Bottum-up tree acceptors

Citation for published version (APA):

Document status and date:
Published: 01/01/1988

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.
Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 01. Dec. 2019
Bottom-up tree acceptors

by

C.Hemerik and J.P. Katoen

October, 1988
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
Bottom-up tree acceptors

C. Hemerik J. P. Katoen
Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, Netherlands

October 28, 1988

Abstract

This paper deals with the formal derivation of an efficient tabulation algorithm for table-driven bottom-up tree acceptors. Bottom-up tree acceptors are based on a notion of match sets. First we derive a naive acceptance algorithm using dynamic computation of match sets. Tabulation of match sets leads to an efficient acceptance algorithm, but tables may be so large that they can not be generated due to lack of space. Introduction of a convenient equivalence relation on match sets reduces this effect and improves the tabulation algorithm.

1 Introduction

Nowadays, many parts of a compiler can be generated automatically. For instance, automatic generation of lexical and syntactic analyzers using notations based on regular expressions and context-free grammars is commonly used (see e.g. [Aho]). However, much research is still going on in the field of universal code generator-generators, which take a description of a machine as input and deliver a (good) code generator for that machine.

Code generation forms an important subject in developing a compiler. Requirements traditionally imposed on a code generator are severe: the generated code must be correct and must utilize the resources of the machine (such as registers) efficiently. A particular and relevant issue in code generation is instruction selection. This forms the subject of the remainder of this section. Nature of the instruction set and addressing modes of the target machine determine the difficulty of instruction selection. As an example, we illustrate the instruction selection of an expression for a register machine with a very simple instruction set. First the addressing modes are presented.
1 INTRODUCTION

Example 1.1

<table>
<thead>
<tr>
<th>addressing mode</th>
<th>format</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate</td>
<td>#c</td>
<td>c</td>
</tr>
<tr>
<td>register</td>
<td>Ri</td>
<td>Ri</td>
</tr>
<tr>
<td>indexed</td>
<td>c(Ri)</td>
<td>M(c + Ri)</td>
</tr>
<tr>
<td>indirect</td>
<td>*Ri</td>
<td>M(Ri)</td>
</tr>
</tbody>
</table>

Here, \( M(a) \) denotes the contents of address \( a \); \( c \) is a constant and \( R_i \) is a register. Suppose our target machine supports the following instructions.

Example 1.2

<table>
<thead>
<tr>
<th>instruction</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) MOV #c, Ri</td>
<td>Ri := c</td>
</tr>
<tr>
<td>(2) MOV *Rj, Ri</td>
<td>Ri := M(Rj)</td>
</tr>
<tr>
<td>(3) MOV c(Rj), Ri</td>
<td>Ri := M(c + Rj)</td>
</tr>
<tr>
<td>(4) ADD Ri, Rj</td>
<td>Ri := Ri + Rj</td>
</tr>
</tbody>
</table>

Now consider the expression \( R_1 := c_1 + M(c_2 + R_2) \). Using the instructions given above we may derive an instruction sequence for this expression as follows. At each step the selected subexpression to code is underlined.

Example 1.3

<table>
<thead>
<tr>
<th>expression</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 := c_1 + M(c_2 + R_2) )</td>
<td>MOV #c_2, R_3</td>
</tr>
<tr>
<td>( R_1 := c_1 + M(R_3 + R_2) )</td>
<td>ADD R_2, R_3</td>
</tr>
<tr>
<td>( R_1 := c_1 + M(R_2) )</td>
<td>MOV *R_2, R_2</td>
</tr>
<tr>
<td>( R_1 := c_1 + R_2 )</td>
<td>MOV #c_1, R_1</td>
</tr>
<tr>
<td>( R_1 := R_1 + R_2 )</td>
<td>ADD R_1, R_2</td>
</tr>
</tbody>
</table>

Observe that the derivation above looks like the parsing of a string. By replacing the definition of an instruction, which is of the form \( R_i := \ldots \), by a production rule of the form \( R_i \rightarrow \ldots \), definitions (1)-(4) of example 1.2 may be considered as a code generation grammar. The above suggests to use traditional parsing techniques for code generation. For instance, Graham & Glanville use LR-parsing for instruction selection (see [Graham]). Main problem with this approach is the solution of the large number of parsing conflicts caused by the fact that code generation grammars are inherently highly ambiguous. For example, an alternative instruction sequence may be derived for the expression \( R_1 := c_1 + M(c_2 + R_2) \) as follows:

...
1 INTRODUCTION

Example 1.4

expression

\[ R_1 := c_1 + M(c_2 + R_2) \]
\[ R_1 := c_1 + R_2 \]
\[ R_1 := R_1 + R_2 \]

instruction

\[ MOV c_2(R_2), R_2 \]
\[ MOV #c_1, R_1 \]
\[ ADD R_1, R_2 \]

A way to overcome this problem could be to use a more general parsing method like Earley parsing, as suggested in [Christopher a], but the resulting space and time complexity is unacceptable for practical use in code generators.

Neither of these methods just mentioned takes into account a special property of code generation grammars, viz. that every operator symbol has a fixed rank. Using this fact leads to the idea of considering the tree representation of an expression, rather than its string representation. To this end, the code generation (string) grammar becomes a so-called tree grammar. For the instructions of example 1.2 the production rules, represented by trees, become:

Example 1.4

instruction

\[ MOV \#c, R_i \]
\[ MOV *R_j, R_i \]
\[ MOV c(R_j), R_i \]
\[ ADD R_i, R_j \]

tree representation

\[ R_i \rightarrow c \]
\[ R_i \rightarrow M \]
\[ R_i \rightarrow c + R_j \]

Several code generation algorithms based on tree grammars have been described [Aho, Christopher b, Turner], but a theoretical framework is painfully missing. This is the more remarkable as a well-developed theory of tree grammars and tree automata already exists for some twenty years [Brainerd, Doner, Rounds, Hoffmann]. A systematic treatment of this theory, aimed at code generation applications, is given in [Van Dinther]; a survey paper is in preparation [Hemerik].

