that $\Gamma_f \subseteq \Omega_1$ and that $\Gamma'_f \subseteq \Omega_0$, where $\Gamma'_f = \{\gamma' \mid \gamma \in \Gamma_f\}$ and every γ' is a new symbol.

We prove that $U \leqslant_d \mathrm{TP}_{ext}(S)$. The input sets of U and of $\mathrm{TP}_{ext}(S)$ are both equal to I.

We define the representation function $h: C' \to (\Gamma \times C)^+$ as follows, where C' is the set of configurations of $\mathrm{TP}_{ext}(S)$:

(i) for every $\langle \gamma', c\rangle \in \Gamma'_f \times C$, $h(\langle \gamma', c\rangle) = (\gamma, c)$;

(ii) for every $\langle \gamma, c\rangle \in \Gamma_f \times C$ and every monadic tree-pushdown t over $(\Gamma_f \times C) \cup (\Gamma'_f \times C)$ with $t \in \mathrm{dom}(h)$, $h(\langle \gamma, c\rangle(t)) = (\gamma, c)h(t)$.

Requirement (1) *of Definition* 4.6: The encoding $e'_2 = \lambda u \in I.\langle \gamma'_0, e(u)\rangle$ satisfies the requirements.

Requirement (2) *of Definition* 4.6: For every predicate ϕ of U we define the $\mathrm{TP}_{ext}(S)$-flowchart ω_ϕ for predicates which is determined entirely by its rules.

(i) $\phi = (\mathrm{top} = \gamma)$:

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma \textbf{ or } \mathrm{call} = \gamma' \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id});$$

(ii) $\phi = (\mathrm{test}(p))$:

$$A_{in} \to \textbf{if } \mathrm{test}(p) \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id}).$$

Requirement (3) *of Definition* 4.6: For every instruction ϕ of U we define the $\mathrm{TP}_{ext}(S)$-flowchart ω_ϕ for instructions which is again totally determined by its rules. For every $\gamma \in \Gamma_f$ and every $\gamma' \in \Gamma'_f$,

(i) $\phi = \mathrm{push}(\delta, f)$:

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma \textbf{ then stop}(\mathrm{expand}(\langle \delta, f\rangle(\langle \gamma, \mathrm{id}\rangle(y_1)))),$$

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma' \textbf{ then stop}(\mathrm{expand}(\langle \delta, f\rangle(\langle \gamma', \mathrm{id}\rangle)));$$

(ii) $\phi = \mathrm{pop}$: $A_{in} \to \textbf{stop}(\mathrm{expand}(y_1))$;

(iii) $\phi = \mathrm{stay}(\delta)$:

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma \textbf{ then stop}(\mathrm{expand}(\langle \delta, \mathrm{id}\rangle(y_1))),$$

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma' \textbf{ then stop}(\mathrm{expand}(\langle \delta', \mathrm{id}\rangle));$$

(iv) $\phi = \mathrm{id}$:

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma \textbf{ then stop}(\mathrm{expand}(\langle \gamma, \mathrm{id}\rangle(y_1))),$$

$$A_{in} \to \textbf{if } \mathrm{call} = \gamma' \textbf{ then stop}(\mathrm{expand}(\langle \gamma', \mathrm{id}\rangle))$$

are rules of ω_ϕ.

It should be clear that these flowcharts fulfil the requirements. □

For the simulation of $\mathrm{TP}_{ext}(S)$, the main work is already done in Lemma 5.6. By an easy modification of the involved flowcharts which simulate expand instructions, the desired simulation is accomplished.

5.19. Lemma. $\mathrm{TP}_{ext}(S) \leqslant \mathrm{P}(S)$.

Proof. We can take over the whole proof of Lemma 5.6, except that we restrict, in part (v) in the proof of requirement (3), the instruction f to range over instructions of F only. Furthermore, we add in part (v), for every $\langle\sigma, \mathrm{id}\rangle(y_1, \ldots, y_k) \in \mathrm{SUB}(\{\zeta_1, \ldots, \zeta_k\})$, the rules

$$B \to \textbf{if } \mathrm{top} = \langle\sigma, \mathrm{id}\rangle(y_1, \ldots, y_k) \textbf{ then } \mathrm{stop}(\mathrm{stay}(\sigma)),$$

$$B \to \textbf{if } \mathrm{top} = (\langle\sigma, \mathrm{id}\rangle(y_1, \ldots, y_k), (\alpha_1, \ldots, \alpha_\eta))$$

$$\textbf{then } \mathrm{stop}(\mathrm{stay}((\sigma, (\alpha_1, \ldots, \alpha_\eta))))$$

to the constructed flowchart.

Intuitively, it is clear that this modification exactly captures the additional features of the extended tree-pushdown. Again, we leave out a formal proof. □

The previous two lemmata prove the equivalence of $\mathrm{P}(S)$ and $\mathrm{TP}_{\mathrm{ext}}(S)$.

5.20. Theorem. $\mathrm{P}(S) \equiv \mathrm{TP}_{\mathrm{ext}}(S)$.

Assuming that the identity is in a storage type S, all the storage type operators discussed in this section yield equivalent storage types when they are applied to S.

5.21. Corollary. *If S contains an identity, then* $\mathrm{TP}_{\mathrm{ext}}(S) \equiv \mathrm{TP}(S) \equiv \mathrm{P}_{\mathrm{bex}}(S) \equiv \mathrm{P}(S)$.

Proof. The statement holds, because $\mathrm{TP}_{\mathrm{ext}}(S) \equiv \mathrm{P}(S)$ (Theorem 5.20), $\mathrm{TP}(S) \equiv \mathrm{P}_{\mathrm{bex}}(S)$ (Theorem 5.13), and $\mathrm{TP}_{\mathrm{ext}}(S) \equiv \mathrm{TP}(S)$ (because S contains an identity). □

In particular, since the trivial storage type has an identity, $\mathrm{TP} \equiv \mathrm{P}$. In [29] it is proved that every pushdown tree automaton can be equivalently transformed into a restricted pushdown tree automaton: instead of a tree-pushdown, only a pushdown is allowed. Actually, since pushdown tree automata recognize range(RT(TP)) and the restricted version corresponds to RT(P)-transducers (in the same sense), the result of [29] is reobtained by applying the justification theorem to $\mathrm{P} \equiv \mathrm{TP}$.

Now we are ready to define the appropriate extension of CFT(S)-transducers.

5.22. Definition. An *extended* CFT(S)*-transducer* (*extended* MAC(S)*-transducer*) is a CFT(S_{id})-transducer (MAC(S_{id})-transducer, respectively) $\mathfrak{M}$, where the rules are restricted as follows: if $A(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ then } \zeta$ is a rule of $\mathfrak{M}$, then the instruction id may only occur in subtrees (subterms, respectively) ζ' of ζ, where ζ' has the form $B(\mathrm{id})(y_1, \ldots, y_k)$.

Note that id is a new instruction symbol in S_{id}, i.e., it does not occur in S (cf. Definition 3.7). Hence, if S already contains an identity, say f, then, of course, f can still occur everywhere in a rule. Clearly, in this case, every extended CFT(S)-transducer can be simulated by an ordinary CFT(S)-transducer (by replacing id by f).

The class of translations induced by extended CFT(S)-transducers is denoted by $\mathrm{CFT}_{\mathrm{ext}}(S)$. For extended MAC($S$)-transducers we use the corresponding notations.

Fortunately, the characterization of Theorem 5.5 also holds for $CFT_{ext}(S)$ and $RT(TP_{ext}(S))$.

5.23. Theorem

$$CFT_{ext}(S) = RT(TP_{ext}(S)), \qquad MAC_{ext}(S) = CF(TP_{ext}(S)),$$

and determinism and totality are preserved.

Proof. In fact, the proofs of Lemma 5.3 and Lemma 5.4 can be taken over literally, by considering a $CFT_{ext}(S)$-transducer and an $RT(TP_{ext}(S))$-transducer just as a $CFT(S_{id})$-transducer and an $RT(TP(S_{id}))$-transducer respectively. It is an easy observation that the requirements put on the instruction id in $RT(TP_{ext}(S))$-transducers and $CFT_{ext}(S)$-transducers respectively, are preserved by the constructions. □

Now we provide the characterization of indexed S-transducers by means of extended $CFT(S)$-transducers.

5.24. Theorem

$$CFT_{ext}(S) = RT(P(S)), \qquad MAC_{ext}(S) = CF(P(S)),$$

and determinism and totality are preserved.

Proof. The statements of this theorem follow from Theorem 5.23, Theorem 5.20, and the justification theorem. □

Since the trivial storage type S_0 contains an identity, $CFT_{ext}(S_0) = CFT(S_0)$. Hence, by considering the ranges, the first equation of the previous theorem turns into a statement of [29]: (OI) context-free tree languages are accepted by restricted pushdown tree automata. In the same way we reobtain from the second equation the equivalence of (OI) macro grammars and indexed grammars (cf. [24]). Confer also point (v) after Definition 3.28.

For the total deterministic case, all the considered modified translation classes fall together. This also holds in general, when S contains an identity (cf. Corollary 5.21).

5.25. Corollary

$$D_tCFT_{ext}(S) = D_tCFT(S) = D_tRT(P(S)) = D_tRT(P_{bex}(S)),$$

$$D_tMAC_{ext}(S) = D_tMAC(S) = D_tCF(P(S)) = D_tCF(P_{bex}(S)).$$

Proof. This immediately follows from Theorem 5.24, Theorem 5.16, and Lemma 5.15. □

6. Characterization of MAC(S) by pushdown^2 S-to-string transducers

In this section we will provide, for MAC(S) and $\mathrm{MAC}_{\mathrm{ext}}(S)$, an iterative, regular characterization by means of $\mathrm{REG}(\mathrm{P}^2_{\mathrm{bex}}(S))$-transducers and $\mathrm{REG}(\mathrm{P}^2(S))$-transducers, respectively. Since Section 5 already offers a first step in the good direction, namely that $\mathrm{MAC}(S) = \mathrm{CF}(\mathrm{P}_{\mathrm{bex}}(S))$ (cf. Theorem 5.14) and $\mathrm{MAC}_{\mathrm{ext}}(S) = \mathrm{CF}(\mathrm{P}(S))$ (cf. Theorem 5.24), the work which is left amounts to proving a characterization of CF(S) by $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$-transducers. Then, by considering storage types S' of the form $\mathrm{P}_{\mathrm{bex}}(S)$ and P(S), respectively, the desired results are obtained (note that $\mathrm{P}_{\mathrm{bex}}(\mathrm{P}(S)) \equiv \mathrm{P}^2(S)$, because P($S$) contains an identity; cf. Corollary 5.21).

Besides the nondeterministic case which will be treated in Section 6.1, we will present a characterization for the total deterministic case in Section 6.2. Unfortunately, the characterization in the latter case does not hold in general but only under a condition. (In Section 6.2 we will try to get a strong condition.) This observation together with the fact that the justification theorem preserves determinism implies that we cannot prove the characterization results along the lines of Section 5 by proving the equivalence of appropriate storage types. This is the reason why we provide direct constructions.

6.1. *Nondeterministic* MAC(S)*-transducers*

Here, we will characterize CF(S) by means of $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$-transducers. This generalizes the result of [40] on E0L and preset pushdown automata (for which S = count-down), cf. the discussion preceding Definition 5.10. We only consider left-to-right derivations of CF(S)-transducers in the usual sense. Obviously, this does not restrict the implied class of translations.

The desired characterization result is also closely related to the equivalence of regular extended top-down tree-to-string transducers (for short: yRT) and of checking-tree pushdown transducers (for short: ct-pd), which is proved in Theorem 4.5 of [19]. As we have already discussed at the end of Section 3.3, the ct-pd transducer is equivalent to the REG(P(TR))-transducer. Moreover, the yRT transducers correspond to the $\mathrm{CF}_{\mathrm{ext}}(\mathrm{TR})$-transducers, where the extended CF(TR)-transducer is obtained by allowing the identity at the rightmost nonterminal of a rule whenever it is not followed by a terminal. Hence, in the terminology of $X(S)$-transducers, the result in [19] would read like $\mathrm{CF}_{\mathrm{ext}}(\mathrm{TR}) = \mathrm{REG}(\mathrm{P}(\mathrm{TR}))$. Actually, in [16] this result is generalized to $\mathrm{CF}_{\mathrm{ext}}(S) = \mathrm{REG}(\mathrm{P}(S))$ for an arbitrary storage type S. Here, we are going to provide the counterpart of this result in the same sense as Section 5.2 is the counterpart of Section 5.3, i.e., we prove that $\mathrm{CF}(S) = \mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$.

In spite of some technical details, the constructions used to prove this equality are closely related to the ones that are given in the proof of $\mathrm{CF}_{\mathrm{ext}}(S) = \mathrm{REG}(\mathrm{P}(S))$ in [16] and in the proof of the corresponding result in [19] for the storage type TR. And, actually, we would like to animate the interested reader to compare them with each other.

Now we start with the implementation of a CF(S)-transducer on a REG($P_{bex}(S)$)-transducer. In order to simulate the application of a rule of a CF(S)-transducer, we just write the right-hand side of the rule on the pushdown (in one square) and work it through from left to right. This corresponds to a left-to-right derivation of the CF(S)-transducer.

6.1. Lemma

$$\mathrm{CF}(S) \subseteq \mathrm{REG}(P_{bex}(S)),$$

and determinism and totality are preserved.

Proof. Let $\mathfrak{M} = (N, e, \Sigma, A_{in}, R)$ be a CF(S)-transducer. Let SUF be the set of suffixes of right-hand sides of rules of R with respect to the alphabet $N(F) \cup \Sigma$. Since SUF is a finite set, we can assume that $\mathrm{SUF} \cup \{[\zeta] \mid \zeta \in \mathrm{SUF}\} \subseteq \Gamma$ (the brackets "[" and "]" are used to mark the bottom square of pushdown configurations). Let F_f be the finite set of instructions occurring in R. We first construct the REG(P(S))-transducer $\mathfrak{M}' = (N', e', \Sigma, *, R')$ as follows: $N' = \{*\} \cup \{q_{A,f} \mid A \in N, f \in F_f\}$; $e' = \lambda u \in I.([A_{in}], e(u))$; and

(i) if $A \to$ **if** b **then** ζ is in R, then

$$* \to \textbf{if top} = A \textbf{ and } \mathrm{test}(b) \textbf{ then } *(\mathrm{stay}(\zeta)),$$

$$* \to \textbf{if top} = [A] \textbf{ and } \mathrm{test}(b) \textbf{ then } *(\mathrm{stay}([\zeta]))$$

are in R';

(ii) for every $a\zeta \in \mathrm{SUF}$ with $a \in \Sigma$, the rules

$$* \to \textbf{if top} = a\zeta \textbf{ then } a*(\mathrm{stay}(\zeta)),$$

$$* \to \textbf{if top} = [a\zeta] \textbf{ then } a*(\mathrm{stay}([\zeta]))$$

are in R';

(iii) for every $A(f)\zeta \in \mathrm{SUF}$ with $A(f) \in N(F_f)$, the rules

$$* \to \textbf{if top} = A(f)\zeta \textbf{ then } q_{A,f}(\mathrm{stay}(\zeta)),$$

$$* \to \textbf{if top} = [A(f)\zeta] \textbf{ then } q_{A,f}(\mathrm{stay}([\zeta]))$$

are in R';

(iv) for every $q_{A,f} \in N'$, the rule $q_{A,f} \to *(\mathrm{push}(A, f))$ is in R';

(v) $* \to$ **if top** $= \lambda$ **then** $*(\mathrm{pop})$ and $* \to$ **if top** $= [\lambda]$ **then** λ are in R'.

Obviously, determinism is preserved by this construction. Moreover, it is clear that $\mathfrak{M}'$ can easily be turned into a REG($P_{bex}(S)$)-transducer, where the bound on the number of excursions is $2 \cdot \mathrm{mx} + 1$, where mx is the maximal length of a right-hand side of a rule of $\mathfrak{M}$. (Such a REG($P_{bex}(S)$)-transducer has the same rules as $\mathfrak{M}'$ and it has $e'' = \lambda u \in I.([A_{in}], e(u), 0, 2, \mathrm{mx}+1)$ as encoding.)

We can prove the equivalence of $\mathfrak{M}$ and $\mathfrak{M}'$ by proving the following two claims.

Claim 10. *For every* $A \in N$, $c \in C$, *and* $w \in \Sigma^*$,

$$A(c) \Rightarrow^*_{\mathfrak{M}} w \text{ iff } *(([A], c)) \Rightarrow^*_{\mathfrak{M}'} w$$

and for every $\beta \in (\Gamma \times C)^+$: $*((A, c)\beta) \Rightarrow^*_{\mathfrak{M}'} w*(\beta)$ *and* β *is not tested in this derivation.*

Claim 11. *For every* $\zeta \in (\Sigma \cup N(F_f))^*$, $c \in C$, *and* $w \in \Sigma^*$,

$$\zeta[f \leftarrow m(f)(c); f \in F] \Rightarrow^*_{\mathfrak{M}} w \text{ iff } *(([\zeta], c)) \Rightarrow^*_{\mathfrak{M}'} w$$

and for every $\beta \in (\Gamma \times C)^+$: $*((\zeta, c)\beta) \Rightarrow^*_{\mathfrak{M}'} w*(\beta)$ *and* β *is not tested in this derivation.*

The term "β is not tested in a derivation" means that the configuration β does not occur in the derivation except in the last sentential form.

The proof of the two claims is an easy simultaneous induction on the length of the considered derivations. Since $\mathrm{dom}(e) = \mathrm{dom}(e'')$ and $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$, totality is preserved. □

For the simulation of a $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$-transducer by a $\mathrm{CF}(S)$-transducer we use the ordinary triple construction (as it is used in [19, Theorem 4.5] and in the proof of $\mathrm{REG}(\mathrm{P}(S)) \subseteq \mathrm{CF}_{\mathrm{ext}}(S)$ in [16]). Note that, since the pushdown is bounded excursion, the right-hand side of any rule for the constructed $\mathrm{CF}(S)$-transducer only needs to contain finitely many nonterminals. Hence, we can live without any extension of the $\mathrm{CF}(S)$-transducer. Recall that the nonterminals of the constructed $\mathrm{CF}(S)$-transducer are triples like $\langle A, \gamma, B\rangle$ which encode the following information of the $\mathrm{REG}(\mathrm{P}(S))$-transducer $\mathfrak{M}$: if a pushdown $(\gamma, c)\beta$ is read with nonterminal A, then, after some excursions from the square (γ, c), $\mathfrak{M}$ will pop this square and continue its computation by reading β with 'return' nonterminal B.

Since the return nonterminals have to be guessed, the construction of the following lemma does not preserve determinism. However, under specific requirements on the storage type S, we can also prove the deterministic version. This situation will be treated in Section 6.2.

6.2. Lemma. $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S)) \subseteq \mathrm{CF}(S)$.

Proof. Let us start with a $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$-transducer $\mathfrak{M}$. We apply some modifications to $\mathfrak{M}$ such that the construction of the $\mathrm{CF}(S)$-transducer is easier.

(1) Every stay instruction in a rule is simulated by an appropriate stay(γ) instruction; the γ's can be determined by a test.

(2) By Lemma 3.31 we can construct a $\mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$-transducer without stay(γ) instructions. The constructed transducer again applies stay instructions but only to pushdown configurations of height 1, i.e., only to the bottom square of the pushdown. Furthermore, the bottom square is marked. Note that the construction of Lemma 3.31 preserves the bounded excursion property. However, if the original transducer is bounded by mx, then the constructed transducer without stay(γ) instructions has bound $\mathrm{mx} + \mathrm{mx}^2$. This is an easy observation: every stay(γ) instruction is replaced

by (pop; push(...)) (except at the bottom square); hence, at every pushdown square z, there may be at most mx pushes plus the number of times which any other square put on top of z could apply a stay(γ) instruction, i.e., at most mx^2 times. Thus, the total number of push instructions is at most $mx+mx^2$.

(3) Since the restriction of $P(S)$ being of bounded excursion does not cause any trouble in the construction of Lemma 3.30, we can assume that the tests of rules have the form top $=\gamma$ **and** test(b), where b is a standard test.

