Macro Tree Transducers in Treebag

Karl Azab

April 5, 2005

Abstract

TREEBAG is a software for experimenting with formal models in Computer Science, one type of such models being tree transducers. Tree transducers translate input trees to output trees, a mechanism used in e.g. compilers and picture generation. The top-down tree transducer is one of the formal models implemented as a component in TREEBAG. Compared to the top-down tree transducer, the macro tree transducer provides a more powerful model of computation. This thesis describes how TREEBAG is extended with a macro tree transducer component.

Contents

Chapter 1

Introduction

A *formal language* is a set of strings. Let L denote a formal language, then each *string* s in the language L is created from characters in an alphabet A [HMU01]. E.g. let A_{ex1} be the set $\{+, *, (,)\} \cup \mathbb{N}^0$ and let L_{ex1} contain all the strings created from A_{ex1} representing arithmetic expressions. Some of the strings in L_{ex1} are shown in Figure 1.1. Additionally, let the operators $*, +$ have the standard semantics (meaning) and precedence. If $\mathcal{S}$ denotes a *semantics evaluator* for strings then $\mathcal{S}(s)$ would denote the value of string s, e.g. $\mathcal{S}(1 + 1 * 1 + 1) = 3$.

$$\begin{array}{ccc} 3 + 1 * 2 & 1 & (12 + 1) * (5 + 11) \\ (3 + 22) * 5 & 1 * 1 & 12 + 1 * 5 + 11 \end{array}$$

Figure 1.1: Six examples of arithmetic expressions created from the characters $\{+, *, (,)\} \cup \mathbb{N}^0$

Now, let the language L_{ex2} contain all arithmetic expressions created from the alphabet $A_{ex2} = \{+\} \cup \mathbb{N}^0$, i.e. all arithmetic expressions denoting additions of natural numbers. E.g. $3 + 2 + 15$ is in L_{ex2}.

Each natural number $n \in \mathbb{N}^0$ is denoted by at least one string in L_{ex1} and one in L_{ex2}[1]. I.e. both languages contain expressions over the same range of natural numbers. Hence it should also be possible to map each element in L_{ex1} to an equivalent element in L_{ex2}. Formally, for each string $s \in L_{ex1}$ there exists a string $f(s) = s'$ where $s' \in L_{ex2}$ and the semantics of both strings are

[1]Since n is an arithmetic expression by itself and exists in both A_{ex1} and A_{ex2}.

equal, $\mathcal{S}(s) = \mathcal{S}(s')$. This mapping could, in our example languages, result in e.g. $f_{ex1,2}(3 * 2 + 1) = 2 + 2 + 2 + 1$.

More generally the mapping described above can be called a *translation* from a *source language* to a *target language*. The process of the translation can be divided into four consecutive steps: *lexical analysis*, *syntactic analysis*, *syntax-directed semantics translation* and *interpretation*; as shown in Figure 1.2.

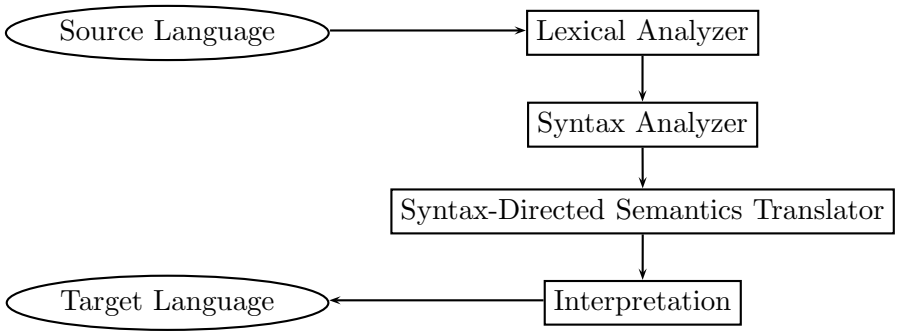
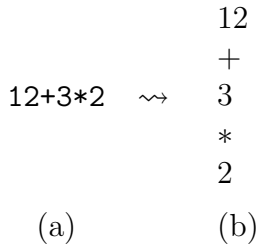


Figure 1.2: Major steps in a simplified language translation process.

The *lexical analyzer* receives a string in the source language and constructs *tokens* of the input. A token is a, longest possible, sequence of characters in the input stream which together constitute a string in the source language. Figure 1.3 shows a data stream being sequenced into tokens. [ASU86]



(a) (b)

Figure 1.3: An example of a character stream, (a), going through a lexical analyzer and becoming a sequence of tokens, (b).

Tokens from the lexical analyzer act as input for the *syntactic analyzer*. The syntactic analyzer arranges the tokens into a *parse tree*. The parse tree

is constructed according to the syntactic rules of the input language, e.g. operator precedence in arithmetic expressions. Figure 1.4 shows an example of how the token sequence from Figure 1.3 is turned into a parse tree. A program creating a parse tree is commonly referred to as a *parser*.

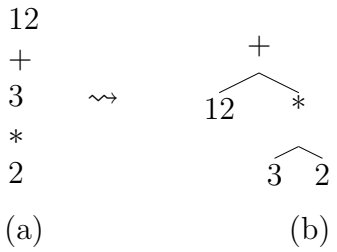


Figure 1.4: Token sequence (a) constructed as parse tree (b) by a syntactic analyzer.

The *syntax-directed semantics translator*, for short *sds translator*, is the most complex part of the translator described in Figure 1.2. A syntax-directed semantics translation assumes that the input language has a syntax-directed semantics, i.e. that the semantics (meaning) of the language depends upon its syntactic structure. The sds evaluator transforms its input, the parse tree, into an output tree which represents a string in the target language. Furthermore, the syntax-directed semantics of the input tree and the output tree equal each other. I.e. the sds translator must perform a tree translation where the input and the generated output tree have the same meaning. See Figure 1.5 for an example of how the parse tree in Figure 1.4 is translated into an output tree representing the same semantics as the input but using only characters from the alphabet of the target language, A_{ex2}.

Finally an *algebra* will perform an interpretation of the output tree. To generate the output, the algebra will perform an operation at each symbol in the tree, eventually yielding a result string in the target language. E.g. Figure 1.6 shows the output of an algebra performing string concatenation at all symbols in a tree. Another interpretation, calculating the result of the tree by using numbers as data values and + as an operator, is shown in Figure 1.7.

In this text, the first two steps of the translation process, lexical and syntactic analysis are not covered further. An introduction to *finite automata* and *context-free grammars* and how they can model lexical analyzers and

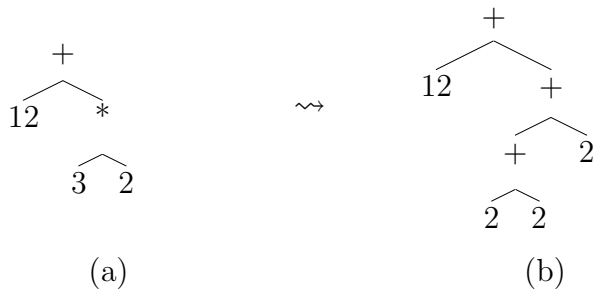


Figure 1.5: Example of an sds translator transforming a parse tree (a) into an output tree (b).

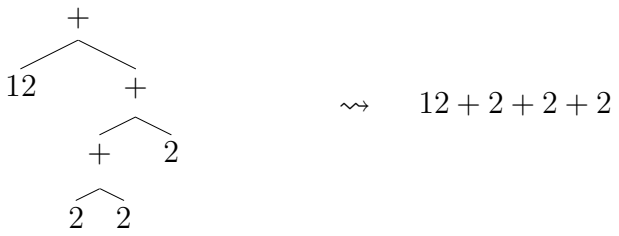


Figure 1.6: Example of an algebra outputting a string corresponding to the expression denoted by the syntactic structure of the tree.

syntactic analyzers, respectively, can be found in [HMU01] and [ASU86].

The focus of this thesis is inside the area of syntax-directed semantics. In the theory of syntax-directed semantics, *formal models of syntax-directed semantics* are used. A formal model of syntax-directed semantics is the combination of a *specification language* and a *computation paradigm*. The specification language describes the syntax-directed semantics of the input, i.e. how the syntactic structure of the input describes the semantics of the input. The computation paradigm transforms input trees to a output trees according to the specification language. [FV98]

One type of formal models of syntax-directed semantics are called *tree transducers*. *Top-down tree transducers* and *macro tree transducers* are two types of tree transducers. Chapter 2 describes these two tree transducers and how their translation capabilities differ.

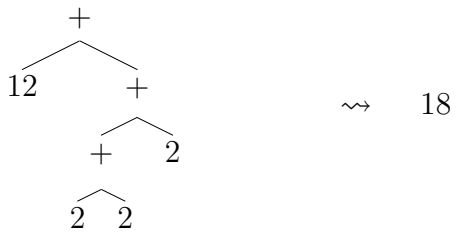


Figure 1.7: Shows an algebra performing addition and outputting the numeric result.

Translations like the one described above, in Figure 1.2, can be used for a number of applications. Two of them are compilers[2] and *picture generation*. In picture generation, the target language is interpreted as geometric objects by algebras and displayed as pictures. TREEBAG is a system which can be used for picture generation. TREEBAG is explained in Section 1.2.

1.1 Tree Terminology

To understand the formal definitions of the tree transducers in Chapter 2, some basic terminology needs to be explained.

A *symbol* consists of a name f and rank n, where n is any integer zero or larger (formally $n \in \mathbb{N}^0$). $f^{(n)}$ denotes a symbol uniquely identified by the combination of its name and rank.

A *ranked alphabet* Σ is a set of symbols. E.g. $\Sigma_{ex1} = \{\ a^{(0)}, b^{(1)}, c^{(1)}, f^{(3)}, g^{(2)}\}$ denotes the ranked alphabet Σ_{ex1} consisting of the symbols named a, b, c, f and g with the ranks 0, 1, 1, 3 and 2 respectively. $\Sigma^{(i)}$, where $i \in \mathbb{N}^0$, denotes the subset of Σ containing all symbols of rank i. In our example, $\Sigma^{(1)}_{ex1} = \{b^{(1)}, c^{(1)}\}$. A *finite ranked alphabet* is a ranked alphabet containing a finite number of symbols.

Trees are constructed of symbols from a ranked alphabet. A tree consists of a top symbol, called *root*, and n *subtrees*, where n equals the rank of the root. In turn, each subtree is also a tree, i.e. they have their own root symbol

[2]This introduction has attempted to explain the language translation process from a very simplified point of view. A real world compiler would however have to consider several more aspects, such as error handling and printing useful error messages, code optimizations and register allocation [ASU86].

with subtrees. Since one symbol can exist at more than one position in the same tree, it might be impossible to use only symbols to distinguish unique positions within the tree. Each unique position in a tree is therefore referred to as a *node*. One way to identify and label nodes is to describe the path from the root to the node in question [Dre05]. The *children* of a node n are the roots of its subtrees, whose *parent* is of course n. A (sub)tree with a root symbol of rank zero has no subtrees or children, hence it contains only a single node which is referred to as a *leaf*.

The definition of a *term* is equivalent to that of a tree, though their graphical representations differ. One may say that a term is the linear, i.e. textual, representation of a tree. An example of a term and its corresponding tree, together with a demonstration of some tree terminology, is given in Figure 1.8.

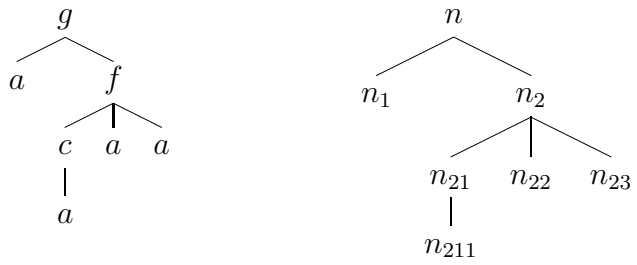


Figure 1.8: (a) Tree representation of the term $g[a, f[c[a], a, a]]$. The root, g, has two subtrees, the first is a and the second is $f[c[a], a, a]$. The children of g are a and f. The four occurances of a are leaves. (b) Shows the node labeling of the positions in tree (a). E.g. n_{23} denotes the node encountered after walking the path from n to n_2 to n_{23}.

T_Σ denotes the set of all possible trees/terms over[3] Σ. More formally, the term t is in T_Σ if and only if:

i. t consists solely of the symbol σ and $\sigma \in \Sigma^{(0)}$ or

ii. t is of the form $\sigma[t_1, \ldots, t_k]$ where $\sigma \in \Sigma^{(k)}$, $k \geq 1$ and $t_1, \ldots, t_k \in T_\Sigma$.

[3] "Over" meaning "constructed of symbols from"

A subset, possibly infinite, of T_Σ is called a *tree language*.

The smallest term in $T_{\Sigma_{ex1}}$ would be the single symbol a of rank zero, where a is both root and leaf. On the other hand, it is also possible to construct an arbitrarily large term over Σ_{ex1} which would still be in $T_{\Sigma_{ex1}}$. E.g. consider a term with root symbol b. It has one child, another b. This child in turn has another b as child, this would be the grandchild of the first b. This recursion may continue indefinitely, creating a very large term of the form $b[b[b[\ldots]]]$. The recursion is stopped when a symbol of rank zero is used as child instead of b.

The height of a tree, height(t), is the number of nodes encountered on the longest path from the root to a leaf. It is recursively defined as

i. 1 if $t = \sigma$ where $\sigma \in \Sigma^{(0)}$.

ii. $1+\max\{\text{height}(t_1), \ldots, \text{height}(t_k)\}$ if $t = \sigma[t_1, \ldots, t_k]$ where $t_1, \ldots, t_k \in T_\Sigma$, $k \geq 1$ and $\sigma \in \Sigma^{(k)}$.

Furthermore, the yield of a tree, yield(t), is the string created by concatenating the symbol names of each leaf in t from left to right. Formally, the string is defined as

i. f if $t = f^{(0)}$ where $f^{(0)} \in \Sigma^{(0)}$.

ii. $\text{yield}(t_1)\ldots\text{yield}(t_k)$ if $t = \sigma[t_1, \ldots, t_k]$ where $t_1, \ldots, t_k \in T_\Sigma$, $k \geq 1$ and $\sigma \in \Sigma^{(k)}$.

E.g. the height and yield of the tree in Figure 1.10 (b) is 4 and *aabbabb*, respectively.

Trees and terms which are solely composed of symbols of rank zero or one are *monadic*. I.e. each node in a monadic tree has only one child, except for the single leaf, which has none. E.g. $b[b[b[a]]]$ is a monadic term.

Another useful notation is $T_\Sigma(S)$, where S is a set of trees. It can be seen as an extension of T_Σ. $T_\Sigma(S)$ denotes all trees t over Σ where t may also have subtrees from S. E.g. if $S_{ex1} = \{i, k[i], j[i, k[i])\}$ then three examples of trees in $T_{\Sigma_{ex1}}(S_{ex1})$ are shown in Figure 1.9 on the following page.

Let X be a ranked alphabet containing *variables*. A variable is always of rank zero and is never used as an ordinary symbol, i.e. $X \cap \Sigma = \emptyset$. In this text the symbols x_i, where $i \in \mathbb{N}^+$, are used as variables, and therefore $X = \{x_1^{(0)}, x_2^{(0)}, \ldots\}$. $X_{[n]}$ denotes the finite set $\{x_1^{(0)}, \ldots, x_n^{(0)}\}$. Furthermore,

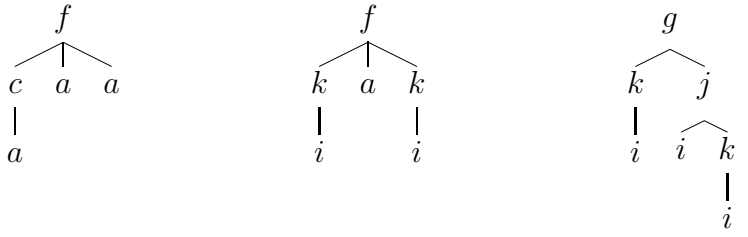


Figure 1.9: Three examples of trees in the set $T_{\Sigma_{ex1}}(S_{ex1})$, where Σ_{ex1} and S_{ex1} are defined as in the text.

let the set of *parameters*, defined similarly as variables, be denoted by $Y = \{y_1^{(0)}, y_2^{(0)}, \ldots\}$ and where $Y_{[n]} = \{y_1^{(0)}, \ldots, y_n^{(0)}\}$.

A *substitution* replaces the variables and parameters in a tree with new subtrees. If $t \in T_\Sigma(X_{[m]} \cup Y_{[n]})$ and $s_1, \ldots, s_{n+m} \in T_\Sigma$ then the substitution of every $x_i \in X_{[m]}$ in t with s_i and every $y_i \in Y_{[n]}$ in t with s_{m+i} is written as

$$t[\![s_1/x_1, \ldots, s_m/x_m, s_{m+1}/y_1, \ldots, s_{m+n}/y_n]\!].$$

Formally, this substitution is inductively defined as

i. s_i if $t = z$, where s_i/z is the substitution and $z \in X_{[m]} \cup Y_{[n]}$.

ii. $\sigma[t'_1, \ldots, t'_k]$ if $t = \sigma[t_1, \ldots, t_k]$ for some $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T_\Sigma(X_{[m]} \cup Y_{[n]})$, where $t'_i = t_i[\![s_1/x_1, \ldots, s_m/x_m, s_{m+1}/y_1, \ldots, s_{m+n}/y_n]\!]$ for all $i \in \{1, \ldots, k\}$.

If only variables are used in t, i.e. $t \in T_\Sigma(X_{[m]})$, there is no need to explicitly specify which variable is to be substituted by which subtree, instead this information is given implicitly by the order the subtrees are listed by. The notation $t[\![s_1, \ldots, s_n]\!]$ is therefore used as an abbreviation for $t[\![s_1/x_1, \ldots, s_n/x_n]\!]$. Figure 1.10 shows a substitution example.

1.2 TREEBAG

TREEBAG (Tree Based Generator) is a software for experimenting with theoretical models from formal language theory, such as tree grammars, tree transducers and algebras. TREEBAG is described in [Dre98] and is written in Java.

