Tree Automata for Code Selection

Christian Ferdinand*
Helmut Seidl†
Reinhard Wilhelm
FB 14 – Informatik, Universität des Saarlandes
Saarbrücken Germany

Abstract

We deal with the generation of code selectors in compiler backends. The fundamental concepts are systematically derived from the theory of regular tree grammars and finite tree automata. We use this general approach to construct algorithms that generalize and improve existing methods.

1 Introduction

A code generator for a compiler is applied to an intermediate representation (*IR*) of the input program that has been computed during preceding phases of compilation. This intermediate representation can be viewed as code for an *abstract machine*. The task of code generation is to translate this code into an efficient sequence of instructions for a concrete machine.

Besides register allocation and instruction scheduling (for processors with pipelined architectures), code selection, i.e., the selection of instructions, is one subtask of code generation. It is especially important for CISC (*C*omplex *I*nstruction *S*et *C*omputer) architectures where there are usually many possibilities to generate code for the same piece of program. However, with the increasing complexity of RISC architectures the problem reappears.

Example 1.1 Consider the simplest member of the MC680x0 family, namely the MC68000. The instruction

MOVE.B *dist*(A1, D1.W), D5

loads one byte into the lowest quarter of data register D5. The address of the operand is determined by adding the lower half of the contents of D1 together with the 8 bit constant *dist* to the contents of base register A1. This addressing mode is called "address register indirect with index and 8 bit displacement". If the costs of instructions are counted in terms of execution time, i.e., the number of necessary processor cycles, then this instruction has cost 14. An alternative code sequence without indexed addressing and displacement is:

*Funded by the ESPRIT Project #5399 (COMPARE)
†Funded by DFG SFB #124 VLSI-Design and Parallelism

ADD.W	#*dist*, A1	cost: 16
ADD.W	D1, A1	cost: 8
MOVE.B	(A1), D5	cost: 8
	with total cost 32	

Another possibility is:

ADD.W	D1, A1	cost: 8
MOVE.B	*dist*(A1), D5	cost: 12
	with total cost 20	

This example shows the necessity of an intelligent code selection algorithm for CISC architectures.

Note, however, that the two alternative code sequences above are equivalent only with respect to the storage and to the result register D5. They are not equivalent with respect to the condition codes and register A1! The code selector has to guarantee that these alternatives are valid in a given context. □

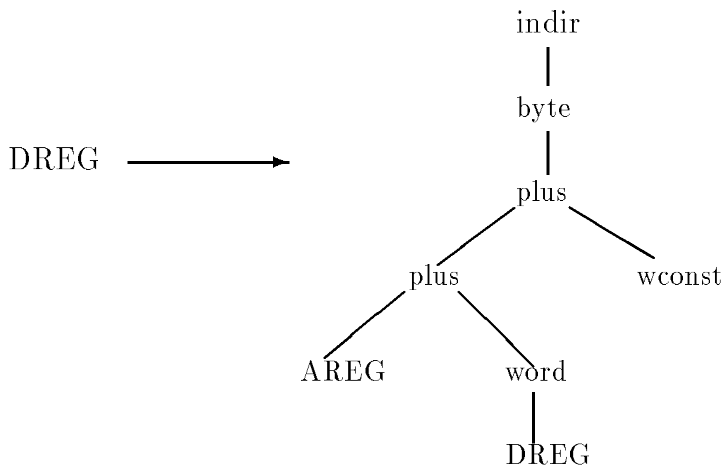
We assume, as usual, that there is some canonical way to translate the control structure of programs, that storage allocation and storage layout of data structures has been determined before, and that access to data are represented appropriately in the IR. What is left to the code selector is the generation of code for expression trees.

The use of code selector *generators* significantly simplifies the development of code generators for new target architectures.

The input of a code selector generator is some kind of machine language description. We expect the *IR* to be a tree form (at least conceptually). One possibility to describe machine instructions is by rules of a regular tree grammar. The right hand side of a rule describes the "effect" of the instruction (in terms of *IR* operators). The terminals denote (sub-)operations that are executed by the instruction. Nonterminals denote locations or resource classes, e.g., different types of registers. The left hand side nonterminal of a rule describes where (i.e., in which resource class) the result of the instruction will be stored.