In this paper we consider a particular class of tree acceptors, called deterministic bottom-up tree acceptors, which have a time complexity proportional to the size of the tree to be analyzed. They can easily be extended to bottom-up tree parsers. Our main purpose is to present algorithms...
for the efficient generation of compressed parse tables, and to show how these rather complex programs can be derived systematically.

The organization of this paper is as follows: in sections 2 and 3 we present a simplified treatment of the theory of tree grammars and deterministic bottom-up tree acceptors. Section 4 shows how the transition functions of the acceptor can be tabulated, thus leading to a linear time acceptance algorithm. In practical applications the size of the resulting tables may be prohibitive however. Therefore in section 5, using ideas of [Chase], an improved algorithm is described which generates compressed transition tables. Finally, section 6 contains some concluding remarks.

2 Tree grammars

In this section we define the basic concepts of the theory of tree grammars. Readers familiar with context-free (string-) grammars will notice that tree grammars are a generalization of context-free (string-) grammars.

**Definition 2.1 { ranked alphabet }**

A ranked alphabet is a pair \((V, r)\) such that \(V\) is a finite set and \(r \in V \rightarrow \mathbb{N}\)

Elements of \(V\) are called symbols and \(r(a)\) is called the rank of symbol \(a\). In the following the set \(V_n\) denotes the set of symbols with rank \(n\), that is, \(V_n = \{v \in V | r(v) = n\}\).

**Definition 2.2 { Tree(V, r), trees over a ranked alphabet }**

The set \(\text{Tree}(V, r)\) of trees over a ranked alphabet \((V, r)\) is the smallest set \(X\) such that:

\[
\begin{align*}
\bullet & \quad V_0 \subseteq X \\
\bullet & \quad \forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in X : a(t_1, \ldots, t_n) \in X
\end{align*}
\]

**Definition 2.3 { tree grammar }**

A tree grammar \(G\) is a 5-tuple \((N, V, r, P, S)\) such that:

\[
\begin{align*}
\bullet & \quad (N \cup V, r) \text{ is a ranked alphabet such that } \forall A \in N : r(A) = 0 \\
\bullet & \quad N \cap V = \emptyset \\
\bullet & \quad P \text{ is a finite subset of } N \times \text{Tree}(N \cup V, r) \\
\bullet & \quad S \in N
\end{align*}
\]
Elements of $N$, $V$, $P$ are called nonterminals, terminals, and production rules, respectively. $S$ is called the start symbol of $G$. Notational remark: upper-case letters are used to denote nonterminals, lower-case letters stand for terminals. An element $(A, t) \in P$ is usually written as $A \rightarrow t$; $A$ is sometimes called the left-hand side, $t$ the right-hand side of the production rule.

**Definition 2.4** \{ $\Rightarrow$, derivation-step (informal) \}

Let $(N, V, r, P, S)$ be a tree grammar. $\forall t_1, t_2 \in \text{Tree}(N \cup V, r)$:

$t_1 \Rightarrow t_2$ if $\exists (A \rightarrow \alpha) \in P$, such that $t_2$ can be obtained from $t_1$ by substituting $\alpha$ for one occurrence of $A$ in $t_1$.

$\Rightarrow$ is the reflexive and transitive closure of $\Rightarrow$. We say that $t$ is derivable from $A$ if $A \Rightarrow t$.

The set of trees, containing only terminals, derivable from the start symbol constitute the language that is generated by the corresponding tree grammar.

**Definition 2.5** \{ $\mathcal{L}$, language generated by a tree grammar \}

Let $G = (N, V, r, P, S)$ be a tree grammar.

- the function $L \in N \mapsto \mathcal{P}(\text{Tree}(V, r))$ is defined by:
  $\forall A \in N : L(A) = \{ t \in \text{Tree}(V, r) \mid A \Rightarrow t \}$

- the language generated by $G$ is $L(S)$

The tree grammar defined in the following example shall be used as a running example throughout this paper.

**Example 2.6**

Let $G = (N, V, r, P, A)$ be a tree grammar, where $N = \{ A, B \}$; $V = \{ a, b, c, d \}$; $r(a) = 2$; $r(b) = 1$; $r(c) = 0$; $r(d) = 0$;

$P = \{ (1) \ A \rightarrow a(b(c), B), \\
                           (2) \ A \rightarrow a(B, d), \\
                           (3) \ A \rightarrow c, \\
                           (4) \ B \rightarrow b(B), \\
                           (5) \ B \rightarrow A, \\
                           (6) \ B \rightarrow d \} \\

An alternative presentation of the elements of $P$ is given in figure 1. Some examples of derivations are:
3 TREE ACCEPTORS

Figure 1: Production rules represented as trees

\[ A \rightarrow a \]  \hspace{1cm} \[ A \rightarrow a \]  \hspace{1cm} \[ A \rightarrow c \]

\[ B \rightarrow B \]  \hspace{1cm} \[ B \rightarrow A \]  \hspace{1cm} \[ B \rightarrow d \]

Some elements of \( \mathcal{L}(G) \) are: \( c, a(b(c),d), a(a(b(c),b(d)),d) \).

\[ A \overset{(1)}{\rightarrow} a(b(c),B) \overset{(2)}{\rightarrow} a(b(c),d). \]
\[ A \overset{(3)}{\rightarrow} a(B,d) \overset{(4)}{\rightarrow} a(b(B),d) \overset{(5)}{\rightarrow} a(b(A),d) \overset{(6)}{\rightarrow} a(b(c),d). \]

3 Tree acceptors

A tree acceptor is a tree automaton which, given a tree grammar \( G = (N, V, r, P, S) \) and a tree \( t \in \text{Tree}(V, r) \), establishes whether \( t \in \mathcal{L}(G) \). In this section we consider a particular kind of tree acceptors, viz. deterministic bottom-up tree acceptors, although we shall not stress the automata-theoretic concepts. The basic idea underlying this kind of acceptor is to extract from the grammar \( G \) a set \( PS \) of patterns, i.e. subtrees of right-hand sides of production rules, and to compute for a tree \( t \) the match set \( MS_t(t) \) of patterns from which it may be derived.