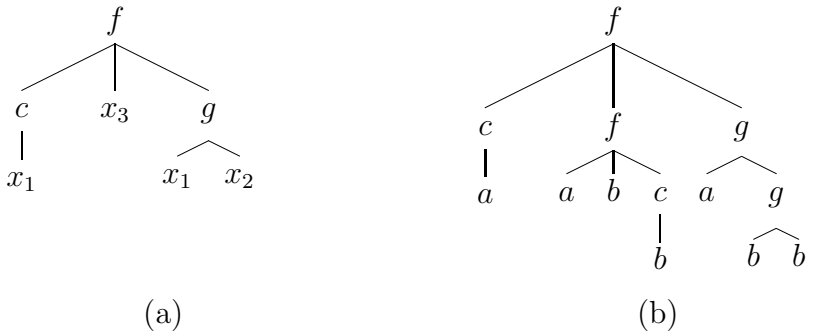


Figure 1.10: (a) shows the example tree $t \in T_\Sigma(X)$ and (b) shows the result of the substitution $t[\![a, g[b, b], f[a, b, c[b]]]\!]$.

The set up of an experiment consists of linking one or more TREEBAG-*nodes* on a worksheet. The TREEBAG-nodes are linked with *edges*. An edge can exist between two nodes, a source node and a destination node, and provides a unidirectional data stream between them. The output of the source node becomes the input of the destination node.

Each TREEBAG-node is an *instance* of a *component*. A *component* is a Java-class representing a particular formal model. There exists components for e.g. tree grammars and top-down tree transducers. An instance of a component is created with an *object file* which specifies how the component should behave. The syntax in the object files are deliberately chosen to be similar to the formal definition of the respective component. This results in an easy conversion from examples found in articles to a working object file in TREEBAG. In the case of a tree transducer, the object file and the component would act as specification language and computation paradigm respectively.

There exists four types of components in TREEBAG:

Generators construct trees according to grammars. Their output are the generated trees. E.g. regular tree grammars are implemented as a generator component in TREEBAG.

Tree transducers requires an input tree, which is translated into an output tree. Typically, the output from a generator or another tree transducer serves as input for a tree transducer component. E.g. the top-down tree transducer is implemented as a tree transducer component.

Algebras which usually interpret trees as something other than trees. Algebras take input from generators and tree transducers. E.g. the *collage-algebra* is an algebra component where the nodes of the input tree are interpreted as operations constructing geometric objects. The output of the *free-term algebra* is identical to its input, i.e. it performs the identity interpretation.

Displays show the output of an algebra component and provide user controls for e.g. zooming. Examples of display components in TREEBAG are the *tree display* which displays trees and the *collage display* which draws two dimensional geometric objects.

An experiment in TREEBAG is shown in Figure 1.11.

1.3 Purpose of Thesis

The purpose of this thesis is to describe an implementation of the macro tree transducer as a component in the TREEBAG-system. Since an implementation of the top-down tree transducer already exists in TREEBAG, the thesis begins by describing both tree transducers in Chapter 2. Chapter 2 also motivates the the need for a macro tree transducer implementation. The implementation is described in Chapter 3 and shown with examples in Chapter 4.

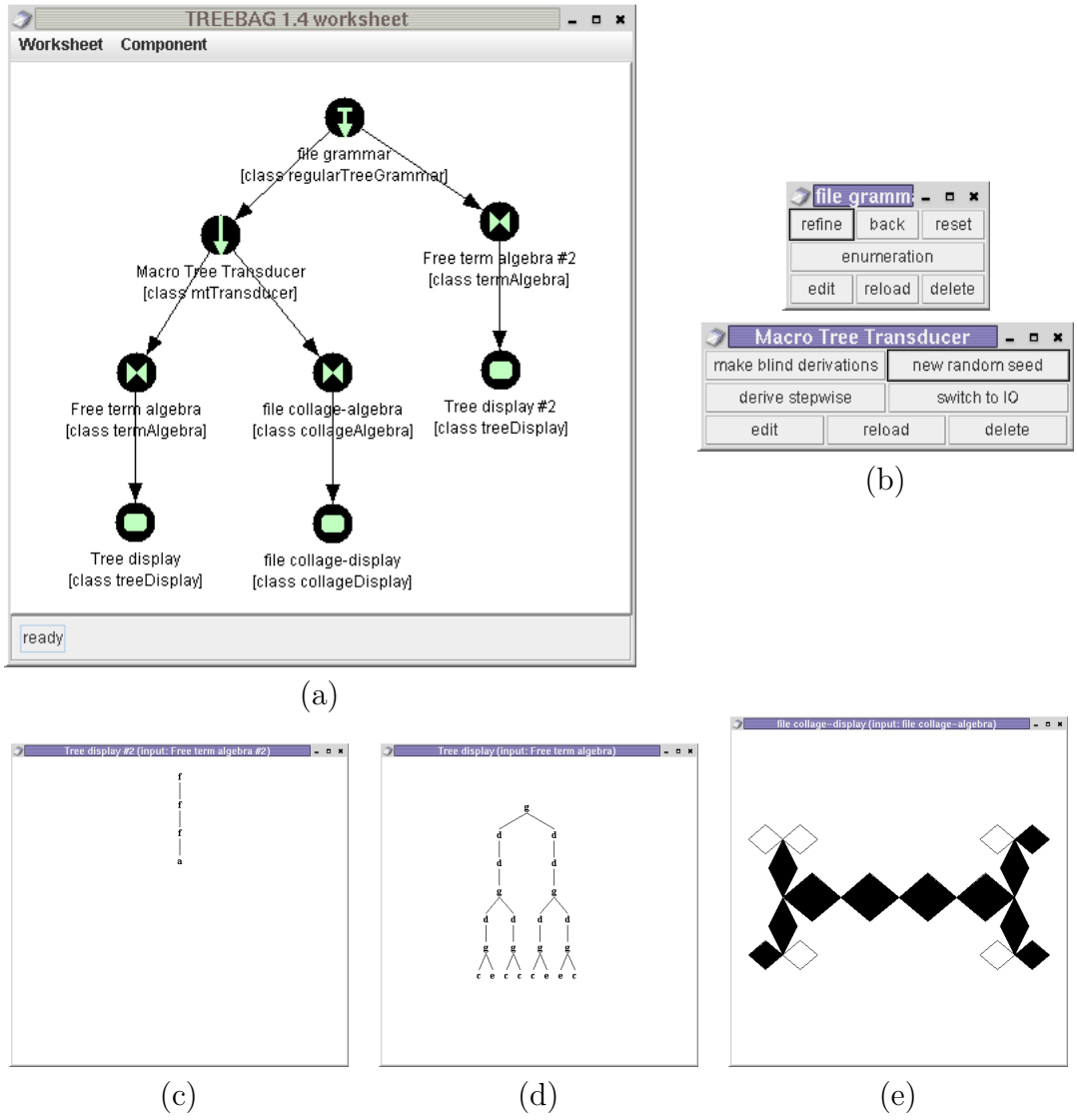


Figure 1.11: Example of a TREEBAG experiment. (a) shows the worksheet with nodes and edges. (b) user interfaces for the tree grammar node (upper window) and the macro tree transducer node (lower window). (c), the output of the tree grammar, is given as input for the macro tree transducer which, in turn, outputs (d). The collage algebra interprets the output of the macro tree transducer, shown in image (e).

Chapter 2

Tree Transducers

As mentioned in Chapter 1, tree transducers are formal models of syntax-directed semantics. Intuitively, a tree transducer works as a tree translator – an input tree over the ranked alphabet Σ is transformed by the tree transducer into an output tree over the ranked alphabet Σ'. Mathematically, a tree transducer can be described as a function $f(t) = T$ where $t \in T_\Sigma$ and T is a set of trees such that $T \subseteq T_{\Sigma'}$. In a *deterministic* tree transducer, the cardinality of T is either zero or one for all trees $t \in T_\Sigma$. A tree transducer is *non-deterministic* if T contains more than one element for any $t \in T_\Sigma$.

Although the purpose of this thesis is to describe the macro tree transducer implementation, this section also describes the theory behind the top-down tree transducer implementation, in order to:

- later in the chapter make it easier to understand the more complex macro tree transducer and

- in more detail point out the limitations of the top-down tree transducer and why a macro tree transducer implementation is needed.

The top-down tree transducer is described in Section 2.1. Section 2.2 describes some limitations of top-down tree transducers and how they can be improved. The macro tree transducer is defined and described in Section 2.3. Section 2.3 also shows how the macro tree transducer overcomes the limitations of the top-down tree transducer.

2.1 Top-Down Tree Transducers

As the name suggests, a top-down tree transducer transforms input trees to output trees in a top-down fashion, starting with the root and working down to the leaves. This tree transducer was introduced in [Rou70] and [Tha70].

The specification for a top-down tree transducer, *td transducer* for short, consists of four finite sets and a symbol. Formally, a top-down tree transducer named T is specified as follows:

$$\mathrm{T} = (\Sigma, \Sigma', Q, R, q_0)$$

where

- Σ and Σ' are the finite ranked alphabets for input and output trees respectively. The only valid input to the td transducer T are trees in T_Σ and T will only generate output trees in $T_{\Sigma'}$.

- Q is the set of states. All elements in Q are symbols of rank one and may neither occur in input nor output trees, hence $Q \cap (\Sigma \cup \Sigma') = \emptyset$ must always hold. States are special symbols which occur only in *sentential trees*. A sentential tree t is a modified input tree which has not yet become a finished output tree. Sentential trees are never used outside the tree transducer and the following holds for all sentential trees t: $t \in T_{\Sigma \cup Q \cup \Sigma'}$, $t \notin T_\Sigma$ and $t \notin T_{\Sigma'}$.

- R denotes the set of rules. Rules specify how trees can be transformed. Rules for td transducers are explained in detail under Section 2.1.1.

- q_0 is the initial state and indicates in which state the translation process shall begin. In td transducers, the initial state is therefore appended above the root. E.g. if $g[a, b[a]]$ is the input tree, then the initial state, q_0, is appended above g so that $q_0[g[a, b[a]]]$ is the tree to be processed.

2.1.1 Rules and Derivation Steps

Let T be a td transducer $(\Sigma, \Sigma', Q, R, q_0)$. Then the rule set, R, contains a finite number of rules. Each rule consists of two trees, referred to as the *left hand side*, *lhs* for short, and *right hand side*, *rhs* for short. Rules are of the form:

$$\text{lhs} \rightarrow \text{rhs}.$$

The left hand side contains information on what kind of subtree the rule can be applied at. The root of the lhs will always be a state, hence rules can only be applied at subtrees rooted at states. Since all states in a td transducer are of rank one, the root will have only one child: an input symbol.

A right hand side describes what the lhs is replaced with after the rule has been applied. The structure of the lhs and rhs of rules in R must follow a general form, formally written as

$$q[\sigma[x_1, \ldots, x_m]] \rightarrow t'[\![q_1[x_{i_1}], \ldots, q_n[x_{i_n}]]\!] \tag{2.1}$$

where $q, q_1, \ldots, q_n \in Q$, $\sigma \in \Sigma^{(m)}$, $t' \in T_{\Sigma'}(X_{[n]})$ and $i_1, \ldots, i_n \in \{1, ..., m\}$.

Let $t \in T_\Sigma$ be the input tree to the top-down tree transducer T. To process an input tree t into an output tree t', a sequence of rules are applied, i.e. a sequence of *derivation steps* are performed.

[Dre05] defines a derivation step as a transformation from tree s into tree s'. The transformation is denoted by $s \Rightarrow s'$ and can be performed if and only if there exists a rule $r \in R$, defined identically as in Equation 2.1, and

$$s = s_0[\![q[\sigma[t_1, \ldots, t_m]]]\!] \text{ and } s' = s_0[\![\sigma'[q_1[t_{i_1}], \ldots, q_n[t_{i_n}]]]\!]$$

where $s_0 \in T_{\Sigma \cup Q \cup \Sigma'}(X_1)$, $t_1, \ldots, t_m \in T_\Sigma$ and all other definitions are identical to those in Equation 2.1.

An example of how a derivation step transforms a tree is given in Figure 2.1. Another result of a derivation step is that the height of the input is *decreased* by one while the height of the output may be *increased* by some constant k. This result tells us that the translation process will eventually run out of input symbols, height zero, and therefore terminate.

To state some final terminology, $t_0 \Rightarrow^n t_n$ denotes a sequence of n consecutive derivation steps and $t \Rightarrow^* s$ denotes an unspecified number of consecutive derivation steps. A (q, σ)-*rule* is a rule which left hand side has $q \in Q$ as root and $\sigma \in \Sigma$ as its child.

2.1.2 A Top-Down Tree Transducer Example

This example is based on an example from [Rou70] on how td transducers can be used for symbolic differentiation.

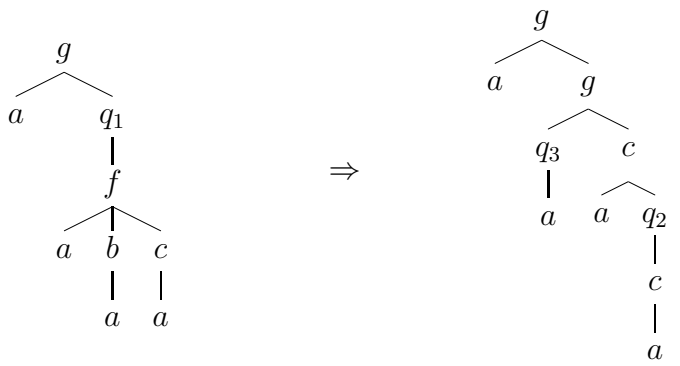


Figure 2.1: Let the td transducer T have Σ_{ex1}, defined on page 5, as both input and output alphabet and let the set of states $Q = \{q_1, q_2, q_3\}$. Then this figure shows how the rule $q_1[f[x1, x2, x3]] \to g[q_3[x_1], c[a, q_2[x_3]]]$ is applied at a tree.

Let the td transducer $\mathrm{T}_{ex1} = (\Sigma, \Sigma', Q, R, dx)$ where $\Sigma = \{\ \cdot^{(2)},\ +^{(2)},\ \sin^{(1)},\ \cos^{(1)},\ p^{(0)}\}$, $\Sigma' = \{\ \cdot^{(2)},\ +^{(2)},\ \sin^{(1)},\ \cos^{(1)},\ p^{(0)},\ p'^{(0)},\ -^{(1)}\}$, $Q = \{\ dx,\ i\}$ and R contains the rules

$$
\begin{array}{rcl}
dx[+[x_1, x_2]] & \to & +[dx[x_1], dx[x_2]] \\
dx[\cdot[x_1, x_2]] & \to & +[\cdot[dx[x_1], i[x_2]], \cdot[i[x_1], dx[x_2]]], \\
dx[\sin[x_1]] & \to & \cdot[dx[x_1], \cos[i[x_1]]], \\
dx[\cos[x_1]] & \to & \cdot[dx[x_1], -[\sin[i[x_1]]]], \\
dx[p] & \to & p', \\
i[+[x_1, x_2]] & \to & +[i[x_1], i[x_2]], \\
i[\cdot[x_1, x_2]] & \to & \cdot[i[x_1], i[x_2]], \\
i[\sin[x_1]] & \to & \sin[i[x_1]], \\
i[\cos[x_1]] & \to & \cos[i[x_1]], \\
i[p] & \to & p
\end{array}
$$

where p denotes a polynomial of the variable x. Then derivation steps to differentiate the input tree $\cdot[\cdot[p, p], \sin[p]]$ is shown in Figure 2.2.

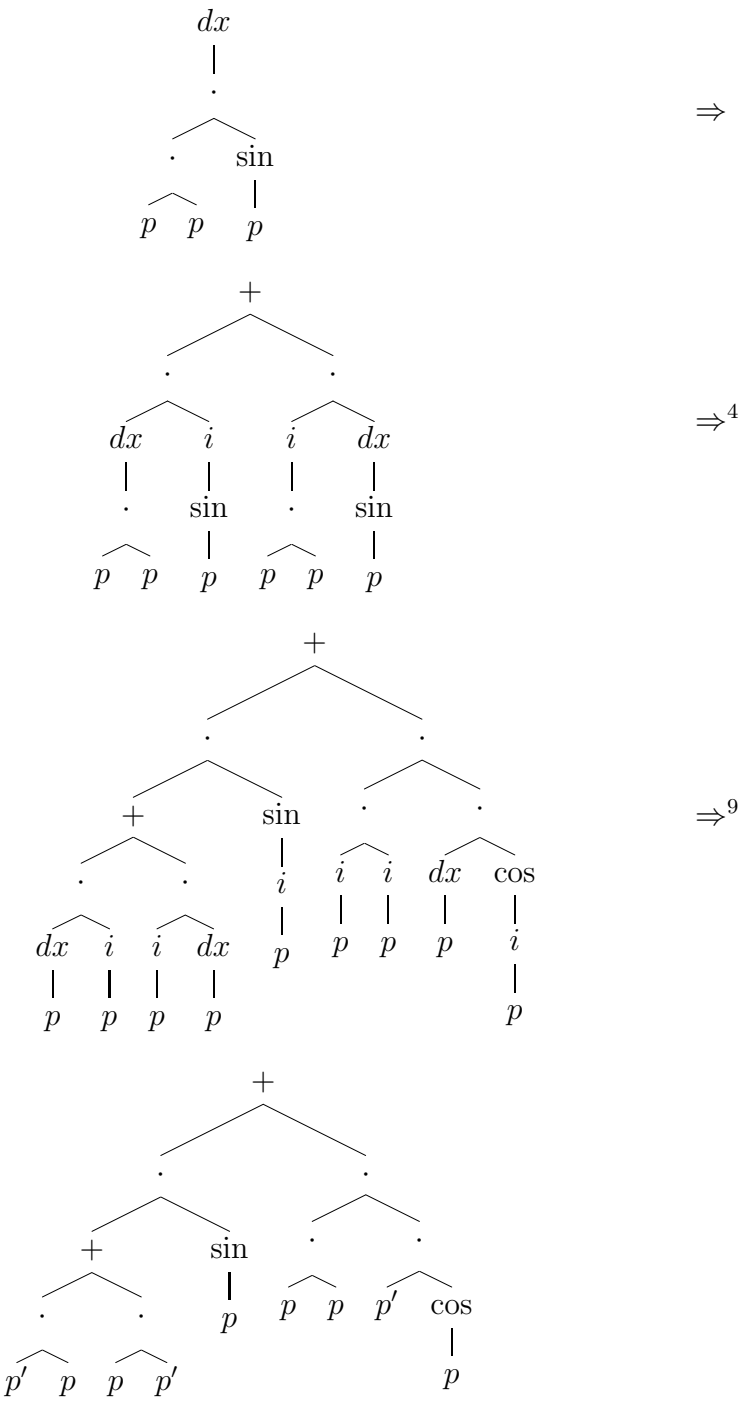


Figure 2.2: The top-down tree transducer T_{ex1} performing symbolic differentiation of the expression $dx(p^2 \cdot \sin p) = 2 \cdot p' \cdot p \cdot \sin p + p' \cdot p^2 \cdot \cos p$. Note that the derivation steps are made in parallel.