Such a *machine grammar* must generate all possible *IR* trees of expressions. A derivation tree for an *IR* tree describes one possibility to generate code for the corresponding expression. In general (especially for CISC processors), the machine grammar is ambiguous. In this case several derivation trees may exist for the same expression corresponding to several possible instruction sequences. To allow for a selection of an efficient instruction sequence one annotates the rules of the grammar with costs that, e.g., give the number of machine cycles to execute the instruction. The goal is to pick the minimal cost derivation tree. This corresponds to a *locally optimal* instruction sequence. However, there are processors (e.g., Motorola 68020) for which the number of necessary machine cycles cannot precisely be predicted since the execution time depends on the runtime context. Here, we have to use approximations.

The number of combinations of instructions, addressing modes and word lengths of customary CISC processors is rather large (in the order of 1000). To avoid the introduction of distinct rules and costs for all combinations one splits the rules for instructions into operator rules, rules for addressing modes and lengths of operands.



$$DREG \quad \rightarrow \quad indir(byte(plus(plus(AREG, word(DREG)), wconst)))$$

Figure 1: Instruction of example 1.1. Nonterminals are in uppercase. Terminals are in lowercase.

To enlarge the applicability of code selection one can allow jump or assignment operators at the root node of a tree. Jumps or assignments do not return a "result" in the above sense. Therefore, rules for such instructions contain a *dummy* nonterminal as left hand sides. This nonterminal does not occur in the right hand sides of any of the rules.

2 Comparison to Related Work

Ripken [26] was the first to suggest a machine description language for processor architectures from which a code generator should be generated. Since the necessary tree parsing algorithms were not available Glanville and Graham [11, 12] proposed a linearization of the intermediate language to apply LALR parser techniques to code selection. This idea was further developed and brought to applicability by Henry [14]. Other approaches combined efficient algorithms for pattern matching in trees [18, 17, 3, 20] with dynamic programming to determine locally optimal instruction sequences [1, 15, 27] or extended the tree pattern matchers to allow for a direct selection of locally optimal instruction sequences [24, 16].

In a sequence of publications Giegerich and Schmal [10, 9] try to develop a unified framework for the two different approaches of tree pattern matching and tree parsing. The problem of code selection is reduced to determining the inverse of a "derivor", i.e., a tree homomorphism from a target algebra to an IR-algebra.

Pelegri-Llopart [23] and Emmelmann [6] use (restricted versions of) term rewriting systems as specification tools for their generators. These systems allow for more convenient machine descriptions. E.g., commutativity of operators can be expressed easily.

Most of these works do not really exploit the theoretically well known connections between algorithms for tree pattern matching and tree parsing on the one side and finite tree automata on the other. A good overview about the classical results on tree automata can be found in [8, 21]. Without explicitly referring to code selection Kron in [18] anticipates many of today's techniques for tree pattern matching.

For simplicity, we restrict ourselves to machine descriptions by means of regular tree grammars. A compilation of ground term rewriting systems or other restricted classes of rewrite systems to tree automata is possible as well (see [23, 4, 6]).

3 Tree Pattern Matching

A *ranked alphabet* is a finite set Σ of operators[1] together with a ranking function $\rho : \Sigma \rightarrow \mathbb{N}_0$. We write Σ_k for $\{a \in \Sigma \mid \rho(a) = k\}$. The *homogeneous tree language* over Σ, T_Σ is the smallest set T such that

- $\Sigma_0 \subseteq T$;
- If $t_1, \ldots, t_n$ are in T and $a \in \Sigma_n$ then $a(t_1, \ldots, t_n) \in T$.

As usual, *nodes* or *occurrences* in a tree are denoted by addresses in $\mathbb{N}^*$. The set $O(t)$ of nodes of a tree $t = a(t_1, \ldots, t_k)$ is inductively defined by:
$O(t) := \{\epsilon\} \cup \bigcup_{j=1}^{k} j.O(t_j)$. The *subtree* t/n of t at node n is defined by $t/\epsilon := t$ and $t/n := t_j/n'$ if $t = a(t_1, \ldots, t_k)$ and $n = j.n'$.

Assume V is a set of variables of rank 0. A member of $T_\Sigma(V) := T_{\Sigma \cup V}$ is called a *pattern* over Σ. A pattern is called *linear* if no variable in it occurs more than once.

Example 3.1 Let $\Sigma = \{a, cons, nil\}$ with $\rho(a) = \rho(nil) = 0$, $\rho(cons) = 2$. Trees over Σ are, e.g., a, $cons(nil, nil)$, $cons(cons(a, nil), nil)$.
Let $V = \{X\}$. Then, X, $cons(nil, X)$, and $cons(X, nil)$ are patterns over Σ. □

A *substitution* is a map $\Theta : V \rightarrow T_\Sigma(V)$. Θ is extended to a map $\Theta : T_\Sigma(V) \rightarrow T_\Sigma(V)$ by $t\Theta := x\Theta$ if $t = x \in V$ and $t\Theta := a(t_1\Theta, \ldots, t_k\Theta)$ if $t = a(t_1, \ldots, t_k)$.[2] We also write $t\Theta = t[t_1\backslash x_1, \ldots, t_k\backslash x_k]$ if the variables occurring in t are from $\{x_1, \ldots, x_k\}$ and $x_j\Theta = t_j$ for all j.