(4) We modify the transducer such that it empties its store in the following sense: if a rule like

$$A \to \textbf{if top} = \gamma \textbf{ and test}(b) \textbf{ then } w,$$

where w is a terminal string, is applied to a pushdown configuration c', then c' contains only one square. This modification can be achieved by introducing a new nonterminal which reduces the pushdown to one square whenever the original transducer tries to erase it.

Now, let $\mathfrak{M}_1 = (N, e, \Sigma, A_{in}, R)$ be the equivalent REG($P_{bex}(S)$)-transducer which is obtained from $\mathfrak{M}$ by modifications (1)-(4). Let $e = \lambda u \in I.([\gamma_0], g(u), 0, mx)$, where g is an encoding of S and $[\gamma_0]$ is the marked bottom symbol. Note that every bottom square of a pushdown which occurs in a derivation of $\mathfrak{M}_1$ contains $[\gamma_0]$, because $\mathfrak{M}_1$ does not contain stay(γ) instructions. Let mx be the bound on the number of excursions and let Γ_f be the finite set of pushdown symbols used in $\mathfrak{M}_1$.

Construct the CF(S)-transducer $\mathfrak{M}' = (N', e', \Sigma, \langle A_{in}, [\gamma_0]\rangle, R')$ as follows:

$$N' = \{\langle A, \gamma, B\rangle \mid A, B \in N, \gamma \in \Gamma_f\} \cup \{\langle A_{in}, [\gamma_0]\rangle\},$$

$e' = g$, and R' is determined by (i)-(iv). For the sake of simplicity we do not specify the types of the involved objects. Let π abbreviate top $=\gamma$ **and** test(b) for some γ and b.

(i) Let $\gamma \neq [\gamma_0]$. If, for some $k \in [mx]$,

$$A \to \textbf{if } \pi \textbf{ then } w_1 B_1(\text{push}(\delta_1, f_1)),$$

and, for every $i \in [k-1]$,

$$B_i' \to \textbf{if } \pi \textbf{ then } w_{i+1} B_{i+1}(\text{push}(\delta_i, f_i)),$$

and

$$B_k' \to \textbf{if } \pi \textbf{ then } w_{k+1} B(\text{pop})$$

are in R, then

$$\langle A, \gamma, B\rangle \to \textbf{if } b \textbf{ then } w_1\langle B_1, \delta_1, B_1'\rangle(f_1) \ldots w_k\langle B_k, \delta_k, B_k'\rangle(f_k) w_{k+1}$$

is in R'.

(ii) Let $\gamma \neq [\gamma_0]$. If $A \to \textbf{if } \pi \textbf{ then } wB(\text{pop})$ is in R, then $\langle A, \gamma, B\rangle \to \textbf{if } b \textbf{ then } w$ is in R'.

(iii) Let $\gamma = [\gamma_0]$. If, for some $k \geq 0$ and some $n_1, \ldots, n_k \geq 0$ such that $s(k+1) - 1 \leq \mathrm{mx}$, where $s(i) = n_1 + \cdots + n_i$, the rules

$$B'_j \to \textbf{if } \pi \textbf{ then } w_{j+1} B'_{j+1}(\mathrm{stay})$$

for every $i \in [k+1]$ and every j with $1 + s(i-1) \leq j \leq s(i) - 1$ with $A_{\mathrm{in}} = B'_1$, and

$$B'_{s(i)} \to \textbf{if } \pi \textbf{ then } w_{s(i)} B_{s(i)+1}(\mathrm{push}(\delta_i, f_i))$$

for every $i \in [k]$, and

$$B'_{s(k+1)} \to \textbf{if } \pi \textbf{ then } w$$

are in R, then

$$\langle A_{\mathrm{in}}, [\gamma_0] \rangle \to \textbf{if } b \textbf{ then}$$

$$w_1 \ldots w_{s(1)} \langle B_{s(1)+1}, \delta_1, B'_{s(1)+1} \rangle (f_1) \ldots w_{s(i-1)+1} \ldots$$

$$w_{s(i)} \langle B_{s(i)+1}, \delta_i, B'_{s(i)+1} \rangle (f_i) \ldots w_{s(k-1)+1} \ldots$$

$$w_{s(k)} \langle B_{s(k)+1}, \delta_k, B'_{s(k)+1} \rangle (f_k) w_{s(k)+1} \ldots w_{s(k+1)} w$$

is in R'.

(iv) Let $\gamma = [\gamma_0]$. If $A_{\mathrm{in}} \to \textbf{if } \pi \textbf{ then } w$ is in R, then $\langle A_{\mathrm{in}}, [\gamma_0] \rangle \to \textbf{if } b \textbf{ then } w$ is in R'.

The equivalence of $\mathfrak{M}_1$ and $\mathfrak{M}'$ immediately follows from the following two claims. Again, we leave out the specification of the types of the involved objects and hope that it does not cause any confusion.

Claim 12. $\langle A, \gamma, B \rangle (c) \Rightarrow^*_{\mathfrak{M}'} w$ *iff for every* β: $A((\gamma, c)\beta) \Rightarrow^*_{\mathfrak{M}_1} wB(\beta)$ *and* β *is not tested in this derivation.*

Claim 13. $\langle A_{\mathrm{in}}, [\gamma_0] \rangle (c) \Rightarrow^*_{\mathfrak{M}'} w$ *iff* $A_{\mathrm{in}}(([\gamma_0], c)) \Rightarrow^*_{\mathfrak{M}_1} w$.

Claims 12 and 13 can be proved by an easy induction on the length of the derivations. We leave out the formal proof. □

The previous two lemmata provide a characterization of $\mathrm{CF}(S)$.

6.3. Theorem. $\mathrm{CF}(S) = \mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(S))$.

By taking $S = \mathrm{TR}$, this provides a (new) characterization of the class of translations induced by top-down tree-to-string transducers by means of checking-tree pushdown transducers which are bounded excursion.

This characterization can be used to find pushdown machines for $\mathrm{MAC}(S)$ and $\mathrm{MAC}_{\mathrm{ext}}(S)$.

6.4. Theorem. $\mathrm{MAC}(S) = \mathrm{REG}(\mathrm{P}^2_{\mathrm{bex}}(S))$ *and* $\mathrm{MAC}_{\mathrm{ext}}(S) = \mathrm{REG}(\mathrm{P}^2(S))$.

Proof. $\mathrm{MAC}(S) = \mathrm{CF}(\mathrm{P}_{\mathrm{bex}}(S)) = \mathrm{REG}(\mathrm{P}^2_{\mathrm{bex}}(S))$ by Theorem 5.14 and Theorem 6.3. $\mathrm{MAC}_{\mathrm{ext}}(S) = \mathrm{CF}(\mathrm{P}(S)) = \mathrm{REG}(\mathrm{P}_{\mathrm{bex}}(\mathrm{P}(S))$ by Theorem 5.24 and Theorem 6.3, and, since, by Corollary 5.21, $\mathrm{P}_{\mathrm{bex}}(\mathrm{P}(S)) \equiv \mathrm{P}^2(S)$, the second equation of the present theorem follows from the justification theorem. □

For the trivial storage type which contains an identity, the second equation says that (OI) macro languages (equivalent to level-2 (OI) grammars, cf. [8]) are accepted by pushdown^2 automata. For an automata-theoretic characterization of the OI-hierarchy which is generated by high-level OI-grammars we refer the reader to [9]. In [36] it is similarly shown that the indexed languages are accepted by pushdown^2 automata (there called indexed pushdown automata).

6.2. *Total deterministic* MAC(S)*-transducers*

We want to prove the total deterministic version of Theorem 6.4. Since in Lemma 6.1 determinism and totality are preserved, we already know that $D_tMAC(S) \subseteq D_tREG(P^2_{bex}(S))$ and that $D_tMAC_{ext}(S) \subseteq D_tREG(P^2(S))$. Unfortunately, in Lemma 6.2 determinism is not preserved, because the CF(S)-transducer $\mathfrak{M}$ has to guess the return nonterminals. Now, if $\mathfrak{M}$ could, by some means, determine the return nonterminals, then the constructed transducer is also deterministic. In fact, in [16] the notion of storage type with look-ahead is introduced to obtain a solution for this problem. In the storage type S with look-ahead, denoted by S_{LA}, special tests are added to the set of predicates of S, which are called look-ahead tests. They have the form $\langle A, \mathfrak{H}\rangle$, where A is a nonterminal of a CF(S)-transducer $\mathfrak{H}$ (also called the look-ahead transducer). The look-ahead test $\langle A, \mathfrak{H}\rangle$ is true on a configuration c of C iff the transducer $\mathfrak{H}$ can derive a terminal string from the sentential form $A(c)$. Actually, this enrichment of S is appropriate to determine the return nonterminals in our original problem. The nondeterministic CF(S)-transducer $\mathfrak{M}'$, which is obtained from the $D_tREG(P_{bex}(S))$-transducer $\mathfrak{M}$ via the construction of Lemma 6.2 may serve as look-ahead transducer. Thus, if the look-ahead test $\langle\langle A, \gamma, B\rangle, \mathfrak{M}'\rangle$ is true on the configuration c of C, then $\mathfrak{M}$ derives, for every β, from $A((\gamma, c)\beta)$ the sentential form $wB(\beta)$ for some terminal string w (note that $\mathfrak{M}$ is deterministic). But this means that B is the correct return nonterminal. By adding these look-ahead tests to the rules of $\mathfrak{M}'$ a $D_tCF(S_{LA})$-transducer $\mathfrak{M}''$, equivalent to $\mathfrak{M}$, is obtained. We refer the reader for more details to [16]. Here, we only recall the formal definition of look-ahead storage types and the consequence for the simulation of $D_tREG(P(S))$-transducers. Note that LA is an operator on storage types (just as P and TP); to increase readability we write S_{LA} rather than LA(S).

6.5. Definition. The storage type S *with look-ahead*, denoted by S_{LA}, is the tuple (C, P', F, m', I, E), where m' restricted to $P \cup F$ is equal to m,

$$P' = P \cup \{\langle A, \mathfrak{H}\rangle \mid \mathfrak{H} \text{ is a CF}(S)\text{-transducer and } A \text{ is one of its nonterminals}\}$$

and for every $c \in C$, $m'(\langle A, \mathfrak{H}\rangle)(c) =$ **true** iff there is a $w \in \Sigma^*$ such that $A(c) \Rightarrow^*_{\mathfrak{H}} w$, where Σ is the terminal alphabet of $\mathfrak{H}$.

Obviously, $S \leqslant S_{LA}$. Note also that the definition of S_{LA} for the special storage type TR is equivalent with the definition of regular look-ahead in [12]. Hence, an

$\mathrm{RT}(\mathrm{TR}_{\mathrm{LA}})$-transducer is a top-down tree transducer with regular look-ahead (and a $\mathrm{CFT}(\mathrm{TR}_{\mathrm{LA}})$-transducer is a macro tree transducer with regular look-ahead, cf. [22]).

The operator LA is monotonic with respect to the simulation relation.

6.6. Lemma. *If* $S_1 \leqslant S_2$, *then* $(S_1)_{\mathrm{LA}} \leqslant (S_2)_{\mathrm{LA}}$.

Proof. Let U_{LA} be a finite restriction of $(S_1)_{\mathrm{LA}}$. Since for every look-ahead test $\langle A, \mathfrak{H}\rangle$ in U_{LA} the $\mathrm{CF}(S_1)$-transducer $\mathfrak{H}$ uses only finitely many predicates and instructions, U_{LA} determines a finite restriction U on S_1 such that all these $\mathfrak{H}$'s are $\mathrm{CF}(U)$-transducers. Since $S_1 \leqslant S_2$, $U \leqslant_{\mathrm{d}} S_2$; assume that h is the involved representation function. We claim that with this h also $U_{\mathrm{LA}} \leqslant_{\mathrm{d}} (S_2)_{\mathrm{LA}}$. It suffices to show that every look-ahead test of U_{LA} can be simulated by an $(S_2)_{\mathrm{LA}}$-flowchart for predicates (actually, by a look-ahead test of $(S_2)_{\mathrm{LA}}$). Let $\langle A, \mathfrak{H}_1\rangle$ be such a look-ahead test, where $\mathfrak{H}_1$ is a $\mathrm{CF}(U)$-transducer and A is a nonterminal of $\mathfrak{H}_1$. Then, by Lemma 4.15(1), there is a simple $\mathrm{CF}(U)$-transducer $\mathfrak{H}_1'$ such that $\mathfrak{H}_1 \leqslant^{(\mathrm{id})} \mathfrak{H}_1'$. By Lemma 4.16 there is a $\mathrm{CF}(S_2)$-transducer $\mathfrak{H}_2$ with chain rules such that $\mathfrak{H}_1' \leqslant^{(h)} \mathfrak{H}_2$. Finally, by Lemma 4.17, there is a $\mathrm{CF}(S_2)$-transducer $\mathfrak{H}_2'$ such that, for every nonterminal A of $\mathfrak{H}_2$, $c \in C$, and for every $w \in \Delta^*$: $A(c) \Rightarrow^*_{\mathfrak{H}_2} w$ iff $A(c) \Rightarrow^*_{\mathfrak{H}_2'} w$.

Now we can construct the $(S_2)_{\mathrm{LA}}$-flowchart ω for predicates by $A_{\mathrm{in}} \rightarrow$ **if** $\langle A, \mathfrak{H}_2'\rangle$ **then true**(id) **else false**(id). We prove that requirement (2) of Definition 4.6 is fulfilled for ω and $\langle A, \mathfrak{H}_1\rangle$.

Since $\mathrm{oper}(\omega)$ is the identity on C_2, (2.1.1) and (2.1.2) are trivially true. Let $c \in \mathrm{dom}(h)$. In the following we will denote the meaning function of $(S_1)_{\mathrm{LA}}$ and $(S_2)_{\mathrm{LA}}$ by $m_{1,\mathrm{LA}}$ and $m_{2,\mathrm{LA}}$, respectively. Then the following equivalences prove (2.2).

$m_{1,\mathrm{LA}}(\langle A, \mathfrak{H}_1\rangle)(h(c)) =$ **true**

iff there is a $w \in \Delta^*$: $A(h(c)) \Rightarrow^*_{\mathfrak{H}_1} w$ (by Definition 6.5)

iff there is a $w \in \Delta^*$: $A(h(c)) \Rightarrow^*_{\mathfrak{H}_1'} w$ (because $\mathfrak{H}_1 \leqslant^{(\mathrm{id})} \mathfrak{H}_1'$)

iff there is a $w \in \Delta^*$: $A(c) \Rightarrow^*_{\mathfrak{H}_2} w$ (because $\mathfrak{H}_1' \leqslant^{(h)} \mathfrak{H}_2$)

iff there is a $w \in \Delta^*$: $A(c) \Rightarrow^*_{\mathfrak{H}_2'} w$ (because of Lemma 4.17)

iff $m_{2,\mathrm{LA}}(\langle A, \mathfrak{H}_2'\rangle)(c) =$ **true**

iff $\mathrm{pred}(\omega)(c) =$ **true**. □

Now we cite the result from [16], which we discussed above.

6.7. Theorem. *If* $S_{\mathrm{LA}} \equiv S$, *then* $\mathrm{D_tCF}(S) = \mathrm{D_tREG}(\mathrm{P}(S))$.

The condition $S_{\mathrm{LA}} \equiv S$ means that S is 'closed under look-ahead', i.e., the storage type S is strong enough to handle its look-ahead test by S-flowcharts. Note that this condition is necessary in Theorem 6.7. If we dropped it, then, by taking $S = \mathrm{TR}$,

it would contradict $D_tREG(P(TR)) = D_tCF(TR_{LA})$ (cf. Theorem 4.7 of [19]) and the fact that total deterministic top-down tree transducers are not closed under look-ahead (cf. [12, Example 2.2]).

We are ready to present the deterministic counterpart of Theorem 6.4. Note that, by Corollary 5.25, the extension does not increase the translation class induced by $D_tMAC(S)$-transducers.

6.8. Theorem. *If* $P(S)_{LA} \equiv P(S)$, *then* $D_tMAC(S) = D_tREG(P^2(S)) = D_tREG(P^2_{bex}(S))$.

Proof. Under the assumption that $P(S)_{LA} \equiv P(S)$, $D_tMAC(S) = D_tREG(P^2(S))$ immediately follows from Theorem 5.16 and Theorem 6.7.

Since Lemma 6.1 preserves determinism, it follows from Theorem 5.14 that $D_tMAC(S) = D_tCF(P_{bex}(S)) \subseteq D_tREG(P^2_{bex}(S))$. Since $P_{bex}(S) \leq P(S)$ (this follows from Lemma 5.12, Lemma 5.6, and the transitivity of $\leq$), the monotonicity of P implies that $P(P_{bex}(S)) \leq P^2(S)$. Hence, by the justification theorem and Lemma 5.15, it follows that $D_tREG(P^2_{bex}(S)) \subseteq D_tREG(P^2(S))$. Hence, $D_tREG(P^2(S)) = D_tREG(P^2_{bex}(S))$. □

The rest of this section is devoted to improve the situation of Theorem 6.8, because, actually, the storage type $P(S)_{LA}$ is complicated and hard to imagine. We will prove that Theorem 6.8 holds under the intuitively weaker condition that $P(S_{LA}) \equiv P(S)$ (since it is easy to see that $P(S)_{LA} \equiv P(S)$ implies $P(S_{LA}) \equiv P(S)$). We do this by proving that, actually, the two conditions are equivalent. In fact, in Lemma 6.14 we will prove that $P(S)_{LA} \equiv P(S_{LA})$. This amounts to simulate a $P(S)_{LA}$ look-ahead test b by a $P(S_{LA})$-flowchart ω for predicates. Since b is determined by a $CF(P(S))$-transducer (and a designated nonterminal) and ω can use look-ahead tests on S-configurations, which are determined by $CF(S)$-transducers, a comparison of $dom(CF(P(S)))$ and $dom(CF(S))$ is involved in the proof of $P(S)_{LA} \leq P(S_{LA})$. Indeed, in Lemma 6.11 we will show the equality of these two classes of domains. However, for the proof of this result we need two more lemmata. The first one states that an $RT(S)$-transducer can check whether its output trees are in a regular tree language and the second one shows that the two classes $dom(CF(S_{id}))$ and $dom(CF(S))$ are the same.

6.9. Lemma. $RT(S) \circ FTA \subseteq RT(S)$.

Proof. The proof consists of an easy product construction. Let $\mathfrak{M} = (N, e, \Delta, A_{in}, R)$ be an $RT(S)$-transducer and let $\mathfrak{A} = (Q, \Delta, q_0, R_A)$ be a finite state tree automaton (cf. Section 2.4).

Construct the $RT(S)$-transducer $\mathfrak{M}' = (N', e, \Delta, \langle A_{in}, q_0\rangle, R')$ as follows: $N' = N \times Q$ and if $A \to$ **if** b **then** ζ is a rule in R and $\zeta = \bar{t}[A_1(f_1), \ldots, A_n(f_n)]$ for some

$\tilde{t} \in T_\Delta(Z_n)$ with $n \geq 0$ and every z_i occurs exactly once in $\tilde{t}$ and for some $A_1(f_1), \ldots, A_n(f_n) \in N(F)$, and if

$$q(\tilde{t}) \Rightarrow^*_{\mathfrak{A}} \tilde{t}[q_1(z_1), \ldots, q_n(z_n)]$$

for some $q_1, \ldots, q_n \in Q$, then

$$\langle A, q\rangle \to \textbf{if } b \textbf{ then } \tilde{t}[\langle A_1, q_1\rangle(f_1), \ldots, \langle A_n, q_n\rangle(f_n)]$$

is in R'.

It is easy to see that, for every $n \geq 0$ and $t \in T_\Delta$, $A(c) \Rightarrow^n_{\mathfrak{M}} t$ and $q(t) \Rightarrow^*_{\mathfrak{M}'} t$ iff $\langle A, q\rangle(c) \Rightarrow^n_{\mathfrak{M}'} t$, and so $\tau(\mathfrak{M}') = \tau(\mathfrak{M}) \circ \tau(\mathfrak{A})$. □

The second lemma uses an easy consequence of the pumping lemma for context-free languages. It says that for every context-free grammar G there is a natural number p such that if there is a word w in $L(G)$, then there is also a word w' in $L(G)$ such that w' is not longer than p and, furthermore, every symbol which occurs in w' also occurs in w (p is also called the pumping index of G). By considering the way in which we have defined the derivation of CF(S)-transducers, it is clear that this pumping lemma is useful to study CF(S)-transducers.

