For: jubo
Printed on: Fri, May 8, 1992 11:00:01

Document: Komprimierung
Last saved on: Fri, May 8, 1992 10:59:36

updated version of AIB No. 87−12; appeared in ACM TOPLAS 13 (3), July 91, 295−314.

Table Compression for Tree Automata[1]

Jürgen Börstler[2], Ulrich Möncke[3] and Reinhard Wilhelm

Universität des Saarlandes, Fachbereich 10−Informatik, Im Stadtwald,

6600 Saarbrücken, West Germany

SUMMARY

The compression of bottom−up tree automata and their representation as implemented in the OPTRAN tree transformation system is described as a four step process. First, the vertically working tree automata traversing one generation in one step are replaced by horizontally working automata splitting this step into rank(operator) steps. This replaces each matrix of dimension rank(operator) by a tree of depth rank(operator). A second step replaces this tree by a shortened DAG, i.e. a DAG where equivalent states of the tree are identified and paths in the tree with no gain in information are condensed. The acceptance property of the automaton is not changed. This and the following size reduction steps critically depend on the "no error" assumption, i.e. the assumption that input trees are correctly built and no error detection is required of the tree automata. The third step embeds the DAG, which is naturally represented by 2−dimensional tables, into a linear array using row displacement and row column schemes. In addition, this step compresses strings of consecutive identical entries which occur often in this type of automaton. The last step exploits the memory structure and addressability of the target machine. Using information on automata and table sizes, the most efficient storage representation is selected. The data structures and access functions are generated as C objects and functions, resp.. Results showing the effectivness of the different steps are given.

KEY WORDS Tree automata Table compression Sparse tables

---

1. Research reported herein was partially supported by the Deutsche Forschungsgemeinschaft, Project "Manipulation of Attributed Trees" and ESPRIT, Project PROSPECTRA, PROgram development by SPECification and TRAnsformation.

2. present address: RWTH Aachen, Lehrstuhl für Informatik III, Ahornstraße 55, D−5100 Aachen.

3. present address: Gesellschaft für Automation und Organisation mbH, D−8000 München.

1. Introduction

The OPTRAN system [6,7,10], designed and implemented at the Universität des Saarlandes, supports the specification and automatic generation of transformers for attributed trees. Originally developed for the optimization phase in compilers it has been and is being used for the generation of source−to−source translators, compiler frontends, and code generators. The system essentially consists of a set of generators each processing some part of a specification and generating automata to be activated at transformation time. In particular, automata for efficient tree pattern matching, attribute evaluation and re−evaluation are generated [11].

Several of these automata are finite−state bottom−up tree automata [5]. They propagate states bottom up through the tree one generation in one step, cf. fig. 1.

Such automata are used in tree rewriting for efficient matching of tree patterns [8,9] against subject trees and for computing strategic information for attribute evaluation, e.g. global attribute dependencies, and for selecting node specific attribute evaluators [11].

Formally, a finite **bottom up tree automaton** is a tuple $T = (S, V, \varrho, \delta)$, where

- S is a finite set of **states**,
- V is a finite ranked alphabet of **operators**,
- $\varrho: V \to \underline{I}N$ gives the **rank** of each operator, and
- $\delta = \{\delta_{op}\}_{op \in V}$ is a family of **transition functions**, where

  $\delta_{op}: S^k \to S$, if $\varrho(op) = k$.

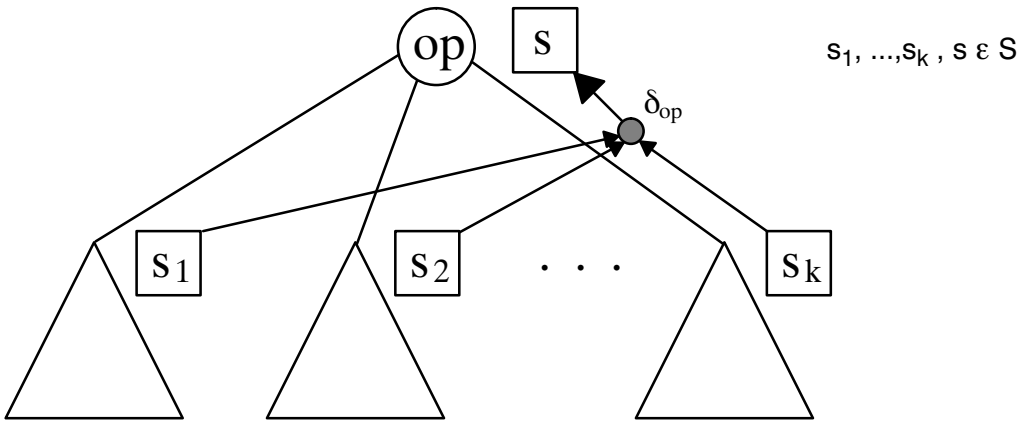

Figure 1

There is a natural way of representing the transition function δ of bottom−up tree automata, i.e. by one *k*−dimensional matrix per operator of rank *k*. But these matrices are unacceptably large. Following Kron [9] we chose a different representation. Such a *k*−dimensional matrix is replaced by *k 2*−dimensional tables. These tables represent a different type of automaton that starts with a horizontal walk through the children in some new initial state, making transitions under the children states into new intermediate states, and finally computing the state at the parent node, cf. fig. 2.

Thus the *k 2*−dimensional tables can better be regarded as a decision tree of depth *k*, whose non−terminal nodes are the newly introduced intermediate states and the branches are labeled with the original states.

Formally, the **horizontal automaton for T**, $T = (S, V, \Omega, \delta)$ is a family of **local automata** $\{HT_{op} = (S, IS_{op}, init_{op}, trans_{op})\}_{op \in V}$
where *S* will now be called the set of **final states** of the horizontal automata and where IS_{op} is a finite set of **intermediate states** such that $S \cap IS_{op} = \{\}$, $init_{op}: - \rightarrow IS_{op}$ determines the initial intermediate state, and $trans_{op}: IS_{op} \times S - \rightarrow IS_{op} \cup S$ defines the horizontal transitions from intermediate states to intermediate states under final states at children nodes and the transition into the final state at the parent, cf. fig. 2.