A pattern $\tau \in T_\Sigma(V)$ with variables from $\{x_1, \ldots, x_k\}$ *matches* a tree t if there are trees $t_1, \ldots, t_k$ in T_Σ such that $t = \tau[t_1\backslash x_1, \ldots\ldots, t_k\backslash x_k]$.

Definition 3.1 (The Tree Pattern Matching Problem)
An instance of the *tree pattern matching problem* consists of a finite set of patterns $T = \{\tau_1, \ldots, \tau_k\} \subset T_\Sigma(V)$ together with an input tree $t \in T_\Sigma$. The solution of the tree pattern matching problem for this instance is the set of all pairs (n, i) such that pattern τ_i matches t/n. An algorithm that for every

[1] in functional languages also called constructors.
[2] Traditionally, the application of a substitution Θ to a pattern t is written $t\Theta$.

input tree $t \in T_\Sigma$ returns the solution of the tree pattern matching problem for (T, t) is called a *tree pattern matcher* for T. An algorithm that on input of a pattern set T returns a tree pattern matcher for T is called a *tree pattern matcher generator*. $\square$

Example 3.2 Assume $\tau_1 = cons(X, nil)$ and $\tau_2 = cons(a, X)$ are patterns, and $t = cons(cons(a, nil), nil)$ is an input tree. The solution of the instance $(\{\tau_1, \tau_2\}, t)$ of the pattern matching problem is the set $\{(\varepsilon, 1), (1, 1), (1, 2)\}$. $\square$

In fact, we are interested in constructing efficient tree pattern matcher generators. The pattern matchers generated by the following algorithms will only work for *linear* patterns. We have two possibilities to generate tree pattern matchers for sets of non-linear patterns. First, we can introduce tests for equality of subtrees whenever needed. This may not be efficient since the pattern matcher may have to visit the same subtree several times. The second alternative is to let the tree pattern matcher execute all possible equality tests in advance. This is achieved by representing the input tree as a directed acyclic graph that contains exactly one node for each set of equal subtrees. This *subtree graph* can be computed in linear time [5].

4 Tree Parsing

A *regular tree grammar* G is a triple (N, Σ, P) where

- N is a finite set of *nonterminals*,
- Σ is the ranked alphabet of *terminals*, [3]
- P is a finite set of *rules* of the form $X \to s$ with $X \in N$ and $s \in T_{\Sigma \cup N}$.

Let $p : X \to s$ be a rule of P. p is called a *chain rule* if $s \in N$, and a *non-chain rule* otherwise. p is of *type* $(X_1, \ldots, X_k) \to X$ if the j-th occurrence of a nonterminal in s is X_j. For a right side s, we define the pattern $\tilde{s}$ as the pattern in $T_\Sigma(\{x_1, \ldots, x_k\})$ that one obtains from s by replacing, for all j, the j-th occurrence of a nonterminal by the variable x_j.

A X*-derivation tree* for a tree $t \in T_{\Sigma \cup N}$ is a tree $\psi \in T_{P \cup N}$ satisfying the following conditions:

- If $\psi \in N$ then $\psi = X = t$.
- If $\psi \notin N$ then $\psi = p(\psi_1, \ldots, \psi_k)$ for some rule $p : X \to s \in P$ of type $(X_1, \ldots, X_k) \to X$ such that $t = \tilde{s}[t_1 \backslash x_1, \ldots, t_k \backslash x_k]$ and ψ_j are X_j–derivation trees for the trees t_j.

Finally, we define the *derivation head*, $head(\psi)$, of a X–derivation tree ψ.

- If $\psi = X$ then $head(\psi) := X$.
- If $\psi = p(\psi_1, \ldots, \psi_k)$ and p is a non-chain rule then $head(\psi) := p$.

[3] The notion "terminals" was chosen in analogy to context free word grammars.

- If $\psi = p(\psi_1)$ and p is a chain rule then $head(\psi) := p \cdot head(\psi_1)$.

The derivation head of a X–derivation tree is also called X–derivation head.

Thus, the derivation head of ψ describes the "upper" part of ψ, which consists of a sequence of applications of chain rules followed by the first non-chain rule or a nonterminal.