The tree \( t \) is accepted if and only if \( S \in MS_1(t) \). As the match set of a computed tree can be simply computed from the match sets of its direct subtrees, an acceptor can be obtained which operates in time proportional to the size of the tree.

The following definitions are all relative to a given tree grammar \( G = (N, V, r, P, S) \). We assume that \( G \) has no useless terminals and nonterminals, i.e. every (non)terminal occurs in some tree derivable from \( S \) and for all \( A \in N : \mathcal{L}(A) \neq \emptyset \).

Definition 3.1 \( \{ \text{sub, subtree relation} \} \)

\[ \forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in \text{Tree}(N \cup V, r) : \forall i : 1 \leq i \leq n : t_i \text{ sub } a(t_1, \ldots, t_n) \]
sub* is the reflexive and transitive closure of sub.

Definition 3.2 \{ PS, pattern set \}
PS = \{ t ∈ Tree(N U V, r) | ∃A, t' : A ∈ N ∧ t' ∈ Tree(N U V, r) : (A → t') ∈ P ∧ t sub* t' \}

Notice that N ⊆ PS holds, since every nonterminal occurs in some tree derivable from S. The closure of a pattern s is the set of patterns containing s and the nonterminals from which s is derivable. Similarly, the closure of a set of patterns is defined as follows.

Definition 3.3 \{ closure \}
\begin{itemize}
  \item the function closure ∈ P(PS) → P(PS) is defined by :
  \item ∀s ∈ P(PS) : closure(s) = s ∪ \{ A ∈ N | ∃α ∈ s : A \xrightarrow{α} A \}
\end{itemize}

Apparently, for all s ∈ P(PS) : closure(s) ⊆ PS. There are various ways to handle the acceptance problem. One possible way, commonly known as the bottom-up method (or bottom-up pattern matching), is to derive the start symbol S starting with a given tree t. The bottom-up method relies on the notion of match sets, sets of subpatterns that match at a particular tree node. These sets are defined recursively as :

Definition 3.4 \{ MS₁, match set \}
\begin{itemize}
  \item the function MS₁ ∈ Tree(V, r) → P(PS) is defined by :
  \item ∀a ∈ V₀ : MS₁(a) = closure(\{a\})
  \item ∀n : 1 ≤ n : ∀a ∈ Vₙ : ∀t₁, ..., tₙ ∈ Tree(V, r) :
    MS₁(a(t₁, ..., tₙ)) = closure(\{a(p₁, ..., pₙ) ∈ PS | ∀i : 1 ≤ i ≤ n : p_i ∈ MS₁(t_i)\})
\end{itemize}

The relevance of match sets is stated in the following lemma.

Lemma 3.5 ∀t ∈ Tree(V, r) : MS₁(t) = \{ t' ∈ PS | t' \xrightarrow{sub*} t \}

Proof : by structural induction over Tree(V, r).

1. base step : let a ∈ V₀,
   
   MS₁(a)
   
   = \{ \text{definition 3.4} \}
3 TREE ACCEPTORS

\[
\text{closure(\{a\})} \\
= \{\text{definition 3.3}\} \\
\{a\} \cup \{A \in N \mid A \rightarrow a\} \\
= \{N \subseteq PS\} \{a \in PS \land a \rightarrow a\} \\
\{t' \in PS \mid t' \rightarrow a\}
\]

2. induction step: let \(a \in V_n, 1 \leq n\), then

\[
\text{MS}_1(a(t_1, \ldots, t_n)) \\
= \{\text{definition 3.4}\} \\
\text{closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall i : 1 \leq i \leq n : p_i \in MS_1(t_i)\})} \\
= \{\text{induction hypothesis}\} \\
\text{closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall i : 1 \leq i \leq n : p_i \in \{t' \in PS \mid t' \rightarrow t_i\}\})} \\
= \{\text{set calculus}\} \\
\text{closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall i : 1 \leq i \leq n : p_i \rightarrow t_i\})} \\
= \{\text{definition 2.4}\} \\
\text{closure(\{a(p_1, \ldots, p_n) \in PS \mid a(p_1, \ldots, p_n) \rightarrow a(t_1, \ldots, t_n)\})} \\
= \{\text{definition 3.3}\} \{\forall s \in \mathcal{P}(PS) : \text{closure}(s) \subseteq PS\} \\
\{t' \in PS \mid t' \rightarrow a(t_1, \ldots, t_n)\}
\]

\[\square\]

Lemma 3.6 \(\forall t \in \text{Tree}(V, r) : S \in MS_1(t) \Leftrightarrow t \in \mathcal{L}(S)\)

**Proof**: use definition 2.5 and lemma 3.5. \(\square\)

So, by computing \(MS_1(t)\) for a given tree \(t\), it is rather simple to decide whether \(t\) belongs to the language generated by \(G\) or not.

**Example 3.7**

Consider the grammar of example 2.6. Its pattern set is:

\[PS = \{a(b(c), B), b(c), c, B, a(B, d), d, b(B), A\}.\]

For \(t = a(b(c), b(a(d, d)))\) bottom-up computation of \(MS_1(t)\) is depicted in figure 2, where each node of \(t\) is annotated with the match set of the tree rooted at that node.