2.2 Compositions with Top-Down Tree Transducers

In a complex translation, it is possible to divide the translation into smaller parts. This might be desirable for abstraction purposes, making it easier to e.g. test separate parts of the translation. Each part is translated by a separate tree transducer. When e.g. the testing, is complete, the abstraction is no longer necessary. For some classes of translations, it is possible to, mechanically, *compose* the translation performed by several tree transducers into a single tree transducer, still performing the same translation. The result, a single, complex, tree transducer instead of several smaller tree transducers might be desirable with regard to memory and computing resources[1]. [FV98]

When processing a tree sequentially by two td transducers, it is sometimes possible to compose them into a single td transducer. However, not all td transducers are composable, details on this subject are not covered here, but conditions for when td transducer composition is possible[2] and an algorithm for composing them is given in [Dre05].

2.2.1 Bottom-Up Tree Automata

The *bottom-up tree automaton*, *bu automaton* for short, processes an input tree starting with the leaves and up to the root and outputs a single symbol. A bu automaton B is a quadruple

$$\mathrm{B} = (\Sigma, Q, Q_a, R).$$

Σ, a finite raked alphabet, is the input alphabet and all input trees to B must be in T_Σ. All states of the bu automaton are of rank zero and are in the set Q. Q_a, a subset of Q, is the set of accepting states. The set R contains rules of the form

$$\sigma[q_1, \ldots, q_k] \to q \tag{2.2}$$

where $\sigma \in \Sigma^{(k)}$ and $q, q_1, \ldots, q_k \in Q$. Derivation steps are made in a similar fashion as in tree transducers. The derivation step $t \Rightarrow s$ may be performed if $t = s_0[\![\sigma[q_1, \ldots, q_k]]\!]$ and $s = s_0[\![q]\!]$ where $s_0 \in T_\Sigma$ and $\sigma, q, q_1, \ldots, q_k$ are

[1] Since e.g. no intermediate trees need to be created in the chain of tree transducers.

[2] Possible in the sense that the resulting composition will also be a td transducer.

defined as in Equation 2.2. An input tree t is *accepted* by B if $t \Rightarrow^* q$ for some $q \in Q_a$. Otherwise, t is *rejected*. Thus the bu automaton generalizes the finite automaton to trees. As a more informal description, the reader may familiarize her-/himself with the bu automata by following the example below.

Let $\mathrm{B}_{ex1} = (\{f^{(2)}, a^{(0)}, b^{(0)}, c^{(0)}\}, \{q_y^{(0)}, q_n^{(0)}\}, \{q_y^{(0)}\}, R)$ where R contains the rules

$$\begin{aligned} a &\rightarrow q_y \\ b &\rightarrow q_n \\ c &\rightarrow q_n \\ f[q_n, q_n] &\rightarrow q_n \\ f[q_n, q_y] &\rightarrow q_n \\ f[q_y, q_n] &\rightarrow q_n \\ f[q_y, q_y] &\rightarrow q_y. \end{aligned}$$

Then Figure 2.3 shows how an example input tree is processed by B_{ex1}. Obviously, B_{ex1} accepts exactly $T_{\{f^{(2)}, a^{(0)}\}}$.

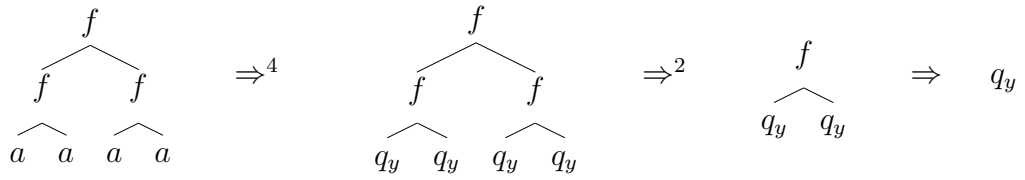


Figure 2.3: B_{ex1} processes the input tree $f[f[a, a], f[a, a]]$

In general, the tree language which a bu automaton B defines is the set of trees $T \subseteq T_\Sigma$ accepted by B. A *regular tree language* L is a tree language where there exists a bu automaton, or an equivalent construction, which accepts precisely L.

More detailed definitions together with instructions on how the bu automata can be extended to a bottom-up tree transducer[3] can found in [Tha73, CDG+02].

[3] A bottom-up tree transducer works similarly as a bottom-up tree automaton but outputs a tree instead of just accepting or rejecting the input.

2.2.2 Regular Look-Ahead

Imagine the translation outputting

- The single symbol a, if the input tree contains only leaves with the symbol a.
- Nothing (undefined result), if the input tree contains a leaf without the symbol a, e.g. b or c.

Assume we want to implement this transformation by means of a td transducer and let this td transducer inspect the tree $f[f[a,a],f[a,a]]$ to determine whether or not, according to the translation above, to generate the symbol a. A state, e.g. q would be appended above the root yielding $q[f[f[a,a],f[a,a]]]$. Considering the general form of td transducer rules, see Equation 2.1 for details, the td transducer is left with two options:

i. Delete one of the subtrees, $f[a,a]$, while continuing to inspect the other. This approach is clearly undesirable as it makes it impossible to check whether the deleted subtree contains b's or c's.

ii. Inspect both subtrees simultaneously by generating a state above each. Since states in td transducers are of rank one, this cannot be accomplished without generating an output symbol above the states[4], e.g. like $h[q[f[a,a]],q[f[a,a]]]$. Since the root of the output tree must be a, this approach cannot be used.

Hence, the translation above cannot be realized by a td transducer. However, [Eng77] offers remedy, a construction of a *top-down tree transducers with regular look-ahead*, capable of the translation above.

A top-down tree transducer with regular look-ahead, *td^R transducer* for short, is a td transducer combined with a deterministic bu automaton. The td^R transducer is a six tuple $\mathrm{T}^R = (\Sigma, \Sigma', Q, R, q_0, \mathrm{B})$ where Σ, Σ' and q_0 are as in a td transducer. B is a bu automaton (Σ, Q, Q_a, R_b). The rules in R are extended td transducer rules and have the form

$$< lhs \rightarrow rhs, D >$$

where D is the new extension.

[4]This is once again due to the general form of td transducer rules.

The set D contains a subset of the states used in B and determines whether or not that specific rule may be applied at subtree t. The rule may be applied at t if B outputs a symbol $q \in D$ when given t as input.

E.g. let the top-down tree transducer with regular look-ahead $\mathrm{T}^R_{ex1} = (\Sigma, \Sigma', Q, R, q_0, \mathrm{B}_{ex1})$ where $\Sigma = \{f^{(2)}, a^{(0)}, b^{(0)}, c^{(0)}\}$, $\Sigma' = \{a^{(0)}\}$, $Q = \{q_0\}$ and R contains the rules

$$\begin{aligned} < q_0[f[x_1, x_2]] &\rightarrow a, \{q_y\} >, \\ < q_0[a] &\rightarrow a, \{q_y\} > . \end{aligned}$$

On the input $f[f[a,a], f[a,a]]$, T^R_{ex1} will perform the single derivation step $f[f[a,a],\ f[a,a]] \Rightarrow a$ since B_{ex1} outputs q_y on that input tree. T^R_{ex1} will not give any output on the input $f[f[b,a], f[a,a]]$ since B_{ex1} will output q_n and therefore there is no appropriate rule which the td^R transducer can apply.

In an efficient implementation of a td^R transducer, the result from the bu automaton can be directly saved in the nodes of the input tree. As a result, the bu automaton will only need to be run a single time, before the top-down phase. The top-down phase then uses the precalculated values, stored locally in the nodes, to determine which rules can be applied.

2.2.3 Composition with the YIELD Algebra

Recall the translation introduced in Chapter 1, expressions using multiplication and addition translated into expressions using only addition. More specifically, imagine the translation of the tree shown in Figure 2.4. For a td transducer there is no way to carry copies of the subtree n_2 to the leaves of the subtree rooted at n_1. A remedy for this problem would be to have states with parameters. E.g. when the tree transducer process the node n_1, the state appended above it would have a parameter containing a copy of the entire n_2 subtree. When the tree transducer eventually processes the leaves n_{111}, n_{112} and n_{12} it replaces them with a copy of the parameter.

The *YIELD algebra* [Mai74], *YIELD* for short[5], is an algebra which copies subtrees, e.g. as in Figure 2.4 (b). Given an input tree t containing variables, YIELD will substitute the variables for copies of subtrees from inside t. There exists three types of special symbols (we use a notation similar to [Dre01]) which are used as operators in YIELD

- *Variables*, $z_i^{(0)}$, which have their regular role in the substitution.

[5]YIELD is not to be confused with the yield of a tree.

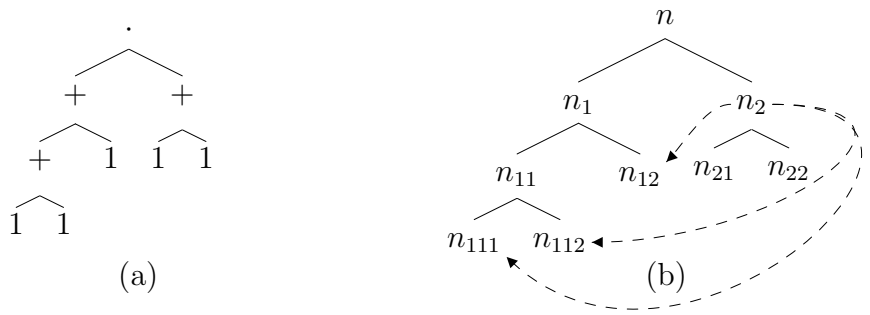


Figure 2.4: (a) and (b) are the same tree. (a) displays the symbols and (b) the node names. Furthermore, the dashed lines in (b) indicate how the subtree rooted at n_2 can be copied into node n_{111}, n_{112} and n_{12} by a tree transducer to translate the tree into one without the ·-operator. In the new tree, n is removed and n_1 is the new root.

- *Substitution symbols*, $\text{subst}^{(k)}$, of different ranks where $\text{subst}[t, t_1, \ldots, t_{k-1}]$ denotes the substitution $t[\![t_1/z_1, \ldots, t_{k-1}/z_{k-1}]\!]$.
- *Function symbols*, $p^{(0)}$, associated with a name f and an integer n. YIELD replaces the function symbol with the tree $f[z_1, \ldots, z_n]$

E.g. let the td transducer $\mathrm{T}_{ex2} = (\Sigma, \Sigma', Q, R, q_0)$ where $\Sigma = \{\cdot^{(2)}, +^{(2)}, 1^{(0)}\}$, $\Sigma' = \{+^{(2)}, 1^{(0)}, \text{subst}^{(2)}, z_1^{(0)}\}$, $Q = \{q, q_0\}$ and R contains the rules

$$\begin{aligned}
q_0[\cdot[x_1, x_2]] &\rightarrow \text{subst}[q[x_1], q_0[x_2]],\\
q_0[+[x_1, x_2]] &\rightarrow +[q_0[x_1], q_0[x_2]],\\
q_0[1] &\rightarrow 1,\\
q[+[x_1, x_2]] &\rightarrow +[q[x_1], q[x_2]],\\
q[1] &\rightarrow z_1,\\
q[\cdot[x_1, x_2]] &\rightarrow \text{subst}[q[x_1], q[x_2]].
\end{aligned}$$

Then a composition of T_{ex2} and a YIELD algebra can be used to translate the arithmetic expression of the tree in Figure 2.4. For details regarding this translation, see Figure 2.5.

2.3 Macro Tree Transducers

The macro tree transducer, *mt transducer* for short, can roughly be seen as a combination of the top-down tree transducer with the YIELD algebra which

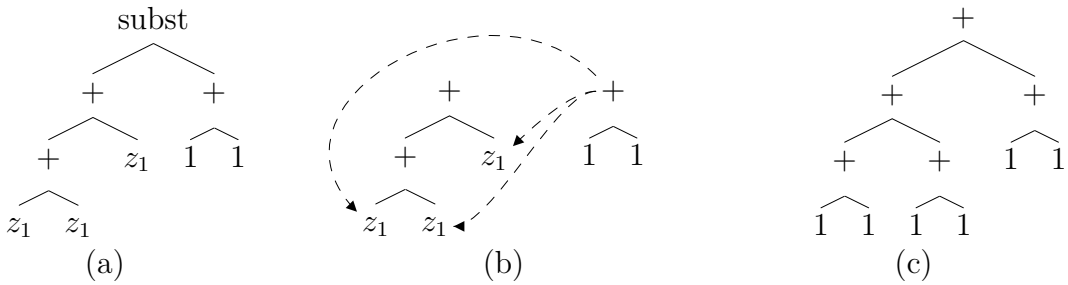


Figure 2.5: (a) shows the output generated by T_{ex2} on the input tree from Figure 2.4 (a). (b) shows how YIELD replaces three leaves with copies of a subtree. (c) displays the output of the YIELD algebra after receiving (a) as input.

results in a more powerful model of computation. It has been shown in [EV85] that all translations performed by a deterministic macro tree transducer can also be realized by a composition of a deterministic top-down tree transducer with a YIELD algebra, and vice versa. Macro tree transducers were first mentioned in [Eng80] and later, more formally defined in [EV85].

The mt transducer named M is written as a five-tuple

$$\mathrm{M} = (\Sigma, \Sigma', Q, R, q_0)$$

where

- Σ and Σ' are the finite ranked input and output alphabets respectively.
- Q is a finite ranked alphabet of states. In a mt transducer states may have ranks greater than one.
- R denotes a finite set of rules. The mt transducer rules are described in Section 2.3.1.
- q_0, the initial state, must be of rank one and works like the initial state of a td transducer.

2.3.1 Rules and Derivation Steps

Like td transducer rules, the macro tree transducer rules consists of a lhs and a rhs and works in a similar fashion. I.e. in a derivation step, the lhs

and rhs describes the pre and post-image, respectively, of the processed tree. Contrary to td transducer rules, the rhs may have

- Nested states, i.e. in a rhs and therefore also in a sentential tree, multiple state symbols may be encountered on a path from the root to a leaf.
- Parameters, y_i, to implicitly pass along data in the translation process. Technically, parameters resemble variables but have lesser restrictions on in which context they may appear[6].

The general form of mt transducer rules is

$$q[\sigma[x_1, \ldots, x_m], y_1, \ldots, y_n] \to t \tag{2.3}$$

where $q \in Q^{(n+1)}$, $\sigma \in \Sigma^{(m)}$ and $t \in T$, where the set T is recursively defined as

i. $y_i \in T$ for all $y_i \in Y_{[n]}$.

ii. $\delta[t_1, \ldots, t_l] \in T$ for all $\delta \in \Sigma'^{(l)}$ and $t_1, \ldots, t_l \in T$.

iii. $q'[x_i, t_1, \ldots, t_l] \in T$ for all $q' \in Q^{(l+1)}$, $x_i \in X_{[m]}$ and $t_1, \ldots, t_l \in T$. The special case $Q = Q^{(1)}$ defines td transducer rules.

The formal definition of a derivation step in a mt transducer extends that of a td transducer to the more general type of rules. A derivation step transforming the tree s into s', denoted by $s \Rightarrow s'$, exists if there is a rule $r \in R$ defined identically as in Equation 2.3 and

$$\begin{aligned} s &= s_0[\![q[\sigma[s_1, \ldots, s_m], s_{m+1}, \ldots, s_{m+n}]]\!], \\ s' &= s_0[\![t[\![s_1/x_1, \ldots, s_m/x_m, s_{m+1}/y_1, \ldots, s_{m+n}/y_n]]\!]\!], \end{aligned}$$

where $s_0 \in T_{\Sigma \cup Q \cup \Sigma'}(X_1)$ and $s_1, \ldots, s_{m+n} \in T_{\Sigma \cup Q \cup \Sigma'}$.

Depending on its set of rules, the macro tree transducer M is said to be

- *linear* if for every rule $r \in R$, every variable, x_i, occurs at most once in the rhs. If the same applies for the parameters, y_i, it is *parameter linear*.

[6]There exists a tree transducer, named *attributed tree transducer*, which passes along data/parameters in an explicit way in attributes. Attributes are of different data types than trees, e.g. integers or strings [Fül81].

- *deleting* if for some rule $r \in R$, a variable, x_i, in the lhs does not occur in the rhs. If a parameter, y_i, in the lhs of a rule does not occur in the rhs, that rule is *parameter deleting*.

- *total* if for every pair $q \in Q, \sigma \in \Sigma$ there exists at least one (q, σ)-rule in R, where the meaning of the term "(q, σ)-rule" is defined as in the case of td transducers.

- *deterministic* if there exists at most one (q, σ)-rule in R for each pair $q \in Q, \sigma \in \Sigma$. If more than one such rule exists for a pair, the mt transducer is *non-deterministic*.

2.3.2 A Simple Example

This section shows how a mt transducer can be used to realize the translation introduced earlier. Translation of expressions containing both multiplication and addition to expressions of only addition.

Let the linear, non-deleting and non-parameter deleting, total, deterministic, mt transducer $\mathrm{M}_{ex1} = (\Sigma, \Sigma', Q, R, q_0)$ where $\Sigma = \{+^{(2)}, \cdot^{(2)}, 1^{(0)}\}$, $\Sigma' = \{+^{(2)}, 1^{(0)}\}$, $Q = \{q^{(2)}, q_0^{(1)}\}$ and R contains the rules

$$\begin{aligned}
q_0[\cdot[x_1, x_2]] &\rightarrow q[x_1, q_0[x_2]], \\
q_0[+[x_1, x_2]] &\rightarrow +[q_0[x_1], q_0[x_2]], \\
q_0[1] &\rightarrow 1, \\
q[+[x_1, x_2], y_1] &\rightarrow +[q[x_1, y_1], q[x_2, y_1]], \\
q[1, y_1] &\rightarrow y_1, \\
q[\cdot[x_1, x_2], y_1] &\rightarrow q[x_1, q[x_2, y_1]].
\end{aligned}$$

Then Figure 2.6 shows how M_{ex1} translates the tree from Figure 2.4.(a) into a multiplication-free tree.