For $X \in N$ we define the *language* of G relative to X as $\mathcal{L}(G, X) := \{t \in T_\Sigma \mid \exists \psi \in T_{P \cup N} : \psi$ is a X–derivation tree for $t\}$.

Example 4.1 Let G_1 be the regular tree grammar (N_1, Σ, P_1);
$\Sigma = \{a, cons, nil\}$ with $\rho(a) = \rho(nil) = 0, \rho(cons) = 2$, $N_1 = \{E, S\}$ and

$$P_1 = \{ \begin{array}{lcl} S & \rightarrow & nil, \\ S & \rightarrow & cons(E, S), \\ E & \rightarrow & a\} \end{array}$$

$\mathcal{L}(G_1, S)$ is the set of linear lists of a's including the empty list, i.e., $\mathcal{L}(G_1, S) = \{nil, cons(a, nil), cons(a, cons(a, nil)), \ldots\}$.

□

Example 4.2 Let G_m be the regular tree grammar (N_m, Σ, P_m);
$\Sigma = \{const, m, plus, REG\}$ with $\rho(const) = 0; \rho(m) = 1, \rho(plus) = 2$,
$N_m = \{REG\}$ and

$$P_m = \{ \begin{array}{llcl} addmc: & REG & \rightarrow & plus(m(const), REG), \\ addm: & REG & \rightarrow & plus(m(REG), REG), \\ add: & REG & \rightarrow & plus(REG, REG), \\ ldmc: & REG & \rightarrow & m(const), \\ ldm: & REG & \rightarrow & m(REG), \\ ldc: & REG & \rightarrow & const, \\ ld: & REG & \rightarrow & REG\} \end{array}$$

P_m describes a subset of an instruction set of a simple processor. The rules are labeled with the names of the corresponding instructions.
The first three rules represent instructions for addition that add

- the contents of a storage cell whose address is given by a constant
- the contents of a storage cell whose address resides in a register
- the contents of a register itself

to the contents of a register and store the resulting value into a register.
The remaining rules describe load instructions that load

- the contents of a storage cell whose address is given by a constant
- the contents of a storage cell whose address is given by the contents of a register
- a constant
- the contents of a register

into a register. □

Definition 4.1 (The Tree Parsing Problem)
An instance of the *tree parsing problem* consists of a regular tree grammar G together with a nonterminal X and an input tree t. The solution of the tree parsing problem for this instance is (a representation of) the set of all X–derivation trees of G for t. A *tree parser* for G is an algorithm that, for every tree t, returns the solution of the instance (G, t) of the tree parsing problem. A *tree parser generator* is an algorithm that, for every regular tree grammar G, returns a tree parser for G. □

Similar to regular word languages, the family of languages generated by regular tree grammars (relative to some nonterminal) has particularly nice properties. For example, it is closed under intersection and complementation. Also, emptyness can be decided in polynomial time.

In the following we show how finite tree automata can be used to implement both tree pattern matchers and tree parsers.

5 Finite Tree Automata

A *finite tree automaton* A is a 4-tuple $A = (Q, \Sigma, \delta, Q_F)$ where

- Q is a finite set of *states*;
- $Q_F \subseteq Q$ is the set of *final* or *accepting* states;
- Σ is the finite ranked input alphabet; and
- $\delta \subseteq \bigcup_{j \geq 0} Q \times \Sigma_j \times Q^j$ is the set of *transitions*.

The automaton A is called *deterministic* if for every $a \in \Sigma_k$ and every sequence $q_1, \ldots, q_k$ of states there is at most one transition $(q, a, q_1 \ldots q_k) \in \delta$. In this case, δ can be written as a partial function

- $\delta : \bigcup_{j \geq 0} \Sigma_j \times Q^j \to Q$

Given an input tree t the automaton A traverses t. At every visited node, A changes its state according to the transition from δ chosen at this node. If A is deterministic then there is at most one possible choice at every node, otherwise there may be several. Technically, we describe such a computation by an *annotation* of the input tree. For this, we introduce an extended alphabet $\Sigma \times Q$ whose operators now consist of *pairs* of operators from Σ and states of A.