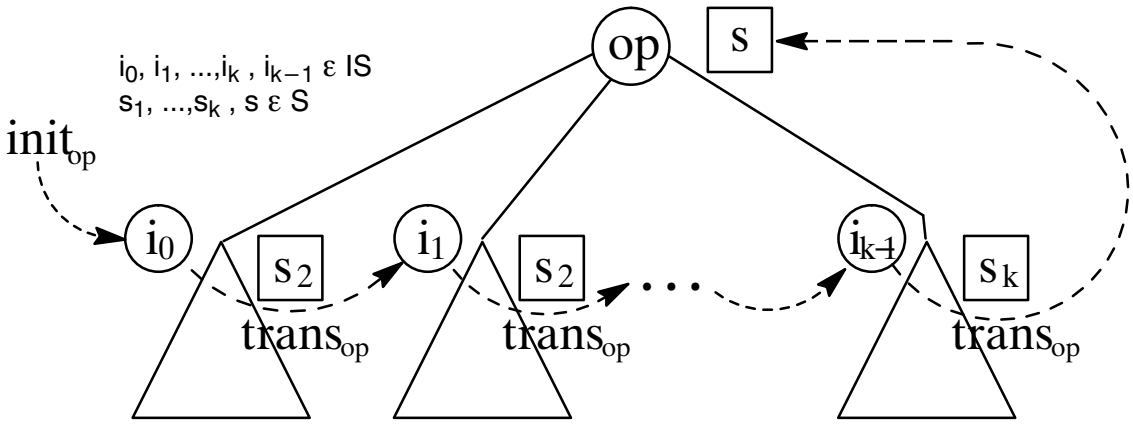

Figure 2

The effect of this change of automaton type together with some additional improvements has been described in [12].

The structure of the paper is as follows. Section 2 describes compression of

the local automata. States and transitions are eliminated as long as the final state reached at the parent remains unchanged. In section 3, two different table compression methods are presented having different size and access speed characteristics. These are modifications of the well known row displacement scheme [1,14] and the row column scheme [4]. Section 4 presents machine level table representations, while sections 2 and 3 are independent of the target machine. Each of these sections contains measurements of savings in space and comparisons in runtime.

2. Reduction in automata sizes

The starting points for the automata size reduction described in this section are automata of known size, i.e. number of states and transitions. They were generated by tools performing extensive precomputation on specifications such as attribute grammars or sets of tree transformation rules. Their type is described in the introduction; they traverse each instance of a production in a (left−to−right) walk through the children computing intermediate states before ascending to the parent with the final state.

Note that the function $trans_{op}$ may not be defined for all pairs of states and i−states, or stated equivalently, the table representing such an automaton may contain empty entries. However, these transitions will never be tried and these entries will never be accessed, since the trees traversed by these automata are correctly built. Any checking and rejecting is done beforehand. Let us call this the ”no error” assumption. For the reduction techniques proposed here the irrelevance of the empty entries is essential.

Each final state can be ”interpreted”, i.e. it corresponds to a characteristic graph of the attribute grammar or a set of tree items, while the intermediate states may not have such an interpretation. Therefore, final states can be checked for equality and thus be made unique for each local automaton belonging to a production. However, the equality of intermediate states can in general only be deduced from the structure of the automaton. All paths from an intermediate state

may be eliminated if they lead to a uniquely determined final state. Identical subautomata can be identified.

Two intermediate states i_1 and i_2 are **superimposable**, if for all final states *s*, if both $trans_{op}(i_1, s)$ and $trans_{op}(i_2, s)$ are defined, then they are equal.

The **level** of an intermediate or final state is the length of the path leading from the initial state to it. We now describe the first step in our compression process. It is only concerned with reducing the size of the horizontal automata.

First, any local automaton for an operator of *rank >= 1* having a unique final state is eliminated; the final state is identified with the initial state. Final states and intermediate states are represented differently (except for this detail representation of the automata plays no role, yet). The generated automaton will notice the immediate transition into a final state, and no further horizontal steps will be performed.

Note that a local automaton for an operator of *rank = 0* always has a unique final state because there are no transitions.

Now, a backwards walk through the intermediate states is performed. All states of a local automaton, which are at the same level, are regarded together. They are called the set of **actual states**. Transitions are eliminated and states are identified in the following way:

- an actual state *i* is identified with a final state *s*, if all transitions out of *i* lead to this final state *s* (*shortening*);
- two states on the same level are identified, if they are superimposable (*overlay*).

The following transition function for an operator *op* of rank *3* will be used as a running example throughout the paper. Figure 3 shows the automaton which is the input to our compression process.

Let $S = (s1, s2, ..., s7)$; let the transition function be:

$\delta_{op}(s[2,3,4], s[1,2,4,5], s[2,3,6,7]) = s2,$

$\delta_{op}(s6, s[1,2,4,5], s2) = s2,$

$\delta_{op}(s6, s[1,2,4,5], s[3,6]) = s3,$

δ_{op} *(s6,* *s*[*1,2,4,5*], *s7)* = *s4,*

δ_{op} *(s7,* *s2,* *s2)* = *s2,*

δ_{op} *(s7,* *s2,* *s*[*3,6*]) = *s3,*

δ_{op} *(s7,* *s2,* *s7)* = *s4,*

δ_{op} *(s7,* *s*[*1,4,5*], *s2)* = *s5,*

δ_{op} *(s7,* *s*[*1,4,5*], *s*[*3,6,7*]) = *s6,*

where δ_{op} *(..., s*[*x,y*], *...)* = *si* is an abbreviation for:

δ_{op} *(..., s*[*x*], *...)* = *si* and δ_{op} *(..., s*[*y*], *...)* = *si*

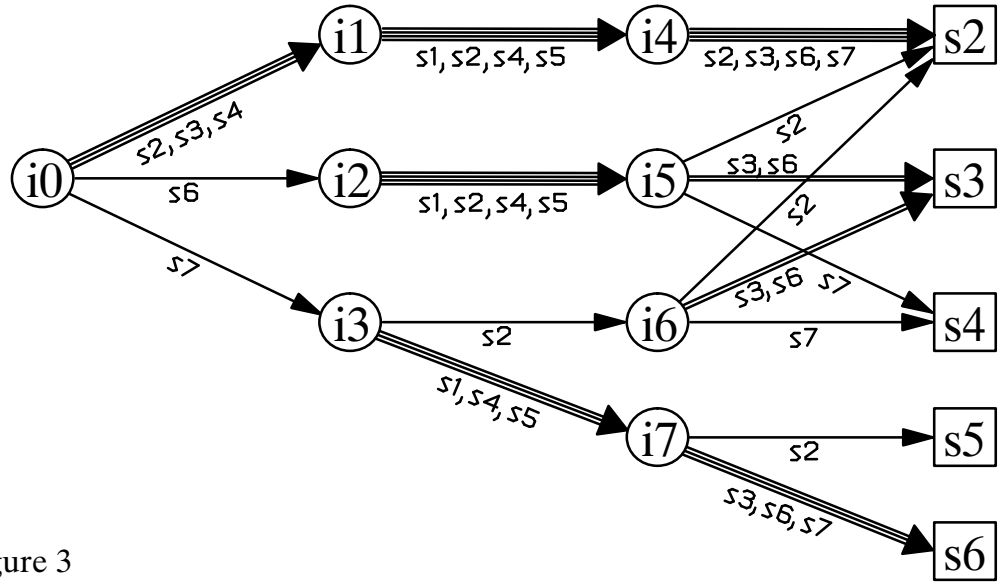

Figure 3

The reduction algorithm starts at the level ϱ*(op)*−*1*, here *2* (note that the final states at the level ϱ*(op)* are unique). First, intermediate states are deleted (shortening) if possible, here intermediate state *i4*. Fig. 4 shows the resulting automaton. At runtime, the automaton will stop when it recognizes a final state. If paths have been shortened at optimization time, the automaton may stop after

the execution of $n < \varrho(op)$ transitions.

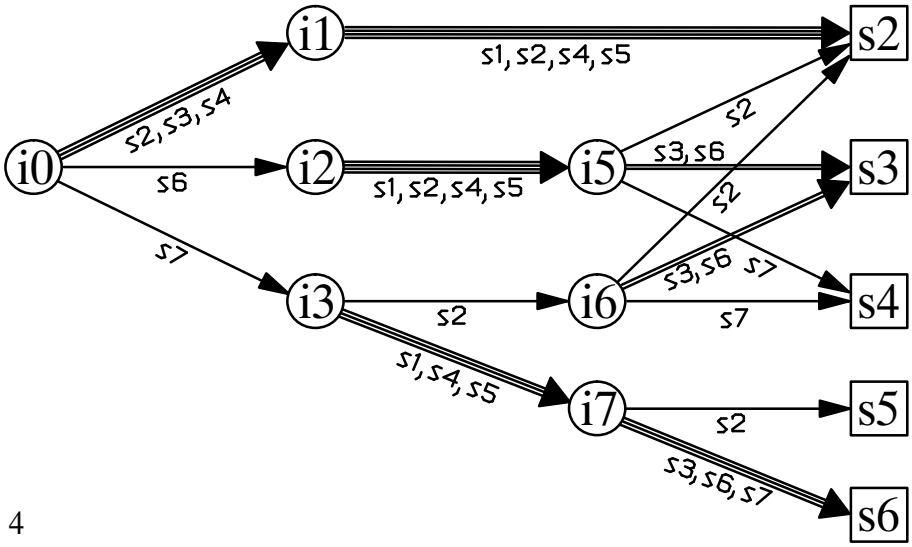

Figure 4

Second, states are merged if they are superimposable (overlay), here intermediate states *i5* and *i6*. The resulting intermediate state is called *i5'* (cf. fig. 5).

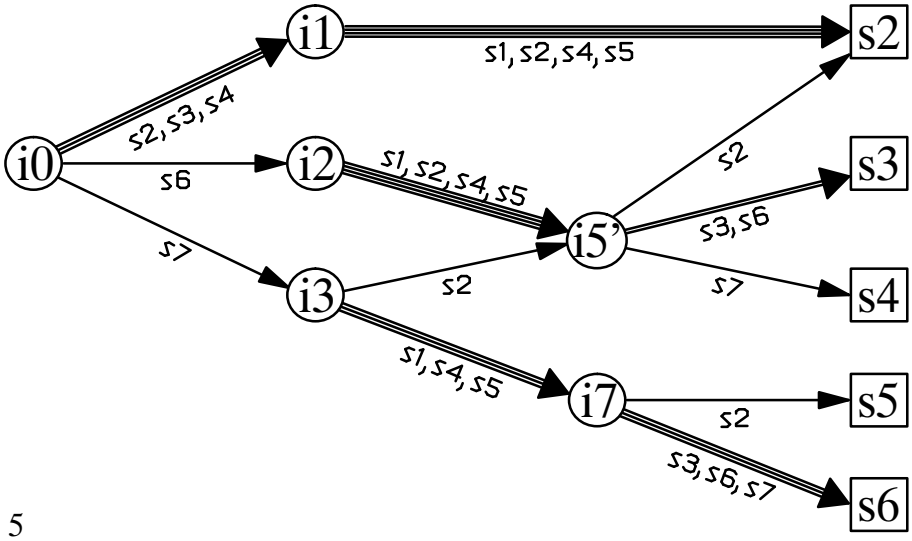

Figure 5

The reduction continues at level *1*, deleting *i1*. Note that according to the above *i2* must not be deleted, although all transitions lead to the same successor. Shortening the path *(i0,i2,i5')* would lead to a transition under state *6* from *i0* to *i5'*. This would cause trouble for the automaton in subsequent transitions (cf. fig. 6).