It shows that \(A \in MS_1(t)\), hence (see lemma 3.6), \(t \in \mathcal{L}(A)\). \(\square\)

Notational remark: elements of a match set added by a closure operation are separated from other elements by a semicolon.
4 Tabulation of match sets

A program computing $MS_1$ is easy implementable, but very inefficient. At each determination of the acceptance of a tree, match sets (and closures) must be recalculated. Fortunately, since $\mathcal{P}(PS)$ is a finite set (due to the fact that $PS$ is finite) the number of match sets is finite. This gives the possibility of tabulation of match sets. Observe that $MS_1(a(t_1, \ldots, t_n))$ is of the form $f_a(MS_1(t_r), \ldots, MS_1(t_n))$, where $f_a$ is defined as follows.

**Definition 4.1** \{ $f_a$, transition function for symbol $a$ \}

- $\forall a \in V_0: f_a \in \mathcal{P}(PS)$, where $f_a = \text{closure}(\{a\})$
- $\forall n: 1 \leq n: \forall a \in V_n: \forall s_1, \ldots, s_n \in \mathcal{P}(PS): f_a \in \mathcal{P}(PS)^n \rightarrow \mathcal{P}(PS)$, where $f_a(s_1, \ldots, s_n) = \text{closure}(\{a(p_1, \ldots, p_n) \in PS | \forall i: 1 \leq i \leq n: p_i \in s_i\})$

Obviously all $f_a$'s have a finite domain (since the number of match sets is finite), so we may tabulate $f_a$. This means that we will have an $n$-dimensional table for a symbol of rank $n$. We do not have to tabulate $f_a$ for the entire powerset $\mathcal{P}(PS)$, but only for its reachable part, i.e. the smallest set $Z \subseteq \mathcal{P}(PS)$ closed under all $f_a$'s.

To compute the reachable part $Z$ and the tabulation of $f_a$, $T_a \in Z^n \rightarrow Z$, where $n = r(a)$, the standard reachability algorithm (see e.g. [Rem]) is used. This leads to the following algorithm, which is called $A_1$.
\[
\begin{align*}
\text{var } & x, y : \mathcal{P}(PS) \\
& Z, W, G := \emptyset, \mathcal{P}(PS), \emptyset \\
& \text{for all } a \in V_0 \\
& \quad \text{do } y := f_a; W, G := W \setminus \{y\}, G \cup \{y\}; T_a := y \text{ od} \\
& \text{for all } a \in V \setminus V_0 \\
& \quad \text{do } \| \text{var } n : \mathcal{N} \\
& \quad \quad n := r(a) \\
& \quad \quad \text{for all } (s_1, \ldots, s_n) \in (Z \cup \{x\})^n \setminus Z^n \\
& \quad \quad \quad \text{do } y := f_a(s_1, \ldots, s_n) \\
& \quad \quad \quad \quad \text{if } y \in W \rightarrow W, G := W \setminus \{y\}, G \cup \{y\} \\
& \quad \quad \quad \quad \text{if } y \not\in W \rightarrow \text{skip} \\
& \quad \quad \quad \quad \text{od} \\
& \quad \quad T_a(s_1, \ldots, s_n) := y \\
& \quad \text{od} \\
& \quad \text{od} \\
& \quad G, Z := G \setminus \{x\}, Z \cup \{x\}\]
\end{align*}
\]

To allow indexing of transition tables with numbers rather than match sets, we introduce an enumeration \( E \in \mathcal{N} \rightarrow_p \mathcal{Z} \) of match sets\(^2\). \( E \) is an injection. The definition of transition tables is stated in terms of the enumeration as follows.

**Definition 4.2** \( \{ T_a, \text{transition table for symbol } a \} \)

- \( \forall a \in V_0 : T_a \in \text{dom}(E) \), where \( E(T_a) = f_a \)

- \( \forall n : 1 \leq n : \forall a \in V_n : \forall s_1, \ldots, s_n \in \text{dom}(E) : T_a \in \text{dom}(E)^n \rightarrow \text{dom}(E) \), where 
  \( E(T_a(s_1, \ldots, s_n)) = f_a(E(s_1), \ldots, E(s_n)) \)

\( \Box \)

The corresponding definition of match set is now.

\(^2\)we use \( \rightarrow_p \) to denote a partial function
Definition 4.3 \{ MS_2, \text{match set} \}

- The function $MS_2 \in Tree(V, r) \rightarrow \text{dom}(E)$ is defined by:

  - $\forall a \in V_0 : MS_2(a) = T_a$
  - $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(V, r) :$
    
    $$MS_2(a(t_1, \ldots, t_n)) = T_a(MS_2(t_1), \ldots, MS_2(t_n))$$

The correspondence between $MS_1$ and $MS_2$ is stated in the following lemma.

Lemma 4.4 \forall t \in Tree(V, r) : MS_1(t) = E(MS_2(t))

Proof: by structural induction over $Tree(V, r)$, using definitions 3.4 and 4.1-4.3.

Match sets containing the start symbol $S$ have a special meaning (see lemma 3.6) and are called accepting states.

Definition 4.5 \{ F, \text{accepting states} \}

$$F = \{ n \in \text{dom}(E) \mid S \in E(n) \}$$

Lemma 4.6 \forall t \in Tree(V, r) : t \in L(S) \Leftrightarrow MS_2(t) \in F

Proof: use definition 4.5 and lemmata 4.4 and 3.6.

As can be observed from definition 3.3, taking the closure of a set of patterns just consists of deriving some left-hand sides of production rules. This calculation can be simplified by tabulating the closure of nonterminals in a table $N$ closure. Formally:

Definition 4.7 \{ N\text{-}closure \}

- $N\text{-}closure \in N \rightarrow \mathcal{P}(N)$

- $\forall A \in N : N\text{-}closure(A) = \{ B \in N \mid B \overset{*}{\rightarrow} A \}$

The $N$ closure of a nonterminal is nothing more than determining the reflexive and transitive closure of $P \cap N \times N$. We use Warshall’s algorithm to calculate the transitive closure.

The relation between closure and $N$ closure is stated in lemma 4.8.

Lemma 4.8 \forall s \in \mathcal{P}(PS) : closure(s) = s \cup (\bigcup A, \alpha \mid (A \rightarrow \alpha) \in P \land \alpha \in s : N\text{-}closure(A))

Proof: use definitions 3.3, 2.4, and 4.7.