2.3.3 IO, OI and Unrestricted Derivation Modes

As mentioned in Section 2.3.1 and displayed in Figure 2.6, nested states may occur in sentential trees. In trees with nested states the question of precedence arises: At which state should a derivation step be performed first? In deterministic mt transducers, the order in which derivation steps are performed does not matter, the results will be identical regardless of

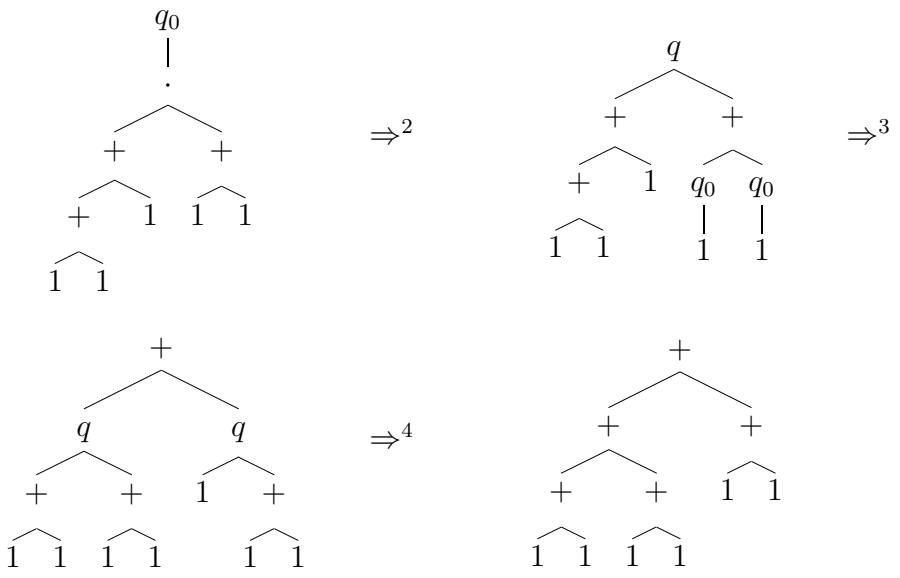


Figure 2.6: Derivation steps conducted by a mt transducer to translate multiplication into addition.

order. The reader may convince him-/herself of this by experimenting with the previous, deterministic, example in Section 2.3.2.

However, the case is different for non-deterministic mt transducers. A non-deterministic mt transducer may yield different sets of results depending on the order which the nested states are processed. In [EV85] three alternatives are given for how to determine the order in which nested states are processed.

- *Innermost-Outermost* (*IO* for short), the states are processed from the bottom up. E.g. in the second tree of Figure 2.6 the two q_0-states would be processed before the q-state.

- *Outermost-Innermost* (*OI* for short), is the opposite of IO, the topmost states are processed first. With OI, q is processed before any of the q_0 in the second tree of Figure 2.6.

- *Unrestricted* (*Unr* for short), no particular order is used, derivation steps can be taken in any order.

A derivation step taken in IO, OI or Unr mode is denoted by $\Rightarrow_{IO}$, $\Rightarrow_{OI}$ and $\Rightarrow_{Unr}$ respectively.

Consider the non-deterministic mt transducer M which can generate several different output trees on the single input tree t. Now consider t as a subtree in a larger tree and t is copied *before* it has been processed (OI). Then the copies of t may result in nonidentical subtrees once they have been processed. However, if t is copied *after* it has been processed (IO), all copies of t will be identical. Let the set of translations realized by the mt transducer M in IO, OI and Unr-mode be denoted by $\tau_{IO}(\mathrm{M})$, $\tau_{OI}(\mathrm{M})$ and $\tau_{Unr}(\mathrm{M})$ respectively.

To illustrate this with a short example, let $\mathrm{M}_{ex2} = (\{f^{(1)}, a^{(0)}\}, \{h^{(2)}, b^{(0)}, c^{(0)}\}, \{q^{(1)}, p^{(2)}\}, R, q)$ where R contains the rules

$$\begin{array}{rcl} q[f[x_1]] & \rightarrow & p[x_1, q[x_1]], \\ p[a, y_1] & \rightarrow & h[y_1, y_1], \\ q[a] & \rightarrow & b, \\ q[a] & \rightarrow & c. \end{array}$$

Then the translations realized by M_{ex2} on the input term $f[a]$ are

$$\begin{array}{lll} f[a] & \Rightarrow^*_{IO} & \{h[b,b], h[c,c]\}, \\ f[a] & \Rightarrow^*_{OI} & \{h[b,b], h[b,c], h[c,b], h[c,c]\}, \\ f[a] & \Rightarrow^*_{Unr} & \{h[b,b], h[b,c], h[c,b], h[c,c]\}, \end{array}$$

In this example, b and c are the non-deterministic subtrees being identically copied in IO but not in OI or Unr mode. Furthermore, the translations realized by IO in this example is a subset of those realized by OI. The OI and Unr result sets equal each other. It has actually been given as a corollary in [EV85] that this also holds in general. Thus for a given mt transducer M

$$\tau_{IO}(\mathrm{M}) \subseteq \tau_{OI}(\mathrm{M}) = \tau_{Unr}(\mathrm{M}).$$

$\tau_{OI}(\mathrm{M}) = \tau_{Unr}(\mathrm{M})$ is particularly interesting, since it tells us that OI derivations are as expressive as Unr derivations. Therefore, Unr derivations does not need to be considered further.

2.3.4 Non-Determinism and Dead Ends

Let us consider the non-deterministic mt transducer $\mathrm{M}_{ex3} = (\{f^{(2)}, a^{(0)}, b^{(0)}\}, \{h^{(2)}, a^{(0)}, b^{(0)}\}, \{q^{(1)}, q_0^{(2)}, q_1^{(2)}\}, R, q)$ where R contains the rules

$$
\begin{array}{lrcl}
(3.1) & q_0[b, y_1] & \to & a, \\
(3.2) & q_1[a, y_1] & \to & a, \\
(3.3) & q[b] & \to & b, \\
(3.4) & q[a] & \to & b, \\
(3.5) & q[f[x_1, x_2]] & \to & q_0[x_1, q[x_2]], \\
(3.6) & q[f[x_1, x_2]] & \to & q_0[x_2, q[x_1]], \\
(3.7) & q[f[x_1, x_2]] & \to & q_1[x_1, q[x_2]], \\
(3.8) & q[f[x_1, x_2]] & \to & q_1[x_2, q[x_1]], \\
(3.9) & q[f[x_1, x_2]] & \to & h[q[x_2], q[x_1]].
\end{array}
$$

On the input term $f[f[a, b], b]$, M_{ex3} could conduct the following derivation steps

$$
\begin{gathered}
q[f[f[a, b], b]] \Rightarrow_{IO} q_1[b, q[f[a, b]]] \Rightarrow_{IO} q_1[b, q_0[b, q[a]]] \Rightarrow_{IO} \\
q_1[b, q_0[b, b]] \Rightarrow_{IO} q_1[b, a] \Rightarrow_{IO} \lambda
\end{gathered}
$$

The second last term, $q_1[b, a]$, is still in sentential form, but no further derivation steps can be applied. The derivation is stuck in a *dead end* and produces an *undefined output*, denoted by λ. However, if another sequence of derivation steps had been chosen, the derivation would have been successful, e.g. the derivation sequence

$$
q[f[f[a, b], b]] \Rightarrow_{IO} h[q[b], q[f[a, b]]] \Rightarrow^2_{IO} h[b, q_0[b, q[a]]] \Rightarrow^2_{IO} h[b, a].
$$

For non-deterministic mt transducer with an input tree where some, but not all, derivation sequences lead to a dead end, there exists an algorithm which avoids sequences leading to dead ends. The algorithm is a type of regular look-ahead. Before any derivation steps are performed, each node in the tree is processed bottom-up. The algorithm stores information in each node on which states can safely be appended above it. This information is later used by the mt transducer to avoid rules which appends states leading to dead ends. The algorithm to avoid dead ends in IO and OI derivations differs slightly. In fact, the algorithm for IO mode is a special case of the one for OI and they are described in Section 2.3.5 and Section 2.3.6, respectively. Implementations of both algorithms are described in Chapter 3.

2.3.5 Avoiding Dead Ends in IO Derivations

Consider a mt transducer $\mathrm{M} = (\Sigma, \Sigma', Q, R, q_0)$ and an input tree $t = \sigma[t_1, \ldots, t_l]$ where $\sigma \in \Sigma^{(l)}$ and $t_1, \ldots, t_l \in T_\Sigma$. Let the boolean value $\Gamma(q, t)$ where $q \in Q$ be true, $\mathcal{T}$, if there exists a derivation sequence $q[t, \ldots] \Rightarrow_{IO}^{*} s$ where $s \in T_{\Sigma'}$, or false, $\mathcal{F}$, if no such derivation sequence exists. The parameters of q can, without risk of ending up in a dead end, be disregarded (denoted by using $\ldots$) since we are assuming IO derivations, i.e. the parameters have already been evaluated. If $\Gamma(q, t) = \mathcal{T}$ then $q[t, \ldots]$ is referred to as *safe* and there exists a derivation sequence which leads to a terminal tree.

Set $\Gamma(q, \sigma[t_1, \ldots, t_l]) = \mathcal{T}$ if there exists a (q, σ)-rule in R with rhs r where $OK_{IO}(\mathrm{r}[\![t_1, \ldots, t_l]\!]) = \mathcal{T}$. For the tree $s \in T_{\Sigma \cup Q \cup \Sigma'}$, $OK_{IO}(s)$ is recursively calculated as

i. If $s \in Y_{[n]}$ then $OK_{IO}(s) = \mathcal{T}$.

ii. If $s = \delta^{(k)}[s_1, \ldots, s_k]$ where $\delta \in \Sigma'^{(k)}$ and $s_1, \ldots, s_k \in T_{\Sigma \cup Q \cup \Sigma'}$ then $OK_{IO}(s) = OK_{IO}(s_1) \wedge \ldots \wedge OK_{IO}(s_k)$. In the special case $k = 0$, $OK_{IO}(s) = \mathcal{T}$.

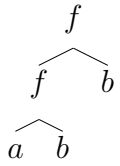
iii. If $s = q'[x_i, s_1, \ldots, s_k]$ where $q' \in Q^{(k+1)}$ and $x_i \in X_{[l]}$ then $OK_{IO}(s) = \Gamma(q', t_i) \wedge OK_{IO}(s_1) \wedge \ldots \wedge OK_{IO}(s_k)$.

An efficient implementation only needs to calculate the information Γ once, before the mt transducer phase. Similarly as described for bu automata in Section 2.2.1.

The information Γ is then used to perform derivation steps which are guaranteed to lead to a terminal tree. A (q, σ)-rule from R with rhs r is safe to apply at the (sub)tree $q[\sigma[t_1, \ldots, t_k], \ldots]$ if and only if $OK_{IO}(\mathrm{r})$.

Continuing on the previous example, M_{ex3} with the input $f[f[a, b], b]$, the calculated information Γ is shown in Figure 2.7. This information is then used in the derivation steps, as shown below:

$q[f[f[a,b],b]]$	Safe rules: (3.6),(3.9)
$\Rightarrow h[q[b], q[f[a,b]]]$	Safe rules: (3.3)
$\Rightarrow h[b, q[f[a,b]]]$	Safe rules: (3.6),(3.7),(3.9)
$\Rightarrow h[b, h[q[b], q[a]]]$	Safe rules: (3.3)
$\Rightarrow h[b, h[b, q[a]]]$	Safe rules: (3.4)
$\Rightarrow h[b, h[b, a]]$	



(a)

$$\begin{array}{rcl} \Gamma(q,a) & = & \mathcal{T} \\ \Gamma(q_0,a) & = & \mathcal{F} \\ \Gamma(q_1,a) & = & \mathcal{T} \end{array} \qquad \begin{array}{rcl} \Gamma(q,f[a,b]) & = & \mathcal{T} \\ \Gamma(q_0,f[a,b]) & = & \mathcal{F} \\ \Gamma(q_1,f[a,b]) & = & \mathcal{F} \end{array}$$

$$\begin{array}{rcl} \Gamma(q,b) & = & \mathcal{T} \\ \Gamma(q_0,b) & = & \mathcal{T} \\ \Gamma(q_1,b) & = & \mathcal{F} \end{array} \qquad \begin{array}{rcl} \Gamma(q,f[f[a,b],b]) & = & \mathcal{T} \\ \Gamma(q_0,f[f[a,b],b]) & = & \mathcal{F} \\ \Gamma(q_1,f[f[a,b],b]) & = & \mathcal{F} \end{array}$$

(b)

Figure 2.7: (a) shows the input tree to the mt transducer M_{ex3}. (b) shows the calculated information Γ for the tree in (a).

2.3.6 Avoiding Dead Ends in OI Derivations

Consider the non-deterministic mt transducer $\mathrm{M}_{ex4} = (\{f^{(2)}, a^{(0)}, b^{(0)}\}, \{f^{(2)}, c^{(0)}, d^{(0)}, e^{(0)}\}, \{q_0^{(1)}, p^{(3)}\}, R, q_0)$ (loosely based on an example from [EV85]), where R contains the rules

$$\begin{array}{lrcl} (4.1) & q_0[a] & \to & c, \\ (4.2) & q_0[a] & \to & d, \\ (4.3) & q_0[a] & \to & e, \\ (4.4) & q_0[f[x_1,x_2]] & \to & p[x_1,c,q_0[x_2]], \\ (4.5) & q_0[f[x_1,x_2]] & \to & p[x_1,q_0[x_1],e], \\ (4.6) & p[a,y_1,y_2] & \to & f[y_1,y_2], \\ (4.7) & p[a,y_1,y_2] & \to & f[y_1,c], \\ (4.8) & p[b,y_1,y_2] & \to & f[c,y_2], \\ (4.9) & p[f[x_1,x_2],y_1,y_2] & \to & f[p[x_1,y_1,y_2],p[x_2,y_1,y_2]]. \end{array}$$

Derivations with M_{ex4} in OI mode may end up in dead ends, similarly as described in the previous section. E.g. consider the derivation sequence

$$q_0[f[a,b]] \Rightarrow_{OI} p[a,c,q_0[b]] \Rightarrow_{OI} f[c,q_0[b]] \Rightarrow_{OI} \lambda.$$

Hence, dead end detection is also necessary in OI mode. However, consider the following derivation sequence

$$q_0[f[a,b]] \Rightarrow_{OI} p[a,c,q_0[b]] \Rightarrow_{OI} f[c,c].$$

The first derivation step creates the subterm $q_0[b]$ which cannot reach a terminal state, it is unsafe. Since the second derivation step deletes that subterm

before it has been processed a terminal tree is nevertheless reached. Hence, the dead end detection algorithm for OI derivations must pay attention to parameter deleting rules which allows for some unsafe subtrees in sentential trees.

Given the mt transducer $\mathrm{M} = (\Sigma, \Sigma', Q, R, q_0)$, let $\Gamma_{OI}(q,t)$, where $q \in Q^{(n+1)}$, $t = \sigma[t_1, \ldots, t_l]$, $\sigma \in \Sigma$ and $t_1, \ldots, t_l \in T_\Sigma$, denote a set of strings. Each string $b \in \Gamma_{OI}(q,t)$ is of the form $b = b_1 \ldots b_n \in \{\mathcal{T}, \mathcal{F}\}^n$. For each string $b_1 \ldots b_n \in \Gamma_{OI}(q,t)$ there exists a derivation sequence $q[t, y_1, \ldots, y_n] \Rightarrow^*_{OI} s$, where s is a terminal tree and for each $b_i = \mathcal{F}$ the parameter y_i is deleted before it is processed. On the other hand, for each $b_i = \mathcal{T}$ the parameter y_i *must* be safe, since it will not be deleted before it has been processed. In the special case where q is of rank one, $\Gamma(q,t)$ equals $\{\lambda\}$ if $q[t]$ is safe[7] or $\emptyset$ if it is not.

$\Gamma_{OI}(q,t)$ contains the string $b = b_1 \ldots b_n$ if there exists a (q,σ)-rule in R with rhs r where $OK_{OI}(\mathrm{r}[\![b_1/y_1, \ldots, b_n/y_n]\!])$. For the tree s, the value of $OK_{OI}(s)$ is recursively calculated as

i. If $s \in \{\mathcal{T}, \mathcal{F}\}$ then $OK_{OI}(s) = s$.

ii. If $s = \delta[s_1, \ldots, s_k]$ where $\delta \in \Sigma'^{(k)}$ then $OK_{OI}(s) = OK_{OI}(s_1) \wedge \ldots \wedge OK_{OI}(s_k)$. In the special case $k = 0$, $OK_{OI}(s) = \mathcal{T}$.

iii. If $s = q'[x_i, s'_1, \ldots, s'_k]$ where $q' \in Q^{(k+1)}$ and $x_i \in X_{[l]}$ then

$$\begin{gathered} OK_{OI}(s) \\ \equiv \\ \exists b'_1 \ldots b'_k \in \Gamma(q', t_i)) : (\neg b'_1 \vee OK_{OI}(s'_1)) \wedge \ldots \wedge (\neg b'_k \vee OK_{OI}(s'_k)). \end{gathered} \tag{2.4}$$

I.e. $OK_{OI}(s) = \mathcal{T}$ if there exists a string $b' = b'_1 \ldots b'_n \in \Gamma(q', t_i))$ where $\neg b'_j \vee OK_{OI}(s'_j)$ holds for each $j \in \{1, \ldots, k\}$. If $k = 0$ and $\Gamma_{OI}(q', t_i) = \{\lambda\}$ then $OK_{OI}(s) = \mathcal{T}$.

In the special case $Q = Q^{(1)}$, the algorithm above is equivalent to the dead-end detection algorithm for td transducers. A second special case, where all parameters of a state must be safe[8], yields the algorithm used for IO derivations.

[7] Zero of zero parameters are safe.

[8] $\Gamma(q^{(n+1)}, t) = \{\mathcal{T}\}^n$

As in the IO version, the calculated information Γ is then used in order to avoid derivation steps leading to dead-ends. Let each node with a symbol $q \in Q^{(n+1)}$ in a sentential tree contain a string $b = b_1 \dots b_n \in \{\mathcal{T}, \mathcal{F}\}^n$ denoting which of its subtrees are safe. The initial state node, of rank one, contains the string $b = \lambda$.