Let *(6, 2, 7)* be the input. Because *i5'* is an intermediate state itself, the automaton will try to execute the transition under *2* from *i5'* to final state *s2*, getting to the wrong final state. Now, let *(6, 1, 7)* be the input. In this case, the automaton

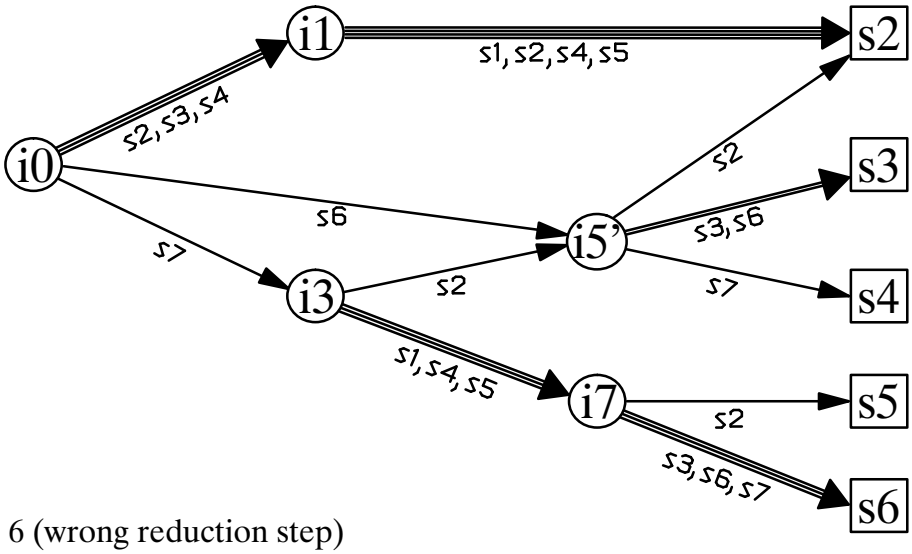

Figure 6 (wrong reduction step)

would have an undefined transition. Of course, adding some information about skipped positions would correct these errors, while complicating the representation (cf. [12]), and complicating the further compression steps discussed in section 4.

Figure 7 shows the final automaton resulting from the local reduction step.

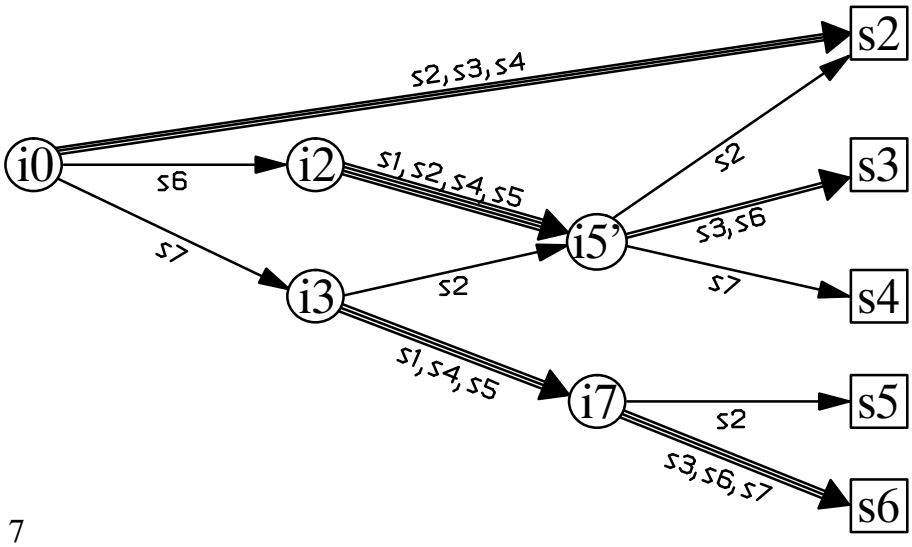

Figure 7

We want to complete this section with a summary of reduction results we gained using this method to reduce local automata. The example automata were generated to compute characteristic graphs resp. tree pattern matchers for attribute grammars resp. tree rewriting systems.

Cg_2, cg_3, cg_4 are automata generated from different attribute grammars. They compute characteristic dependency graphs. Cg_1 describes formulae of

	cg_1	cg_2	cg_3	cg_4	pm_1	pm_2	pm_3	pm_4	pm_5	pm_6	pm_7
size of original local automaton:											
(i−)states	41	425	436	390	396	202	183	58	84	156	177
transitions	74	4225	2552	1984	5768	1375	340	486	539	597	237
size of reduced local automaton:											
(i−)states	16	143	123	100	319	145	87	48	49	86	75
transitions	10	903	448	239	4526	1034	131	384	309	218	47
savings in %											
(i−)states	60.98	66.35	71.79	74.36	19.44	28.22	52.46	17.24	41.67	44.87	57.63
transitions	86.49	78.63	82.45	87.95	21.53	24.80	61.47	20.99	42.67	63.48	80.17

Figure 8

first order predicate calculus. Cg_2, cg_3, and cg_4 specify the static semantics of Mini−PASCAL. The good reduction results result from the uniformity of the attribute dependencies. Pm_1, ..., pm_7 are automata for pattern matching. Pm_1 results from a specification of code generation. This specification contains a lot of special cases, which makes the pattern sets less uniform. Pm_2, pm_3, pm_6, and pm_7 are different versions of optimization phases for Mini−PASCAL. Pm_2 treats the highest number of special cases which leads to the least uniform set of patterns. The transformation supported by pm_4 transforms logical formulae into disjunctive normal form. It again has a very heterogeneous set of patterns. As a conclusion one can say, that higher uniformity in the specification, e.g. attribute dependencies and tree patterns, leads to higher uniformity in the automata and thus to better results in the reduction process.

3. Table Compression

The transition functions of the local automata (as produced by the mechanisms of section 2) can be represented by $\varrho(op)$ 2−dimensional tables per automaton, or since all have width $|S|$, by one such table altogether. These are now embedded into a linear array plus some additional data structures for access and control.