The elaborated version of algorithm $A_1$ (named $A_2$) is presented below. Here, the sets $Z, G,$ and $W$ are characterized by $\{ E(i) \mid 0 \leq i < p \}, \{ E(i) \mid p \leq i < q \},$ and $\mathcal{P}(PS) \setminus (Z \cup G)$, respectively.
con $G = (N, V, r, P, S)$: tree grammar
\[ \text{var } E : N \rightarrow \mathcal{P}(\mathcal{P}(S)) \]
\[ F : \mathcal{P}(N) \]
\[ p, q : N \]
\[ T_a : N \]
\[ T_a : N^n \rightarrow N \]
\[ N\text{closure} : N \rightarrow \mathcal{P}(N) \]
\[ \text{proc compute_N}\text{closure} = \]
\[ \text{for all } A \in N \text{ do } N\text{closure}(A) := \{ B \in N \mid (B \rightarrow A) \in P \} \text{ od} \]
\[ \text{for all } B \in N \]
\[ \text{do for all } A \in N \]
\[ \text{do if } B \in N\text{closure}(A) \rightarrow N\text{closure}(A) := N\text{closure}(A) \cup N\text{closure}(B) \]
\[ \text{if } B \notin N\text{closure}(A) \rightarrow \text{skip} \]
\[ \text{od (second, reflexive closure *)} \]
\[ \text{for all } A \in N \text{ do } N\text{closure}(A) := N\text{closure}(A) \cup \{ A \} \text{ od} \]
\[ \]
\[ \text{func closure =} \]
\[ (\lambda s : \mathcal{P}(PS) \mid \mathcal{P}(PS)) \]
\[ \text{var } r : \mathcal{P}(PS) \]
\[ r := s \]
\[ \text{for all } (A \rightarrow \alpha) \in P \]
\[ \text{do if } \alpha \in s \rightarrow r := r \cup N\text{closure}(A) \]
\[ \text{if } \alpha \notin s \rightarrow \text{skip} \]
\[ \text{od} \]
\[ r \]
\[ \]
\[ (* \text{main program *} ) \]
\[ \text{compute_N}\text{closure}() \]
\[ q := 0; p := 0 \]
\[ \text{for all } a \in V_0 \]
\[ \text{do } E(q) := \text{closure}\{a\} (* \text{new match set *}) \]
\[ T_a, q := q, q + 1 \]
Given the transition tables, the acceptance problem is easily solved: some simple table look-ups do the job. The table-driven acceptor is described by the following algorithm. Time complexity of this program is proportional to the size of the tree.
5 OPTIMIZED TABULATION

```
// con F : \mathcal{P}(N)
; T_a : N
; T_a : N^n \rightarrow N
; t : Tree(V, r)

; var accepted : bool

:func ms_2 =
( \{ t : Tree(V, r) | N
| if t :: a \rightarrow T_a
| t :: (t_1, ..., t_n) \rightarrow T_a(ms_2(t_1), ..., ms_2(t_n))
fi
)
| accepted := ms_2(t) \in F
```

Example 4.9
Consider the grammar of example 2.6. The tables generated by algorithm A2 are:

<table>
<thead>
<tr>
<th>E</th>
<th>match set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; A, B}</td>
</tr>
<tr>
<td>1</td>
<td>{d; B}</td>
</tr>
<tr>
<td>2</td>
<td>{a(B, d); A, B}</td>
</tr>
<tr>
<td>3</td>
<td>{b(c), b(B); B}</td>
</tr>
<tr>
<td>4</td>
<td>{b(B); B}</td>
</tr>
<tr>
<td>5</td>
<td>{a(b(c), B); A, B}</td>
</tr>
<tr>
<td>6</td>
<td>{a(b(d), a(b(c), B); A, B}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>match set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; A, B}</td>
</tr>
<tr>
<td>1</td>
<td>{d; B}</td>
</tr>
<tr>
<td>2</td>
<td>{a(B, d); A, B}</td>
</tr>
<tr>
<td>3</td>
<td>{b(c), b(B); B}</td>
</tr>
<tr>
<td>4</td>
<td>{b(B); B}</td>
</tr>
<tr>
<td>5</td>
<td>{a(b(c), B); A, B}</td>
</tr>
<tr>
<td>6</td>
<td>{a(b(d), a(b(c), B); A, B}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T_a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3 2 2 2 2 2 2 0 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 3 2 2 2 2 2 2 1 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2 2 2 2 2 2 2 2 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 3 2 2 2 2 2 2 3 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 6 7 2 6 6 6 6 6 4 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 2 3 2 2 2 2 2 2 5 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 2 3 2 2 2 2 2 2 6 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 2 3 2 2 2 2 2 2 7 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ T_c = 0, T_d = 1; F = \{0, 3, 6, 7\} \]

Table-driven bottom-up accepting of an input tree, for instance \( t = a(b(c), b(a(d, d))) \) proceeds as demonstrated in figure 3. Each node of \( t \) is now annotated with a number corresponding to the match set as it was annotated with in figure 2.
Since \( 6 \in F, t \in \mathcal{L}(G) \).
\[ \Box \]

5 Optimized tabulation

In practice, code generation tree grammars (see introduction) are rather extensive. This means that transition tables may be very large. Compression of these tables can be applied after com-
The optimization is based on an equivalence relation on match sets. The basic idea of the equivalence relation is the observation that some patterns only occur as j-th subtree of a tree labelled with a symbol of rank \( n \) \((n \geq j)\). Main advantage is that with generation of match sets one can iterate over the equivalence classes instead of the match sets. This is quite lucrative, provided that the mapping of match sets on equivalence classes is not (nearly) a bijection (then no improvement is made).

The idea of this optimization originated with David Chase, but he only gave an informal treatment of his ideas (see [Chase]). Here, we derive an improved algorithm for the generation of match sets based on the material presented in [Chase].