At the subtree $q[\sigma[t_1, \dots, t_k], s_1, \dots, s_n]$ where $q \in Q^{(n+1)}$, $\sigma \in \Sigma^{(k)}$, $t_1, \dots, t_k \in T_\Sigma$ and $s_1, \dots, s_n \in T_{\Sigma \cup Q \cup \Sigma'}$ the rule $q[\sigma[x_1, \dots, x_k], y_1, \dots, y_n] \to \mathrm{r}$ may be applied if $OK_{OI}(\mathrm{r}[\![b_1/y_1, \dots, b_n/y_n]\!])$. Similarly as in the IO case, an efficient implementation need only compute the information Γ once, before any derivation step is performed, and store it locally in each respective node.

After the rule has been applied, yielding the resulting tree t'. For all newly created subtrees of the form $q'[s'_0, \dots, s'_i]$ where $q \in Q^{(i+1)}$, create the string $b' = b'_1 \dots b'_i$ where $b'_j = OK_{OI}(s'_j)$ where $j \in \{1, \dots, i\}$ and store b' in the respective node of q'.

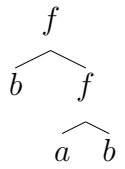
As a more intuitive explanation, one can say that Γ contains the static information telling us for each subtree t and each state q which combination of parameters $t'_1, \dots, t'_k$ *must* be safe in order to make $q[t, t'_1, \dots, t'_k]$ safe. The strings stored in the states during execution time provide the dynamic information telling us which parameters actually *are* safe in the given situation. E.g. consider the input term $f[b, f[a, b]]$ to M_{ex4}. Figure 2.8 shows the calculated information Γ of the tree. A safe derivation sequence, which considers the information from Figure 2.8, is shown in Figure 2.9.

There exists an optimization which reduces the number of strings in Γ. First let us define the partial order $\leq$ for the strings considered in this section as

$$b_1, \dots, b_k \leq c_1, \dots, c_k$$

if for every b_i that is true, c_i is also true, where $i \in \{1, \dots, k\}$. E.g. $\mathcal{TF} \leq \mathcal{TT}$ and $\mathcal{TT} \leq \mathcal{TT}$ but $\mathcal{FT} \not\leq \mathcal{TF}$. The careful reader might have noticed that for any string $c \in \Gamma_{OI}(q', t_i)$ which satisfies $OK_{OI}(s)$ in Equation 2.4 all strings $b \in \Gamma_{OI}(q', t_i)$, such that $b \leq c$, will also satisfy $OK_{OI}(s)$. Hence, the algorithm for dead end detection does not need to include any string c if there already exists a $b \leq c$ in Γ. An efficient implementation should therefore test "smaller" strings first when generating Γ, i.e. $\mathcal{FF} \dots \mathcal{F}$ is the first string to be tested. Intuitively, the optimization can be explained as disregarding all larger ("safer") strings if the algorithm has already proven a smaller ("less safe") string to be large enough ("safe enough"). E.g. for Γ in Figure 2.8 (b), the sets $\{\mathcal{TT}, \mathcal{TF}\}$ and $\{\mathcal{TT}, \mathcal{FT}\}$ can, without altering the

result of the algorithm, be reduced to $\{\mathcal{TF}\}$ and $\{\mathcal{FT}\}$, respectively.



(a)

$$
\begin{aligned}
\Gamma_{OI}(q_0, a) &= \{\lambda\} \\
\Gamma_{OI}(p, a) &= \{\mathcal{TT}, \mathcal{TF}\} \\
\\
\Gamma_{OI}(q_0, b) &= \emptyset \\
\Gamma_{OI}(p, b) &= \{\mathcal{TT}, \mathcal{FT}\} \\
\\
\Gamma_{OI}(q_0, f[a, b]) &= \{\lambda\} \\
\Gamma_{OI}(p, f[a, b]) &= \{\mathcal{TT}\} \\
\\
\Gamma_{OI}(q_0, f[b, f[a, b]]) &= \{\lambda\} \\
\Gamma_{OI}(p, f[b, f[a, b]]) &= \{\mathcal{TT}\}
\end{aligned}
$$

(b)

Figure 2.8: (a) the input to M_{ex4}. (b) the calculated information Γ, strings indicating which subtrees of the corresponding state can be deleted ($\mathcal{F}$).

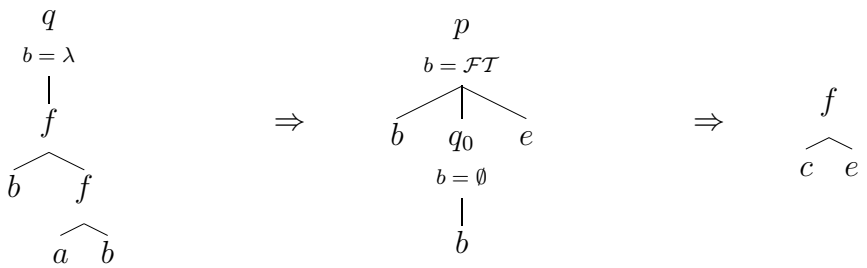


Figure 2.9: In the first derivation step, rule (4.4) and (4.5) are safe. The second derivation step has only one safe rule to choose from, rule (4.8).

2.3.7 Regular Look-Ahead

Recall the translation described in Section 2.2.2: Output a single a if and only if all leaves in the input are of the symbol a. To demonstrate that mt transducers have the ability to simulate regular look-ahead (shown in general

in [EV85] for both the IO and OI case), let $\mathrm{M}_{ex5} = (\Sigma, \Sigma', Q, R, q_0)$ where $\Sigma = \{f^{(2)}, a^{(0)}, b^{(0)}\}$, $\Sigma = \{a^{(0)}\}$, $Q = \{q_0^{(1)}, q^{(2)}\}$ and R contains the rules

$$\begin{array}{rcl} q_0[f[x_1, x_2]] & \rightarrow & q[x_1, q_0[x_2]], \\ q_0[a] & \rightarrow & a, \\ q[f[x_1, x_2], y_1] & \rightarrow & q[x_1, q[x_2, y_1]], \\ q[a, y_1] & \rightarrow & y_1 \end{array}$$

Then the following derivation steps show how M_{ex5} processes the input term $f[f[a, a], f[a, a]]$:

$$q_0[f[f[a,a], f[a,a]]] \Rightarrow q[f[a,a], q_0[f[a,a]]] \Rightarrow q[f[a,a], q[a, q_0[a]]] \Rightarrow$$
$$q[f[a,a], q[a,a]] \Rightarrow q[f[a,a], a] \Rightarrow q[a, q[a,a]] \Rightarrow q[a,a] \Rightarrow a.$$

On an input term containing a b, M_{ex5} will not generate any output. E.g. on the input $f[f[a, a], b]$, M_{ex5} performs the following derivation steps:

$$q_0[f[f[a,a], b]] \Rightarrow q[f[a,a], q_0[b]] \Rightarrow \lambda.$$

In the above derivation, a dead-end was encountered at the subtree $q_0[b]$ and the tree transducer gives an undefined output.

Chapter 3

Implementation

This chapter describes the macro tree transducer implementation, a TREEBAG-component written in Java and named `mtTransducer`. When possible and appropriate, code and ideas from the already existing td transducer component, `tdTransducer`, have been modified or extended to fit the `mtTransducer`. The chapter begins with Section 3.1 which, in retrospect to Chapter 2, sums up the necessary requirements for an implementation of the mt transducer. Section 3.2 and Section 3.3 describe the general design of the component and the API[1] through which it communicates with the TREEBAG system, respectively. The user interface of the component is demonstrated in Section 3.4.

3.1 Requirements

From a user's point of view, a number of soft and hard requirements on the `mtTransducer` component can be identified. The component should

- Use an object file syntax which resembles both the syntax used in other TREEBAG components[2] and the mathematical notation used in the literature on macro tree transducers. A similar syntax among components allow users to more easily start using the `mtTransducer` and a similarity to the mathematical notation simplifies the process of using examples from articles in journals.

[1] Application Programming Interface

[2] In particular the td transducer component.

- Have the ability to switch between derivations made in IO and OI mode.
- Detect and avoid dead-ends in derivation sequences, using the algorithms described in Section 2.3.5 and Section 2.3.6.
- Allow the user to follow the derivation sequence of an input tree with commands such as *single step* and *parallel step*. A single step applies a rule at a state in the tree, that state is found by a post-order tree traversal in IO and a pre-order tree traversal in OI. In a parallel step, multiple states are processed simultaneously, at least from a user point of view. Those states are the first detected state of each subtree in a post-order and pre-order tree traversal in IO and OI, respectively. E.g. it should be possible to perform the last shown derivation in Figure 2.2 with only one parallel step command.
- Record the sequence of issued single and parallel steps and allow for a *back* command. The back command rewinds the sequence of step commands.
- Deterministic and non-deterministic mt transducers should output a single output tree. In the non-deterministic case, where more then one rule can be applied at a node, the mt transducer should pseudo randomly choose one of them. The implementation should therefore allow the user to issue a command, *new random seed*, which reperforms the derivation choosing rules in a new pseudo random order.

3.2 General Design

Given the requirements from the previous section, the implementation can be divided into three subproblems: A

- *Parser*, `mtTransducerParser`, to create an internal representation from a description given in the Treebag object file syntax. Section 3.5 describes the parser.
- *Computation tree*, `mtComputationTree`, which transforms the input to an output tree. It also records user commands such as single-/parallel step and back, in order to always output the desired result. This data structure is described in detail in Section 3.6.

- Wrapper class, `mtTransducer`, for the parser and the computation tree. This class interfaces with the rest of the TREEBAG-system via the TREEBAG *Component API*, described in Section 3.3.

Figure 3.1 show the general design of the mt transducer implementation and how the three subproblems above are interconnected.

3.2.1 Basic Data Types

In TREEBAG ranked symbols are represented by the Java class `symbol` which contains two variables, a string and an integer for its name and rank, respectively.

The class `finiteSignature`[3] contains instances of the class `symbol`. A `finiteSignature` can be indexed, e.g. like the Java `Vector`, and supports some basic set operations, e.g. contains, $\in$. In the mt transducer component, instances of `finiteSignature` store the sets Σ, Σ' and Q which are parsed from the object file by the parser.

TREEBAG uses the class `term` for its internal tree representation. Each instance of `term` symbolizes a node and contains a `symbol` and references to its subterms. Each `term` contains as many subterms as the rank of its symbol.

The rule set of the implementation, representing R, consists of a two dimensional `term` array, where the rule number is one dimension and lhs/rhs is the other.

The class `extendedTerm`, only used inside of the mt transducer component, inherits `term` and has the ability to locally, in the respective node, store the information Γ used by the dead-end detection algorithms.

To store the Γ in the IO, the class `extendedTerm` contains a `BitSet`[4] which stores which nodes may safely be appended above the respective node. If an index i of the `BitSet` maps to true, it means that the symbol at index i in the `finiteSignature` representing Q, is safe to append above this `extendedTerm`.

In OI, each `extendedTerm` uses an array of `Vector` instances instead of a single `BitSet`. The array contains a vector for each rule in the rule set. The vector at index i in node n represents the set of strings $\Gamma_{OI}(q, n)$ where q is the symbol at index i in the `finiteSignature` representing the state set.

[3] Finite signature is synonymous with finite ranked alphabet.

[4] A `BitSet` maps a positive integer to a boolean value [Mic04].

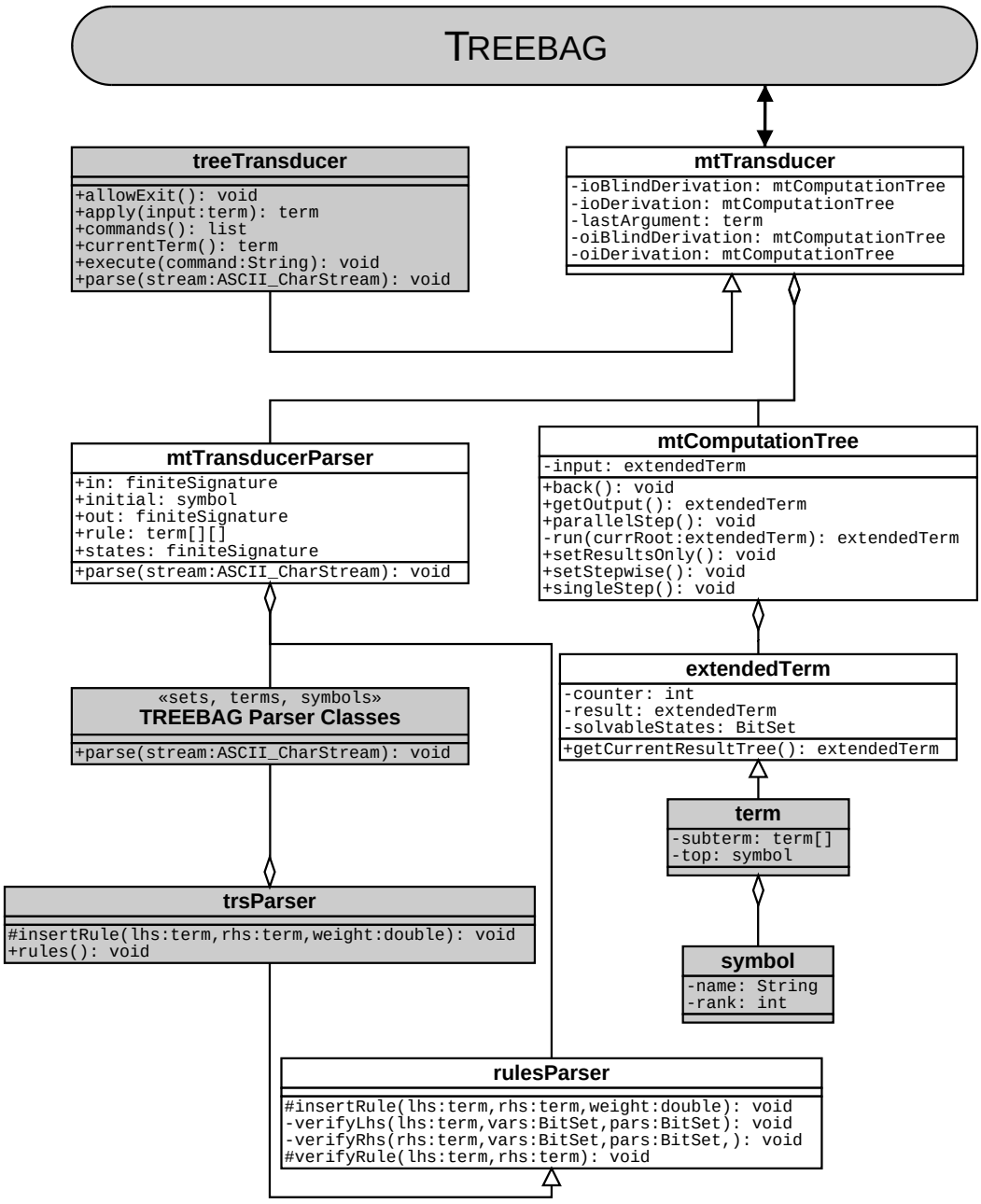


Figure 3.1: Schematic diagram of the classes in the mt transducer component. White classes are part of the mt transducer component and grey classes are part of the standard TREEBAG-system. `mtTransducer` interfaces with TREEBAG and forwards data streams to and from the parser and the computation tree. The `mtTransducerParser` takes advantage of a number of already existing TREEBAG-parsers, parsers used for e.g. symbols, terms and ranked alphabets. `mtComputationTree` uses the class `extendedTerm` to internally represent trees. The `extendedTerm` inherits the class `term`, which in turn contains a `symbol`.

Each element in the `Vector` at that index is a `BitSet` instance representing a specific string in $\Gamma_{OI}(q, n)$. Section 3.6.1 describes how this information is generated.

The component uses the class `Random`, available through the Java 2 API [Mic04], for random number generation.

3.3 TREEBAG Component API

As shown in Figure 3.1, the `mtTransducer` handles the communication between the component and the TREEBAG-system. A tree transducer component in TREEBAG is created by letting the component wrapper class, in this case `mtTransducer`, inherit the class `treeTransducer`. Similarly, any implementations of the other three component types discussed in Section 1.2, generators, algebras and displays inherits the classes `treeGrammar`, `algebra` and `display`, respectively.

The class `mtTransducer` inherits a number of abstract methods from `treeTransducer` to implement. When these methods are called by TREEBAG, data is transferred, via the arguments of the methods, to the component. Data in the other direction, from the component to TREEBAG, is transferred via the return values of these methods. This bidirectional flow of data works as the TREEBAG Component API, through which all communications between the component and TREEBAG are made. Furthermore, the TREEBAG Component API provides the components with a GUI[5] for e.g. computation control and result displays.

More specifically the inherited methods from `treeTransducer` are

`void parse(ASCII_CharStream stream)` — When a new `mtTransducer` component is placed on the worksheet, it is initialized by parsing a text file representing a mt transducer definition.

`term apply(term t)` — This method is called when there is a new input tree available on the inbound link of the component node. The method has one argument, `t`, the new input tree. TREEBAG anticipates the return value to be the resulting output tree.

`list commands()` — Each component on the worksheet has its own *menu*, e.g. as shown in Figure 1.11.(b) on page 11. A menu contains a number

[5] Graphical User Interface

of *command items*. The return value of `commands` is basically a list of strings which TREEBAG will display as command items in the menu of the component. E.g. if "single step" is one of the returned strings, then the mt transducer menu will display that string as a command item. Since the available commands may change, `commands` is always called after a call to `apply` or `execute` (see below).

`void execute(String command)` — Called when the user has executed the command item (pushed a button) corresponding to the argument `command`.

`term currentTerm()` — Returns the current output tree. This method is therefore called when the TREEBAG system expects an updated output, e.g. after a call to `execute`, or a new edge has been established whose source is this component.