6.10. Lemma. $\mathrm{dom}(\mathrm{CF}(S_{\mathrm{id}})) = \mathrm{dom}(\mathrm{CF}(S))$.

Proof. Let $\mathfrak{M} = (N, e, \Sigma, A_{\mathrm{in}}, R)$ be a CF(S_{id})-transducer in standard test form.

First, we mark in the rules of $\mathfrak{M}$ every nonterminal which is applied to the identity. For this purpose we construct the CF(S_{id})-transducer $\mathfrak{M}' = (N', e, \Sigma, A_{\mathrm{in}}, R')$ by $N' = N \cup [N]$, where $[N] = \{[A] \mid A \in N\}$ and, if $A \to \textbf{if } b \textbf{ then } \zeta$ is in R, then $A \to \textbf{if } b \textbf{ then } [\zeta]$ and $[A] \to \textbf{if } b \textbf{ then } [\zeta]$ are in R', where $[\zeta]$ is obtained from ζ by replacing every $B(\mathrm{id})$ by $[B](\mathrm{id})$. Obviously, $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$.

For every standard test b we now extract from $\mathfrak{M}'$ a context-free grammar $G(b)$ which works only with the marked nonterminals of $\mathfrak{M}'$ and contains only the rules with test b. Define $G(b) = ([N], \Psi, \_, R(b))$ by

$$\Psi = \{B(f) \mid B(f) \in N(F) \text{ and } B(f) \text{ occurs in a rule of } R\} \cup \Sigma,$$

and if $[A] \to \textbf{if } b \textbf{ then } [\zeta]$ is in R', then $[A] \to [\tilde{\zeta}]$ is in $R(b)$, where $[\tilde{\zeta}]$ is obtained from $[\zeta]$ by dropping every id instruction and the parentheses around it.

Then it is easy to prove (by induction on the length of the derivation) that, for every $[B] \in [N]$, $c \in C$, $v \in \Sigma^*$, and standard test b such that $m(b)(c) = \textbf{true}$, if $[B](c) \Rightarrow^*_{\mathfrak{M}'} v$, then there is a $\zeta \in \Psi^*$ such that $[B] \Rightarrow^+_{G(b)} \zeta$ and $\zeta[f \leftarrow m(f)(c); f \in F] \Rightarrow^*_{\mathfrak{M}'} v$.

We denote this statement by (*). Let $p(b)$ the pumping index of $G(b)$. Define the set

$$T_{B,b} = \{\zeta \in \Psi^+ \mid [B] \Rightarrow^*_{G(b)} \zeta \text{ and } \zeta \text{ is not longer than } p(b)\}.$$

Note that $T_{B,b}$ is a finite set.

By means of these context-free grammars we can construct the desired CF(S)-transducer $\mathfrak{M}'' = (N, e, \Sigma, A_{\text{in}}, R'')$. If $A \to \textbf{if } b \textbf{ then } [\zeta]$ is in R', then, for every $[\zeta]'$, the rule $A \to \textbf{if } b \textbf{ then } [\zeta]'$ is in R'', where $[\zeta]'$ is obtained from $[\zeta]$ by replacing every $[B](\text{id})$ by an element of $T_{B,b}$, if $T_{B,b} \neq \emptyset$.

Now we prove that $\text{dom}(\tau(\mathfrak{M}')) = \text{dom}(\tau(\mathfrak{M}''))$. Since every rule of $\mathfrak{M}''$ is just a contracted representation of a derivation of $\mathfrak{M}'$, it is obvious that $\text{dom}(\tau(\mathfrak{M}'')) \subseteq \text{dom}(\tau(\mathfrak{M}'))$. For the proof of the other direction, we consider an element $u \in \text{dom}(\tau(\mathfrak{M}'))$. Then there is a $w \in \Sigma^*$ such that $A_{\text{in}}(e(u)) \Rightarrow^*_{\mathfrak{M}'} w$. By using $(*)$ we can reorder this derivation in such a way that after an application of a rule to a nonterminal $A \in N$, every nonterminal of the form $[B]$ is derived first, until we obtain a sentential form in $(N(C) \cup \Sigma)^*$, i.e., there are $\xi_1, \ldots, \xi_n \in (N(C) \cup \Sigma)^*$ and there are $\xi_1', \ldots, \xi_n' \in (N'(C) \cup \Sigma)^*$ such that $\mathfrak{M}'$ can compute

$$\begin{aligned} A_{\text{in}}(e(u)) = \xi_1 &\Rightarrow \xi_1' \Rightarrow^* \xi_2, \\ \xi_2 &\Rightarrow \xi_2' \Rightarrow^* \xi_3, \\ &\ldots \\ \xi_{n-1} &\Rightarrow \xi_{n-1}' \Rightarrow^* \xi_n = w, \end{aligned}$$

and in every subderivation from ξ_i' to ξ_{i+1} no rule is applied to a nonterminal in N.

From the rules which are applied to the ξ_i's and from the subderivations starting with them, we find rules of $\mathfrak{M}''$ which contribute to a successful derivation of $\mathfrak{M}''$ starting with $e(u)$. Let r: $A_{\text{in}} \to \textbf{if } b \textbf{ then } [\zeta]$ be the rule of $\mathfrak{M}'$ which is applied to ξ_1. Let $[B]$ occur in $[\zeta]$. Then there is a $\xi' \in \Psi^*$ such that $[B] \Rightarrow^*_{G(b)} \xi'$ and $\xi'[f \leftarrow m(f)(e(u))]$ is a substring of ξ_2. If ξ' is not longer than $p(b)$, then replace $[B](\text{id})$ in the right-hand side of r by ξ'. If ξ' is longer than $p(b)$, then, by the above mentioned special form of the pumping lemma, there is a ξ'' not longer than $p(b)$ and $[B] \Rightarrow^*_{G(b)} \xi''$, i.e., $\xi'' \in T_{B,b}$. Then replace $[B](\text{id})$ by ξ''. Moreover, every nonterminal which occurs in ξ'' occurs also in ξ'. This replacement process is done for every nonterminal of $[N]$ in $[\zeta]$ and the result is a rule $A_{\text{in}} \to \textbf{if } b \textbf{ then } [\zeta]'$ in R'' such that every nonterminal of N which occurs in $[\zeta]'$ occurs also in ξ_2.

Since from every nonterminal in ξ_2 there is a successful derivation which starts with one of the rules applied to the ξ_i's in the denoted derivation, the same holds for every nonterminal in $[\zeta]'[f \leftarrow m(f)(e(u)); f \in F]$. Then we can transform the rules which correspond to the nonterminals in this sentential form in the same way as the initial rule. It is obvious that, by putting the constructed rules together, a successful derivation of $\mathfrak{M}''$ starting from $e(u)$ is obtained. Hence, $\text{dom}(\tau(\mathfrak{M}'')) \subseteq \text{dom}(\tau(\mathfrak{M}'))$. □

Now we can prove the mentioned equality of dom(CF(P(S)) and dom(CF(S)). In the following lemma also a slightly extended result is proved which we will use in Section 8.

6.11. Lemma. $\text{dom}(\text{CFT}_{\text{ext}}(S) \circ \text{FTA}) = \text{dom}(\text{CF}(S))$ *and* $\text{dom}(\text{CF}(\text{P}(S)))$ $\text{dom}(\text{CF}(S))$.

Proof. Obviously, for both equations we only have to prove one direction.

$$\begin{aligned}
&\mathrm{dom}(\mathrm{CFT}_{\mathrm{ext}}(S)\circ\mathrm{FTA})\\
&\quad\subseteq \mathrm{dom}(\mathrm{CFT}(S_{\mathrm{id}})\circ\mathrm{FTA})\\
&\quad\subseteq \mathrm{dom}(\mathrm{RT}(S_{\mathrm{id}})\circ\mathrm{CFT}(\mathrm{TR})\circ\mathrm{FTA}) && \text{(by Corollary 3.27)}\\
&\quad\subseteq \mathrm{RT}(S_{\mathrm{id}})^{-1}(\mathrm{CFT}(\mathrm{TR})^{-1}(\mathrm{dom}(\mathrm{FTA}))) && \text{(by definition of domain)}\\
&\quad\subseteq \mathrm{RT}(S_{\mathrm{id}})^{-1}(\mathrm{MT}_{\mathrm{OI}}^{-1}(\mathrm{dom}(\mathrm{FTA}))) && \text{(by Theorem 3.22(b))}\\
&\quad\subseteq \mathrm{RT}(S_{\mathrm{id}})^{-1}(\mathrm{dom}(\mathrm{FTA}))
\end{aligned}$$

because RECOG is closed under the inverse of macro tree transducers (cf. [22, Theorem 7.4] and $\mathrm{dom}(\mathrm{FTA}) = \mathrm{RECOG}$;

$$\begin{aligned}
&\subseteq \mathrm{dom}(\mathrm{RT}(S_{\mathrm{id}})\circ\mathrm{FTA})\\
&\subseteq \mathrm{dom}(\mathrm{RT}(S_{\mathrm{id}})) \quad \text{(by Lemma 6.9)}\\
&\subseteq \mathrm{dom}(\mathrm{CF}(S_{\mathrm{id}}))\\
&\subseteq \mathrm{dom}(\mathrm{CF}(S)) \quad \text{(by Lemma 6.10).}
\end{aligned}$$

Since $\mathrm{dom}(\mathrm{CF}(\mathrm{P}(S))) = \mathrm{dom}(\mathrm{MAC}_{\mathrm{ext}}(S))$ (cf. Theorem 5.24), the second equation follows from the first one. □

In order to simplify the proof of $\mathrm{P}(S)_{\mathrm{LA}} \equiv \mathrm{P}(S_{\mathrm{LA}})$, we introduce an extension of the look-ahead concept. Roughly speaking, a look-ahead test of S_{LA} checks whether a configuration is in the domain of a $\mathrm{CF}(S)$-transducer. Hence, Lemma 6.11 justifies to allow look-ahead tests which decide the membership of domains of $\mathrm{CF}(\mathrm{P}(S))$-transducers without increasing the power of S_{LA}. For a precise formulation of this property of S_{LA}, we introduce the storage type $S_{\text{ind-LA}}$ in which the above mentioned enrichment is implemented.

6.12. Definition. Let $S_{\mathrm{LA}} = (C, P, F, m, I, E)$. The storage type S *with indexed look-ahead*, denoted by $S_{\text{ind-LA}}$, is the tuple (C, P', F, m', I, E), where

$$P' = P \cup \{\langle A, \gamma, \mathfrak{H}\rangle \mid \mathfrak{H} \text{ is a } \mathrm{CF}(\mathrm{P}(S))\text{-transducer, } A \text{ is a nonterminal of } \mathfrak{H}, \text{ and } \gamma \in \Gamma\},$$

m' restricted to $P \cup F$ is equal to m, and, for every $c \in C$, $m'(\langle A, \gamma, \mathfrak{H}\rangle)(c) =$ **true** iff there is a $w \in \Sigma^*$ such that $A((\gamma, c)) \Rightarrow_{\mathfrak{H}}^{*} w$, where Σ is the terminal alphabet of $\mathfrak{H}$. $\langle A, \gamma, \mathfrak{H}\rangle$ is also called an indexed look-ahead test.

Note that $S_{\text{ind-LA}}$ contains the usual type of look-ahead tests as in Definition 6.5 and indexed look-ahead tests.

6.13. Lemma. $S_{\mathrm{LA}} \equiv S_{\text{ind-LA}}$.

Proof. It clearly suffices to prove that $S_{\text{ind-LA}} \leq S_{\text{LA}}$. Let $S = (C, P, F, m, I, E)$ and let m_1 and m_2 be the meaning functions of $S_{\text{ind-LA}}$ and S_{LA} respectively. Let U be a finite restriction of $S_{\text{ind-LA}}$. We take the representation function to be the identity on C. Of course, for an indexed look-ahead test $\langle A, \gamma, \mathfrak{H}_1\rangle$ of U, we only have to provide an S_{LA}-flowchart for predicates which simulates $\langle A, \gamma, \mathfrak{H}_1\rangle$.

Consider the look-ahead test $\langle A, \gamma, \mathfrak{H}_1\rangle$ of U with the CF(P(S))-transducer $\mathfrak{H}_1 = (N, \_, \Sigma, \_, R)$.

In order to transform the set $C_1 = \{c \in C \mid m_1(\langle A, \gamma, \mathfrak{H}_1\rangle)(c) = \textbf{true}\}$ into the domain of a transducer (and to then being able to apply Lemma 6.11), we define the storage type $S' = (C, P, F, m, C, \{\mathrm{id}_C\})$, where id_C denotes the identity on C. Then, obviously, $C_1 = \mathrm{dom}(\tau(\mathfrak{H}_1'))$ where $\mathfrak{H}_1'$ is the CF(P(S'))-transducer (N, e, Σ, A, R) with $e = \lambda c \in C.(\gamma, c)$.

By Lemma 6.11 there is an CF(S')-transducer $\mathfrak{H}_2'$ such that $\mathrm{dom}(\tau(\mathfrak{H}_1)) = \mathrm{dom}(\tau(\mathfrak{H}_2'))$. Let $\mathfrak{H}_2' = (N_2, \mathrm{id}_C, \Sigma_2, A_2, R_2)$ for some N_2, Σ_2, A_2, and R_2.

But obviously, taking the CF(S)-transducer $\mathfrak{H}_2 = (N_2, \_, \Sigma_2, A_2, R_2)$, we obtain

$$\begin{aligned}\mathrm{dom}(\tau(\mathfrak{H}_2')) &= \{c \in C \mid \text{there is a } w \in \Sigma^* \text{ such that } A_2(c) \Rightarrow_{\mathfrak{H}_2}^* w\}\\ &= \{c \in C \mid m_2(\langle A_2, \mathfrak{H}_2\rangle)(c) = \textbf{true}\}.\end{aligned}$$

Hence, we have in total:

$$C_1 = \{c \in C \mid m_2(\langle A_2, \mathfrak{H}_2\rangle)(c) = \textbf{true}\}.$$

Now, the S_{LA}-flowchart ω for predicates which simulates the look-ahead test $\langle A, \gamma, \mathfrak{H}_1\rangle$ is entirely determined by its only rule which is

$$A_{\text{in}} \to \textbf{if } \langle A_2, \mathfrak{H}_2\rangle \textbf{ then true}(\mathrm{id}) \textbf{ else false}(\mathrm{id}). \qquad \square$$

Now we can show that P(S)$_{\text{LA}}$ is equivalent to P(S_{LA}). To prove this equivalence, by the previous lemma and the monotonicity of P (cf. Theorem 4.22), it clearly suffices to simulate P(S)$_{\text{LA}}$ by P($S_{\text{ind-LA}}$). This amounts to simulating a look-ahead test on a P(S)-configuration by a P($S_{\text{ind-LA}}$)-flowchart ω for predicates. The main idea of simulating such a look-ahead test $\langle A, \mathfrak{H}\rangle$ (for a CF(P(S))-transducer $\mathfrak{H}$) is to divide into two parts the derivation of $\mathfrak{H}$ on a P(S)-configuration $(\gamma, c)\beta$ starting with nonterminal A. In the first part, only those derivation steps are considered which do not test the rest-pushdown β (i.e., they end by popping (γ, c)); in the second part, the remaining steps of the considered derivation on β are collected (note that the derivation of $\mathfrak{H}$ can block already in the first part).

The first part can be simulated by the indexed look-ahead test $\langle A, \gamma, \mathfrak{H}'\rangle$ on c, where $\mathfrak{H}'$ is only a slight modification of $\mathfrak{H}$ on pushdown configurations of height one (note that we are allowed to use indexed look-ahead tests in the flowchart to be constructed).

For the simulation of the second part, we assume for the moment that we have the termination behavior of β at the topmost square (γ, c) available. The termination behavior of β indicates, for every nonterminal B of $\mathfrak{H}$, whether there is a successful

derivation of $\mathfrak{H}$ starting from $B(\beta)$ or not. Then, by using the assumption, this behavior of the rest-pushdown can be tested by the simulating flowchart ω. But how do we obtain the termination behavior of a pushdown configuration? This information is inductively computed during the growth of the configuration when simulating a push instruction. If a push(δ, f) is applied to a pushdown configuration $(\gamma, c)\beta$ where θ is the termination behavior of β, then the termination behavior of $(\gamma, c)\beta$ can be computed (by the same arguments as discussed above!) from the values of the indexed look-ahead tests on (γ, c) and θ. The formal construction is provided in the following theorem.

6.14. Theorem. $\mathrm{P}(S)_{\mathrm{LA}} \equiv \mathrm{P}(S_{\mathrm{LA}})$.

Proof. ($\mathrm{P}(S_{\mathrm{LA}}) \leqslant \mathrm{P}(S)_{\mathrm{LA}}$): A predicate like test($\langle A, \mathfrak{H}\rangle$) (with look-ahead test $\langle A, \mathfrak{H}\rangle$) can be simulated by first testing the pushdown symbol and then, if it is e.g. γ, by testing $\langle A, \gamma, \mathfrak{H}_\gamma\rangle$, where $\mathfrak{H}_\gamma$ is obtained from $\mathfrak{H}$ by replacing every predicate p by test(p) and every instruction f by stay(γ, f).

($\mathrm{P}(S)_{\mathrm{LA}} \leqslant \mathrm{P}(S_{\mathrm{LA}})$): Let U be a finite restriction of $\mathrm{P}(S)_{\mathrm{LA}}$. By Lemma 6.13, it is sufficient to prove that $U \leqslant_{\mathrm{d}} \mathrm{P}(S_{\text{ind-LA}})$. Let m_1 and m_2 be the meaning functions of U and of $\mathrm{P}(S_{\text{ind-LA}})$ respectively.

Let Γ_{f} be the finite set of pushdown symbols which are used in U. Since U uses only finitely many look-ahead tests, we can put all look-ahead transducers together, and thus assume that in all of these tests the same CF(P(S))-transducer $\mathfrak{H}$ is involved. Let $\mathfrak{H} = (N, \_, \Sigma, \_, R)$ with $N = \{A_1, \ldots, A_r\}$ for some $r \geqslant 1$.

Since Γ is infinite and Γ_{f} finite, we can assume that $\Gamma_{\mathrm{LA}} \subseteq \Gamma$, where $\Gamma_{\mathrm{LA}} = \{\langle \gamma, \theta\rangle \mid \gamma \in \Gamma_{\mathrm{f}}, \theta \in \{0, 1\}^r\}$. An element θ of Γ_{LA} will represent a termination behavior. In the sequel, $\theta(i)$ denotes the ith component of θ, and M_θ is the set of those nonterminals of N for which the corresponding component in θ contains a 1 (i.e., $M_\theta = \{A_i \in N \mid \theta(i) = 1$ for some $i \in [r]\}$).

We define the representation function $h : (\Gamma \times C)^+ \to (\Gamma \times C)^+$ as follows:

(i) For every $\gamma \in \Gamma_{\mathrm{f}}$ and $c \in C$, $c' = (\langle \gamma, \theta_0\rangle, c) \in \mathrm{dom}(h)$ where $\theta_0 = (0, \ldots, 0)$, and $h(c') = (\gamma, c)$;

(ii) for every $\langle \gamma, \theta\rangle \in \Gamma_{\mathrm{LA}}$, $c \in C$, and $\beta \in \mathrm{dom}(h)$: if, for every $i \in [r]$, it holds that $\theta(i) = 1$ iff there is a $w \in \Sigma^*$: $A_i(h(\beta)) \Rightarrow_{\mathfrak{H}}^* w$, then $c' = (\langle \gamma, \theta\rangle, c)\beta \in \mathrm{dom}(h)$, and $h(c') = (\gamma, c)h(\beta)$. (Thus, $h(c') = c''$ means that c' is obtained from c'' by adding all termination behaviors of all its suffixes to the appropriate pushdown squares; clearly, at the bottom square there is only one termination behavior represented by θ_0.)

Before we provide the flowcharts, we will introduce special sets of configurations of C which play a central role in the first parts of the derivations of $\mathfrak{H}$, as discussed before this lemma.

For every $A \in N$, $V \subseteq N$, and $\gamma \in \Gamma_{\mathrm{f}}$,

$$L(A, \gamma, V) = \{c \in C \mid \text{there are } B_1, \ldots, B_n \in V \text{ and there are } w_0, \ldots, w_n \in \Sigma^* \text{ for some } n \geqslant 0 \text{ such that, for every } \beta \in (\Gamma \times C)^+,$$

$A((\gamma, c)\beta) \Rightarrow_{\mathfrak{H}}^{*} w_0 B_1(\beta) w_1 \ldots B_n(\beta) w_n$ and β is not tested in this derivation}.