Let the transition functions $trans_{op}$ be represented by one array

T_{op}: **array** [*1..*$|IS_{op}|$, *1..*$|S_{op}|$] **of** target, where

target = (IS_{op} − {*initial−state*}) Y S_{op} Y {*empty*}.

$S_{op} \subseteq S$ denotes the set of final states occuring at operator *op*, *empty* denotes entries corresponding to undefined transitions (cf. the beginning of sec. 2). Due to the no error−assumption we call them insignificant entries, as opposed to the significant ones related to defined transitions. Note that an initial−state can't occur as a target of any transition, therefore it can be used to denote insignificant entries.

Figure 9 shows the table representations of the running example automaton.

before reduction (cf. fig. 3):

states	s1	s2	s3	s4	s5	s6	s7
i−states							
i0	−	i1	i1	i1	−	i2	i3
i1	i4	i4	−	i4	i4	−	−
i2	i5	i5	−	i5	i5	−	−
i3	i7	i6	−	i7	i7	−	−
i4	−	s2	s2	−	−	s2	s2
i5	−	s2	s3	−	−	s3	s4
i6	−	s2	s3	−	−	s3	s4
i7	−	s5	s6	−	−	s6	s6

after reduction (cf. fig. 7):

	s1	s2	s3	s4	s5	s6	s7
i0	−	s2	s2	s2	−	i2	i3
i1	−	−	−	−	−	−	−
i2	i5'	i5'	−	i5'	i5'	−	−
i3	i7	i5'	−	i7	i7	−	−
i4	−	−	−	−	−	−	−
i5'	−	s2	s3	−	−	s3	s4
i6	−	−	−	−	−	−	−
i7	−	s5	s6	−	−	s6	s6

Figure 9 table entry i_j, s_k represents the transition $trans(i_j, s_k)$;
− denotes an undefined transition

This table T_{op} is now embedded into a linear array by two different compression methods, a Row Displacement Scheme (RDS) resp. a Row Column Scheme (RCS).

3.1. Row Displacement Scheme

The original Row Displacement Scheme (RDS) [13,14] successively embeds the rows of T_{op} into the array

Value : **array** $[1..(|S_{op}| + \max_{k \in IS_{op}} v_{op,k})]$ **of** target,

where the $v_{op,k}$ are obtained as follows: The actual row of T_{op} (index *k*) is shifted right over Value, until there are no collisions between significant entries of row *k* and significant entries already entered into Value. Note that in the original version, no merging of identical significant entries is allowed. $V_{op,k}$ then is the number of

positions row k of T_{op} is shifted. The entries of row k are entered into Value starting at $v_{op,k}$.

In a second array

RowIndex : **array** $[1..(|S_{op}| + \max_{k \in IS_{op}} v_{op,k})]$ **of** IS_{op}

the row index in T_{op} for each entry in Value is stored to distinguish insignificant entries from overlayed significant ones. Insignificant entries in Value are marked with a special symbol in RowIndex. Note that one cannot avoid the storage of insignificant entries in this scheme. A third array

RowPtr : **array** $[1..|IS_{op}|]$ **of** $0..\max_{k \in IS_{op}} v_{op,k}$

contains the values of the $v_{op,k}$, i.e. the index into Value, where the embedded rows of T_{op} start.

In our setting, we can improve on this scheme. Under the no error−assumption, we can remove the array RowIndex, which only served for error checking purposes. In addition, we can work with a looser definition of collision. When embedding a row of T_{op} into Value, only collisions of **different** significant entries have to be avoided. This looser definition of collision may sometimes lead to a complete overlay of two identical rows, at least to an overlay of suffixes.

Finding the optimal row shift, i.e. the placement of rows leading to the minimal length of Value is an NP−complete problem. We prefer the first−fit shifting of rows, in contrast to first−fit−decrease which first orders the rows according to the number of significant values [13,14]. This choice has practical reasons. We decrease the need of dynamic memory, because there is no need to collect all rows to decide the order of embedding. Despite this we still obtain short arrays Value (cf. fig. 10 and fig. 12). In [2] different row−shifting schemes are compared. The results show that there is no scheme which always leads to good results. In addition there is no scheme which is always the best one.

Figures 10a to 10c show the table compression. The matrix (fig. 10a) gives an overview of the successive embedding of rows. Each row shows the embedding of the transitions according to a given intermediate state. The istate → index−

position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
istate → index																						
i0 → 0	−	s2	s2	s2	−	i2	i3															
i1 → 0	−	−	−	−	−	−	−															
i2 → 7								i5'	i5'	−	i5'	i5'	−	−								
i3 → 9										i7	i5'	−	i7	i7	−	−						
i4 → 0	−	−	−	−	−	−	−															
i5' → 13														−	s2	s3	−	−	s3	s4		
i6 → 0	−	−	−	−	−	−	−															
i7 → 15																−	s5	s6	−	−	s6	s6

Figure 10a

column indicates for each intermediate state, what the starting index in Value of its row is. The results of this embedding are the arrays Value and RowPtr (cf. fig. 10b).

Value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	−	s2	s2	s2	−	i2	i3	i5'	i5'	i7	i5'	i5'	i7	i7	s2	s3	s5	s6	s3	s4	s6	s6

RowPtr	i0	i1	i2	i3	i4	i5'	i6	i7
	0	0	7	9	0	13	0	15

Figure 10b

Table access is done by the following formula:

```
state := i0;                    (* the first element of RowPtr contains a pointer to v_op,i0  *)
level := 1;                     (* level : 1..ϱ(op) denotes the actual level of the automata *)
while state ε IS_op do
    state := follow_state( state, si_level );      (* si_level : transition to be taken at level i *)
    level := level + 1
od;
```

where

```
follow_state( state, si_level ) := Value[ RowPtr[ state] + i];      (* the si are coded as i *)
```

We can exploit another property to achieve further space reductions. Any sequence of transitions of the automaton starts with state *i0* and always continues with a transition from the intermediate state just reached, until a transition to a final state is taken. This means that a sequence of accesses to the array Value does not need the array RowPtr if we replace the intermediate states by the offset of their rows in the array Value. These offsets were represented in the istate →

index−column in Fig. 10a. With this improvement the array RowPtr becomes obsolete. The following table shows the resulting array Value'.