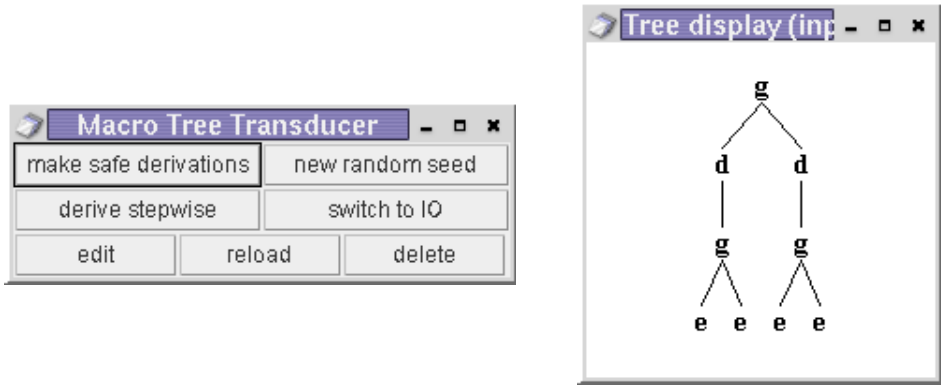
`void allowExit()` — When TREEBAG calls a method to provide a new input tree or to execute a command, it creates a thread where the method is executed. A call to `allowExit` terminates that thread if there is a new input tree available, if not, the thread continues to execute. The method `allowExit` can therefore be called from inside of the component, e.g. in a time consuming loop, allowing for a quick reaction to new user actions.

3.4 User Interface

In this section, the implementation is described from a user conceptual level. I.e. how the GUI reacts to user actions. As a start, the `mtTransducer` component is added to a worksheet in the same manner as any other component. That is, either via a worksheet specification file or from the "Component" menu in TREEBAG, see [Dre01] for instructions.

Similarly as other components, TREEBAG provides a menu for the `mtTransducer`. The menu is displayed as a rectangular box containing several buttons. Figure 3.2.(a) shows the mt transducer menu in its initial mode. For each string in the returned `list` from the method `commands`, described in Section 3.3, TREEBAG creates a button labeled with that string. When the user pushes a button, TREEBAG calls the method `execute` of the component with the label of the button as argument. This allows the component to

interact with the user. From the menu of the mt transducer, the user can order the actions



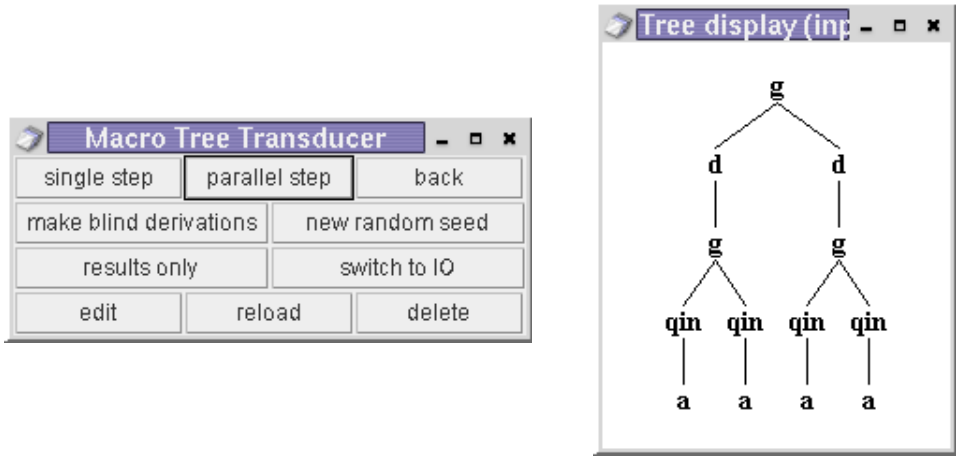
(a) (b)

Figure 3.2: The component in its initial mode, occurs after e.g. a new input. (a) displays the menu and (b) an example output of the component.

- *Switch derivation mode* from IO to OI and vice versa. On a switch of derivation modes, the entire tree is recomputed in the new mode. I.e. in a derivation sequence all derivation steps must be performed in the same mode.
- Choose between *complete* and *stepwise derivations*. Complete derivations directly outputs the final output tree, e.g. see Figure 3.2 (b). From the menu in Figure 3.2 (a), the user can chose to make stepwise derivations ("derive stepwise"). With stepwise derivations, each derivation step is shown in a sequence, e.g. as in Figure 3.3 (b) and the menu displays three more actions, see Figure 3.3 (a). These three actions allow the user to navigate through the sequence of derivations steps in
  - *single step*, which displays the next sentential tree in the derivation sequence.
  - *parallel step*, which displays the sentential tree corresponding to the previous tree where a parallel step was performed.
  - *back*, cancels the previous step command.

- Make *safe* or *blind derivations*, where the latter makes derivations without bothering about dead-end detection. The user can observe derivation sequences leading to dead-ends if stepwise derivations are made in blind mode. The safe/blind choice is only available for non-total non-deterministic mt transducers.

- Chose a *new random seed* in non-deterministic mt transducers. A new `mtComputationTree` with a different random seed for the random number generator is created and used by the `mtTransducer`. This results in a possibly new output tree.



(a) (b)

Figure 3.3: Shows the component performing stepwise derivations. The menu (a) has new menu items and (b) displays a sentential tree. If "single step" is pressed in (a), the leftmost q_{in} will be processed. But if "parallel step" is pressed then all four states will be processed simultaneously. The command "back" returns to the sentential tree previously displayed by the component.

The three bottommost commands in the component menu act as in all other TREEBAG components. *Edit* opens the object file of the component in a text editor. *Reload* reinitializes the component with its object file and *delete* removes the component from the worksheet.

3.5 Object File Parser

The object file parser combines several old TREEBAG classes to parse object files. Three of these classes are `symbolParser`, `finiteSignature` and `termParser` which parses symbols, finite ranked alphabets and terms, respectively. Additionally `nameParser` parses the initial state, q_0.

The only modified parser is the `rulesParser` which basically uses `termParser` to parse the lhs and rhs of the rule. A rule is inserted into the mt transducer rule set with the method `insertRule`. The rule is only inserted into the rule set if a call to `verifyRule` proves successful[6]. The method `verifyRule` uses `verifyLhs` and `verifyRhs` to confirm that the respective rule comply with the general form of mt transducer rules, shown in Equation 2.3.

In `mtTransducerParser`, the parsers above are combined to parse a mt transducer definition following the syntax given in Figure 3.4. Besides the slightly more complex rule syntax, there are only trivial differences between `mtTransducerParser` and `tdTransducerParser` The parsers are created with the *Java Compiler Compiler*, a tool for generating parsers to be used in Java programs.

3.6 Computation Tree

The computation tree class, `mtComputationTree`, is the most complex part of the implementation. Each instance of `mtComputationTree` is initiated with an input tree and a derivation mode, IO or OI. The computation tree calculates Γ, depending on whether the mode is IO or OI, as described in Section 3.6.1. With respect to this information, it performs a sequence of derivation steps until an output tree is reached, see Section 3.6.3 for details. The user may then, as previously described in Section 3.4, view the final output tree or navigate through the derivation sequence with the step commands. Section 3.6.2 and Section 3.6.3 motivates and describes the implemented mechanism for performing derivation steps and navigating through them.

[6]Otherwise the user is prompted with an error message stating the specific syntactic error

$$
\begin{array}{rcl}
< component > & ::= & \texttt{generators.mtTransducer}[(< name >)] :< instance > \\
< instance > & ::= & (< input >, < output >, < states >, < rules >, \\
 & & < initial\ state >)[\texttt{, performance}] \\
< states > & ::= & \{< symbol^{+} > \ (, < symbol^{+} >)*\} \\
< input > & ::= & \{< symbol > \ (, < symbol >)*\} \\
< output > & ::= & \{< symbol > \ (, < symbol >)*\} \\
< initial\ state > & ::= & < symbol^{+} > \\
< rules > & ::= & \{< rule > \ (, < rule >)*\} \\
< rule > & ::= & < tree > \ \texttt{->} < tree > \\
< tree > & ::= & < symbol > \\
 & | & < symbol > [< symbol > \ (, < symbol >)*] \\
< symbol > & ::= & < name >:< nat > \\
< symbol^{+} > & ::= & < name >:< nat^{+} >
\end{array}
$$

Figure 3.4: Object file syntax of the mt transducer component expressed as a grammar, using the same notation as [Dre98]. Strings inside < and > are non-terminals which are expanded to the string in rhs. $< name >$ denotes any alphanumeric string. $< nat >$ and $< nat^{+} >$ denotes any integer in $\mathbb{N}^0$ and $\mathbb{N}^+$, respectively. The option `performance` is explained in Section 3.6.3. See Appendix A for examples.

3.6.1 Algorithms for Dead-End Detection

To avoid dead-ends in IO derivations, the algorithm from Section 2.3.5 is implemented in the computation tree. This section describes how Γ is computed in the IO case and Section 3.6.3 describes how the implementation uses the generated information to avoid dead-ends.

The implemented algorithm is shown as pseudo code in Figure 3.5. For OI derivations, a different algorithm is used, described in Section 2.3.6 and shown as pseudo code in Figure 3.7.

```
1  Procedure calcIOGamma(node n)
2  /* n.symbol is of rank k and child[0], ..., child[k − 1] denote its
   children */
3  if n is a leaf then
4  |  /* All rules can be applied */
5  |  foreach (q, n.symbol)-rule in R do
6  |  |  Γ_IO(q, n) ← true
7  else
8  |  foreach child of n do
9  |  |  /* Γ_IO is generated bottom-up */
10 |  |  calcIOGamma(child)
11 |  foreach (q, n.symbol)-rule in R with rhs r do
12 |  |  if OK_IO(r[[child[0], ..., child[k − 1]]]) then
13 |  |  |  Γ_IO(q, n) ← true
```

Figure 3.5: Pseudo code describing how the Γ_{IO} information is generated. The recursive procedure `calcIOGamma` receives the root of an input tree and calculates the Γ_{IO} information of the nodes bottom up. Pseudo code for the procedure `OK_IO` is given in Figure 3.6.

3.6.2 Computation Tree Approaches

To allow for user commands such as single/parallel steps and back, two (naive) implementation approaches come to mind

```
1  Function OK_IO (node n) returns boolean

2  /* child[0] denotes the leftmost child of n */
3  if n.symbol ∉ Q then
4  |  foreach child of n do
5  |  |  if !OK_IO (child) then
6  |  |  |  return false
7  else if Γ_IO(n.symbol, child[0]) then
8  |  /* Check each parameter */
9  |  foreach child of n except child[0] do
10 |  |  if !OK_IO (child) then
11 |  |  |  return false
12 |  return true
13 else
14 |  return false
```

Figure 3.6: `OK_IO` is a pseudo code implementation of OK_{IO} from Section 2.3.5.

```
1 Procedure calcOIGamma(node n)
2 /* n.symbol is of rank k and child[0],...,child[k-1] denotes its
  children */
3 foreach child of n do
4     calcOIGamma(child)
5 foreach (q, n.symbol)-rule in R with rhs r do
6     /* Where q ∈ Q^(n+1) */
7     foreach b = b_1 ... b_n ∈ {T, F}^n do
8         if OK_OI(r[[child[0],...,child[k-1]]], b) then
9             Γ_OI(q, n) ← Γ_OI(q, n) ∪ {b}
```

Figure 3.7: Generates Γ in the OI case. Similarily as for the IO case, the algorithm processes the input tree bottom up and stores the generated Γ locally in each respective node. Pseudo code for OK_{OI} is found in Figure 3.8.

- Perform the derivation sequence and save a copy of each sentential tree in a list. In results only mode, the last tree in the list is displayed for the user. Then in stepwise derivations, the user can navigate through the list with step and back commands. This approach will however be inefficient with respect to memory space and problematic in parallel steps. There is no way for the component to know which sequence of commands the user will use. A solution where each possible sentential tree, with respect to possible sequences of single or parallel steps, is calculated and stored would be even more inefficient with respect to both time and space.

- A second approach is to keep a single internal tree, which is first processed into an output tree. During this initial derivation, each derivation step pushes a reference to the node, where the derivation step was performed, onto a stack. Such nodes, where a rule was applied, contains a variable which identifies which specific rule that was applied at this node. With this information the output tree can be reversed into a previous sentential tree by performing a reversed derivation step, i.e. rhs $\rightarrow$ lhs. In a switch to stepwise derivations all derivation steps are

```
1  Function OK_OI (node n, BitSet b) returns boolean
2  /* n.symbol is of rank k and child[0], ..., child[k − 1] denotes its
   children */
3  if n.symbol is a parameter y_i then
4      return b[i]
5  else if n.symbol ∉ Q then
6      foreach child of n do
7          if !OK_OI (child, b) then
8              return false
9      return true
10 else
11     foreach b' = b'_1 ... b'_l ∈ Γ_OI(n.symbol, child[0]) do
12         boolean okRhs ← true
13         for j ← 1 to l do
14             if b'_j == true and !OK_OI (n.child[j], b) then
15                 okRhs ← false
16         if okRhs then
17             return true
18     return false
```

Figure 3.8: Pseudo code implementation of OK_{OI} from Section 2.3.6.

reversed, yielding the first sentential tree. For each step command, a set[7] containing references to all nodes which were processed is pushed onto a stack. When a back command is issued, a set is popped from the stack and the elements in it are used for the necessary backward derivation steps. However in backwards derivations where deleting mt transducer rules have been used, the component would need to store deleted subtrees, so that they could be properly restored in the reversed derivation step. Similarly, a problem also occurs in backwards derivations when the tree transducers would have to remember which rule was applied in *future* states[8]. To solve this, one would essentially have to store (sub) trees of future sentential trees, resulting in a greater space complexity, actually similar to the one in the first approach. Another, greater problem, occurs when the user switches from results only to stepwise mode, and vice versa. Each time such a switch is made, the current internal tree is reversed into the desired sentential tree, or output tree. In the worst case the entire tree must be reversed or reprocessed, clearly an undesirable property.

The `tdTransducer` component uses a more efficient approach. Given the input tree t, a copy of t is processed into the output tree t'. In each node n in t' where a rule has been applied, a reference `initial` exists. The reference `initial` was saved during the derivation step which created the output node n and it links to a tree s which is a subtree in t. The tree s is the tree which produced the tree rooted at n. I.e. `initial` represents the tree rooted at n before it was processed, so the component can either display the processed tree, node n, or the unprocessed tree, the reference `initial`.

The td transducer component equips all nodes n with an integer variable c, used as a counter. For complete derivations, the counter is ignored and the tree t' is displayed. I.e. the node n is displayed instead of the one referenced by `initial`. In stepwise derivations, the component displays tree t' as output, and at a node n with a counter larger or equal to zero, n is displayed, just as in results only derivations. But if the counter $c = -1$, it outputs the tree linked to by the reference `initial` instead of the node n itself. This allows the component to increase the counter above -1 where the processed output is supposed to be shown and keep it at -1 where it is not supposed to be shown. I.e. single/parallel step and back is implemented by increasing and

[7]For single step, this set would only contain one element.

[8]Recall that a derivation step may produce new nodes containing state symbols.

decreasing the counters of the nodes, respectively. E.g. in a step, a breadth first search is made in t' and and the following actions are taken at node n with counter c

- If $c > -1$, increase c by one and continue with the subtrees of n.
- If $c = -1$, set c to zero and ignore the subtrees of n. If this is a single step, then at most one node with a counter $c = -1$ is processed.

Figure 3.9 explains the general idea of the td transducer computation tree. Without some modifications, this approach can however not be used for the mt transducer component. E.g. consider IO derivations with a sentential tree t_s containing nested states. Since nodes deeper in the tree t_s would then be processed before nodes above it, the counter values of the deeper nodes would also become greater than those above it. This would result in a computation tree where the counters of the nodes are not monotonically decreasing with increasing depth. Hence the computation tree would need to cover such scenarios and the `tdTransducer` computation tree can not be used in the mt transducer without some modifications.

The approach used in the mt transducer implementation is a combination of the two informal approaches and the one used in `tdTransducer`. It is described in detail in the following section.

3.6.3 The Implemented Approach

The general idea of the approach used in the `mtTransducer` is to save copies of future subtrees at nodes with state symbols. Let s be a subtree in the input tree t. Node n is the root of s and n contains a state symbol. When a rule is applied at n, the component leaves s unmodified and generates the result tree s'. The reference `result` in n refers to the generated tree s'. This allows the component to either show the tree before it was processed, s, or after, s'. In future derivation steps, the tree referred to by `result` is processed, i.e. used as input in the derivations. Similarly as in the td transducer, each node in the `mtTransducer` contains a counter c which is used for stepwise derivations. When $c \geq 0$, the tree referred to by `result`, s', is displayed and otherwise the node and its subtrees, s, are shown.

Since saving each pair of unprocessed and processed subtrees, s and s', is memory demanding, there exists a performance option, see Figure 3.4, which disables the computation tree from saving the unprocessed subtree, i.e. s is

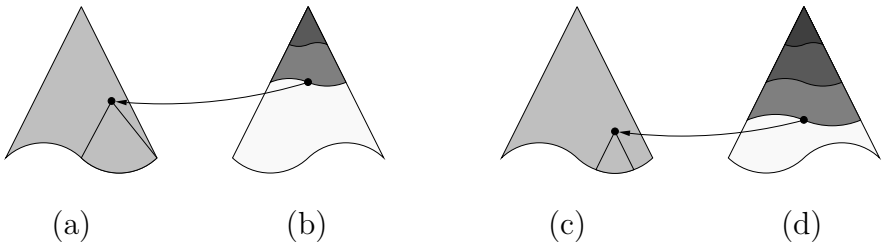


Figure 3.9: General computation tree idea for td transducers. (a) represents the input tree t and (b) the generated output tree t'. The shades of grey in (b) represent the counter values of the nodes in that area of the tree. The lighter and darker grey represents the counter values 0 and 1, respectively, while white represents nodes with a $c = -1$. The arrow from (b) to (a) represents the reference `initial` of one node in t' referencing to an unprocessed subtree. (c) and (d) represents (a) and (b), respectively, after a parallel step has been applied to t'. The counter values of the nodes in the greyed areas are now 2 in the topmost, 1 in the middle and 0 in the bottommost greyed area. If a back command is issued at (c)/(d) then (a)/(b) would be the result.

deleted. Consequently, the computation tree with the performance option enabled can not be used for navigating[9] through the derivation sequence, but less memory is consumed allowing for larger inputs to be processed.

The computation tree processes its input tree[10] to an output tree in either IO or OI mode. Pseudo code for the algorithm, processing the input tree is shown in Figure 3.10. Essentially the implementation makes a depth first search through the tree in a search for nodes with state symbols. In IO, the subtrees of a node are processed before the node itself and in OI the node is processed before its subtrees, see line five and sixteen, respectively, in Figure 3.10.