For every $V \subseteq N$ we construct the CF(P(S))-transducer $\mathfrak{H}(V)$ such that, for every $A \in N$ and every $\gamma \in \Gamma_f$,

$$L(A, \gamma, V) = \{c \in C \mid \text{there is a } w \in \Sigma^* \text{ such that } A((\gamma, c)) \Rightarrow_{\mathfrak{H}(V)}^{*} w\}.$$

The transducer $\mathfrak{H}(V)$ is obtained from $\mathfrak{H}$ by replacing every occurrence of a B(pop) in a rule which is applied to a pushdown of height 1 by an arbitrary terminal symbol, whenever $B \in V$. The separation between rules which are applied to pushdowns of height 1 and the other rules can be achieved, as usual, by marking the pushdown symbol of the bottom square.

Requirement (1) *of Definition* 4.6: Let $e_1 = \lambda u \in I.(\gamma, e(u))$ be the encoding of U. Then $e_2 = \lambda u \in I.(\langle \gamma, \theta_0 \rangle, e(u))$ fulfils requirement (1).

Requirement (2) *of Definition* 4.6: Every predicate p of U is simulated by a P($S_{\text{ind-LA}}$)-flowchart ω_p for predicates. Since all the information of a pushdown square (γ, c) is also contained in every corresponding representation, the flowcharts for top $= \gamma$ and test(p) are obvious. The flowchart $\omega = \omega_{\langle A, \mathfrak{H} \rangle}$ uses the nonterminals A_{in}, term(γ, θ) for every $\gamma \in \Gamma_f$ and $\theta \in \{0, 1\}^r$, and **true** and **false**, and is determined by the following rules: for every $\langle \gamma, \theta \rangle \in \Gamma_{\text{LA}}$ and every $\theta \in \{0, 1\}^r$, the rules

$$A_{\text{in}} \to \textbf{if top} = \langle \gamma, \theta \rangle \textbf{ then } \text{term}(\gamma, \theta)(\text{id}),$$

$$\text{term}(\gamma, \theta) \to \textbf{if test}(\langle A, \gamma, \mathfrak{H}(M_\theta) \rangle) \textbf{ then true}(\text{id}) \textbf{ else false}(\text{id})$$

are in ω.

Since oper(ω) is the identity, requirements (2.1.1) and (2.1.2) are trivially true. For the proof of (2.2), we only consider a pushdown configuration of height greater than 1. Let $c_2 = (\langle \gamma, \theta \rangle, c)\beta$ and let $c_2 \in \text{dom}(h)$, i.e., θ is the termination behavior of β. Then we obtain the following equivalent statements (note that the third statement defines the above-mentioned division of the derivation of $\mathfrak{H}$, in the second statement):

$m_1(\langle A, \mathfrak{H} \rangle)(h(c_2)) = \textbf{true}$

iff there is a $w \in \Sigma^*$ such that $A(h(c_2)) \Rightarrow_{\mathfrak{H}}^{*} w$

iff there are $A_{\nu(1)}, \ldots, A_{\nu(k)} \in N$ and $w_0, \ldots, w_k \in \Sigma^*$ with $k \geq 0$ such that

$A((\gamma, c)h(\beta)) \Rightarrow_{\mathfrak{H}}^{*} w_0 A_{\nu(1)}(h(\beta)) w_1 \ldots A_{\nu(k)}(h(\beta)) w_k$ and in this derivation $h(\beta)$ is not tested, and there are $v_1, \ldots, v_k \in \Sigma^*$ such that $A_{\nu(i)}(h(\beta)) \Rightarrow_{\mathfrak{H}}^{*} v_i$ for every $i \in [k]$

iff $c \in L(A, \gamma, M_\theta)$

iff there is a $w \in \Sigma^*$ such that $A((\gamma, c)) \Rightarrow_{\mathfrak{H}(M_\theta)}^{*} w$

iff $m_2(\text{test}(\langle A, \gamma, \mathfrak{H}(M_\theta) \rangle))(c_2) = \textbf{true}$.

Requirement (3) *of Definition* 4.6: Every instruction ϕ of U is simulated by a $\mathrm{P}(S_{\text{ind-LA}})$-flowchart ω_ϕ for instructions. In the case that $\phi \in \{\text{pop}, \text{id}, \text{stay}(\delta), \text{stay}(\delta, f)\}$, ω_ϕ is just the obvious embedding of ϕ in a flowchart. Note that, for the stay instructions, the termination behavior has to be kept.

In the flowchart $\omega = \omega_{\text{push}(\delta,f)}$, we have to make sure that the new pushdown square contains the correct termination behavior. For the sake of convenience, we define, for every $\gamma \in \Gamma_{\mathrm{f}}$ and every $\theta \in \{0,1\}^r$, the set

$$\mathrm{SEQ}(\gamma, \theta) = \{\tilde{t}_1 \textbf{ and } \ldots \textbf{ and } \tilde{t}_r \mid \tilde{t}_i \in \{\mathrm{test}(\langle A_i, \gamma, \mathfrak{H}(M_\theta)\rangle), \textbf{not}\mathrm{test}(\langle A_i, \gamma, \mathfrak{H}(M_\theta)\rangle)\} \text{ for every } i \in [r]\}.$$

Then, ω is determined as follows: for every $\langle \gamma, \theta\rangle \in \Gamma_{\mathrm{LA}}$ and every $\mathrm{seq} \in \mathrm{SEQ}(\gamma, \theta)$,

$$A_{\text{in}} \to \textbf{if } \mathrm{top} = \langle \gamma, \theta\rangle \textbf{ and } \mathrm{seq} \textbf{ then stop}(\mathrm{push}(\langle \delta, \mathrm{seq}'\rangle, f))$$

is a rule of ω, where $\mathrm{seq}' \in \{0,1\}^r$ and

$$\mathrm{seq}'(i) = 1 \text{ iff } \mathrm{seq}(i) = \mathrm{test}(\langle A_i, \gamma, M(\theta)\rangle).$$

Obviously, it is sufficient to prove that ω transforms $\mathrm{dom}(h)$ into itself. But this immediately follows from the way in which ω computes the new termination behavior from the old one, the way in which $\mathrm{dom}(h)$ is defined, and the above equivalences in the proof of requirement (2). □

Clearly, $(S_0)_{\mathrm{LA}} \equiv S_0$ and it follows from the monotonicity of P that $\mathrm{P}_{\mathrm{LA}} \equiv \mathrm{P}$. This equivalence is closely related to a result of [31, Section 12.3] where a predicting machine for a deterministic pushdown machine and a finite automaton is constructed to recognize the quotient of the corresponding languages. This construction actually motivated us to obtain Theorem 6.14.

Now, we can pick the flowers of our work and present the main result of this section.

6.15. Theorem. *If* $\mathrm{P}(S_{\mathrm{LA}}) \equiv \mathrm{P}(S)$, *then* $\mathrm{D_tMAC}(S) = \mathrm{D_tREG}(\mathrm{P}^2(S)) = \mathrm{D_tREG}(\mathrm{P}^2_{\text{bex}}(S))$.

Proof. The implication of the theorem immediately follows from Theorems 6.8 and 6.14. □

Here, we only mention that for the storage type TR, the assumption of Theorem 6.15 holds. Actually, it has already been proven in [19, Theorem 4.7] that the regular look-ahead can be deterministically simulated by a ct-pd transducer, which, in our terminology, means that a look-ahead test on TR-configurations can be simulated by a P(TR)-flowchart (cf. Theorem 8.1).

Note that $S_{\mathrm{LA}} \equiv S$ implies $\mathrm{P}(S_{\mathrm{LA}}) \equiv \mathrm{P}(S)$, but not vice versa (e.g., $\mathrm{TR}_{\mathrm{LA}}$ is not equivalent to TR because $\mathrm{RT}(\mathrm{TR}) \subsetneq \mathrm{RT}(\mathrm{TR}_{\mathrm{LA}})$, see [12]). Thus, in this sense, Theorem 6.15 is stronger than Theorem 6.8.

7. Pushdown^2 versus nested stack

This section is devoted to a space optimization of the storage type $P^2(S)$. In fact, $P^2(S)$ is equivalent to the nested stack of S, denoted by $NS(S)$, where the storage type operator NS is obtained in the same way from the usual nested stack of [1] as the pushdown operator is obtained from the usual pushdown: besides a usual stack symbol, every square of a nested stack configuration contains also a configuration of S. The equivalence $P^2(S) \equiv NS(S)$ (cf. Theorem 7.4), together with the justification theorem and the characterization of $MAC(S)$ by pushdown^2 S-to-string transducers offers a second characterization of $MAC(S)$ by means of $REG(NS(S))$-transducers (cf. Theorem 7.6). The equivalence of $P^2(S)$ and $NS(S)$ is claimed and also partly proved in [17]. Here, we want to give a complete formal proof.

The storage type nested stack of S can be understood as an extension of the storage type $P(S)$ with the operational features of the usual nested stack of [1]. A configuration of $NS(S)$ can be viewed as a collection of stacks such that each of them is nested between two squares of another one, except one stack which we will call the main stack. Furthermore, this nesting relation is acyclic. From this point of view, the structure of an $NS(S)$-configuration may be called a tree of stacks (cf. [23, 21]). Fig. 10 shows an $NS(S)$-configuration as a tree of stacks.

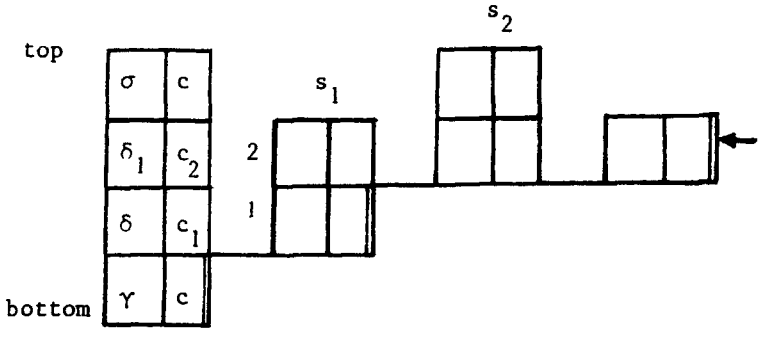


Fig. 10. An $NS(S)$-configuration viewed as a tree of stacks. Stack s_2 is nested between squares 1 and 2 of stack s_1. (For the explanation of the double bars and the arrow cf. next paragraph.)

In the sequel we will describe the instructions of $NS(S)$ informally. Besides the usual instructions push(γ, f), pop, stay(γ, f), stay(γ), and the identity id (cf. Fig. 11(a) for the first two instructions), the nested stack provides the ability to read and test the contents of a large part of its squares. The instructions by which these squares can be reached are called 'movedown' and 'moveup' (cf. Fig. 11(b)). In order to indicate the square which is presently scanned we use the notion of stack pointer (the arrow in Fig. 10). All squares which are reachable from the stack pointer are indicated by a double vertical bar at the right in Fig. 10. Furthermore, in a nested stack one can create a new stack, which is nested between the current stack square and the one below (cf. Fig. 11(c)). The corresponding instruction is create(γ), where γ indicates the initial inscription of the new stack square. Via the inverse instruction, called 'destruct', a (nested) stack which contains only one square is

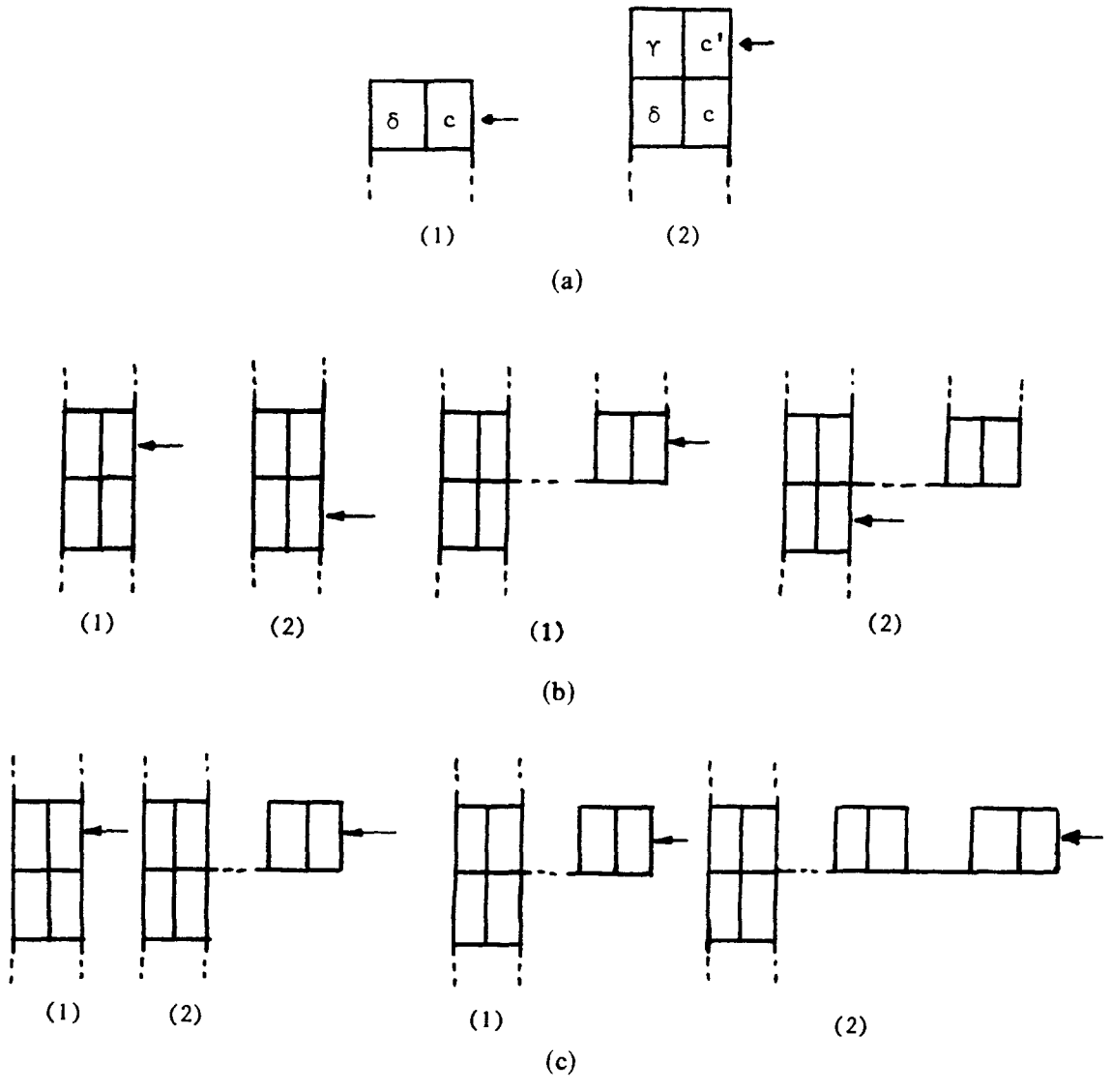


Fig. 11. Instructions of NS(S). (a) push(γ, f): $1 \Rightarrow 2$; pop: $2 \Rightarrow 1$. (b) movedown: $1 \Rightarrow 2$; moveup: $2 \Rightarrow 1$. (c) create: $1 \Rightarrow 2$; destruct: $2 \Rightarrow 1$.

destructed, i.e., it is erased from the total configuration (cf. Fig. 11(c)). The stack pointer moves to that square that was immediately above the destructed stack. The situation in which at the bottom square of a stack a movedown instruction is applied, is shown in Fig. 11(b).

Now we provide the formal definition of the nested stack operator.

7.1. Definition. The *nested stack of* S, denoted by NS(S), is the storage type (C', P', F', m', I', E'), where

(i) $C' = \mathrm{NS}(\Gamma, C)^*.\{!\}.\mathrm{NS}(\Gamma, C)^+$ is the set of NS(S)-configurations, where Γ is a fixed infinite set,

$$\mathrm{NS}(\Gamma, C) = \mathrm{Sym}(\Gamma) \times C, \qquad \mathrm{Sym}(\Gamma) = \Gamma_{\#} \cup \{\$\}.\Gamma_{\#}, \Gamma_{\#} = \Gamma \cup \Gamma.\{\#\}$$

for some symbols !, \$, and $\# \notin \Gamma$ (the dot denotes string concatenation; intuitively, "\$" denotes the top of a stack and "#" its bottom, where the top is at the left; "!"

is the stack pointer; a general element of C' looks like $\alpha_1\alpha_2 \ldots \alpha_i ! \alpha_{i+1} \ldots \alpha_n$, where $\alpha_j \in \mathrm{NS}(\Gamma, C)$ for every $j \in [n]$ and $n \geqslant 1$; α_{i+1} is the current stack square);

(ii) $P' = \{\mathrm{sym} = \zeta \mid \zeta \in \mathrm{Sym}(\Gamma)\} \cup \{\mathrm{test}(p) \mid p \in P\}$;

(iii) $F' = \{\mathrm{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{pop}\} \cup \{\mathrm{movedown}, \mathrm{moveup}\}$
$\cup \{\mathrm{create}(\gamma) \mid \gamma \in \Gamma\} \cup \{\mathrm{destruct}\} \cup \{\mathrm{stay}(\gamma, f) \mid \gamma \in \Gamma, f \in F\}$
$\cup \{\mathrm{stay}(\gamma) \mid \gamma \in \Gamma\} \cup \{\mathrm{id}\}$.

(iv) We use w_1 and w_2 to range over $\mathrm{NS}(\Gamma, C)^*$. For every $w_1 ! (\zeta, c) w_2 \in C'$ with $\zeta \in \mathrm{Sym}(\Gamma)$ and $c \in C$

$$m'(\mathrm{sym} = \xi)(w_1 ! (\zeta, c) w_2) = \textbf{true} \text{ iff } \xi = \zeta,$$
$$m'(\mathrm{test}(p))(w_1 ! (\zeta, c) w_2) = m(p)(c).$$

The meaning of each instruction of F' is provided as a relation R on C'. In parentheses we mention (as a comment) the condition under which the application of an instruction is 'defined', i.e., gives a configuration as result. (Of course, for $\mathrm{push}(\gamma, f)$ and $\mathrm{stay}(\gamma, f)$ instructions definedness depends also on the definedness of f.)

$$m'(\mathrm{push}(\gamma, f)) = \{(w_1 ! (\$\alpha, c) w_2, w_1 ! (\$\gamma, c')(\alpha, c) w_2) \mid \alpha \in \Gamma_*, c \in C, \text{ and } c' = m(f)(c) \in C\} \quad \text{(only at the top)},$$

$$m'(\mathrm{pop}) = \{(w_1 ! (\$\gamma, c')(\alpha, c) w_2, w_1 ! (\$\alpha, c) w_2) \mid c, c' \in C, \gamma \in \Gamma, \alpha \in \Gamma_*\} \quad \text{(only at top, not at bottom)},$$

$$m'(\mathrm{moveup}) = \{(w_1 \kappa ! (\alpha, c) w_2, w_1 ! \kappa (\alpha, c) w_2) \mid \kappa \in \mathrm{NS}(\Gamma, C), \alpha \in \Gamma_*, c \in C\} \quad \text{(not at top)},$$

$$m'(\mathrm{movedown}) = m'(\mathrm{moveup})^{-1} \quad \text{(not at lowest bottom)},$$

$$m'(\mathrm{create}(\gamma)) = \{(w_1 ! (\zeta, c) w_2, w_1 (\zeta, c) ! (\$\gamma \#, c) w_2) \mid \zeta \in \mathrm{Sym}(\Gamma), c \in C\} \quad \text{(everywhere)},$$

$$m'(\mathrm{destruct}) = \{(w_1 \kappa ! (\$\gamma \#, c) w_2, w_1 ! \kappa w_2) \mid \kappa \in \mathrm{NS}(\Gamma, C), \gamma \in \Gamma, c \in C\} \quad \text{(only at one square stacks)},$$

$$m'(\mathrm{stay}(\gamma, f)) = \{(w_1 ! (\$\delta Z, c) w_2, w_1 ! (\$\gamma Z, c') w_2) \mid \delta \in \Gamma, Z \in \{\lambda, \#\}, c \in C, c' = m(f)(c) \text{ and } c' \in C\} \quad \text{(only at top)},$$

$$m'(\mathrm{stay}(\gamma)) = \{(w_1 ! (\$\delta Z, c) w_2, w_1 ! (\$\gamma Z, c) w_2) \mid \delta \in \Gamma, Z \in \{\lambda, \#\}, c \in C\} \quad \text{(only at top)},$$

$$m'(\mathrm{id}) = \{(w_1 ! \kappa w_2, w_1 ! \kappa w_2) \mid \kappa \in \mathrm{NS}(\Gamma, C)\} \quad \text{(everywhere)}.$$

The input device of $\mathrm{NS}(S)$ is defined as follows:

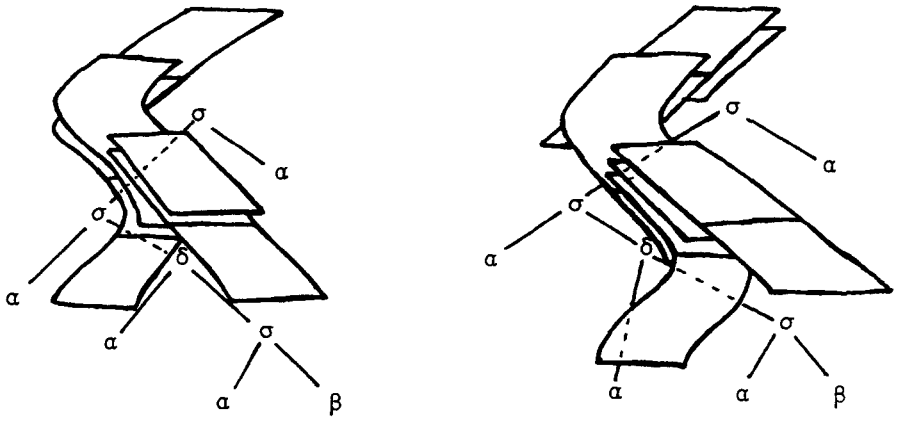
(v) $I' = I$;

(vi) $E' = \{\lambda u \in I . ! (\$\gamma_0 \#, e(u)) \mid \gamma_0 \in \Gamma, e \in E\}$.