So, let $\Sigma \times Q$ be the ranked alphabet $\{\langle a, q\rangle \mid a \in \Sigma, q \in Q\}$ where $\langle a, q\rangle$ has the same rank as a. A *q–computation* ϕ of the finite tree automaton A on the input tree $t = a(t_1, \ldots, t_m)$ is inductively defined as a tree $\phi = \langle a, q\rangle(\phi_1, \ldots, \phi_m) \in T_{\Sigma \times Q}$ where ϕ_j are q_j–computations for the subtrees t_j, $j = 1, \ldots, m$, for some states q_j such that $(q, a, q_1 \ldots q_m)$ is a transition from δ. ϕ is called *accepting*, if $q \in Q_F$. The language $\mathcal{L}(A)$ accepted by A consists of all trees for which an accepting computation of A exists. A transition $\tau \in \delta$ is called *superfluous* if it does not occur in any computation of A. Clearly, superfluous transitions can be removed from δ without changing the "behavior" of A.

In the literature a distinction is sometimes made between bottom-up tree automata and top-down tree automata which in the nondeterministic case are equivalent [8]. According to our definition of a computation the direction is irrelevant. In informal descriptions we allow ourselves any of the two views to support intuition.

If A is deterministic then at most one computation exists for every input tree. We can then extend the partial function δ to a partial function $\delta^* : T_\Sigma \to Q$ by: $\delta^*(t) = \delta(a, \delta^*(t_1) \ldots \delta^*(t_k))$ whenever $t = a(t_1, \ldots, t_k)$. For simplicity, we also denote δ^* by δ. Induction on the structure of t shows that $\delta^*(t)$ is defined and equals q if and only if a q–computation of A for t exists.

As a first application consider the tree pattern matching problem. Let τ be a linear pattern in $T_\Sigma(V)$. We want to construct a (possibly non-deterministic) finite tree automaton A_τ that detects whether or not τ matches the given input tree. Intuitively, A_τ works as follows. Outside of pattern τ, A_τ is in an unspecific state $\perp$; whereas inside of pattern τ the state memorizes the subpattern already traversed. Since we are not interested in the precise numbering of the variables we replace all variables by $\perp$ ("everything is matched by a variable"). Hence in the sequel, we assume that $\tau \in T_{\Sigma \cup \{\perp\}}$. Then we define $A_\tau := (Q_\tau, \Sigma, \delta_\tau, Q_{\tau,F})$ where $Q_\tau := \{s \mid s \text{ subpattern of } \tau\} \cup \{\perp\}$, $Q_{\tau,F} := \{\tau\}$, and δ_τ is defined as follows.

- $(\perp, a, \perp \ldots \perp) \in \delta$ for all $a \in \Sigma$;
- If $s \in Q_\tau$ and $s = a(s_1, \ldots, s_k)$ then $(s, a, s_1 \ldots s_k) \in \delta$.

Obviously, we have:

1. For every tree t there is a $\perp$–computation;
2. For a tree t there is a τ–computation if and only if τ matches t.

The example easily generalizes to the case of a set $T = \{\tau_1, \ldots, \tau_n\}$ of linear patterns of which we again w.l.o.g. assume that all occurrences of variables are replaced with $\perp$. As the set of states for the finite tree automaton A_T we now choose $Q_T := \bigcup_{j=1}^n Q_{\tau_j}$ with $Q_{T,F} := T$. The definition of the set of transitions does not change.

If we want to determine which patterns of T match an input tree t we just have to compute the set of all final states of accepting computations of A_T for t. A method to determine this set can be obtained from A_T by means of the *subset construction* for finite tree automata.

Definition 5.1 (Subset Construction I)
Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton. The corresponding subset automaton is the deterministic finite tree automaton $P(A) = (Q_1, \Sigma, \delta_1, Q_{1,F})$ with

- $Q_1 := 2^Q$ is the power set of Q;
- $Q_{1,F} := \{B \subseteq Q \mid B \cap Q_F \neq \emptyset\}$;
- δ_1 is the (total) function with
$\delta_1(a, B_1 \ldots B_k) = \{q \in Q \mid \exists q_1 \in B_1, \ldots, q_k \in B_k : (q, a, q_1 \ldots q_k) \in \delta\}$.

□

By induction on the structure of an input tree one finds:

Lemma 5.1 *Let $t \in T_\Sigma$. Then $\delta_1(t)$ is the set of all states $q \in Q$ for which there is a q–computation of A for t. Especially, $\mathcal{L}(A) = \mathcal{L}(P(A))$.* □

The subset construction provides an algorithm to generate tree pattern matchers. The generation proceeds in two stages. Given pattern set T, we first construct the finite tree automaton A_T. Then we apply the subset construction to obtain $P(A_T) = (Q, \Sigma, \delta, Q_F)$. Whenever an input tree t is given, the set $\delta(t) \cap T$ consists precisely of all patterns from T that match t.