When `run` finds a node where a rule may be applied, a call is made to the method `applyRuleAt` with the respective node as argument, see Figure 3.11. The method `applyRuleAt` returns the generated tree and `run` stores it in the `result` reference of the node. The `run` method continues the derivation by recursively calling itself with the new `result` variable as parameter, see line nine and fourteen in Figure 3.10.

[9]The final output tree is of course still available.

[10]Where the input tree has already had Γ calculated and stored, as described in Section 3.6.1

```
1  Procedure run(extendedTerm n)
2  if n is a leaf then
3      return
4  else if IO-mode then
5      foreach child of n do
6          run(child)
7      if n.symbol ∈ Q then
8          if performance then
9              n← applyRuleAt(n)
10             run(n)
11         else
12             n.result ← applyRuleAt(n)
13             run(n.result)
14 else
15     /* OI-mode */
16     if n.symbol∈ Q then
17         if performance then
18             n← applyRuleAt(n)
19             run(n)
20         else
21             n.result ← applyRuleAt(n)
22             run(n.result)
23     else
24         foreach child of n do
25             run(child)
```

Figure 3.10: Pseudo code for how the method `run` performs derivation steps in the mt transducer component.

```
1 Function applyRuleAt(extendedTerm n) returns
  extendedTerm
2 extendedTerm result← getCurrentResultTree(n)
3 int ruleNumber ← chooseRule(result)
4 HashTable hTrees ← substituteBind(result, ruleNumber)
5 result ← ruleSet[ruleNumber].rhs
6 substituteReplace(result, hTrees)
7 if OI-mode then
8   calcNewSafeParameters(ruleSet[ruleNumber].rhs, result)
9 return result
```

Figure 3.11: Pseudo code explaining how a derivation step is performed. `substituteBind` and `substituteReplace` together performs substitutions on trees. `substituteBind`(*tree*, *rule*) returns a `HashTable` in which each distinct variable and parameter found in the lhs of *rule* is mapped to a copy of the appropriate subtree in *tree*. `substituteReplace`(*tree*, *hTable*) uses the hash table from `substituteBind` to replace every variable and parameter in *tree* with the copy stored in *hTable*, thus completing the substitution. `HashTable` is a hash table datatype in the Java API, see [Mic04] for details.

With the help of the stored information Γ, `applyRuleAt` searches through the rule set for a method which can be applied at the node. If more than one such rule exists, the choice of rule is made pseudo randomly using the random generator of the `mtTransducer`. Pseudo code explaining `applyRuleAt` is shown in Figure 3.11. Here, `calcNewSafeParameters` calculates which parameters of a state node are safe, using the algorithm described in Section 2.3.6. Pseudo code for `calcNewSafeParameters` is given in Figure 3.12.

```
1  Function calcNewSafeParameters(term rhs, extendedTerm n)
2  /* rhs is of rank k */
3  /* rhs.child[0], ..., rhs.child[k − 1] denotes the children of rhs */
4  /* n.child[0], ..., n.child[k − 1] denotes the children of n */
5  /* b[0], ..., b[k − 2] denotes the characters (boolean values) of the
     string b belonging to a node with a state symbol of rank k */
6  if rhs is a leaf then
7      return
8  else
9      for i ← 0 to k − 1 do
10         calcNewSafeParameters(rhs.child[i], n.child[i])
11     if rhs.symbol ∈ Q then
12         BitSet b
13         for i ← 1 to k − 1 do
14             if OK_OI(n.child[i]) then
15                 b[i − 1] ← true
16         n.safeParameters ← b
```

Figure 3.12: `calcNewSafeParameters` calculates which parameters/subtrees of state node n are safe and stores the results in a `BitSet` b inside of n. Each index i of the `BitSet` b refers to a boolean value, indicating whether or not subtree $i + 1$ of n is safe/true or unsafe/false.

When a step command or back is issued, the methods `step` and `back`, respectively, are called with an argument node n, the root of the tree first processed by `run`: `step` increases the counter c of the nodes in the tree and `back` decreases them, somewhat similarly as in the `tdTransducer`, explained

```
1  Function step(extendedTerm n) returns boolean
2  if IO-mode then
3      if n.counter≥ 0 then
4          n.counter ← n.counter + 1
5          return step(n.result)
6      else
7          boolean applied ← false
8          foreach childof n do
9              if step(child) then
10                 applied ← true
11         if singleStepPerformed and singleStepMode then
12             return true
13         else if n.result != null and !applied then
14             n.counter ← n.counter+ 1
15             if n.counter== 0 then
16                 singleStepPerformed ← true
17                 return true
18         else
19             return applied
20 else
21     if singleStepPerformed and singleStepMode then
22         return false
23     else if n.result != null then
24         n.counter ← n.counter + 1
25         if n.counter == 0 then
26             singleStepPerformed ← true
27         else
28             step(n.result)
29         return false
30     else
31         foreach child of n do
32             step(child)
33 return false
```

Figure 3.13: Method which performs a single or parallel step. Before the method is called for a single step, the booleans singleStepMode and singleStepPerformed are set to the values true and false, respectively. The boolean variable singleStepPerformed then prevents `step` from increasing more than one negative counter per single step.

```
1  Procedure back(extendedTerm n)
2  if IO-mode then
3      if n.counter ≥ 0 then
4          back(n.result)
5          n.counter ← n.counter - 1
6          if n.counter == -1 then
7              back(n)
8      else
9          foreach child of n do
10             back(child)
11 else
12     /* OI-mode */
13     n.counter ← n.counter - 1
14     if n.result != null then
15         back(n.result)
16     else
17         foreach child of n do
18             back(child)
```

Figure 3.14: Figure 1.14 Cancels the last step command by decreasing all counters $c \geq 0$ in the tree by one.

above. Figure 3.13 and Figure 3.14 shows pseudo code for `step` and `back`, respectively, which describes the algorithms in more detail.

As an example, consider the mt transducer M_{ex1} from Section 2.3.2 on page 25. Let `run` perform the same derivation sequence on the input shown in Figure 2.6. Then Figure 3.15 and Figure 3.16 illustrate the resulting internal trees created by `run` and modified by `step`.

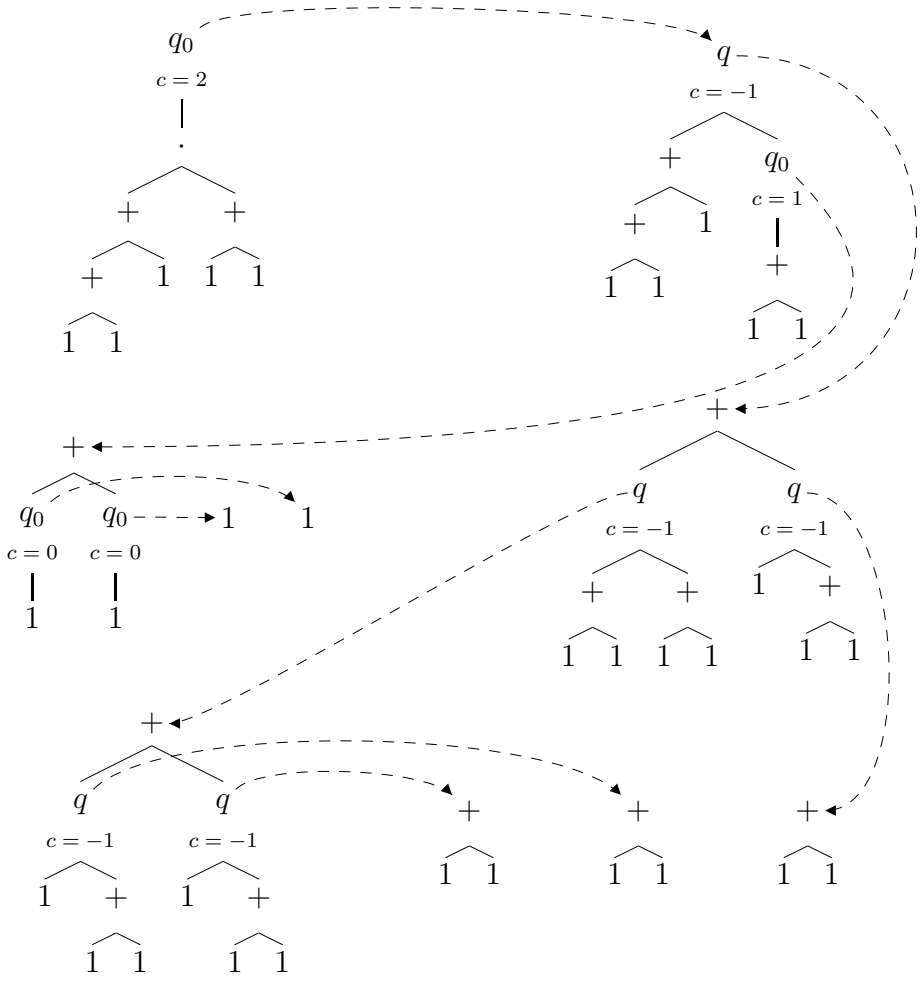


Figure 3.15: Computation tree example representing the derivation sequence from Figure 2.6 on page 26. The dashed lines symbolize the reference `result` and the counters of the nodes represent the state achieved after two single steps and one parallel step. The computation tree outputs the term $q[+[+[1,1],1],+[1,1]]$ in this state.

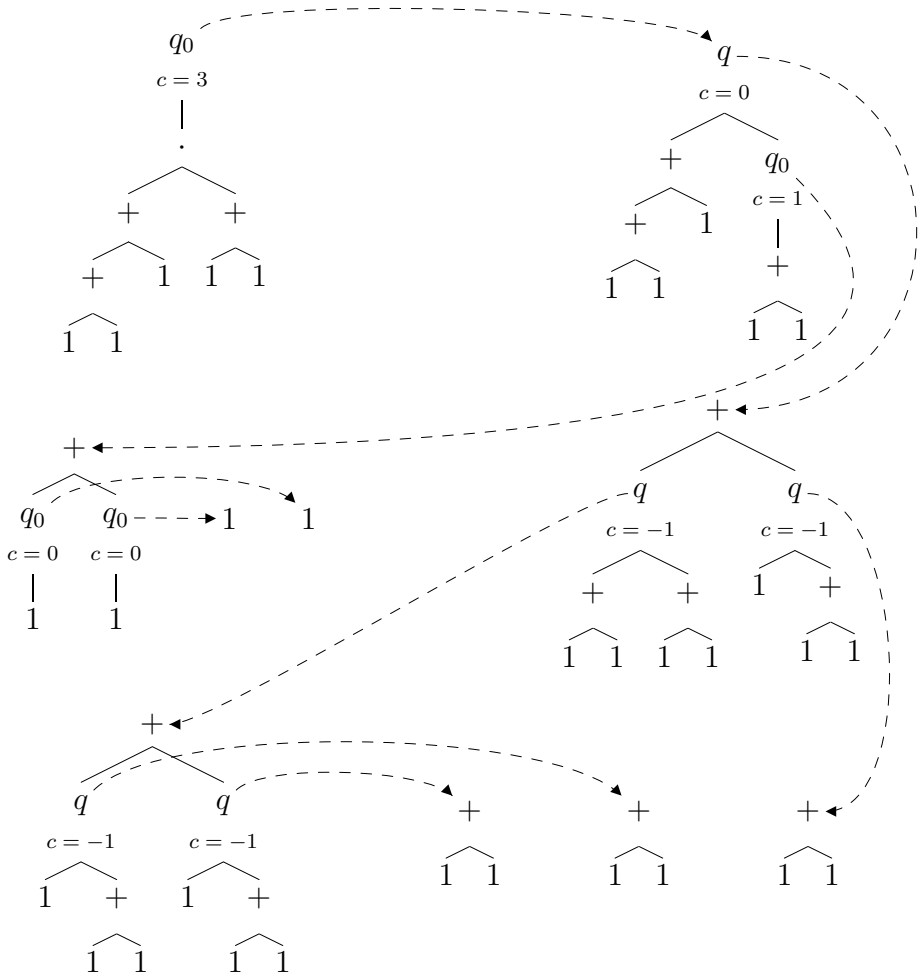


Figure 3.16: Represents the computation tree from Figure 3.15 after another single step. Note that only two counters have been increased. In this state, the computation tree outputs the term $+[q[+[1,1],+[1,1]],q[1,+[1,1]]]$.

Chapter 4

Results

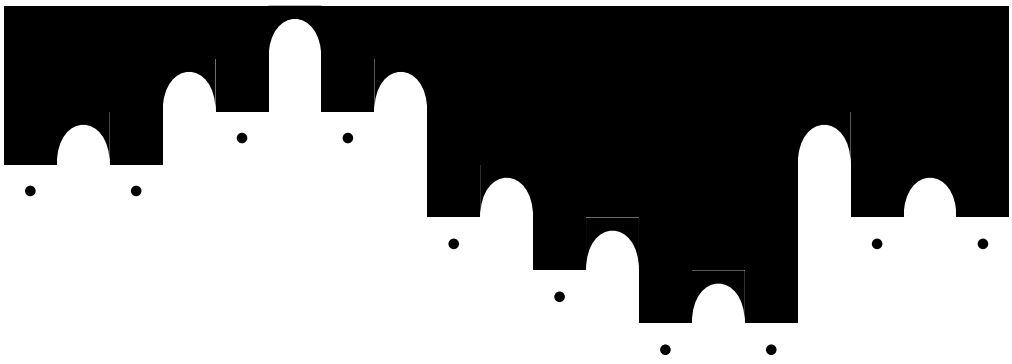
This chapter starts off by showing the mt transducer component in two example scenarios: picture generation and assembler code generation. Results from both scenarios are shown in Section 4.1. The chapter ends with Section 4.2, where some performance measures are given.

4.1 The Component in Example Scenarios

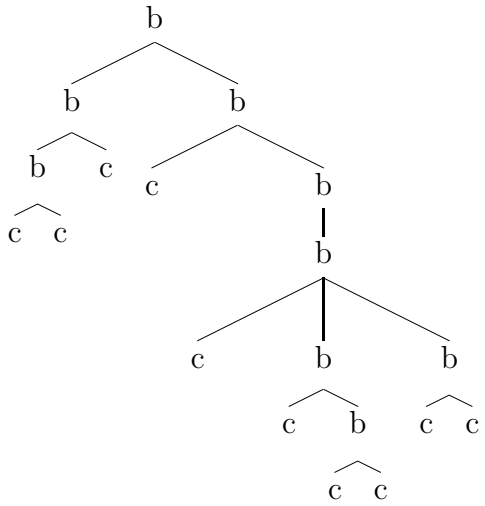
A TREEBAG generated graphical representation of a binary tree is shown in Figure 4.1. In TREEBAG, the binary tree is transformed by a mt transducer into a tree which is interpreted by a collage algebra as a graphical representation of the binary tree. The mt transducer uses a parameter to carry information on the width of the first subtree of a node so that the second subtree can be drawn by the collage-algebra at a correct distance from the first subtree.

Another interpretation of a tree as a picture, by a different mt transducer and collage algebra, is shown in Figure 4.2. Here the mt transducer transforms a monadic tree of height k into a tree with a root of rank two which has two monadic subtrees, each of height $k-1$, both followed by a symbol of rank two with two monadic subtrees, but now of height $k-2$, and so on (the monadic subtrees of height 0 become the leaves). In this translation the mt transducer uses parameters to keep track of the height of the monadic subtrees. See Figure 1.11 (c) and (d) on page 11 for a small example of how the mt transducer translates an example input tree.

A small assembler example, borrowed from [EV85], shows the mt trans-



(a)



(b)

Figure 4.1: (a) shows a picture representing the binary tree in (b). The symbols b and c are displayed as arches and dots, respectively.

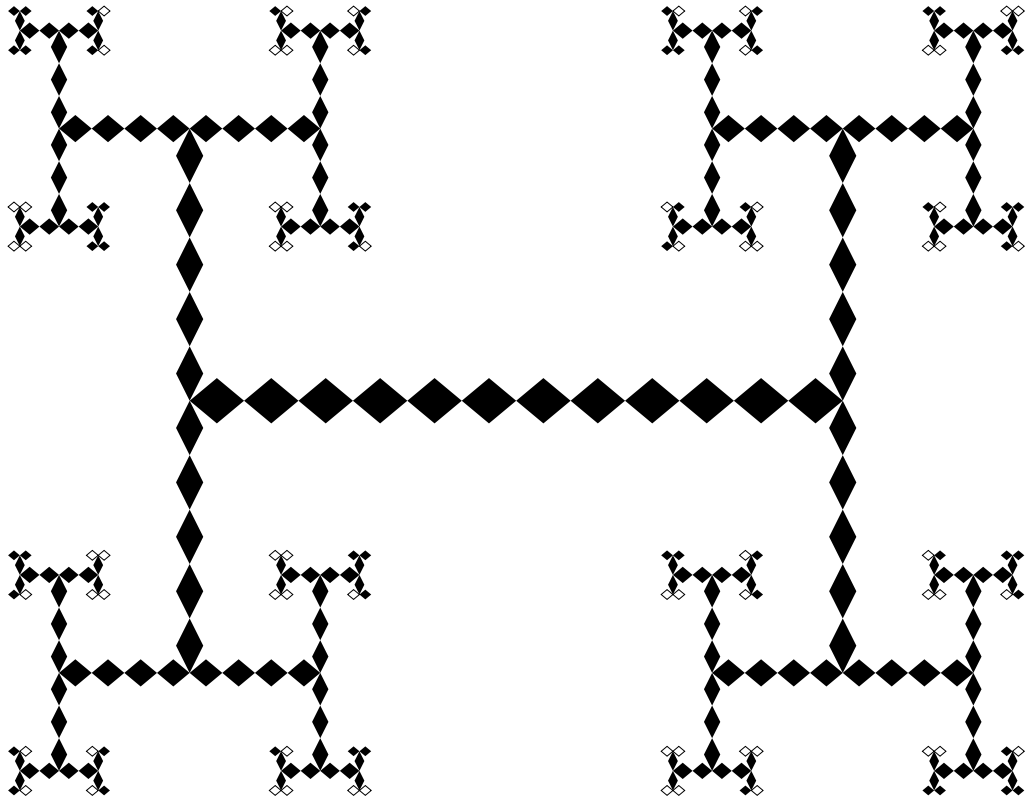


Figure 4.2: A picture representation of a tree over the alphabet $\{g^{(2)}, d^{(1)}, e^{(0)}, c^{(0)}\}$. Each intermediary node with a symbol d is represented by a black quadrangle and each g is deleted while both its subtrees are rotated 90 degrees to each side and decreased in size. The quadrangles representing leaves of symbol e and c are filled black and white, respectively. The root of the tree is located at the center of the picture while the leaves appear in small quadrangles on the outer edges.

ducer outputting trees whose yields form computer programs. The input of the mt transducer is a tree representing a binary number and the output is a tree which leaves represents assembler instructions for a stack machine. Together the instructions form a program which, when run to termination, stores the value of the binary number in the accumulator[1] of the stack machine. In the translation, the mt transducer uses a parameter to keep track of the significance of the current node in the binary number. Figure 4.3 shows such a binary number with the resulting assembler program.