The j-th childset of a symbol, say \( a \), \( 1 \leq j \leq r(a) \), is the set of patterns that appear as j-th subtree of a tree in \( PS \) labelled with \( a \). Formally:

**Definition 5.1** \( \{ CS_{a,j}, j \text{-th childset of symbol } a \} \)

\[
\forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n : \forall t_1, \ldots, t_n \in \text{Tree}(N \cup V, r) : \quad CS_{a,j} = \{ t \in \text{Tree}(N \cup V, r) | \exists t' \in PS : t' :: a(t_1, \ldots, t_n) \land t_j = t \}
\]

**Example 5.2**
The childsets of the symbols of the grammar of example 2.6 are: \( CS_{a,1} = \{ b(c), B \} \), \( CS_{a,2} = \{ B, d \} \), and \( CS_{b,1} = \{ c, B \} \).

**Lemma 5.3** \( \forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n : CS_{a,j} \subseteq PS \)

**Proof**: \( PS \) is closed under taking subtrees.
Using the childsets we may refine the definition of match set as follows.

**Definition 5.4** \{ *MS*ₙ, match set \}  
- the function \( MSₙ \in Tree(V, r) \rightarrow \mathcal{P}(PS) \) is defined by:
- \( \forall a \in V_0 : MSₙ(a) = \text{closure}\{a\} \)
- \( \forall n : 1 \leq n : \forall a \in V_n : \forall t₁, \ldots, tₙ \in Tree(V, r) : MSₙ(a(t₁, \ldots, tₙ)) = \text{closure}\{(a(p₁, \ldots, pₙ) \in PS | \forall i : 1 \leq i \leq n : pᵢ \in MSₙ(tᵢ) \cap CSₙ)\} \)

The only difference with definition 3.4 is the intersection with \( CSₙ \). This does not affect the value of \( MSₙ(a(t₁, \ldots, tₙ)) \), because the only patterns missing from \( MSₙ(tᵢ) \cap CSₙ \) are those patterns that do not appear as the i-th subtree of a tree in \( PS \) labelled with \( a \).

For the same reasons as mentioned in section 4 a program computing \( MSₙ \) is easy implementable, but rather inefficient. Again, tabulation is possible. Observe that \( MSₙ(a(t₁, \ldots, tₙ)) \) is now of the form \( f_a(gₙ₁(MSₙ(t₁)), \ldots, gₙₙ(MSₙ(tₙ))) \) where \( f_a \) is defined as before (see definition 4.1) and \( gₙₖ \) is defined as follows.

**Definition 5.5** \{ *gₙₖ*, map function for j-th child of symbol a \}  
- the function \( gₙₖ \in \mathcal{P}(PS) \rightarrow \mathcal{P}(PS) \), is defined by:
- \( \forall s \in \mathcal{P}(PS) : gₙₖ(s) = s \cap CSₙ \)

For practical reasons we use an enumeration \( E \in \mathcal{N} \rightarrow \mathcal{P}(PS) \) of match sets. The definition of transition tables is stated in terms of the map function and the enumeration \( E \) as follows.

**Definition 5.6** \{ *Tₙ*, transition table for symbol a \}  
- \( \forall a \in V_0 : Tₙ \in \text{dom}(E), \text{ where } E(Tₙ) = f_a \)
- \( \forall n : 1 \leq n : \forall a \in V_n : \forall s₁, \ldots, sₙ \in \text{dom}(E) : Tₙ \in \text{dom}(E) \rightarrow \text{dom}(E), \text{ where } E(Tₙ(s₁, \ldots, sₙ)) = f_a(gₙ₁(E(s₁)), \ldots, gₙₙ(E(sₙ))) \)

Apparently, the transition tables are similar with those defined in definition 4.2.

An important observation is that intersection of a match set with some childset \( CSₙ \), for some symbol \( a \in V_n, 1 \leq n \) and \( j : 1 \leq j \leq n \), induces an equivalence relation over match sets.
Definition 5.7 { equivalence relation $\approx_{a,j}$ }
$\forall s, s' \in \mathcal{P}(\mathcal{S}) : \forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n : s \approx_{a,j} s' \iff s \cap CS_{a,j} = s' \cap CS_{a,j}$

Definition 5.8 { equivalence class $\varepsilon_{\approx_{a,j}}$ }
$\forall s \in \mathcal{P}(\mathcal{S}) : \forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n : \varepsilon_{\approx_{a,j}}(s) = \{s' \in \mathcal{P}(\mathcal{S}) | s \approx_{a,j} s'\}$

In other words: the equivalence class $\varepsilon_{\approx_{a,j}}(s)$ is the set of all match sets that are equivalent (under $\approx_{a,j}$) to $s$. An equivalence class $\varepsilon_{\approx_{a,j}}(s)$ is represented by $s \cap CS_{a,j}$, which is called the representer-set of $\varepsilon_{\approx_{a,j}}(s)$.

Tabulation of representer-sets is possible, since the number of equivalence classes is finite. We introduce for all $n : 1 \leq n$, for all $a \in V_n$, for all $j : 1 \leq j \leq n$ an enumeration $Ra,j \in \mathcal{N} \rightarrow \mathcal{P}(\mathcal{S})$ of representer-sets. The mapping of (the enumeration of) match sets on (the enumeration of) representer-sets is performed by an index map table $\mu_{a,j}$.

Definition 5.9 { $\mu_{a,j}$, index map table for $j$-th child of symbol $a$ }
$\forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n :$

- the function $\mu_{a,j} \in \text{dom}(E) \longrightarrow \text{dom}(Ra,j)$ is defined by:
- $\forall s \in \text{dom}(E) : R_{a,j}(\mu_{a,j}(s)) = E(s) \cap CS_{a,j}$

Notice that $\mu_{a,j}$ is, in fact, nothing else than the tabulation of $g_{a,j}$, defined above. This means that transitions have to be tabulated for representer-sets only (instead of match sets). This is reflected in the following definition.