Value'	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	s2	s2	s2	−	7	9	13	13	15	13	13	15	15	s2	s3	s5	s6	s3	s4	s6	s6	

Figure 10c

The access formula changes to:

state := 0; *(* because $v_{op,i0} = 0$ *)*

level := 1;

while *state* ε IS_{op} **do**

state := follow_state(state, si_{level});

level := level + 1

od;

where

follow_state(state, si_{level}) := Value[*state + i*]; *(* the si are coded as i *)*

Example (the final states *si* are coded as *i* here):

Initial state = *0*; (because $v_{op,i0} = 0$)

Note that the automaton will stop when it recognizes a final state.

transition tuple *(s2,s5,s7)*:

transition under *s2* −› Value[*0 + 2*] = *s2* (final state *2*);

transition tuple *(s7,s2,s7)*:

transition under *s7* −› Value[*0 + 7*] = *9;*

transition under *s2* −› Value[*9 + 2*] = *13*;

transition under *s7* −› Value[*13 + 7*] = *s4* (final state *4*).

So far, we have only considered embedding for a **single** local automaton table T_{op}. Globally, the embedding process is the execution of a doubly nested loop: the outer loop runs over *op* ε *OP*, the inner for each intermediate state over *i = 1..*ϱ*(op)*. So all local automata tables are embedded in **one** array Value. In this way we also superimpose rows belonging to automata for different operators. In the extreme case, identical automata for different operators are merged. The reduction of automata (cf. section 2) reduces the number of rows and therefore

the time needed to search in Value. In addition, an array

OpPtr : **array** [*1*..maxop] **of** $1..(\max_{op \in Op}\{|S_{op}|\} + \max_{op \in Op}\{\max_{k \in IS_{op}} v_{op,k}\})$

is introduced, where OpPtr[op] contains the pointer to the first row of T_{op} in Value, i.e. $v_{op,i0}$. (Note that the $v_{op,i}$, $i <> i0$ are substituted in Value as described before).

Any saving in the length of Value decreases the maximal row shift, and thus decreases the size of the entries of OpPtr. This lets us choose shorter representations for the components of OpPtr. Also, the indices for intermediate states, which are substituted in Value' for the intermediate states may be better represented. Such representation details will be further discussed in section 4.

3.2. Row Column Scheme

The Row Column Scheme (RCS) [4] stores the table T_{op} into an array

Value : **array** [*1*.. $\underset{k \in IS_{op}}{f} w_{op,k}$] **of** target

where $w_{op,k}$ is the number of significant entries in row *k* of T_{op}. Thus, RCS stores only significant entries of T_{op} into Value by compressing them into a sequence of consecutive entries.

The tables resulting from RCS are smaller than those obtained from RDS, but the direct access is lost. The entries of one row have to be searched sequentially. Table size is traded for speed of access when compared to RDS.

ColIndex : **array** [*1*.. $\underset{k \in IS_{op}}{f} w_{op,k}$] **of** state

contains the original column indices of the entries in Value.

RowPtr : **array** [*1*.. $|IS_{op}| + 1$] **of** *1*.. $\underset{k \in IS_{op}}{f} w_{op,k} + 1$

contains for each original rowindex *k* the index of the first entry of $T_{op}[k,*]$ in Value.

The additional Element RowPtr[$m+1$] = $\underset{k \in IS_{op}}{f} w_{op,k} + 1$ is needed to determine the end of the last row (cf. access formula).

Access formula for a **single** transition:

column := *RowPtr*[*state*];

(column contains the first element of the list of transitions out of state *)*

while *column* < *RowPtr*[*state* + *1*] **and** si_{level} <> *ColIndex*[*column*] **do**

(finding the right transition *)*

column := *column* + *1*

od;

if *column* < *RowPtr*[*state* + *1*] **then**

(now :* si_{level} = *ColIndex*[*column*]! **)*

target := *Value*[*column*];

else *insignificant*

To speed up access we introduce a special symbol determining the end of each list of significant entries, corresponding to a row of T_{op}. Thus the array ColIndex changes to

ColIndex : **array** [*1*..($\underset{k\ \varepsilon\ IS_{op}}{f}\ w_{op,k}$) + $|IS_{op}|$] **of** state Υ {*end_element*}.

Then the search is done in the interval ColIndex[RowPtr[state] + state .. RowPtr[state + 1] + state], where ColIndex[RowPtr[state + 1] + state] = *end_element*. This improvement avoids the first test in the while−loop.

Again, note that the no error−assumption is exploited. For this reason we can also avoid the if−statement, which made sure that the right transition was found. As in the previous section we want to embed all tables T_{op}, *op* ε *OP* into one array Value (ColIndex). We therefore introduce an array OpPtr, where OpPtr[op] contains the pointer to the first row of T_{op} in Value, in an analogous way.

As an overall access formula we obtain:

state := *OpPtr*[*op*];

(pointer into Value, containing first transition out of initial−state of op *)*

level := *1;*

(level* = 1..ϱ(op) *denotes the actual level of the automaton *)*

while *state* ε IS_{op} **do**

state := *follow_state(state,* si_{level} *);*

level := *level* + *1*

```
od;
```

and

```
follow_state( state, si_level ) :=
    column := RowPtr[ state] + state;
    (*  column now contains the first element of the list of transitions out of state;      *)
    (*  + state, because of the added elements to determine the list ends, see above       *)
while si_level > ColIndex[ column] do
    column := column + 1
od;
(* now si_level = ColIndex[ column] holds *)
target := Value[ column − state];
(* note that there are no additional elements in Value *)
```

(a) List construction:

The reduced automaton of Fig. 9 is the input to our example. Note that *end_element* was introduced to determine the list ends (see above).

i0:	Value	s2	s2	s2	i2	i3	
	ColIndex	2	3	4	6	7	⊥
i2:	Value	i5’	i5’	i5’	i5’		
	ColIndex	1	2	4	5	⊥	
i3:	Value	i7	i5’	i7	i7		
	ColIndex	1	2	4	5	⊥	
i5’:	Value	s2	s3	s3	s4		
	ColIndex	2	3	6	7	⊥	
i7:	Value	s5	s6	s6	s6		
	ColIndex	2	3	6	7	⊥	

As a further improvement (assuming the no error−condition), default lists are constructed by holding the most frequent element at most once. We prefer placing the default element at the head of the list. So we avoid scanning the entire list to access it, as described in [1].