The interested reader may have a look at Appendix A which contains component files for the mt transducer examples of this section.

4.2 Performance

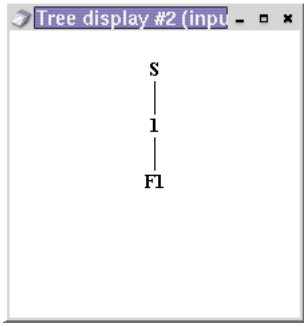
Even though performance was not the top priority of the implementation, it might be interesting to know how the `mtTransducer`, with the performance option enabled, compares to other Treebag components. This section discusses two such performance measures[2].

Figure 4.4 shows the performance results in a picture generating context, using the example translation from Figure 4.2. With the information from Figure 4.4 we can construct a more informative graph, Figure 4.5, which displays the *relative* time used by each component. From Figure 4.5 we see that the portion of the total execution time used by the mt transducer component decreases as the problem set increases.

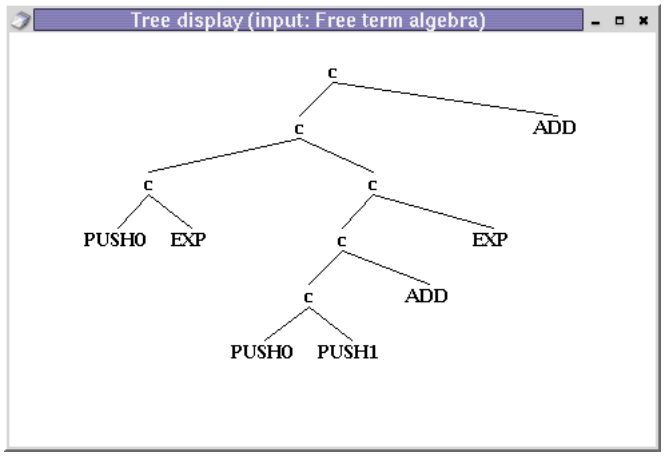
As mentioned in Section 2.3, for a given total deterministic mt transducer there exists a construction of a td transducer together with a YIELD algebra which can perform the same translation. E.g. recall the multiplication to addition example described in Section 2.3.2 for mt transducers and Section 2.2.3 for td transducers composed with YIELD. Since both a td transducer and YIELD component already exist in Treebag, it is interesting to compare their performance with the mt transducer component. Figure 4.6 gives performance measures on the mt transducer compared to the td trans-

[1] A stack machine has a single register, referred to as the accumulator.

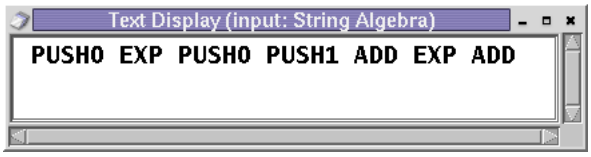
[2] For all test runs, the component was compiled and executed using version 1.5.0 of the Java Compiler and Runtime Environment (JRE), respectively. Execution time was measured with `System.nanoTime()` and the JRE had a maximum limit of 800 MB of heap space and ran on a PC running Linux with an Intel Pentium 4 HT 2.8 GHz CPU and 1024 MB RAM.



(a)



(b)



(c)

Figure 4.3: (a) shows the tree representing the binary number 11. (b) shows the resulting output tree of the mt transducer component and (c) the yield of that tree.

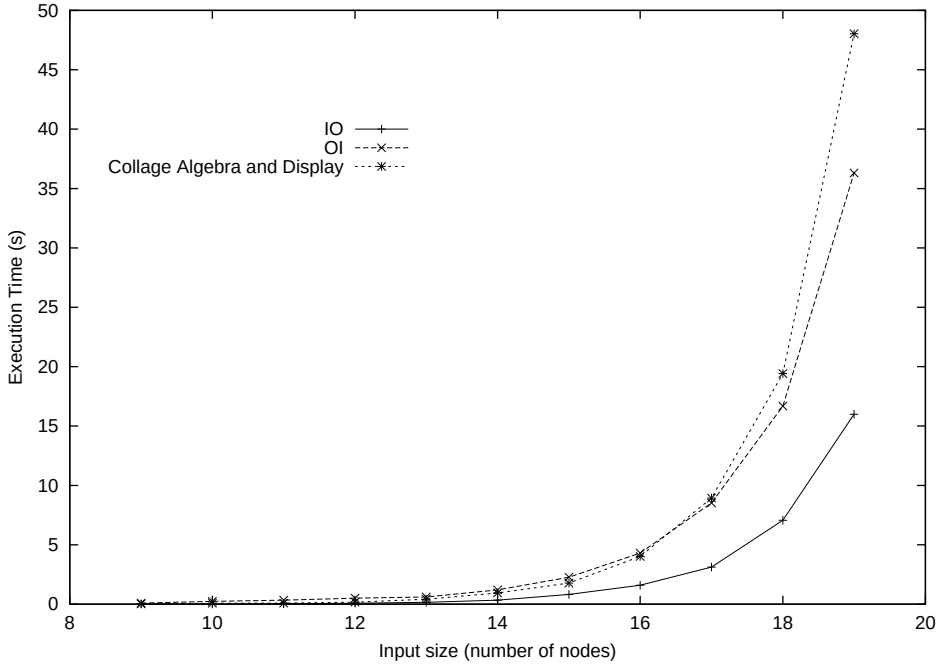


Figure 4.4: Execution times for a picture generating example, see Appendix A for details. "IO" and "OI" refers to the time consumed by the mt transducer (performance option enabled) in IO and OI mode, respectively. "Collage Algebra and Display" refers to the time used by the collage algebra and display to generate the picture from the mt transducer output. Note that the output trees of this translation are exponentially larger than the input trees and the exponential increase of execution time is therefore expected.

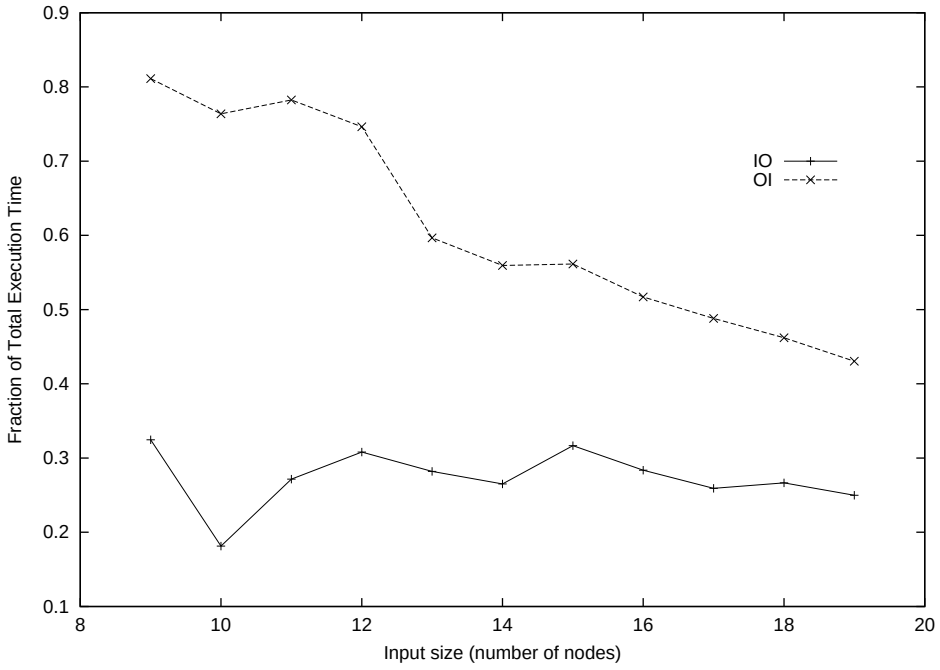


Figure 4.5: Ratio of the execution time of the `mtTransducer` in performance mode to the total time used by both the mt transducer and the collage algebra and display.

ducer composed with YIELD. At each point in Figure 4.6 we see that the execution time of the `mtTransducer` in IO mode is faster then the equivalent td transducer/YIELD construction and that the `mtTransducer` in OI mode is the slowest of them all. Since OI derivations copies its subtrees before processing them, it is not very surprising that OI derivations result in the worst performance of the three.

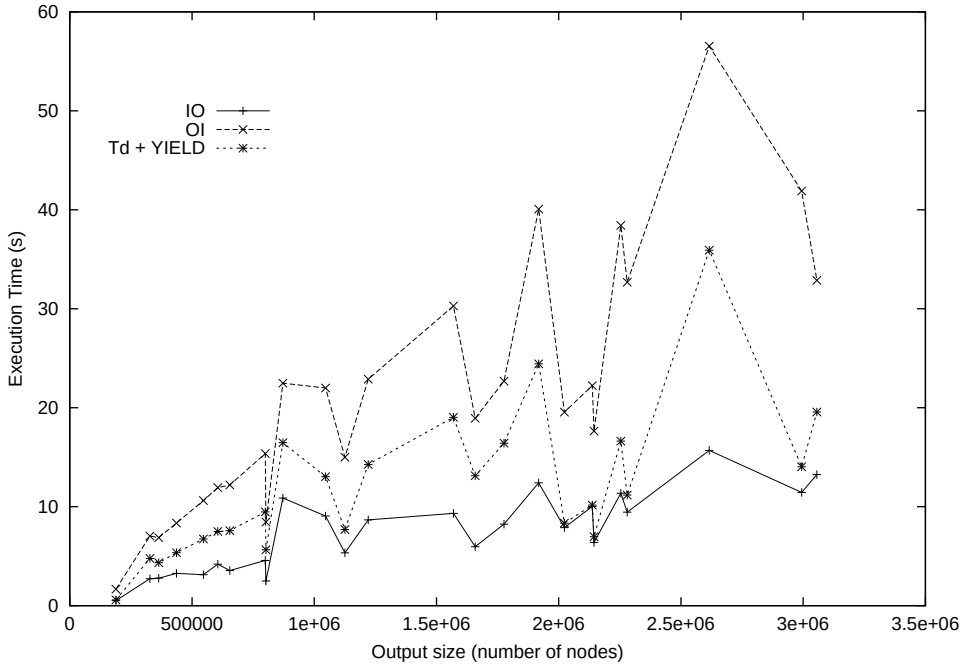


Figure 4.6: Execution times of three components performing the same translation on 25 different input trees. The output size of the tree is not a perfect estimate of the complexity of the translation and can explain the fluctuation of the graphs, e.g. one could consider the amount of multiplication operators in the input tree for a better estimate. Note that the overhead created by TREEBAG for moving the intermediary tree from the td transducer to YIELD is not included.

Chapter 5

Summary and Conclusions

Chapter 2 motivated the need for a mt transducer component in Treebag. The chapter also described how dead-ends could be efficiently avoided in non-deterministic mt transducers. The same algorithm used for the td transducer implementation could, slightly modified, be used for dead-end detection in IO derivations for a mt transducer. For OI derivations a new dead-end detection algorithm, somewhat more complex than the one for the IO case, was introduced. The chapter also mentioned some algorithmic improvements which can be used to improve the performance of a mt transducer implementation.

The mt transducer implementation was described in Chapter 3. The component can be used in two different modes, standard and performance mode. In standard mode, the user may issue commands such as single/parallel step and back to interactively observe the mt transducer performing derivation steps. The standard mode may be useful for testing new mt transducer definitions and may have an educational benefit when teaching students how mt transducers work. However, a very space consuming computation tree is used in standard mode to keep track of the derivation sequence. Consequently, for large problem sets the computation tree will be time consuming. Alternatively the user can, via the object files of the component, set the mt transducer in a performance mode, disabling the mt transducer from creating a computation tree. In performance mode the user obtains a more efficient mt transducer component at the expense of sacrificing the possibility to use the step and back commands. Dividing the component into those two modes should not prove too problematic for most types of usage since any translation large enough to cause performance problems in standard mode probably also has a too long derivation sequence for a normal user to be interested in

observing.

Finally Chapter 4 briefly mentioned two tree translation scenarios in TREEBAG where the mt transducer component was used. Chapter 4 also showed us that the component, in performance mode, will not become the performance bottleneck in a picture generating context. It was also shown that the component has a performance not much different from that of an equivalent construction of the td transducer component together with the YIELD component. However, the mt transducer component has certain advantages. As a concept for tree transformation, mt transducers are more intuitive and therefore easier to use. Other benefits are the possibility to switch between the IO and OI modes and the availability of stepwise derivations.

5.1 Future Work

There are several theoretical and practical problems related to the subject of this thesis which are interesting to study further:

- Proving the correctness of the dead end detection algorithms for IO and OI derivations.
- Improving the component by implementing the optimization mentioned on page 32 for OI dead end detection.
- Finding an efficient computation tree algorithm and implementing it, thus making the performance option obsolete.
- Extending the TREEBAG system with a class `extendedTreeTransducer` which contains more support for tree transducer implementations. E.g. functionality for some general tree substitutions and derivation steps. This would make it easier to implement some of the other classes of tree transducers, e.g. attributed tree transducers and macro attributed tree transducers [FV98], as components in TREEBAG.

5.2 Acknowledgement

The author would like to thank his thesis advisor, Frank Drewes, for his excellent suggestions regarding the algorithms for the implementation and

thorough proofreading of the report. In particular, Drewes has also contributed with the example translation displayed in Figure 4.1 and the collage algebra generating the picture in Figure 4.2, see also Section A.1.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Priciples, Techniques, and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, world student series edition edition, 1986. "The Dragon Book".

[CDG+02] Hubert Comon, Max Dauchet, Rmi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications – Chapter 1: Recognizable Tree Languages and Finite Tree Automata. Internet publication available at `www.grappa.univ-lille3.fr/tata`, 2002. Visited January 17, 2005.

[Dre98] Frank Drewes. TREEBAG—a tree-based generator for objects of various types. Report 1/98, Univ. Bremen, 1998.

[Dre01] Frank Drewes. TREEBAG Manual. Internet publication available at `http://www.informatik.uni-bremen.de/theorie/treebag/manual/`, 2001. Visited January 24, 2005.

[Dre05] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005. To appear.

[Eng77] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[Eng80] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 241–286. Academic Press, New York, 1980.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

[Fül81] Zoltán Fülöp. On attributed tree transducers. *Acta Cybern.*, 5:261–279, 1981.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics : Formal Models Based on Tree Transducers*. Springer-Verlag, 1998.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *"Introduction to Automata Theory, Language, and Computation"*. Addison–Wesley, 2nd edition edition, 2001.

[Mai74] Thomas S. E. Maibaum. A generalized approach to formal languages. *Journal of Computer and System Sciences*, 8:409–439, 1974. See Erratum in Journal of Computer and System Sciences 14(3), p. 369.

[Mic04] Sun Microsystems. Java$^{\text{TM}}$2 platform, standard edition, v.1.4.2 api specification. Web page available at `http://java.sun.com/j2se/1.4.2/docs/api/index.html`, December 2004. Visited December 9, 2004.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[Tha70] James. W. Thatcher. Generalized sequential machine maps. *Journal of Computer and System Sciences*, 4(4):339–367, 1970.

[Tha73] James W. Thatcher. Tree automata: An informal survey. In *Alfred V. Aho, Currents in the Theory of Computing, Prentice-Hall*, chapter 4, pages 143–172. 1973.

Appendix A

Examples

This Appendix contains three TREEBAG object files for initiating the mt transducer component. The object files in Section A.1, Section A.2 and Section A.3 are used for the example translations displayed in Figure 4.1, Figure 4.2 and Figure 4.3, respectively. Note that the object files in Section A.2 and Section A.3 are created from examples in [EV85].

A.1 Binary Tree

```
generators.mtTransducer("Macro Tree Transducer"):
(
  { b:2, c:0 },
  { down:1, .:2, leaf:0, l:1, sh:1, empty:0, connect:1 },
  { L:1, shift:2, line:2 },
  {
    L[b[x1,x2]] -> (down[(L[x1] . sh[shift[x1,L[x2]]])] . line[x1,connect[line[x2,empty]]]),
    L[c] -> leaf,

    shift[b[x1,x2],y1] -> sh[shift[x1,shift[x2,y1]]],
    shift[c,y1] -> sh[y1],

    line[b[x1,x2],y1] -> l[line[x1,line[x2,y1]]],
    line[c,y1] -> l[y1]
  },
  L
)
```

A.2 Monadic Tree

```
generators.mtTransducer("Macro Tree Transducer"):
(
  { f:1, a:0, b:0 },
  { g:2, d:1, e:0, c:0 },
  { p:2, qin:1},
  {
    qin[f[x1]] -> p[x1,qin[x1]],
    qin[a] -> e,
    qin[a] -> c,
    p[a,y1] -> g[y1, y1],
    p[b,y1] -> e,
    p[f[x1],y1] -> p[x1,d[y1]]
  },
  qin
)
```

A.3 Stack Machine Assembler

```
generators.mtTransducer("Macro Tree Transducer"):
(
  { S:1, 0:1, 1:1, F1:0 },
  { c:2, "ADD ":0, "EXP ":0, "PUSH0 ":0, "PUSH1 ":0 },
  { qin:1, q:2 },
  {
    qin[S[x1]] -> q[x1,"PUSH0 "],
    q[0[x1],y1] -> q[x1, c[c[y1,"PUSH1 "],"ADD "]],
    q[1[x1],y1] -> c[c[c[y1,"EXP "], q[x1,c[c[y1,"PUSH1 "],"ADD "]]], "ADD "],
    q[F1,y1] -> c[y1,"EXP "],
    q[F1,y1] -> c[y1,"EXP "]
  },
  qin
)
```