Definition 5.10 { $T'_a$, transition table for symbol $a$ }
- $\forall a \in V_0 : T'_a \in \text{dom}(E)$, where $E(T'_a) = f_a$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall j : 1 \leq j \leq n : \forall s_1, \ldots, s_n \in \text{dom}(E) :$
  - $\mu_{a,j} \in \text{dom}(E) \longrightarrow \text{dom}(Ra,j)$
  - $T'_a \in (\text{dom}(Ra,1) \times \ldots \times \text{dom}(Ra,n)) \longrightarrow \text{dom}(E)$
  - $E(T'_a(\mu_{a,1}(s_1), \ldots, \mu_{a,n}(s_n))) = f_a(g_{a,1}(E(s_1)), \ldots, g_{a,n}(E(s_n)))$

The corresponding definition of match set is changed into:

Definition 5.11 { $M S_2$, match set }
- $M S_2 \in \text{Tree}(V, r) \longrightarrow \text{dom}(E)$
• \( \forall a \in V_0 : MS_2(a) = T'_a \)

• \( \forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots , t_n \in Tree(V, r) : \)
  \( MS_2(a(t_1, \ldots , t_n)) = T'_a(\mu_{a,1}(MS_2(t_1)), \ldots , \mu_{a,n}(MS_2(t_n))) \)

Notice that \( MS_2 \) is still related to \( MS_1 \) by lemma 4.4.

Using the definitions above and the invariants given below we may derive a tabulation algorithm (named \( A_3 \)), which is presented here. First, we give the invariants of the program.

**P1**: \((\forall n : 1 \leq n : \forall a \in V_n : new_a = (\exists j : 1 \leq j \leq n : p_{a,j} < q_{a,j}))\)

**P2**: \(0 \leq p \leq q \land (\forall n : 1 \leq n : \forall a \in V_n : (\forall j : 1 \leq j \leq n : 0 \leq p_{a,j} \leq q_{a,j}) \land (\forall i : 0 \leq i < q \land 0 \leq \mu_{a,j}(i) < q_{a,j} : R_{a,j}(\mu_{a,j}(i)) = E(i) \cap CS_{a,j})) \land (\forall i_1, \ldots , i_n : \forall j : 1 \leq j \leq n : 0 \leq i_j < p \land 0 \leq \mu_{a,j}(i_j) < p_{a,j} \land 0 \leq T'_a(\mu_{a,1}(i_1), \ldots , \mu_{a,n}(i_n)) < q : \)

\[ E(T'_a(\mu_{a,1}(i_1), \ldots , \mu_{a,n}(i_n))) = f_a(g_a, E(i_1)), \ldots , g_a, E(i_n)))) \]

\[| con G = (N, V, r, P, S) : tree grammar\]

\[ ; var E : N \rightarrow P(PS) ; F : P(N) \]
\[ ; T'_a : N \rightarrow P(PS) \]
\[ ; T'_a : N \rightarrow P(PS) \]
\[ ; new_a : bool \]
\[; proc compute_N_closure (* see algorithm A2 *) \]
\[; func closure (* see algorithm A2 *) \]
\[; proc compute_childsets = \]
\[| var j, n : N \]
\[| for all a \in V \setminus V_0 \]

\[| con G = (N, V, r, P, S) : tree grammar\]

\[ ; var E : N \rightarrow P(P_S) \]
\[ ; F : P(N) \]
\[ ; T'_a : N \rightarrow P(N) \]
\[ ; T'_a : N \rightarrow P(N) \]
\[ ; new_a : bool \]
\[ ; N_{\text{closure}} : N \rightarrow P(N) \]
\[ ; proc compute_N_{\text{closure}} (* see algorithm A2 *) \]
\[ ; func closure (* see algorithm A2 *) \]
\[ ; proc compute_childsets = \]
\[| var j, n : N \]
\[| for all a \in V \setminus V_0 \]
do $j, n := 1, r(a)$
  ;do $j \neq n + 1$ --- $CS_{a,j} := \emptyset ; j := j + 1$ od
od
;for all $a(t_1, \ldots, t_n) \in PS$
  do $j := 1$
    ;do $j \neq n + 1$ --- $CS_{a,j} := CS_{a,j} \cup \{t_j\}$ od
od
]

; proc compute_reprsets =
  (↓ p : N
  || (\* compute representer-sets of matchset $E(p)$ for all $a \in V \setminus V_0 \*$)
  (\* and fill $\mu_{a,j}$-tables, $1 \leq j \leq r(a)$, for $E(p)$ \*)
  for all $a \in V \setminus V_0$
  do || var $j, n : N$
    \| $j, n := 1, r(a)$
    ;do $j \neq n + 1$ --- $R_{a,j}(q_{a,j}) := E(p) \cap CS_{a,j}$
    \| var $k : N$
      || $k := 0$; do $R_{a,j}(k) \neq R_{a,j}(q_{a,j})$ --- $k := k + 1$ od
      ;if $k \neq q_{a,j}$ --- skip
      \| $k = q_{a,j}$ --- (\* new representer-set \*)
      \| $q_{a,j}, new_a := q_{a,j} + 1, true$
    \| fi
    ;$\mu_{a,j}(p) := k$
    \| $j := j + 1$
  od
  \||
od
||
(\* main program \*)
| compute_N_closure()
| compute_childsets()
; for all $a \in V \setminus V_0$
  do || var $j, n : N$
    \| $j, n := 1, r(a)$; do $j \neq n + 1$ --- $q_{a,j} := 0 ; p_{a,j} := 0 ; j := j + 1$ od
  \|
| $new_a := false$
The table-driven acceptor is described by algorithm \( A_4 \) which is presented below.
\[ \begin{align*}
\| \text{con } F : \mathcal{P}(N) \\
; & \quad T'_a : N \\
; & \quad T'_a : N^n \rightarrow N \\
; & \quad \mu_{a,j} : N \rightarrow N \\
; & \quad t : \text{Tree}(V, r) \\
; & \var accepted : \text{bool} \\
; \text{func } mS_2 = \\
\quad (t : \text{Tree}(V, r) | N) \\
\quad | \text{if } t :: a \rightarrow T'_a \\
\quad \quad t :: (a(t_1, \ldots, t_n) \rightarrow T'_a(\mu_{a,1}(mS_2(t_1)), \ldots, \mu_{a,n}(mS_2(t_n)))) \\
\quad \quad \text{fi} \\
\quad | accepted := mS_2(t) \in F \\
\| 
\end{align*} \]

Compare the results given in the following example with those of example 4.9.