(b) List construction with default elements (default elements are printed in bold−face).

i0:					
	Value	**s2**	i2	i3	
	ColIndex	6	7	⊥	

i2:			
	Value	**i5'**	
	ColIndex	⊥	

i3:				
	Value	**i7**	i5'	
	ColIndex	2	⊥	

i5':					
	Value	**s3**	s2	s4	
	ColIndex	2	7	⊥	

i7:				
	Value	**s6**	s5	
	ColIndex	2	⊥	

As in the previous section we replace the intermediate states by their indices. The complete data structure resulting from the RCS is shown in figure 11.

position	1	2	3	4	5	6	7	8	9	10	11	12
Value	**s2**	i2	i3	**i5'**	**i7**	i5'	**s3**	s2	s4	**s6**	s5	−
ColIndex	−	6	7	⊥	⊥	2	⊥	2	7	⊥	2	⊥
Value'	**s2**	4	5	**7**	**10**	7	**s3**	s2	s4	**s6**	s5	

Figure 11

The access for follow_state changes to (note that OpPtr[op] = 1 for our running example automaton):

```
follow_state( state, si_level ) :=
    column := state + 1;            (* first non−default transition out of state *)
    while ColIndex[ column] < si_level  do
        column := column + 1            (* scanning through transitions out of state *)
    od;
    if ColIndex[ column] = si_level  then
        target := Value[ column];   (* transition under si_level  was found *)
    else    target := Value[ state];                    (* take default transition *)
```

Because many of the lists in ColIndex are identical or suffixes of another one, some more improvements are possible. Also, we want to merge identical automata, which often occur in some special contexts, i.e. we want to overlay the (default) lists embedded in Value and/ or ColIndex as in the row−

displacement−scheme. The two lists are embedded independently to gain the best results. But this leads to losing the correspondence between the two related sublists according to one row of the table T_{op}. We therefore introduce an additional element to restore the correspondence between the entries in ColIndex and Value. This element is stored at the head of each default list embedded in Value and contains a pointer to the beginning of the corresponding entries in ColIndex. Such improvements of this scheme are shown in detail in [3].

We now show some compression results for the example inputs of fig. 8 which we obtained using the row−displacement−scheme (RDS) as described before, resp. the row−column−scheme (RCS) including the further improvements mentioned above. This improvement usually leads to different lengths of Value and ColIndex. The length of the array *OpPtr* is the same for both schemes. In both schemes we use one additional storage unit to refer to insignificant entries (RDS) resp. list delimiters (RCS).

RDS	cg_1	cg_2	cg_3	cg_4	pm_1	pm_2	pm_3	pm_4	pm_5	pm_6	pm_7
time used user	0:233	7:666	5:383	3:816	22:73	4:467	0:833	1:383	1:200	1:183	0:533
system	0:050	0:600	0:516	0:567	1:116	0:367	0:217	0:250	0:200	0:150	0:150
length of Value	8	475	303	170	2940	1150	173	505	354	181	79
insign. el.	1	7	12	42	241	311	43	132	58	26	31
in %	14.19	1.48	3.97	24.85	8.20	27.07	25.00	26.19	16.43	14.44	39.74
RCS											
time used user	0:216	7:633	5:283	3:600	12:43	2:800	0:783	0:966	1:000	0:967	0:600
system	0:017	0:450	0:483	0:550	0:866	0:217	0:100	0:150	0:067	0:266	0:084
listlengths Value	7	306	209	91	467	150	43	104	20	56	22
ColIndex	5	142	72	30	110	31	10	48	5	11	2
default lists maximum	3	21	13	9	46	9	3	19	3	4	2
average	2.67	10.52	6.64	6.00	3.10	2.57	1.69	7.67	2.33	3.00	1.44
length of OpPtr	13	53	55	56	65	56	55	13	28	56	55

Figure 12

The times mentioned above were taken on a SUN−3 Workstation in single−user mode (shown as <seconds>:<milliseconds>). These measures include the compression of the local automata as described in section 2. It is surprising that RCS is almost always faster than RDS although one does additional work in constructing default−lists. But this additional work creates very

small lists to embed. Note that embedding one list into another is a time consuming process.

The maximum and average lengths of default−lists are computed over all rows of all operators, i.e. over the whole compression process.

4. Machine level table compression

Until now all compressions were independent of the target machine. To perform additional machine level compressions a detailed description of the target machine is needed. We concentrate on compressions at the C language level, because C has enough features for this purposes.

C provides types for byte, word and double−word storage units, which can be restricted to positive integers to augment the interval of positive integers that can be stored. To decide which C−type to choose for the table representation we collect some additional information during the compression phase.

We avoid compression methods below the byte level. That means that we do not pack more than one information unit into the storage unit selected for table representation. This low level packing would lead to complex and thus slow access procedures on most machines.