By applying NS to the trivial storage type, we reobtain the usual nested stack of [1, 2].

In the sequel we will often use the notion of reachable nested stack configuration. This is a configuration c' of NS(S) such that there is an encoding e', an input element u, and a sequence ϕ of NS(S)-instructions with $c' = m'(\phi)(e'(u))$, where m' denotes the meaning function of NS(S). Thus, the reachable configurations are the ones which are actually used by X(NS(S))-transducers. We note that in a reachable configuration c of NS(S) the stack pointer can only occur between the rightmost \$ and the right end of c.

For the time being let us consider the storage type NS(TR) in which no stay(γ, f) instructions are present. (Actually, they are not essential and can be removed by the same technique as for stay(γ, f) instructions in P(S); cf. Theorem 4.21). For a configuration of this storage type it is an easy observation that the structure of the tree of stacks nicely fits to the structure of the underlying tree. This provides a very intuitive picture of NS(TR)-configurations (cf. Fig. 12; the nested stack in Fig. 12(a) corresponds to the nested stack of Fig. 10). Note that the nested stack is put upside down on the tree; thus, the bottom of the nested stack is at the root of the tree; in particular, the NS instruction 'movedown' moves up in the tree (and 'moveup' moves down)!



(a) (b)

Fig. 12. Two configurations of NS(TR) viewed as trees of stacks with an underlying tree.

In fact, the configuration shown in Fig. 12(a) is reachable from the configuration (γ, t), where

$$t = \sigma(\sigma(\alpha, \delta(\alpha, \sigma(\alpha, \beta))), \alpha),$$

by applying the following sequence of instructions to (γ, t), where we drop the pushdown symbols:

push(sel$_1$); push(sel$_2$); push(sel$_1$); movedown; movedown;

create; push(sel$_2$); create; push(sel$_2$); movedown; create.

Now we discuss the simulation of $P^2(S)$ by $NS(S)$ and consider the case that $S = TR$. Actually, since $P_1(S) \equiv P(S)$ (cf. Theorem 4.21) and the operator P_1 is monotonic (cf. Theorem 4.22), it is sufficient to simulate $P_1^2(TR)$ by NS(TR). Hence, we can keep the intuitive picture of a $P_1^2(TR)$-configuration in our mind (as discussed at the end of Section 3.3) and view it as a collection of layers on a tree. Fig. 13 shows again an example of such a configuration.

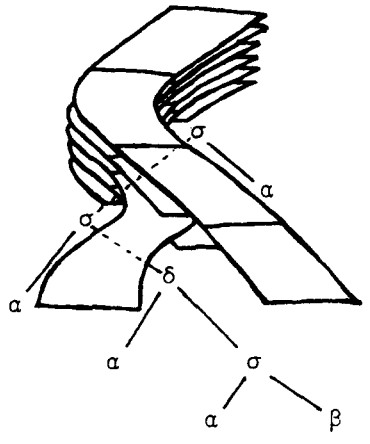


Fig. 13. A configuration of $P^2(TR)$ viewed as a collection of layers with an underlying tree.

In Section 3.3 we already mentioned that a lot of redundant information is stored in a $P_1^2(TR)$-configuration c. If, for example, a push(γ, ϕ)-instruction is applied to c, then another layer s is put on top of the already existing ones. But s differs from its underlying neighbour s' only at its lowest end (with respect to Fig. 13): if $\phi = \text{pop}$, then s is one cell shorter than s'; if $\phi = \text{push}(\gamma, \text{sel}_i)$, then s is one cell longer than s'. All the rest of s' is stored again in the resulting $P_1^2(TR)$-configuration. This is, of course, superfluous, because the information is already contained in c. And, in fact, the nested stack of TR is an equivalent storage type in which this redundant storing of information can be avoided. Instead of putting another layer on top of the already existing collection (i.e., instead of a push(γ, ϕ)), a new stack is created: if $\phi = \text{pop}$, then first, the stack pointer moves down, and second, a stack is created; if $\phi = \text{push}(\gamma, \text{sel}_i)$, a stack is created and the NS(TR)-instruction push(γ, sel_i) is applied. Clearly, in every NS(TR)-configuration c' which simulates c, every stack is associated to one layer of c and vice versa. A layer s itself is reobtained from c' by applying movedown instructions as often as possible, when starting from the top of the corresponding stack. Actually, the NS(TR)-configuration in Fig. 12(b) represents the $P_1^2(TR)$-configuration in Fig. 13.

In the next lemma we will give a formal description of the simulation of $P^2(S)$. For the sake of clarity, a layer is also called a secondary pushdown. The pushdown of layers is called the main pushdown.

7.2. Lemma. $P^2(S) \leqslant NS(S)$.

Proof. Since $P^2(S) \equiv P_1^2(S)$ (by Theorem 4.21 and Theorem 4.22), we consider a finite restriction U of $P_1^2(S)$ and prove that $U \leqslant_d NS(S)$. Let Γ_f be the finite set of

pushdown symbols which are used in U. Recall that S denotes the storage type (C, P, F, m, I, E). Let $\mathrm{NS}(S) = (C', P', F', m', I, E')$. Note that the input set of U and $\mathrm{NS}(S)$ are both equal to I.

For the proof of the requirements (1)–(3) of Definition 4.6, we first define the set R of reachable nested stack configurations c', in which the stack pointer points to the top of that stack that can be reached from the bottom of c' by repeated application of moveup instructions. More precisely,

$$R = \{c' \in C' \mid c' \text{ is reachable and its stack pointer is at the rightmost square which contains a \$}\}.$$

For $c' \in C'$, let $p(c') \in \mathrm{NS}(\Gamma, C)^+$ be obtained from c' by dropping the stack pointer (!). Let $\tilde{R} = \{p(c') \mid c' \in R\}$.

Since the idea is to unify parts of secondary pushdowns (which together form a $\mathrm{P}^2(S)$-configuration), the representing $\mathrm{NS}(S)$-configuration should keep track of the pushdown symbol of the main pushdown. This is done by storing such a symbol in the topmost square of that stack which corresponds to the secondary pushdown. For this purpose, we restrict R by requiring that, additionally, the current stack square contains an element of the set

$$\{\$\langle \gamma, \delta\rangle Z \mid \gamma, \delta \in \Gamma_f \text{ and } Z \in \{\lambda, \#\}\}.$$

In this way, we obtain the set $R(\Gamma)$; again $\tilde{R}(\Gamma)$ is obtained from $R(\Gamma)$ by dropping the stack pointer.

Then, the representation function $h: C' \to (\Gamma \times (\Gamma \times C)^+)^+$ is defined by $\mathrm{dom}(h) = R(\Gamma)$ and, for every $c' \in R(\Gamma)$, $h(c') = g(p(c'))$, and

$$g: \tilde{R}(\Gamma) \cup \{\lambda\} \to (\Gamma \times (\Gamma \times C)^+)^*$$

is the total function given as follows:

(i) if $c' = w_1!(\$\langle \sigma, \delta\rangle, c_1)w_2(\gamma\#, c_2)w_3 \in R(\Gamma)$ and w_2 does not contain a #, then

$$g(p(c')) = (\sigma, (\delta, c_1)w_2(\gamma, c_2)w_3')g(w_1 w_3),$$

where w_3' is obtained from w_3 by dropping every #;

(ii) if $c' = w_1!(\$\langle \sigma, \delta\rangle\#, c)w_3 \in R(\Gamma)$, then

$$g(p(c')) = (\sigma, (\delta, c)w_3')g(w_1 w_3),$$

and w_3' as in (i);

(iii) $g(\lambda) = \lambda$.

Note that since the place of the stack pointer is fixed (in the configuration of $R(\Gamma)$), this is an unambiguous definition of g.

Requirement (1) *of Definition* 4.6: Let e_1' be the encoding of U. Then there are $\gamma_{01}, \gamma_{02} \in \Gamma_f$ and there is an encoding $e \in E$ such that $e_1' = \lambda u \in I.(\gamma_{01}, (\gamma_{02}, e(u)))$. Then, define $e_2' \in E'$ by $e_2' = \lambda u \in I.!(\$\langle \gamma_{01}, \gamma_{02}\rangle\#, e(u))$. Obviously, requirement (1) holds.

Requirement (2) *of Definition* 4.6: Since all the information of a U-configuration c' which can be tested is contained in the current stack square of the representing nested stack configuration, it is easy to construct flowcharts for the simulation of the predicates: top $= \gamma$ and test(top $= \delta$) are true if the current stack square contains $\$\langle\gamma, \delta\rangle$ or $\$\langle\gamma, \delta\rangle\#$, and test(test($p$)) is simulated by test($p$).

Requirement (3) *of Definition* 4.6: Every instruction ϕ of U is simulated by an NS(S)*-flowchart* ω_ϕ for instructions. We will describe the flowcharts by defining their sets of rules.

(i) $\delta = \text{push}(\gamma, \phi')$ with $\phi' \in \{\text{push}(\delta, f), \text{stay}(\delta)\}$: for every $\sigma, \alpha \in \Gamma_f$,

$$A_{\text{in}} \to \textbf{if } \text{sym} = \$\langle\sigma, \alpha\rangle \textbf{ or } \text{sym} = \$\langle\sigma, \alpha\rangle\# \textbf{ then } A(\text{create}(\alpha))$$

and $A \to \textbf{stop}(\phi'')$ are rules of ω_ϕ, where ϕ'' is obtained from ϕ' by replacing δ by $\langle\gamma, \delta\rangle$.

(ii) $\phi = \text{push}(\gamma, \text{id})$: for every $\sigma, \alpha \in \Gamma_f$,

$$A_{\text{in}} \to \textbf{if } \text{sym} = \$\langle\sigma, \alpha\rangle \textbf{ or } \text{sym} = \$\langle\sigma, \alpha\rangle\# \textbf{ then stop}(\text{create}(\langle\gamma, \alpha\rangle)).$$

(iii) $\phi = \text{push}(\text{pop})$: for every $\alpha \in \Gamma_f$,

$$A_{\text{in}} \to A(\text{movedown}),$$

$$A \to \textbf{if } \text{sym} = \alpha \textbf{ or } \text{sym} = \alpha\# \textbf{ then stop}(\text{create}(\langle\gamma, \alpha\rangle)).$$

(iv) $\phi = \text{pop}$: Let bottom? be the disjunction of the predicates $\text{sym} = \$\alpha\#$ and $\text{sym} = \$\langle\sigma, \alpha\rangle\#$ for every $\sigma, \alpha \in \Gamma_f$, and let top? be the disjunction of the predicates $\text{sym} = \$\langle\sigma, \alpha\rangle$ and $\$\langle\sigma, \alpha\rangle\#$ for every $\sigma, \alpha \in \Gamma_f$. Then,

$$A_{\text{in}} \to \textbf{if bottom? then } A(\text{destruct}) \textbf{ else } A_{\text{in}}(\text{pop}),$$

$$A \to \textbf{if top? then stop}(\text{id}) \textbf{ else } A(\text{moveup})$$

are rules in ω_ϕ.

(v) $\phi = \text{stay}(\gamma)$: for every $\sigma, \alpha \in \Gamma_f$,

$$A_{\text{in}} \to \textbf{if } \text{sym} = \$\langle\sigma, \alpha\rangle \textbf{ or } \text{sym} = \$\langle\sigma, \alpha\rangle\# \textbf{ then stop}(\text{stay}(\langle\gamma, \alpha\rangle)).$$

(vi) $\phi = \text{id}$: $A_{\text{in}} \to \textbf{stop}(\text{id})$.

It is easy to check that (3.1.1) and (3.1.2) hold. For the proof of (3.2) we only consider the case $\phi = \text{push}(\gamma, \text{push}(\delta, f))$ and the configuration $c' = w_1!(\$\langle\sigma, \alpha\rangle, c_1)w_2(\beta\#, c_2)w_3 \in R(\Gamma)$, where w_2 does not contain a #. The other cases are similar and omitted here. Let $m'(\phi)(c')$ be defined. Then ω_ϕ can compute as follows.

$$A_{\text{in}}(c') \Rightarrow A(w_1(\$\langle\sigma, \alpha\rangle, c_1)!(\$\alpha\#, c_1)w_2(\beta\#, c_2)w_3) \Rightarrow \textbf{stop}(c''),$$

with

$$c'' = w_1(\$\langle\sigma, \alpha\rangle, c_1)!(\$\langle\gamma, \delta\rangle, m(f)(c))(\alpha\#, c_1)w_2(\beta\#, c_2)w_3.$$

Then,

$$h(\text{oper}(\omega_\phi)(c')) = h(c'') = \zeta_1 g(w_1(\$\langle\sigma, \alpha\rangle, c_1)w_2(\beta\#, c_2)w_3),$$

where $\zeta_1 = (\gamma, (\delta, m(f)(c_1))(\alpha, c_1)w_2(\beta, c_2)w_3')$

$= \zeta_1\zeta_2 g(w_1 w_3),$

where $\zeta_2 = (\sigma, (\alpha, c_1)w_2(\beta, c_2)w_3')$

$= m'(\phi)(\zeta_2 g(w_1 w_3)) = m'(\phi)(h(c)).$ □

For the simulation of NS(S) by P^2(S), we take a more operational method to explain the representation of a nested stack configuration. Consider an initial configuration c'_{01} of NS(S) (i.e., there is a symbol γ, an encoding e, and an input element u such that $c'_{01} = !(\$\gamma\#, e(u))$) and a nested stack configuration c'_1 which is reachable from c'_{01}. Then, it is an easy observation that c'_1 can also be obtained (and, in fact, uniquely) by the application of an instruction sequence ϕ to c'_{01}, where the form of ϕ is described by the regular expression (push* movedown* create)* (for the sake of clarity, the arguments of 'push' and 'stay' are left out here). Now we replace in ϕ the NS(S)-instructions by P^2(S)-instructions: push is replaced by stay(push), movedown by push(pop), and create by push(stay) (where we have dropped again the arguments of the instructions). If the obtained sequence is applied to the initial configuration $c'_{02} = (\$, (\gamma, e(u)))$ of P^2(S), then we obtain the configuration c'_2 of P^2(S), which represents c'_1 (cf. Fig. 14, where $c'_{01} = !(\$\gamma\#, c)$,

$\phi = \text{push}(\delta, f); \text{push}(\sigma, g); \text{push}(\gamma, f); \text{movedown};$

$\text{movedown}; \text{create}(\sigma); \text{push}(\gamma, f),$

$c_1 = m(f)(c), \quad c_2 = m(g)(c_1), \quad c_3 = m(f)(c_2), \quad \text{and} \quad c_4 = m(f)(c_1)).$

In outline this is the idea of the representation of NS(S)-configurations, but there is still an important detail left to be mentioned. Intuitively, there is some more operational freedom on an P^2(S)-configuration (like c'_2) than on the NS(S)-configuration which is represented. After having simulated a movedown instruction, the P^2(S) can continue by simulating a push or a pop (of NS(S)) although this

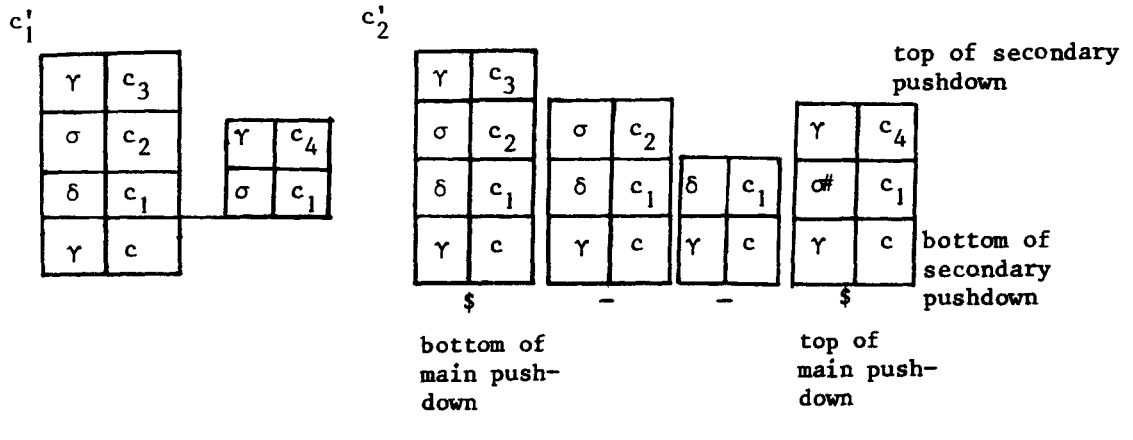


Fig. 14. Representation of the NS(S)-configuration c'_1 by the P^2(S)-configuration c'_2. For the meaning of \$ and _, see text.

does not correspond to a legal action in NS(S). In order to forbid this possibility, we attach the symbol \$ to those secondary pushdowns at which a push or pop can be simulated.

Now we provide the formal construction.

7.3. Lemma. $\mathrm{NS}(S) \leqslant \mathrm{P}^2(S)$.

Proof. Consider the finite restriction U of NS(S) and let $\Gamma_f \subseteq \Gamma$ be the finite set of symbols which is used in U. Let C' be the set of configurations of NS(S) and let m' denote the meaning function of U. We show that $U \leqslant_d \mathrm{P}^2(S)$. The input sets of U and $\mathrm{P}^2(S)$ are both equal to I.

The representation function $h: (\Gamma \times (\Gamma \times C)^+)^+ \to C'$ is defined to be the inverse of a function $g: C' \to (\Gamma \times (\Gamma \times C)^+)^+$. The domain of g is the set of reachable NS(S)-configurations. Then g is defined as follows, where we use w_1, w_2, and w_3 to range over $\mathrm{NS}(\Gamma, C)^*$ and c_1, c_2, and c_3 to range over C.