**Example 5.12**

Consider again our running example and consider the childsets of our grammar as given in example 5.2. The tables generated by algorithm $A_3$ are:

<table>
<thead>
<tr>
<th>$E$</th>
<th>match set</th>
<th>$\mu_{a,1}$</th>
<th>$\mu_{a,2}$</th>
<th>$\mu_{b,1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; A, B}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{d; B}</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>\ø</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>{a(B, d) ; A, B}</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>{b(c), b(B) ; B}</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>{b(B) ; B}</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>{a(b(c), B) ; A, B}</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>{a(B, d), a(b(c), B) ; A, B}</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
T'_a & 0 & 1 & 2 \\
\hline
T'_a & 0 & 1 & 2 \\
\end{array}
\]

\[
\begin{array}{ccc}
T'_b & 0 & 1 & 2 \\
\hline
T'_b & 0 & 1 & 2 \\
\end{array}
\]

\[
\begin{array}{ccc}
T'_c & 0 & 1 & 2 \\
\hline
T'_c & 0 & 1 & 2 \\
\end{array}
\]

$T'_c = 0, T'_b = 1; F = \{0, 3, 6, 7\}$

For example, take $E(4)$, which equals \{b(c), b(B), B\}.

$E(4) \cap CS_{a,1} = E(4) \cap \{b(c), B\} = \{b(c), B\} = R_{a,1}(2)$, so $\mu_{a,1}(4) = 2$
6 Concluding remarks

We derived a rather efficient tabulation algorithm for table-driven bottom-up tree acceptors. The base idea for the optimized tabulation is quite simple, nevertheless it leads to a complex algorithm. The presented algorithms are derived by step-wise refinement: starting with the well-known reachability algorithm we elaborate this towards an algorithm for the efficient generation of compressed parse tables. Our opinion is that such a systematical derivation gives us more insight in a complex algorithm.

Experiments with an implementation (in Pascal) of the algorithm have demonstrated a considerable improvement in table generation. For example, a code generation grammar with 33 production rules (representing a part of the Intel 8085 instruction set) gave an improvement of 9208 table entries to only 635, of which 468 index map table entries. Index map tables take up most of the space, but traditional compression techniques can be used to reduce that space since index maps are inherently sparse.

Although the tabulation algorithm has an exponential time complexity we believe that for code generation grammars the mapping of match sets on equivalence classes is such that a passable improvement is made. Though much work remains to be done in the field of universal code generator-generators, we hope to have made a contribution to bottom-up parsing algorithms.
Acknowledgements
Thanks are due to Huub ten Eikelder, Berry Schoenmakers and Pieter Struik (all of the Eindhoven University of Technology) for reading and commenting on draft versions of this paper.

References


REFERENCES


<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff, H.M.J.L. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specifications of information systems</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several parallel systems</td>
</tr>
<tr>
<td>86/05</td>
<td>Jan L.G. Dietz, Kees M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IFIP86)</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes</td>
</tr>
<tr>
<td>Year/Number</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of sound and complete axiomatizations of the monitor concept</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J.Paredaens</td>
<td>A formal approach to distributed information systems</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification.</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch).</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic.</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection.</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata.</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee, A. Lapinski</td>
<td>OR and AI approaches to decision support systems.</td>
</tr>
<tr>
<td>87/13</td>
<td>J.C.S.P. van der Woude</td>
<td>Playing with patterns, searching for strings.</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hooman</td>
<td>A compositional proof system for an occam-like real-time language</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>87/15</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>A compositional semantics for statecharts</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.-J. Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples</td>
</tr>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence</td>
</tr>
<tr>
<td>88/02</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specification for Information Systems</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples</td>
</tr>
<tr>
<td>88/04</td>
<td>G.J. Houben, J. Paredaens, D. Tahon</td>
<td>The Nested Relational Algebra: A Tool to handle Structured Information</td>
</tr>
<tr>
<td>88/05</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specifications for Information Systems</td>
</tr>
<tr>
<td>88/06</td>
<td>H.M.I.L. Schols</td>
<td>Notes on Delay-Insensitive Communication</td>
</tr>
<tr>
<td>88/07</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Modelling Statecharts behaviour in a fully abstract way</td>
</tr>
<tr>
<td>88/08</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>A Formal model for System Specification</td>
</tr>
<tr>
<td>88/09</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A Tutorial for Data Modelling</td>
</tr>
</tbody>
</table>
88/10 J.C. Ebergen  A Formal Approach to Designing Delay Insensitive Circuits
88/11 G.J. Houben  A graphical interface formalism: specifying nested relational databases
J.Paredaens
88/12 A.E. Eiben  Abstract theory of planning
88/13 A. Bijlsma  A unified approach to sequences, bags, and trees
88/14 H.M.M. ten Eikelder  Language theory of a lambda-calculus with recursive types
R.H. Mak
88/15 R. Bos  An introduction to the category theoretic solution of recursive domain equations
C. Hemerik
88/16 C. Hemerik  Bottom-up tree acceptors
J.P. Katoen
88/17 K.M. van Hee  Executable specifications for discrete event systems
G.J. Houben
L.J. Somers
M. Voorhoeve