The RDS sheme leads to tables with three 'classes' of entries: (1) Final states, which are coded as negative integers, (2) intermediate states, which were substituted by their shift widths (cf. sec. 3.1), and therefore are positive and (3) the empty entries, which are denoted by *0*. Thus the array Value contains entries out of the interval $[-max\{\text{state}\}.. \max_{op \in Op} \{ \max_{k \in IS_{op}} v_{op,k} \}]$. If each of these elements can be stored in a single byte, a byte array is used. Otherwise one can check if the number of different elements is smaller than $2^8 - 1$ ($2^{16} - 1$). This number can be computed easily during the compression phase. The different elements are then enumerated. Replacing each entry in Value by its order in this enumeration will often cause smaller entries in Value, so that often smaller storage units can be used. Our experiments showed that the number of disjoint (pairwise different) elements lies between 7% and 42% of all entries in the tables. This percentage

decreases with the increase of overall table size, so that often significant savings are possible. Note that the additional space needed to store the enumeration information may sometimes outweigh these savings. This can be checked easily after the enumeration step. As a negative side effect each table access causes one more array access to receive the enumeration information.

Using RCS analogous observations can be made [3]. Because the default list construction avoids storing numerous identical elements we exclude this in our setting.

In conclusion, we show some experimental data about the storage the computed tables need, and the time to access entries of the generated tables. The access procedures are C−functions according to the algorithms shown in sections 3.1 and 3.2. The results are compared to results gained using techniques as shown in [12]. The method in [12] only performs compression of local automata, represented as linked data structures. For that purpose an improvement of Kron's [9] version is used, which results in rather small local automata. After the compression step the compressed local automata are emitted to a file. For later uses this file is read and the compressed local automata are reconstructed.

	cg_1	cg_2	cg_3	cg_4	pm_1	pm_2	pm_3	pm_4	pm_5	pm_6	pm_7
size of reduced local automaton:											
(i−)states	16	143	123	100	319	145	87	48	49	86	75
transitions	10	903	448	239	4526	1034	131	384	309	218	47
storage needed for compressed local automata (in bytes)											
file in [12]	4696	120593	80167	65284	167246	49521	23155	16589	19793	27802	20152
RDS	21	1055	715	451	6009	2411	455	1035	763	473	134
RDS+*	−	638	448	290	3437	1406	296	610	464	315	−
RCS	25	807	599	177	1109	387	108	165	53	123	79

* Including machine dependent compression. Empty entries show that no additional gains were possible

Figure 13

We achieved a reduction by a factor of at least 100 using our schemes. Note that we don't need additional dynamic data structures.

Fiugure 14 gives some run−time data used to access the compressed local automata, resp. the generated tables. We therefore generated pattern matchers and attribute evaluaters for each compression scheme using OPTRAN. Then we

timings §	table_1	table_2	table_3	table_4	table_5	table_6	table_7	table_8
[12]	67	67	267	534	784	216	334	167
RDS	34	33	250	383	648	133	266	283
RDS+*	34	50	232	333	649	135	366	251
RCS	17	51	233	434	617	216	317	301

§ in milliseconds on a SUN/3 Workstation in single−user mode
* Including machine dependent compression.

Figure 14

ran the generated pattern matchers and attribute evaluaters with several input trees and recorded the times spent in the procedures computing the state transitions in using the compressed local automata, resp. the generated tables.

The table shows that access by table is almost as fast as access by the compressed (dynamic) automata. Note that using the scheme described in [12] additional time is needed to read the files and reconstruct the automata as a linked data structure. In our examples this needed an additional amount of time between 234 and 266 milliseconds.

5. Conclusion

We presented a compression process for bottom−up tree automata, which proceeds in three steps. The first step reduces the initial automata in size. Two competing schemes are described for the second step, a modified row displacement scheme and a row column scheme. The third step takes the output of either of these compressions and does machine level compression. Since the first two compression steps are independent of each other, these steps can be interleaved with the construction phase of the automata. This significantly decreases the need of dynamic memory.

Acknowledgements

We thank Robert Henry for his careful reading of the paper.

References

1. A. Aho, R. Sethi and J. Ullman, *Compilers – Principles, Techniques and Tools*, Addison−Wesley, 1986.
2. M. Ai−Suwaiyel and E. Horowitz, 'Algorithms for Trie Compaction', *ACM TODS* **9** (2), pp 243−263, (1984).
3. J. Börstler, 'Komprimierung statischer Tabellen für ein baumtransformierendes System', *Diploma Thesis*, Universität des Saarlandes, 1987 (in German).
4. P. Dencker, K. Dürre and J. Heuft, 'Optimization of Parser Tables for Portable Compilers', *ACM TOPLAS* **6** (4), pp 546−572, (1984).
5. J. Engelfriet, 'Bottom−up and Top−down Tree Transformations – a Comparison', *Math. Systems Theory* **9** (3), pp 198−231, (1975).
6. I. Glasner, U. Möncke and R. Wilhelm, 'OPTRAN, a Language for the Specification of Program Transformations', *Informatik−Fachberichte* **50**, Springer, 1981.
7. M. Greim, S. Pistorius, M. Solsbacher and B. Weisgerber, 'POPSY and OPTRAN Manual', *Tech. Report A 08/86*, Universität des Saarlandes, 1986.
8. Ch. D. Hoffman and M. O'Donnel, 'Pattern Matching in Trees', *JACM* **29** (1), pp 68−95, (1982).
9. H. Kron, 'Tree Templates and Subtree Transformational Grammars', *Ph.D. Thesis*, University of California, Santa Cruz, 1975.
10. P. Lipps, U. Möncke and R. Wilhelm, 'OPTRAN – A Language System for the Specification of Program Transformations: System Overview and Experiences', *Proc. of the Workshop on Compiler−Compilers and High−Speed Compilation, Oct. 88*, Springer LNCS, 1989.
11. P. Lipps, U. Möncke, M. Olk and R. Wilhelm, 'Attribute (Re)evaluation in the OPTRAN System', *Acta Informatica* **26**, pp. 213−239, (1988).
12. U. Möncke, B. Weisgerber and R. Wilhelm, 'Generative Support for Transformational Programming', *ESPRIT '85 Status Report*, pp. 511−527, North Holland, 1985.
13. R. Tarjan and A. Yao, 'Storing a Sparse Table', *CACM* **22** (11), (1979).
14. S. Ziegler, 'Smaller Faster Table Driven Parser', *unpublished manuscript*, Madison Academic Comptg. Ctr., 1977.