Example 5.1 *Let $T = \{\tau_1, \tau_2\}$ with*

$$\tau_1 = b(a(a(X_1, X_2), X_3), X_4) \text{ and } \tau_2 = b(X_1, c(X_2, c(X_3, X_4))).$$

Then, $A_T = (Q_T, \Sigma, \delta_T, Q_{T,F})$ where

$$\begin{array}{llll} Q_T = \{\bot, & a(\bot,\bot), & a(a(\bot,\bot),\bot), & b(a(a(\bot,\bot),\bot),\bot), \\ & c(\bot,\bot), & c(\bot, c(\bot,\bot)), & b(\bot, c(\bot, c(\bot,\bot)))\ \}. \end{array}$$ □

A_T needs 7 states. If we apply Construction I we find that for our example (as for most practical cases) this construction is hopelessly inefficient: the generated automaton for the given two small (!) patterns has already $2^7 = 128$ states. It can be shown that in the worst case exponentially many states are inevitable. However, often most of the states introduced by Construction I are superfluous. For instance in our example the set $\{a(\bot,\bot), b(\bot, c(\bot, c(\bot,\bot)))\}$ is generated which consists of "contradicting" patterns, i.e., patterns that cannot match the same tree. Therefore, we present a "cheaper" Construction II, which from the beginning generates only such subsets of states that actually occur in computations of the subset automaton.

Definition 5.2 (Subset Construction II)
Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton. The corresponding (reduced) subset automaton is the deterministic finite tree automaton $P_r(A) = (Q_r, \Sigma, \delta_r, Q_{r,F})$ with $Q_{r,F} := \{B \in Q_r \mid B \cap Q_F \neq \emptyset\}$ whose sets of states and transitions are iteratively determined by $Q_r := \bigcup_{n \geq 0} Q_r^{(n)}$ and $\delta_r := \bigcup_{n \geq 0} \delta_r^{(n)}$ where

- $Q_r^{(0)} := \emptyset$; $\delta_r^{(0)} := \emptyset$;
- Assume $n > 0$. For $a \in \Sigma_k$ and $B_1, \ldots, B_k \in Q_r^{(n-1)}$ let $B := \{q \in Q \mid \exists q_1 \in B_1, \ldots, q_k \in B_k : (q, a, q_1 \ldots q_k) \in \delta\}$. If $B \neq \emptyset$ then $B \in Q_r^{(n)}$ and $(B, a, B_1 \ldots B_k) \in \delta_r^{(n)}$. □

Since for all n, $Q_r^{(n)} \subseteq Q_r^{(n+1)}$ and $\delta_r^{(n)} \subseteq \delta_r^{(n+1)}$ iteration can be terminated as soon as no new states are generated. Therefore, $Q_r = Q_r^{(n)}$ and $\delta_r = \delta_r^{(n)}$ for the first n with $Q_r^{(n)} = Q_r^{(n+1)}$. Consequently, the algorithm terminates after at most $2^{|Q|}$ iterations.
By induction on the structure of an input tree one finds:

Lemma 5.2

1. *For every $t \in T_\Sigma$ the following holds.*
   - *If $\delta_r(t)$ is undefined then no computation of A for t exists.*
   - *If $\delta_r(t)$ is defined then $\delta_r(t)$ is the set of all states q for which a q–computation of A for t exists.*
2. $\mathcal{L}(A) = \mathcal{L}(P_r(A))$.
3. *For every state $B \in Q_r$, a tree t exists such that $\delta_r(t) = B$.* □

Consider the finite tree automaton A_T. We find that no longer *all* subsets of subpatterns are generated as new states but only subsets that are *maximally compatible*. Here, a set $S \subseteq T$ of patterns is called *compatible* if a tree t exists that is matched by every pattern in S. S is called *maximally* compatible if there is a tree that is matched by all patterns from S and not matched by any of the patterns in $T \backslash S$.

The set of states Q_r of the reduced subset automaton for A_T consists exactly of the maximally compatible subsets S of subpatterns. Thus, we obtain for our example:

$$\begin{array}{llll} Q_r = \{ & \{\bot\}, & \{\bot, b(a(a(\bot,\bot),\bot),\bot),\ b(\bot, c(\bot, c(\bot,\bot)))\}, & \\ & \{\bot, a(\bot,\bot)\}, & \{\bot, a(\bot,\bot), a(a(\bot,\bot),\bot)\}, & \{\bot, b(a(a(\bot,\bot),\bot),\bot)\}, \\ & \{\bot, c(\bot,\bot)\}, & \{\bot, c(\bot,\bot), c(\bot, c(\bot,\bot))\}, & \{\bot, b(\bot, c(\bot, c(\bot,\bot)))\}\} \end{array}$$

The reduced subset automaton possesses eight states! Compared with 128 states, according to Construction I, this is a significant improvement.