(1) For every $\psi \in \mathrm{Sym}(\Gamma)$ and every $\gamma \in \Gamma_\#$,

$$g(w_1(\psi, c_1)!(\gamma, c_2)w_2) = (\_, (\gamma, c_2)w_2)g(w_1!(\psi, c_1)(\gamma, c_2)w_2).$$

(2) For every $\gamma, \gamma' \in \Gamma$, $\psi \in \mathrm{Sym}(\Gamma)$, and w_2 not containing \#,

(a) $$g(w_1(\psi, c_1)!(\$\gamma, c_2)w_2(\gamma'\#, c_3)w_3) = (\$, (\gamma, c_2)w_2(\gamma'\#, c_3)w_3)g(w_1!(\psi, c_1)w_3)$$

and

$$g(w_1(\psi, c_1)!(\$\gamma\#, c_2)w_3) = (\$, (\gamma\#, c_2)w_3)g(w_1!(\psi, c_1)w_3),$$

(b) $$g(!(\$\gamma, c_2)w_2(\gamma'\#, c_3)) = (\$, (\gamma, c_2)w_2(\gamma'\#, c_3)),$$

$$g(!(\$\gamma\#, c_2)) = (\$, (\gamma\#, c_2)).$$

Note that we keep \# in the $\mathrm{P}^2(S)$-configurations in order to detect the bottom of a stack.

Intuitively, in (1), g moves up and indicates this on the main pushdown by "_": at the corresponding secondary pushdown no push or pop instruction (of NS(S)) may be simulated. Case (2) treats the situation that g has reached the top of a stack; then this stack is represented by a secondary pushdown with \$ as information, which indicates the permit of the simulation of push and pop.

Note that, in (1), a movedown is replaced by push(pop), whereas in (2) a number of pushes are replaced by stay(push)'s and a 'create' by push(stay).

Note that g is injective. Hence, we can define the inverse of g, which, finally, is the representation function h for the desired simulation. The domain of h is equal to the range of g and, for every $c' \in \mathrm{dom}(h)$, $h(c') = g^{-1}(c')$.

Requirement (1) *of Definition* 4.6: Let $e_1' = \lambda u \in I.!(\$\gamma_0\#, e(u))$ be the encoding of U for some $\gamma_0 \in \Gamma_f$ and some encoding e of E. Then, $e_2' = \lambda u \in I.(\$, (\gamma_0\#, e(u))$ satisfies the conditions.

Requirement (2) *of Definition* 4.6: Since all the information of a nested stack configuration c' which can be tested (by predicates $\mathrm{sym}=\gamma$ and $\mathrm{test}(p)$) is also contained at the top of the topmost secondary pushdown of the $\mathrm{P}^2(S)$-configuration that represents c', the flowcharts are very easy to construct. We leave this to the reader. The proof of requirement (2) is an easy calculation, which takes into account that the operation induced by the flowcharts is the identity.

Requirement (3) *of Definition* 4.6: Every instruction ϕ of U is simulated by the $\mathrm{P}^2(S)$-flowchart ω_ϕ for instructions; ω_ϕ is determined by its rules. Since Γ_f is a finite set, we clearly can use the test "$\mathrm{test}(\mathrm{top}\in\Gamma_\mathrm{f})$" with its obvious meaning; note that the secondary pushdown symbols are in $\Gamma_\mathrm{f}\cup\Gamma_\mathrm{f}\#$.

(i) $\phi\in\{\mathrm{push}(\gamma,f),\mathrm{stay}(\gamma,f),\mathrm{stay}(\gamma)\}$:

$$A_\mathrm{in}\to\textbf{if top}=\$\ \textbf{then}\ \mathrm{stop}(\mathrm{stay}(\$,\phi))$$

is in ω_ϕ.

(ii) $\phi=\mathrm{pop}$: $A_\mathrm{in}\to\textbf{if top}=\$\ \textbf{and}\ \mathrm{test}(\mathrm{top}\in\Gamma_\mathrm{f})\ \textbf{then stop}(\mathrm{stay}(\$,\mathrm{pop}))$ is in ω_ϕ.

(iii) $\phi=\mathrm{create}(\gamma)$: $A_\mathrm{in}\to\textbf{stop}(\mathrm{push}(\$,\mathrm{stay}(\gamma\#)))$ is in ω_ϕ.

(iv) $\phi=\mathrm{destruct}$:

$$A_\mathrm{in}\to\textbf{if top}=\$\ \textbf{and}\ \mathrm{test}(\mathrm{top}\in\Gamma_\mathrm{f}\#)\ \textbf{then stop}(\mathrm{pop})$$

is in ω_ϕ.

(v) $\phi=\mathrm{movedown}$: $A_\mathrm{in}\to\textbf{stop}(\mathrm{push}(\_,\mathrm{pop}))$ is in ω_ϕ.

(vi) $\phi=\mathrm{moveup}$: $A_\mathrm{in}\to\textbf{if top}=\_\ \textbf{then stop}(\mathrm{pop})$ is in ω_ϕ.

(vii) $\phi=\mathrm{id}$: $A_\mathrm{in}\to\textbf{stop}(\mathrm{id})$ is in ω_ϕ.

For the proof of (3.1.1), (3.1.2), and (3.2) we will use the following two claims which correspond to (3.1.2) and (3.2) respectively.

Claim 14. *For every* $c'\in\mathrm{dom}(g)$ *and every instruction symbol* ϕ *of* U, $m'(\phi)(c')$ *is defined iff* $\mathrm{oper}(\omega_\phi)(g(c'))$ *is defined.*

Claim 15. *For every* $c'\in\mathrm{dom}(g)$ *and every instruction symbol* ϕ *of* U, *if* $m'(\phi)(c')$ *is defined, then* $g(m'(\phi)(c'))=\mathrm{oper}(\omega_\phi)(g(c'))$.

The proof of Claim 14 is an easy inspection of the different instruction symbols and the corresponding flowcharts. For the proof of Claim 15 we only give one example, where $\phi=\mathrm{create}(\delta)$ and $c'=w_1(\psi,c_1)!(\$\gamma,c_2)w_2(\gamma'\#,c_3)w_3)$ and the objects are quantified as in part (2) of the definition of g (note that w_2 does not contain $\#$). Let $m'(\phi)(c')$ be defined. Then,

$$\begin{aligned}g(m'(\phi)(c')) &= g(w_1(\psi,c_1)(\$\gamma,c_2)!(\$\delta\#,c_2)w_2(\gamma'\#,c_3)w_3)\\ &= \mathrm{pd}_1 g(w_1(\psi,c_1)!(\$\gamma,c_2)w_2(\gamma'\#,c_3)w_3),\end{aligned}$$

with $\mathrm{pd}_1=(\$,(\delta\#,c_2)w_2(\gamma'\#,c_3)w_3)$,

$$= \mathrm{pd}_1\mathrm{pd}_2 g(w_1!(\psi,c_1)w_3),$$

with $\mathrm{pd}_2 = (\$, (\gamma, c_2)w_2(\gamma' \#, c_3)w_3)$,

$$= \mathrm{oper}(\omega_\phi)(\mathrm{pd}_2 g(w_1!(\psi, c_1)w_3))$$

$$= \mathrm{oper}(\omega_\phi)(g(w_1(\psi, c_1)!(\$\gamma, c_2)w_2(\gamma' \#, c_3)w_3))$$

$$= \mathrm{oper}(\omega_\phi)(g(c')).$$

Proof of (3.1.1): Let ϕ be an instruction of U and let $c_1' \in \mathrm{dom}(h) \cap \mathrm{dom}(\mathrm{oper}(\phi))$; then there is a $c_2' \in C'$ such that $g(c_2') = c_1'$. By Claim 14, $m'(\phi)(c_2')$ is defined. Then, by Claim 15, $\mathrm{oper}(\omega_\phi)(c_1') = g(m'(\phi)(c_2'))$. Since $c_2' \in \mathrm{dom}(g)$, c_2' is reachable, and, hence, $m'(\phi)(c_2') \in \mathrm{dom}(g)$. Then, $g(m'(\phi)(c_2')) \in \mathrm{range}(g) = \mathrm{dom}(h)$, i.e., $\mathrm{oper}(\omega_\phi)(c_1') \in \mathrm{dom}(h)$.

The proofs of (3.1.2) and of (3.2) are immediately obtained from Claims 14 and 15. □

The previous two lemmata prove the equivalence of $\mathrm{P}^2(S)$ and $\mathrm{NS}(S)$.

7.4. Theorem. $\mathrm{P}^2(S) \equiv \mathrm{NS}(S)$.

Since nested stacks can clearly be viewed as elaborated pushdown devices, we obtain our second characterization of $\mathrm{MAC}(S)$ by means of pushdown machines. We will introduce a name for the involved regular, sequential transducer.

7.5. Definition. A $\mathrm{REG}(\mathrm{NS}(S))$-transducer is called a *nested stack S-to-string transducer.*

A nested stack TR-to-string transducer is referred to as a nested stack tree-to-string transducer.

7.6. Theorem. $\mathrm{MAC}_{\mathrm{ext}}(S) = \mathrm{REG}(\mathrm{NS}(S))$ *and, if* $\mathrm{P}(S_{\mathrm{LA}}) \equiv \mathrm{P}(S)$, *then* $\mathrm{D_tMAC}(S) = \mathrm{D_tREG}(\mathrm{NS}(S))$.

Proof. This immediately follows from the characterization of $\mathrm{MAC}(S)$ by $\mathrm{REG}(\mathrm{P}^2(S))$-transducers (cf. Theorem 6.4 and Theorem 6.15), the equivalence of $\mathrm{P}^2(S)$ and $\mathrm{NS}(S)$ (cf. Theorem 7.4), and the justification theorem (cf. Theorem 4.18). □

Thus extended macro S-transducers are characterized by nested stack S-to-string transducers. For $S = \mathrm{TR}$ this is already conjectured in [15]. Since the trivial storage type contains an identity and hence $\mathrm{MAC}_{\mathrm{ext}}(S_0) = \mathrm{MAC}(S_0)$, we reobtain from the second equation the equivalence of (OI) macro grammars and nested stack automata which is proved in [24, 1].

Now, the reader might expect that we will offer a restriction of NS(S) which is equivalent to $P^2_{bex}(S)$. However, we can only provide a conjecture how such a restriction could look like. Of course, we have to count excursions and, for a 'bounded excursion' NS(S) (for short: $NS_{bex}(S)$), there are three bounds which have to be respected. Each square has two counters: one to count the pushes applied at the square (starting with 0 when the square is pushed, just as for $P_{bex}(S)$), and the other to count the movedowns applied at the square (also starting with 0 when the square is pushed, and moreover reset to 0 when a moveup is applied to the square). Thus, the push-counter counts the excursions below the square and the movedown counter counts the excursions above the square. Moreover, a counter is attached to every stack, namely, to every \$: it is used to count the number of 'creates' applied when this \$ is the rightmost \$-symbol in the linear representation of the NS(S)-configuration (started with 0 when the \$ is created). The instruction 'create' copies the push-counter and the movedown-counter of the current square. Note that the sequence of \$'s in a nested stack configuration acts as a pushdown with the top at the right (rather than at the left, as for all the stacks), and 'create' and 'destruct' play the role of 'push' and 'pop', respectively; the counters at the \$-symbols check that this \$-pushdown is bounded-excursion. This is in an outline a possible definition of a bounded-excursion nested stack of S; moreover, we claim that the suggested storage type is equivalent to $P^2_{bex}(S)$.

8. Characterizations of the macro tree transducer

Now the time is ripe to harvest the fruits from the trees which we have planted in Sections 3–7: in Section 8.1 we will give a complete survey of the studied pushdown machines for the macro tree transducer. The power of the concept of separating program and storage type (as discussed in Section 3) and the concept of storage type simulation (cf. Section 4) will again become apparent in Section 8.2. There we will provide pushdown machines with iterated pushdowns as storage for the characterization of compositions of total deterministic macro tree transducers. Recall from Theorem 3.22 that the class of translations induced by macro tree transducers is CFT(TR) and similarly for the total deterministic case.

8.1. Total deterministic and nondeterministic transducers

We are interested in characterizations of the macro tree(-to-string) transducer by regular transducers (i.e., RT(S)- and REG(S)-transducers) and have considered several pushdown devices, namely, P(S), $P_{bex}(S)$, TP(S), $TP_{ext}(S)$, and NS(S). Since total deterministic macro tree transducers are more relevant as metalanguage for the description of the semantics of programming languages than their nondeterministic versions, we will first summarize (in Theorem 8.2) those pushdown machines that characterize D_tCFT(TR) and D_tMAC(TR). However, the results of Section 6.2 show that we can obtain characterizations of D_tMAC(S) by regular transducers

only under the condition that $P(S_{LA}) \equiv P(S)$ (cf. Theorem 6.15). But this condition holds for the storage type TR and, actually, this has already been proved in Theorem 4.7 of [19]. There it is shown that a deterministic checking-tree pushdown transducer (for short: dct-pd transducer) can handle regular look-ahead. Since a dct-pd transducer is equivalent to a DREG(P(TR))-transducer, it can be used as a P(TR)-flowchart for the simulation of look-ahead tests on TR-configurations. We reformulate the result of [19] as an equivalence of storage types and give a short description of the proof.

8.1. Theorem. $P(TR_{LA}) \equiv P(TR)$.

Proof. We only have to prove $P(TR_{LA}) \leq P(TR)$. The representation function is the identity on P(TR)-configurations and the instructions of $P(TR_{LA})$ are simulated by themselves in the usual sense.

Consider the predicate $p = \text{test}(\langle A, \mathfrak{H} \rangle)$, where $\mathfrak{H}$ is a CF(TR)-transducer, and the P(TR)-configuration $c = (\gamma, t)\beta$ to which p is applied (note that t is a tree). Now, the success of derivations of $\mathfrak{H}$ starting with $A(t)$ is checked by the P(TR)-flowchart ω which simulates p. First, ω marks the topmost square of c. Then it inscribes this square with a rule, by which $\mathfrak{H}$ could start a derivation on $A(t)$, and checks whether the obtained derivation step can be completed to a successful derivation. (It does this by going from left to right through the right-hand side of the rule; whenever it meets a $B(\text{sel}_i)$, it applies a $\text{push}(B, \text{sel}_i)$ and checks, in the same way, whether there is a successful derivation from $B(m(\text{sel}_i)(t))$.) If ω does not succeed, then the next rule is considered (note that there are only finitely many). Of course, the question whether a prefix of a derivation of $\mathfrak{H}$ can be prolonged to a successful derivation arises at every node of t at which ω 'arrives' (note that, since $\mathfrak{H}$ may delete subtrees of t, not necessarily every node of t is considered). By using backtracking, ω can answer this question in a sequential way rather than in a parallel way, as $\mathfrak{H}$ does. □

Now we will provide the pushdown machines for the $D_tCFT(TR)$- and $D_tMAC(TR)$-transducer.

8.2. Theorem. $D_tCFT(TR) = D_tRT(P(TR))$ *and* $D_tMAC(TR) = D_tREG(P^2(TR)) = D_tREG(NS(TR))$.

Proof. The first equation holds because of Theorem 5.16. Theorems 8.1, 6.15, and 7.6 prove the second equation. □

Thus, in the total deterministic case, the class of translations induced by macro tree transducers is characterized by indexed tree transducers and the class of translations induced by macro tree-to-string transducers is characterized by pushdown^2 tree-to-string transducers and by nested stack tree-to-string transducers.

Since every noncircular attribute grammar can be considered as a $D_tRT(P(TR))$-transducer (cf. remark behind Theorem 5.16), Theorem 8.2 also repeats the result of [25] that $AG \subseteq D_tCFT(TR)$. Hence, also, $yield(AG) \subseteq D_tMAC(TR)$ and so $yield(AG) \subseteq D_tREG(P^2(TR))$. This fits with the fact that the tree-walking pushdown transducer, defined in [32] for the simulation of attribute grammars (viewed as tree-to-string transducers), clearly is a special case of a $D_tREG(P^2(TR))$-transducer.

We also note that we can now give an alternative proof of the fact (cf. [22]) that total deterministic macro tree transducers are closed under (regular) look-ahead:

$$\begin{aligned} D_tCFT(TR_{LA}) &= D_tRT(P(TR_{LA})) && \text{(by Theorem 8.2)} \\ &= D_tRT(P(TR)) && \text{(by Theorem 8.1 and the justification theorem)} \\ &= D_tCFT(TR) && \text{(by Theorem 8.2).} \end{aligned}$$

The next theorem collects the results for the nondeterministic case, which are more or less of theoretical importance only.

8.3. Theorem

(a) $CFT(TR) = RT(P_{bex}(TR))$ *and* $CFT_{ext}(TR) = RT(P(TR))$.

(b) $MAC(TR) = REG(P^2_{bex}(TR))$ *and* $MAC_{ext}(TR) = REG(P^2(TR)) = REG(NS(TR))$.

Proof. Theorems 5.14 and 5.24 prove the equations in (a); Theorems 6.4 and 7.6 imply the equations of (b). □

Note that in the total deterministic case, all classes in (a) coincide, and similarly for (b) (cf. Corollary 5.25 and Theorem 6.15).

In the previous two theorems, P can be replaced by TP_{ext}, and P_{bex} by TP; note that

$$MAC(S) = CF(TP(S)) = REG(P_{bex}(TP(S))) = REG(TP^2(S)).$$

Since all proofs in this paper are effective, the equalities in Theorems 8.2 and 8.3 are all effective. From this, it follows in particular that *totality* (defined in a nonconstructive way in Definition 3.12) *is decidable* for all the transducers mentioned in Theorem 8.3. Proof: it suffices to prove this for $CFT_{ext}(TR)$. By Lemma 6.11 and Corollary 3.20, $dom(CFT_{ext}(TR)) = dom(CF(TR)) = dom(T)$: the class of domains of top-down tree transducers. Hence [37], $dom(CFT_{ext}(TR)) = RECOG$, effectively. Thus, to decide whether an extended macro tree transducer is total, one can decide whether its recognizable domain equals T_Σ, where Σ is the input alphabet specified by its encoding.

8.2. Composition of total deterministic macro tree transducers

In this section, we will consider the n-fold composition of total deterministic macro tree transducers. We will provide a characterization of the induced class of

translations $D_tCFT(TR)^n$ by means of pushdown machines with an iterated pushdown device as storage (i.e., by means of $D_tRT(P^n(TR))$- and $D_tREG(P^{n+1}(TR))$-transducers), cf. Theorem 8.12. Composition of tree transducers correspond to specifying syntax-directed semantics in several phases. Whereas the class of total deterministic top-down tree transducers is closed under composition [37], i.e., any number of phases can also be done in one phase, composition of total deterministic macro tree transducers gives rise to a proper hierarchy (i.e., $D_tCFT(TR)^n \subsetneq D_tCFT(TR)^{n+1}$ for all n, cf. [22, Theorem 4.16]. In [15] it is shown that compositions of macro tree transducers can be simulated by compositions of attribute grammars (viewed as tree transducers), and vice versa (note that composition of attribute grammars is different from, but closely related to, multi-pass attribute grammars). The above-mentioned characterization result says that the composition of macro tree transducers (and thus of attribute grammars) can be realized in one phase, i.e., by one tree transducer, with an iterated pushdown storage type. Iterated pushdown automata are a natural (the natural?) extension of pushdown automata, related to higher-level macro languages [35, 9] and super-exponential complexity classes [17].

The way in which we will prove this characterization result again stresses the usefulness of the separation of grammar and storage (as discussed in Section 3) and the machinery of storage type simulation (cf. Section 4). The first approach will be used to consider transducer classes with simple storage types. Instead of dealing with the relatively complex $D_tRT(P^n(TR))$-transducers, we will consider $D_tRT(P(S))$-transducers. We will prove a connection between the increase of the level of iteration of the pushdown operator in $D_tRT(P(S))$-transducers and the increase of the level of composition of $D_tCFT(TR)$-transducers, viz., $D_tRT(P(S)) = D_tRT(S) \circ D_tCFT(TR)$ (cf. Corollary 8.11). Actually, this connection is an invariant that holds on every level of iteration of P and composition of $D_tCFT(TR)$-transducers. However, this invariant only holds under the condition that S is closed under look-ahead. Since this property holds for $P^n(TR)$, which will be proved with the help of the second approach (using $P(S)_{LA} \equiv P(S_{LA})$ of Theorem 6.14 and the monotonicity of P), we can 'shuffle' the invariant through the hierarchies $D_tCFT(TR)^n$ and $D_tRT(P^n(TR))$. The proof of the above-mentioned invariant (Corollary 8.11) amounts to prove a characterization of $D_tCFT(S)$-transducers, (by $D_tRT(S) \circ D_tCFT(TR)$), where the decomposition (cf. Lemma 8.5) follows the idea of its nondeterministic counterpart (cf. Theorem 3.26 and Corollary 3.27). For the composition (cf. Lemma 8.9) we will provide a direct construction.