6 How to Generate Tree Parsers

Let $G = (N, \Sigma, P)$ be a regular tree grammar and $X \in N$. To obtain a description of all possible X–derivation trees for a given input tree, we proceed as in the case of tree pattern matching. First, we construct a (possibly non-deterministic) finite tree automaton $A_{G,X}$ whose computations represent derivation trees of G. Then, we apply Subset Construction II to $A_{G,X}$. This automaton is the basis of our tree parser.

Intuitively, the finite tree automaton $A_{G,X}$ operates on input $t \notin N$ as follows. At the root, $A_{G,X}$ guesses an X–derivation head $p_1 \ldots p_k p$ of an X–derivation tree for t with $p : X' \to s$. Then, $A_{G,X}$ verifies that s in fact "fits" i.e., that $\tilde{s}$ matches t. If during verification, $A_{G,X}$ hits a node where the right hand side s contains a nonterminal X_j, then $A_{G,X}$ again guesses an X_j–derivation head and so on. Formally, we therefore define $A_{G,X} = (Q_G, \Sigma, \delta_G, \{X\})$ where $Q_G = N \cup \{s' \mid \exists X \to s \in P \text{ with } s' \text{ subpattern of } s\}$.

δ_G consists of two components: the first one is responsible for the verification of the right hand sides whereas the second one performs the guesses. We define:

$$\begin{array}{rl} \delta_G := & (\{(s, a, s_1 \ldots s_k) \mid s = a(s_1, \ldots, s_k) \in Q_G\}) \ \cup \\ & (\{(X, s, \epsilon) \mid s \in \Sigma \text{ and } \exists X\text{–derivation tree for } s\} \ \cup \\ & \{(X, a, s_1 \ldots s_k) \mid \exists X' \to s \in P : \exists X\text{–derivation tree for } X' \\ & \ \text{and } s = a(s_1, \ldots, s_k)\}) \end{array}$$

It can be shown that several of the tree parsers described in the literature can be obtained from $A_{G,X}$ as defined above by means of Subset Construction

II. However, taking a closer look at the definition one finds that δ_G contains several superfluous transitions! The automaton also allows $(s, a, s_1 \dots s_k)$ if s is a right hand side but itself is not a proper subpattern of another right hand side. Obviously, such transitions cannot be used by any X–computation. Therefore, we define instead:

$$\begin{array}{rl} \delta_G := & \{(s, a, s_1 \dots s_k) \mid s = a(s_1, \dots, s_k) \text{ is a proper subpattern of a right} \\ & \text{hand side}\} \;\cup \\ & \{(X, s, \epsilon) \mid s \in \Sigma \text{ and } \exists X\text{–derivation tree for } s\} \;\cup \\ & \{(X, a, s_1 \dots s_k) \mid \exists X' \to s \in P : \exists X\text{–derivation tree for } X' \\ & \text{ and } s = a(s_1, \dots, s_k)\} \end{array}$$

Example 6.1 Let G_m be the grammar from example 4.2. The non-deterministic automaton $A = (Q, \Sigma, \delta, Q_F)$ for G_m has the set of states

$$Q = \{const, REG, m(const), m(REG)\}$$

and the set of transitions:

$$\begin{array}{ll} \delta = \{ & (const, const, \epsilon) \\ & (REG, const, \epsilon) \\ & (REG, REG, \epsilon) \\ & (m(const), m, const) \\ & (REG, m, const) \\ & (m(REG), m, REG) \\ & (REG, m, REG) \\ & (REG, plus, m(const)\;\; REG) \\ & (REG, plus, m(REG)\;\; REG) \\ & (REG, plus, REG\;\; REG)\} \end{array}$$

The reduced subset automaton $A_r = (Q_r, \Sigma, \delta_r, Q_{F,r})$ for A has the set of states $Q_r = \{q_1, q_2, q_3, q_4\}$ where

$$\begin{array}{rcl} q_1 & = & \{REG\} \\ q_2 & = & \{const, REG\} \\ q_3 & = & \{m(REG), REG\} \\ q_4 & = & \{m(const), REG, m(REG)\} \end{array}$$

We have:

Lemma 6.1 *Let G be a regular tree grammar and t an input tree.*

- *A X–derivation tree of G exists for t if and only if a X–computation of $A_{G,X}$ exists for t.*

  Especially, $\mathcal{L}(G, X) = \mathcal{L}(A_{G,X})$.