For getting prepared for Lemma 8.5, let us recall the technique of Theorem 3.26 used in the decomposition of a $CFT(S)$-transducer $\mathfrak{M}$. The derivations of $\mathfrak{M}$ starting with an S-configuration c are simulated by a $CFT(TR)$-transducer $\mathfrak{M}'$ which works on approximations of c. The approximations are trees (the nodes of which are labeled by a standard test and some instructions) and are nondeterministically 'generated' from c by an approximator $\mathfrak{A}$, which is a particular $RT(S)$-transducer. The nondeterminism comes in by the fact that $\mathfrak{A}$ does not know in advance how far the approximation should be developed such that $\mathfrak{M}'$ can perform a successful

derivation on it. But in our present situation we deal with a $D_tCFT(S)$-transducer $\mathfrak{M}$, which executes exactly one derivation for a given S-configuration. Hence, it is possible to determine with which nonterminals an S-configuration is associated in a derivation of $\mathfrak{M}$ and, furthermore, which instructions are applied to it. This information about the nonterminals of $\mathfrak{M}$ can be kept track of in the approximator $\mathfrak{A}$ by coding it into $\mathfrak{A}$'s nonterminals (i.e., the nonterminals of $\mathfrak{A}$ are sets of nonterminals of $\mathfrak{M}$). Using this technique, the approximator becomes total deterministic; it can stop the approximation as soon as the set of nonterminals of $\mathfrak{M}$ becomes empty (indicating that $\mathfrak{M}$ will not use further S-configurations).

However, now we have cheated the reader a bit and have kept secret a small detail: the transducer $\mathfrak{M}$ can delete a parameter in which a nonterminal occurs. Thus, although the approximator has computed beforehand that a certain nonterminal will be applied to a certain configuration and contribute to the derived terminal tree, actually, the nonterminal will be deleted. This is the reason why we can prove the decomposition result of $D_tCFT(S)$-transducers only for those transducers that preserve, in every derivation step, the presence of nonterminals occurring in parameter positions. The preservation property of a $D_tCFT(S)$-transducer $\mathfrak{M}$ can always be obtained if the initial term of $\mathfrak{M}$ is a single nonterminal and if $\mathfrak{M}$ is enriched by look-ahead tests (cf. Lemma 8.7).

But let us now define the notion of preservation of nonterminals and then turn to the decomposition.

8.4. Definition. Let $\mathfrak{M} = (N, e, \Delta, A_{in}, R)$ be a $CFT(S)$-transducer. $\mathfrak{M}$ *preserves nonterminals in parameter positions* if $\tau(\mathfrak{M}) = \tau_p(\mathfrak{M})$, where

$$\tau_p(\mathfrak{M}) = \{(u, t) \mid u \in I,\ t \in T_\Delta, \text{ and } A_{in}(e(u)) \Rightarrow^*_{p,\mathfrak{M}} t\}$$

and $\Rightarrow_{p,\mathfrak{M}} \subseteq F_{CFT}(N(C), \Delta)^2$ is defined as follows: let $\xi_1, \xi_2 \in F_{CFT}(N(C), \Delta)$: $\xi_1 \Rightarrow_{p,\mathfrak{M}} \xi_2$ iff $\xi_1 \Rightarrow_{\mathfrak{M}} \xi_2$ and if in this step the rule $A(y_1, \ldots, y_k) \to$ **if** b **then** t is applied to the subtree $A(c)(t_1, \ldots, t_k)$ of ξ_1, then, for every $i \in [k]$ such that t_i contains a nonterminal, y_i occurs in t.

The class of translations induced by $D_tCFT(S)$-transducers which preserve nonterminals in parameter positions, is denoted by $p\text{-}D_tCFT(S)$. (And recall that a subscript 1 means that the initial term is a nonterminal.)

8.5. Lemma. $p\text{-}D_tCFT_1(S) \subseteq D_tRT(S) \circ D_tCFT_1(TR)$.

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{in}, R)$ be a $D_tCFT_1(S)$-transducer in standard test form which preserves nonterminals in parameter positions. Let P_f be the finite set of predicates which are used in rules of $\mathfrak{M}$ and let $\{f_1, \ldots, f_r\}$ be the set of instructions which occur in right-hand sides of rules of $\mathfrak{M}$.

Construct the DRT(S)-transducer $\mathfrak{M}_1 = (N_1, e, \Sigma, \{A_{\text{in}}\}, R_1)$ by

(i) $N_1 = \{V \mid V \text{ is a nonempty subset of } N\}$;

(ii) $\Sigma = A(P_f, \tilde{\psi})$, where $\tilde{\psi} = \langle f_1, \ldots, f_r \rangle$ (recall from Definition 3.23 the notion of approximation alphabet);

(iii) and R_1 is defined as follows: we define, for every standard test b over P_f, every $U \subseteq N$ and $f \in \{f_1, \ldots, f_r\}$, the auxiliary set

$$V(U, b, f) = \{B \in N \mid B(f) \text{ occurs in the right-hand side of an } (A, b)\text{-rule for some } A \in U\};$$

then, for every standard test b over P_f, every $U \subseteq N$, and every $\nu(1), \ldots, \nu(n) \in [r]$ such that $j \in \{\nu(1), \ldots, \nu(n)\}$ iff $V(U, b, f_j) \neq \emptyset$, and for every $\rho \in [n-1]$, $\nu(\rho) < \nu(\rho+1)$, the rule

$$U \to \textbf{if } b \textbf{ then } \langle b; f_{\nu(1)}, \ldots, f_{\nu(n)} \rangle (V(U, b, f_{\nu(1)})(f_{\nu(1)}), \ldots, V(U, b, f_{\nu(n)})(f_{\nu(n)}))$$

is in R.

Construct the DCFT$_1$(TR)-transducer $\mathfrak{M}_2 = (N, e_2, \Delta, A_{\text{in}}, R_2)$, where e_2 is the identity on the set of trees over $A(P_f, \tilde{\psi})$ and R_2 is defined as follows: if $A(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ then } \zeta$ is in R, then, for every $\mu(1), \ldots, \mu(n) \in [r]$ with $n \geq 0$ such that, for every $\rho \in [n-1]$, $\mu(\rho) < \mu(\rho+1)$, the rule

$$A(y_1, \ldots, y_k) \to \textbf{if } \text{root} = \langle b; f_{\mu(1)}, \ldots, f_{\mu(n)} \rangle \textbf{ then } \zeta[f_{\mu(i)} \leftarrow \text{sel}_i; i \in [r]]$$

is in R_2.

For the proof of $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2)$, we consider a derivation $d = (\xi_1 \Rightarrow_{p,\mathfrak{M}} \cdots \Rightarrow_{p,\mathfrak{M}} \xi_k)$ of $\mathfrak{M}$ where $\zeta_1 = A_{\text{in}}(c)$ with $c \in \text{range}(e)$ and $\xi_k \in T_\Delta$. The question is whether $\mathfrak{M}_1$ can derive an approximation of c such that $\mathfrak{M}_2$ can perform on it a derivation which corresponds to d (cf. Definition 3.23 for the notion of approximation). In fact, this question is positively answered and in the sequel we give an informal proof.

Assume that s is an approximation of c such that, for every interior node n of s, the instruction f occurs in the label of n iff $m(f)(c')$ occurs in a sentential form of d, where c' corresponds to n. (For the notion of correspondence of configuration and approximation, confer Theorem 3.26.) Intuitively, the condition on s means that perhaps s is not high enough for $\mathfrak{M}_2$ to perform a derivation corresponding to d on it, but, at every node n of s, s is 'thick' enough, i.e., the label of n already contains the complete sequence of instructions which will be applied to the configuration corresponding to n. Then, we can prove the following claim by induction on the size of s.

Claim 16. *$\{A_{\text{in}}\}(c) \Rightarrow^*_{\mathfrak{M}_1} \langle s \rangle$, and $\langle s \rangle$ is obtained from s by substituting $U_k(c')$ for every leaf κ of s, where c' is the S-configuration which corresponds to κ and $U_\kappa = \{B \in N \mid B(c')$ occurs in $d\}$.*

The proof of this claim uses the fact that $\mathfrak{M}$ preserves nonterminals in parameter positions.

Actually, as in the proof of Theorem 3.26, we consider the approximation $s = \mathrm{ap}(d)$ on which $\mathfrak{M}_2$ can clearly simulate d: every node of $\mathrm{ap}(d)$ corresponds to a configuration of d and vice versa. It is clear that the above condition on s holds for $\mathrm{ap}(d)$. Hence, $\{A_{\mathrm{in}}\} \Rightarrow^*_{\mathfrak{M}_1} \langle \mathrm{ap}(d) \rangle$. Now, consider a leaf κ of $\mathrm{ap}(d)$ which is labeled by $\langle b; \rangle$. Then, for every instruction f in $\{f_1, \ldots, f_r\}$, $V(U_\kappa, b, f) = \emptyset$. Otherwise, there is a nonterminal B in N such that $B(f)$ occurs in the right-hand side tree of an (A, b)-rule for some $A \in U_\kappa$. Hence, by the preservation property of $\mathfrak{M}$, $B(m(f)(c'))$ occurs in d, where c' corresponds to κ. But this contradicts the definition of $\mathrm{ap}(d)$ and the fact that κ is a leaf of $\mathrm{ap}(d)$. Hence, there is a rule $U_\kappa \to \textbf{if } b \textbf{ then } \langle b; \rangle$ in $\mathfrak{M}_1$. Since this argumentation holds for every leaf of $\mathrm{ap}(d)$, it is clear that $\mathrm{ap}(d)$ can be derived from $\{A_{\mathrm{in}}\}(c)$ by $\mathfrak{M}_1$. This proves that $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2)$.

Since $\tau(\mathfrak{M})$ is a total function on $\mathrm{dom}(e)$ and $\tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2)$ is a partial function on $\mathrm{dom}(e)$, actually, the inclusion $\tau(\mathfrak{M}) \subseteq \tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2)$ can be turned into an equality. Hence, $\mathfrak{M}_1$ is total. But $\tau(\mathfrak{M}_2)$ may be a partial function, because we only know that $\mathrm{range}(\tau(\mathfrak{M}_1)) \subseteq \mathrm{dom}(\tau(\mathfrak{M}_2))$. We can repair this by adding, for every pair $(A, \mathrm{root} = \sigma)$ for which there is no $(A, \mathrm{root} = \sigma)$-rule in $\mathfrak{M}_2$, a dummy rule without changing the translation of $\mathfrak{M}_2$. Then the modification of $\mathfrak{M}_2$, denoted by $\mathfrak{M}_2'$, can be viewed as a total deterministic macro tree transducer (cf. Theorems 3.19 and 3.22), which computes a total function. Hence, $\mathfrak{M}_2'$ is total in the sense of Definition 3.12. □

As mentioned above, the property that nonterminals in parameter positions are preserved can be checked by look-ahead (cf. Lemma 8.7). To facilitate the construction we will enrich the storage type S_{LA} by another type of look-ahead tests in which the look-ahead transducer is a CFT(S)-transducer. Furthermore, such look-ahead tests can check whether the terminal tree produced by the look-ahead transducer is in a given regular tree language.

8.6. Definition. Let $S_{\mathrm{LA}} = (C, P, F, m, I, E)$. The storage type S *with macro look-ahead*, denoted by $S_{\text{mac-LA}}$, is the tuple (C, P', F, m', I, E), where

$$P' = P \cup \{\langle A, \mathfrak{H}, R \rangle \mid \mathfrak{H} \text{ is a CFT}(S)\text{-transducer, } A \text{ is a nonterminal of } \mathfrak{H}, \text{ and } R \text{ is a regular tree language}\}$$

and m' restricted to $P \cup F$ is equal to m, and, for every $c \in C$, $m'(\langle A, \mathfrak{H}, R \rangle)(c) = \textbf{true}$ iff there is a $t \in R$ such that $A(c)(y_1, \ldots, y_k) \Rightarrow^*_{\mathfrak{H}} t$, where k is the rank of A. $\langle A, \mathfrak{H}, R \rangle$ is also called a macro look-ahead test. (Note that we consider symbolic derivations of $\mathfrak{M}$ in which the y's may also occur in a sentential form as element of rank 0.)

In the next lemma we will construct for a $\mathrm{D_tCFT}(S)$-transducer $\mathfrak{M}$ an equivalent p-$\mathrm{D_tCFT}(S_{\text{mac-LA}})$-transducer. The idea behind the construction is the following. We consider constructs like $A(f)$ (for a nonterminal A and an instruction f) in right-hand sides of rules of $\mathfrak{M}$. During a derivation of $\mathfrak{M}$ a parameter of such a nonterminal may be deleted. Since this deletion can be checked by macro look-ahead

(see later), we can replace every tree in a deleted parameter position by an arbitrary terminal symbol. This means that we delete nonterminals from rules, before the parameter in which they occur is deleted in one of the following derivation steps. Then, obviously, the constructed transducer preserves nonterminals in parameter positions. But how can the deletion of a parameter y_j of a construct $A(f)$ (with A of rank k) be checked by look-ahead? For this purpose we use the macro look-ahead test $\langle A, \mathfrak{M}, R\rangle$ in which $\mathfrak{M}$ itself acts as look-ahead transducer. The regular tree language R contains only those trees in which at least one leaf is labeled by y_j. Then, $\langle A, \mathfrak{M}, R\rangle$ is true on the configuration $m(f)(c)$ of C iff in the derivation of $\mathfrak{M}$ starting from $A(m(f)(c))(y_1, \ldots, y_k)$ the jth parameter y_j of A is preserved (note that $\mathfrak{M}$ is deterministic). In this way the constructed transducer can determine the deletion pattern of a nonterminal. However, the deletion pattern of the initial term of a $D_tCFT(S)$-transducer cannot be determined by look-ahead tests. Since this would require an additional checking rule being put in front of every derivation, we would need the identity as instruction to 'reconstruct' the correct configuration. But, in general, such an instruction is not available. Hence, we only provide the preservation result for $D_tCFT_1(S)$-transducers.

8.7. Lemma. $D_tCFT_1(S) \subseteq \text{p-}D_tCFT_1(S_{\text{mac-LA}})$.

Proof. Let $\mathfrak{M} = (N, e, \Delta, A_{\text{in}}, R)$ be a $D_tCFT_1(S)$-transducer. As look-ahead transducer for the macro look-ahead tests we use slightly modified copies of $\mathfrak{M}$. For every $A \in N_k$ with $k \geq 1$ and every $f \in F$, define the $CFT(S)$-transducer $\mathfrak{M}(A,f) = (N', e, \Delta, A', R(A,f))$, where $N' = N \cup \{A'\}$ and A' is a new nonterminal of rank k, and $R(A,f) = R \cup \{A'(y_1, \ldots, y_k) \to A(f)(y_1, \ldots, y_k)\}$. For every $j \geq 0$ we define the regular tree language $R_j = \{t \in T_{\Delta'} \mid y_j \text{ occurs in } t\}$, where $\Delta' = \Delta \cup Y_k$. In the sequel the macro look-ahead tests $\langle A', \mathfrak{M}(A,f), R_j\rangle$ and **not**$\langle A', \mathfrak{M}(A,f), R_j\rangle$ are abbreviated by $\langle A, f, j\rangle$ and $\langle A, f, j\rangle'$, respectively.

For the construction of a $\text{p-}D_tCFT_1(S_{\text{mac-LA}})$-transducer $\mathfrak{M}'$ we define three auxiliary mappings.

The mapping θ considers a right-hand side of a rule of $\mathfrak{M}$ and produces, for every possible deletion pattern of the involved nonterminals, a right-hand side of the same structure, attaching to every nonterminal its particular deletion pattern. A deletion pattern of a nonterminal A of rank $k \geq 0$ is coded as a sequence $s \in \{0,1\}^k$, where 0 occurs in the jth component of s iff the jth parameter of A is deleted. Formally, the mapping

$$\theta : F_{\text{CFT}}(N(F), \Delta \cup Y) \to \mathcal{P}(F_{\text{CFT}}(N(F)', \Delta \cup Y)),$$

where

$$N(F)' = \{\langle A, f, s\rangle \mid A \in N_k \text{ with } k \geq 0,\ f \in F, \text{ and } s \in \{0,1\}^k\}$$

and $\mathcal{P}(V)$ denotes the power set of V, is defined as a finite substitution. For $\zeta \in F_{\text{CFT}}(N(F), \Delta \cup Y)$,

$$\theta(\zeta) = \zeta[A^{(k)}(f) \leftarrow \{\langle A, f, s\rangle \mid s \in \{0,1\}^k\};\ A(f) \in N(F)].$$

The mappings ψ_1 and ψ_2 use the information about the deletion pattern of a right-hand side tree, which θ has attached to it. For every tree $\zeta \in F_{\mathrm{CFT}}(N(F)', \Delta \cup Y)$, ψ_1 provides a sequence of macro look-ahead tests such that their conjunction is true on a configuration c iff the information about the deletion pattern of ζ is correct, whenever the derivation of $\mathfrak{M}$ starts with $\zeta[f \leftarrow m(f)(c)]$. ψ_2 replaces a tree in the parameter position j of a nonterminal by an arbitrary terminal, whenever, according to the deletion pattern, the jth parameter of that nonterminal is deleted. Formally, define $\psi_1 : F_{\mathrm{CFT}}(N(F)', \Delta \cup Y) \to \mathrm{TEST}^*$, with

$$\mathrm{TEST} = \{\langle A, f, j\rangle, \langle A, f, j\rangle' \mid A \in N_k \text{ with } k \geq 1, f \in F, j \in [k]\},$$

and $\psi_2 : F_{\mathrm{CFT}}(N(F)', \Delta \cup Y) \to F_{\mathrm{CFT}}(N(F), \Delta \cup Y)$ inductively as follows:

(i) for every $\delta \in \Delta_k$ with $k \geq 0$ and $\zeta_1, \ldots, \zeta_k \in F_{\mathrm{CFT}}(N(F)', \Delta \cup Y)$,

$$\psi_1(\delta(\zeta_1, \ldots, \zeta_k)) = \psi_1(\zeta_1) \ldots \psi_1(\zeta_k),$$

$$\psi_2(\delta(\zeta_1, \ldots, \zeta_k)) = \delta(\psi_2(\zeta_1), \ldots, \psi_2(\zeta_k));$$

(ii) for every $y \in Y$, $\psi_1(y) = \lambda$ and $\psi_2(y) = y$;

(iii) for every $\langle A, f, s\rangle$ with $A \in N_k$, $k \geq 0$, $s \in \{0, 1\}^k$, and $\zeta_1, \ldots, \zeta_k \in F_{\mathrm{CFT}}(N(F)', \Delta \cup Y)$,

$$\psi_1(\langle A, f, s\rangle(\zeta_1, \ldots, \zeta_k)) = \xi_1 \ldots \xi_k,$$

where, for every $j \in [k]$, $\xi_j = \langle A, f, j\rangle\psi_1(\zeta_j)$ if $s(j) = 1$ and $\langle A, f, j\rangle'$ otherwise, and

$$\psi_2(\langle A, f, s\rangle(\zeta_1, \ldots, \zeta_k)) = A(f)(\zeta_1', \ldots, \zeta_k')$$

where, for every $j \in [k]$, $\zeta_j' = \psi_2(\zeta_j)$ if $s(j) = 1$ and b otherwise (b is an arbitrary symbol of Δ_0).

Finally, we can construct the p-$\mathrm{DCFT}_1(S_{\text{mac-LA}})$-transducer $\mathfrak{M}' = (N, e, \Delta, A_{\mathrm{in}}, R')$ by providing its set of rules. If $A(y_1, \ldots, y_k) \to$ **if** b **then** ζ is in R, then, for every $\tilde{\zeta} \in \theta(\zeta)$, the rule

$$A(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ and } \psi_1(\tilde{\zeta}) \textbf{ then } \psi_2(\tilde{\zeta})$$

is in R'.

Since for every $c \in C$ and $\zeta \in F_{\mathrm{CFT}}(N(F), \Delta \cup Y)$ there is a $\tilde{\zeta} \in \theta(\zeta)$ such that $m(\psi_1(\tilde{\zeta}))(c) =$ **true**, the addition of look-ahead tests does not provide any additional blocking. Hence, $\mathfrak{M}'$ is also total. Since ψ_2 just cuts out those parameters which will be deleted in any case, it should be clear that $\tau(\mathfrak{M}) = \tau(\mathfrak{M}')$. □

As for $S_{\text{ind-LA}}$, the enrichment by macro look-ahead tests does not increase the power of S_{LA}.

8.8. Lemma. $S_{\mathrm{LA}} \equiv S_{\text{mac-LA}}$.

Proof. The proof is very similar to the proof of Lemma 6.13. Hence, we only stress the main points and the differences here.

Consider a macro look-ahead test $\langle A, \mathfrak{H}_1, R\rangle$ and a finite state tree automaton $\mathfrak{M}$ such that $\mathrm{dom}(\tau(\mathfrak{M})) = R$. Define the set $C_1 = \{c \in C \mid \langle A, \mathfrak{H}_1, R\rangle \text{ is true on } c\}$. Then, $C_1 = \mathrm{dom}(\tau(\mathfrak{H}_1') \circ \tau(\mathfrak{M}))$ where $\mathfrak{H}_1'$ is the CFT(S')-transducer obtained from $\mathfrak{H}_1$ by the following modifications. There is a new initial nonterminal A_{in}' of rank 0, the encoding is id_C (the identity on C), and if $A(y_1, \ldots, y_k) \to$ **if** b **then** ζ is a rule of $\mathfrak{H}_1$, then $\mathfrak{H}_1'$ additionally contains the rule $A_{\mathrm{in}}' \to$ **if** b **then** ζ, where the y's in ζ are viewed as terminal symbols of rank 0. The storage type S' is defined by $(C, P, F, m, C, \{\mathrm{id}_C\})$. By Lemma 6.11, there is a CF(S')-transducer $\mathfrak{H}_2'$ such that $\mathrm{dom}(\tau(\mathfrak{H}_1') \circ \tau(\mathfrak{M})) = \mathrm{dom}(\tau(\mathfrak{H}_2'))$. Hence, there is a CF($S$)-transducer $\mathfrak{H}_2$ with initial nonterminal A_{in} such that $\mathrm{dom}(\tau(\mathfrak{H}_2')) = \{c \in C \mid \langle A_{\mathrm{in}}, \mathfrak{H}_2\rangle \text{ is true on } c\}$. But this means that the macro look-ahead test $\langle A, \mathfrak{H}_1, R\rangle$ can be simulated by the look-ahead test $\langle A_{\mathrm{in}}, \mathfrak{H}_2\rangle$. □

Now we will show that the composition of a $\mathrm{D_tRT}(S)$-transducer and a $\mathrm{D_tCFT(TR)}$-transducer can be realized by a $\mathrm{D_tCFT}(S)$-transducer. This will be proved in the next lemma via a direct construction.

8.9. Lemma. $\mathrm{D_tRT}(S) \circ \mathrm{D_tCFT(TR)} \subseteq \mathrm{D_tCFT}(S)$.

Proof. Let $\mathfrak{M}_1 = (N_1, e_1, \Sigma, A_{\mathrm{in}}^1, R_1)$ be a $\mathrm{D_tRT}(S)$-transducer and let $\mathfrak{M}_2 = (N_2, e_2, \Delta, A_{\mathrm{in}}^2, R_2)$ be a $\mathrm{D_tCFT(TR)}$-transducer. By Theorem 3.22(a) we can assume that $A_{\mathrm{in}}^2 \in N_2$, i.e., that $\mathfrak{M}_2$ is a $\mathrm{D_tCFT_1(TR)}$-transducer. Without loss of generality, we can assume that e_2 is the identity on T_Σ and, by Lemma 3.18, that the tests of rules of R_2 have the form root $= \sigma$. Moreover, we can assume that, for every $B \in N_2$ and every $\sigma \in \Sigma$, there is a $(B, \mathrm{root} = \sigma)$-rule in R_2 (if not, we can just add a dummy rule). By the last assumption we can view $\mathfrak{M}_2$ as a total deterministic macro tree transducer (cf. Section 2.4). Hence, for every $A \in N_2$ of rank $k \geq 0$, and every $s \in T_\Sigma$ there is exactly one tree $t \in T_\Delta(Y_k)$ such that $A(s)(y_1, \ldots, y_k) \Rightarrow_{\mathfrak{M}_2}^* t$. We denote this tree by $M(A(s)(y_1, \ldots, y_k))$.

Now we construct the DCFT(TR)-transducer $\mathfrak{M}_2' = (N_2, e_2', \Delta', A_{\mathrm{in}}^2, R_2')$ which translates right-hand sides of rules of $\mathfrak{M}_1$ according to $\mathfrak{M}_2$.

(i) e_2' is the identity on $T_{\Sigma'}$, where $\Sigma' = \Sigma \cup N_1(F_{\mathrm{f}})$, F_{f} is the finite set of instructions which are used in rules of $\mathfrak{M}_1$, and the elements of $N_1(F_{\mathrm{f}})$ have rank 0;

(ii) $\Delta' = \Delta \cup N_3(F_{\mathrm{f}})$, where N_3 is the ranked alphabet defined by $\{\langle B, A\rangle^{(k)} \mid A \in N_1, B \in N_2 \text{ of rank } k \geq 0\}$;

(iii) R_2' is determined by (a) and (b).

(a) $R_2 \subseteq R_2'$,

(b) for every $B \in N_2$ of rank $k \geq 0$ and every $A(f) \in N_1(F_{\mathrm{f}})$,

$$B(y_1, \ldots, y_k) \to \textbf{if root} = A(f) \textbf{ then } \langle B, A\rangle(f)(y_1, \ldots, y_k)$$

is in R_2'.

It is an easy observation that the translation of $\mathfrak{M}_2'$ is still a total function. Hence, for every $\zeta \in T_{\Sigma'}$ and $B \in N_2$ of rank k, there is exactly one tree $t \in T_{\Delta'}(Y_k)$ such that $B(\zeta)(y_1, \ldots, y_k) \Rightarrow_{\mathfrak{M}_2'}^* t$. Again we denote this tree by $M(B(\zeta)(y_1, \ldots, y_k))$.

Finally, we define the DCFT(S)-transducer $\mathfrak{M}_3 = (N_3, e_1, \Delta, A^3_{\text{in}}, R_3)$ with $A^3_{\text{in}} = \langle A^2_{\text{in}}, A^1_{\text{in}}\rangle$ and R_3 constructed as follows. If $A \to$ **if** b **then** ζ is in R_1, then, for every $B \in N_2$ of rank k with $k \geqslant 0$, the rule

$$\langle B, A\rangle(y_1, \ldots, y_k) \to \textbf{if } b \textbf{ then } M(B(\zeta)(y_1, \ldots, y_k))$$

is in R_3.

Now, the following claim can be proved by an easy induction on the length of the derivation of $\mathfrak{M}_1$.

Claim 17. *For every $A \in N_1$, $c \in C$, $t \in T_\Sigma$, and $B \in N_2$ of rank $k \geqslant 0$, if $A(c) \Rightarrow^*_{\mathfrak{M}_1} t$, then*

$$\langle B, A\rangle(c)(y_1, \ldots, y_k) \Rightarrow^*_{\mathfrak{M}_3} M(B(t)(y_1, \ldots, y_k)).$$

Clearly, for $A = A^1_{\text{in}}$ and $B = A^2_{\text{in}}$, this claim induces that $\tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2) \subseteq \tau(\mathfrak{M}_3)$. Since $\tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2)$ is a total function on $\text{dom}(e_1)$, also $\tau(\mathfrak{M}_3)$ is a total function on $\text{dom}(e_1)$. Hence, $\tau(\mathfrak{M}_1) \circ \tau(\mathfrak{M}_2) = \tau(\mathfrak{M}_3)$. □

We need another lemma that shows that the condition of Lemma 3.21 on a $\text{D}_t\text{CFT}(S)$-transducer can be realized by look-ahead on S.

8.10. Lemma. $\text{D}_t\text{CFT}(S) \subseteq \text{D}_t\text{CFT}_1(S_{\text{LA}})$.

Proof. Let $\mathfrak{M}$ be a $\text{D}_t\text{CFT}(S)$-transducer. For every instruction f occurring in a rule of $\mathfrak{M}$, define the CF(S)-transducer $\mathfrak{H}(f)$ by $A_{\text{in}} \to A(f)$ and $A \to a$. Obviously, $\mathfrak{H}(f)$ translates a configuration c into a iff $m(f)$ is defined on c. We abbreviate the look-ahead test $\langle A_{\text{in}}, \mathfrak{H}(f)\rangle$ by $\langle f\rangle$. Then we can define the $\text{D}_t\text{CFT}(S_{\text{LA}})$-transducer $\mathfrak{H}'$, which is determined by the same components as $\mathfrak{H}$ except that, if $A(y_1, \ldots, y_n) \to$ **if** b **then** ζ is a rule of $\mathfrak{H}$ and $f_1, \ldots, f_k$ are all instructions occurring in ζ, then

$$A(y_1, \ldots, y_n) \to \textbf{if } b \textbf{ and } \langle f_1\rangle \textbf{ and } \ldots \textbf{ and } \langle f_k\rangle \textbf{ then } \zeta$$

is a rule of $\mathfrak{H}'$. Since, by definition, a rule is only applicable to a configuration c if the instructions of the right-hand side are defined on c, $\mathfrak{H}$ and $\mathfrak{H}'$ are equivalent. Now, $\mathfrak{H}'$ fulfils the requirement of Lemma 3.21. Hence, there is a $\text{D}_t\text{CFT}_1(S_{\text{LA}})$-transducer equivalent to $\mathfrak{H}'$. □

As an immediate consequence of the previous lemmata we obtain the invariant which connects the level of iteration of the pushdown operator in $\text{D}_t\text{RT}(\text{P}^n(S))$-transducers and the level of composition of total deterministic macro tree transducers.

8.11. Corollary. *If $S_{\text{LA}} \equiv S$, then* $\text{D}_t\text{RT}(\text{P}(S)) = \text{D}_t\text{RT}(S) \circ \text{D}_t\text{CFT}(\text{TR})$.

Proof. Assume that $S_{\text{LA}} \equiv S$. Hence, by the monotonicity of LA (cf. Lemma 6.6), $(S_{\text{LA}})_{\text{LA}} \equiv S_{\text{LA}}$ and, by Lemma 8.8, $(S_{\text{LA}})_{\text{mac-LA}} \equiv S_{\text{LA}} \equiv S$. Then

$$\begin{aligned}
D_tRT(P(S)) &= D_tCFT(S) && \text{(by Theorem 5.16)}\\
&\subseteq D_tCFT_1(S_{LA}) && \text{(by Lemma 8.10)}\\
&\subseteq \text{p-}D_tCFT_1((S_{LA})_{\text{mac-LA}}) && \text{(by Lemma 8.7)}\\
&\subseteq D_tRT((S_{LA})_{\text{mac-LA}}) \circ D_tCFT_1(TR) && \text{(by Lemma 8.5)}\\
&\subseteq D_tRT(S) \circ D_tCFT(TR)
\end{aligned}$$

(by the fact that $S_{\text{mac-LA}} \equiv S$ and the justification theorem)

$$\subseteq D_tCFT(S) \qquad \text{(by Lemma 8.9).} \quad \square$$

Finally, we can prove the desired characterization of the n-fold composition of total deterministic macro tree(-to-string) transducers by pushdown machines which use $P^n(TR)$ (and $P^{n+1}(TR)$, respectively) as storage. Note that

$$\text{yield}(D_tCFT(TR)^n) = D_tCFT(TR)^{n-1} \circ D_tMAC(TR).$$

The following theorem contains the deepest result of this paper.

8.12. Theorem. *For every* $n \geq 1$, $D_tCFT(TR)^n = D_tRT(P^n(TR))$ *and* $\text{yield}(D_tCFT(TR)^n) = D_tREG(P^{n+1}(TR))$.

Proof. First, we prove by induction on $n \geq 1$ that $P^n(TR)_{LA} \equiv P^n(TR)$.

($n = 1$): Immediate from Theorem 6.14 and Theorem 8.1.

($n \to n+1$): By I.H., $P^n(TR)_{LA} \equiv P^n(TR)$. Since P is monotonic (cf. Theorem 4.22), this induces $P(P^n(TR)_{LA}) \equiv P^{n+1}(TR)$. Then, by Theorem 6.14, it follows that $P^{n+1}(TR)_{LA} \equiv P^{n+1}(TR)$.

We now prove the theorem by induction on n. For $n = 1$ the results are stated in Theorem 8.2. The induction step is shown as follows: for every $n \geq 1$,

$$\begin{aligned}
D_tCFT(TR)^{n+1} &= D_tCFT(TR)^n \circ D_tCFT(TR)\\
&= D_tRT(P^n(TR)) \circ D_tCFT(TR) \quad \text{(by I.H.)}\\
&= D_tRT(P^{n+1}(TR)),
\end{aligned}$$

by Corollary 8.11 and the fact that $P^n(TR)_{LA} \equiv P^n(TR)$. Hence,

$$\begin{aligned}
\text{yield}(D_tCFT(TR)^n) &= \text{yield}(D_tRT(P^n(TR)))\\
&= D_tCF(P^n(TR)) \quad \text{(because yield}(RT(S)) = CF(S))\\
&= D_tREG(P^{n+1}(TR)),
\end{aligned}$$

by Theorem 6.7 and the fact that $P^n(TR)_{LA} \equiv P^n(TR)$. $\square$

We end this section with four remarks. Using the nondeterministic decomposition result of Corollary 3.27, it can easily be shown by induction that $RT(P^n_{\text{bex}}(TR)) \subseteq CFT(TR)^n$ and $RT(P^n(TR)) \subseteq CFT_{\text{ext}}(TR)^n$. However, we think that these inclusions are proper; we do not know a regular machine characterization of $CFT(TR)^n$.

Iterated pushdown automata (i.e., ranges of REG(P^n)-transducers, where P^n denotes $P^n(S_0)$) accept the high-level (OI) macro languages [9]. Thus one may expect a relationship between total deterministic high-level macro tree transducers (which can be viewed as a 'better' formal model of denotational semantics than macro tree transducers) and compositions of macro tree transducers (see [10] for a related result). This will be the subject of a next paper.

From the equivalence of $P^2(S)$ and $NS(S)$ (cf. Theorem 7.4) it can easily be shown that, for every $n \geq 1$, $P^{2n}(S) \equiv NS^n(S)$. Thus, we obtain from Theorem 8.12 that $D_tCFT(TR)^{2n} = D_tRT(NS^n(TR))$ and, in particular,

$$D_tCFT(TR) \circ D_tCFT(TR) = D_tRT(NS(TR)).$$

Let AG denote the class of tree translations realized by attribute grammars. Then, since for every $n \geq 0$, $AG^n \subseteq D_tCFT(TR)^n \subseteq AG^{n+1}$ [15], the composition of attribute grammars is characterized by iterated pushdown tree transducers; more precisely, $AG^* = \bigcup_{n \geq 0} D_tRT(P^n(TR))$.

References

[1] A.V. Aho, Nested stack automata, *J. ACM* **16** (1969) 383-406.

[2] A.V. Aho, Indexed grammars, an extension of context-free grammars, *J. ACM* **15** (1968) 647-671.

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

[4] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. 1, 2* (Prentice-Hall, Englewood Cliffs, NJ, 1973).

[5] L.M. Chirica and D.E. Martin, An order algebraic definition of Knuthian semantics, *Math. Systems Theory* **13** (1979) 1-27.

[6] K.L. Clark and D.F. Cowell, *Programs, Machines, and Computation* (McGraw-Hill, London, 1976).

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

[8] W. Damm, The IO- and OI-hierarchies, *Theoret. Comput. Sci.* **20** (1982) 95-206.

[9] W. Damm and A. Goerdt, An automata-theoretic characterization of the OI-hierarchy, *Proc. 9th ICALP*, Aarhus (1982) 141-153; also *Inform. and Control*, to appear.

[10] W. Damm and I. Guessarian, Implementation techniques for recursive tree transducers on higher-order data types, Rept. 83-16, Laboratoire Informatique Théorique et Programmation, Université de Paris VII, 1983.

[11] J. Engelfriet, Bottom-up and top-down tree transformations—a comparison, *Math. Systems Theory* **9** (1975) 198-231.

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

[13] J. Engelfriet, Two-way automata and checking automata, *Math. Centre Tracts* **108** (1979) 1-69.

[14] J. Engelfriet, Some open questions and recent results on tree transducers and tree languages; in: R.V. Book, ed. *Formal Language Theory, Perspective and Open Problems* (Academic Press, New York, 1980).

[15] J. Engelfriet, Tree transducers and syntax-directed semantics, TW-Memorandom Nr. 363, 1981, Twente Univ. of Technology; also: *Proc. 7th CAAP*, Lille (1982) 82-107.

[16] J. Engelfriet, Recursive automata, Unpublished notes, 1982.

[17] J. Engelfriet, Iterated pushdown automata and complexity classes, *Proc. 15th STOC*, Boston (1983) 365-373.

[18] J. Engelfriet and G. Filè, The formal power of one-visit attribute grammars, *Acta Inform.* **16** (1981) 275-302.

[19] J. Engelfriet, G. Rozenberg and G. Slutzki, Tree transducers, L-systems, and two-way machines, *J. Comput. System Sci.* **20** (1980) 150-202.
[20] J. Engelfriet and E.M. Schmidt, IO and OI, *J. Comput. System Sci.* **15** (1977) 328-353, and *J. Comput. System Sci.* **16** (1978) 67-99.
[21] J. Engelfriet and G. Slutzki, Extended macro grammars and stack controlled machines, *J. Comput. System Sci.* **29** (1984) 366-408.
[22] J. Engelfriet and H. Vogler, Macro tree transducers, *J. Comput. System Sci.* **31** (1985) 71-146.
[23] G. Filè, The characterization of some families of languages by classes of indexed grammars, M.Sc. Thesis, Pennsylvania State University, U.S.A., June 1977.
[24] M.J. Fischer, Grammars with macro-like productions, Ph.D. Thesis, Harvard University, U.S.A., 1968.
[25] P. Franchi-Zannettacci, Attributs sémantiques et schemas de programmes, Thèse d'Etat, Univ. de Bordeaux, France, 1982.
[26] F. Gecseg and M. Steinby, *Tree Automata* (Akademiai Kiado, Budapest, 1984).
[27] M.J.C. Gordon, *The Denotational Description of Programming Languages, an Introduction* (Springer, New York, 1979).
[28] S.A. Greibach, Full AFLs and nested iterated substitution, *Inform. and Control* **16** (1970) 7-35.
[29] I. Guessarian, Pushdown tree automata, *Math. Systems Theory* **16** (1983) 237-263.
[30] C.A.R. Hoare, Proof of correctness of data representations, *Acta Inform.* **1** (1972) 271-281.
[31] J.E. Hopcroft and J.D. Ullman, *Formal Languages and their Relation to Automata* (Addison-Wesley, Reading, MA, 1969).
[32] T. Kamimura, Tree automata and attribute grammars, *Inform. and Control* **57** (1983) 1-20.
[33] D.E. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145; correction in *Math. Systems Theory* **5** (1971) 95-96.
[34] B. Mayoh, Attribute grammars and mathematical semantics, *SIAM J. Comput.* **10** (1981) 503-518.
[35] A.N. Maslov, Multi-level stack automata, *Problems Inform. Transmission* **12** (1976) 38-43.
[36] R. Parchmann, J. Duske and J. Specht, On deterministic indexed languages, *Inform. and Control* **45** (1980) 48-67.
[37] W.C. Rounds, Mappings and grammars on trees, *Math. Systems Theory* **4** (1970) 257-287.
[38] D. Scott, Some definitional suggestions for automata theory, *J. Comput. System Sci.* **1** (1967) 187-212.
[39] J.W. Thatcher, Generalized^2 sequential machine maps, *J. Comput. System Sci.* **4** (1970) 339-367.
[40] J. van Leeuwen, Notes on pre-set pushdown automata, in: G. Rozenberg and A. Salomaa, eds., *L-Systems*, Lecture Notes in Computer Science **15** (Springer, Berlin, 1974) 177-188.
[41] D.S. Scott and C. Strachey, Toward a mathematical semantics for computer languages, in: J. Fox, ed., *Proc. Symp. on Computers and Automata* (Polytechnic Institute of Brooklyn Press, New York, 1971).
[42] G. Huet, Confluent reductions: Abstract properties and applications to term-rewriting systems, *J. ACM* **27** (1980) 797-821.