- *Let $A = (Q, \Sigma, \delta, Q_F)$ be the (reduced) subset automaton for $A_{G,X}$. Then $\delta(t) \cap N = \{X' \in N \mid \exists X'\text{–derivation tree for } t\}$.* □

It is easy to reconstruct the X–derivation trees of G for a given input tree t from the X–computations of $A_{G,X}$ for t.[4] Hence, the tree parsing problem

[4]However, observe that the number of X–computations for t is always *finite* even if the number of X–derivation trees may be infinite. This is due to the fact that the set of derivation heads corresponding to a guessing transition may form an *infinite* regular set.

for a regular tree grammar G can be reduced to the problem of determining all accepting computations of a finite tree automaton A for a given input tree.

Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton and $A_r = (Q_r, \Sigma, \delta_r, Q_{r,F})$ the corresponding (reduced) subset automaton. The following algorithm nondeterministically traverses a q–computation of a tree t, given a B–computation of A_r for t. A corresponding deterministic algorithm, enumerating all q-computations, can be derived by systematically making all possible choices.

Let $\tau = (B, a, B_1 \dots B_k) \in \delta_r$ and $q \in B$. Define

$$\Theta(\tau)_q := \{(q, a, q_1 \dots q_k) \in \delta \mid q_1 \in B_1, \dots, q_k \in B_k\}$$

$\Theta(\tau)_q$ denotes the set of transitions of A corresponding to τ with q as successor state.

Assume $\phi = \langle a, B\rangle(\phi_1, \dots, \phi_k)$ is the B–computation of A_r for t. The algorithm traverses ϕ in pre-order. Let τ be the transition chosen at the root of ϕ. The algorithm selects a transition $(q, a, q_1 \dots q_k) \in \Theta(\tau)_q$. Then, recursively q_j–computations ψ_j, $j = 1, \dots, k$, are determined from the B_j–computations ϕ_j. Finally, the algorithm returns $\langle a, q\rangle(\psi_1, \dots, \psi_k)$. Different choices of transitions correspond to different derivation trees.

7 Application to Code Selection

We would like to apply our tree parser generator to the code selection problem. In this setting we have to select the "best" derivation tree from the possibly huge set of all derivation trees. As for the generation of tree parsers, we proceed in three steps. We assume that the rules of the grammar G are annotated with *cost functions* that describe the "costs" of the corresponding instruction. We translate these cost functions into cost functions for the transitions of the tree automaton $A_{G,X}$. Finally, we show how a cheapest computation of $A_{G,X}$ can be determined from the computation of the subset automaton for $A_{G,X}$.

Hence, assume that for every rule p of type $(X_1, \dots, X_k) \to X$ we are given a k-ary function $C(p) : \mathbb{N}_0{}^k \to \mathbb{N}_0$. C can be extended to a function that maps every derivation tree ψ to costs $C(\psi) \in \mathbb{N}_0$. If $\psi = X \in N$ then $C(\psi) := 0$. If $\psi = p(\psi_1, \dots, \psi_k)$ then $C(\psi) := C(p)C(\psi_1) \dots C(\psi_k)$, i.e., we apply the function $C(p)$ to the already computed values $C(\psi_1), \dots, C(\psi_k)$.

Analogously, we can attach cost functions to transitions of finite tree automata and use them to compute the costs of computations.

A cost measure C is called *monotone* or *additive* if $C(p)$ is monotone or of the form $C(p) = c_p + x_1 + \dots + x_k, c_p \in \mathbb{N}_0$, respectively, for all $p \in P$. Cost measures that are used in practice are usually additive. Examples are cost measures that compute the number of necessary processor cycles[5], the number of referenced memory cells, or the number of the operands of an instruction. A cost measure that is monotone but not additive is given by C_R, which computes the minimal number of registers necessary to evaluate an expression. The advantage of additive cost measures (even if they approximate "reality" in complex processor architectures rather roughly) is that they are easy to implement and maintain.

[5]This measure is not additive for all machines (e.g., Motorola 68020).

A simple cost measure is $C_\sharp$ which maps every chain rule p to the cost function $C_\sharp(p) = x_1$ and every non-chain rule p of type $(X_1, \ldots, X_k) \to X$ to the cost function $C_\sharp(p) = 1 + x_1 + \cdots + x_k$. Then for every derivation tree ψ, $C_\sharp(\psi)$ evaluates to the number of occurrences of non-chain rules in ψ.

We now translate the cost annotation C of the grammar G into a cost annotation C^* of the corresponding tree automaton $A_{G,X}$ for some nonterminal X of G. The proposed constructions assume an additive cost measure. In this case, we can represent the cost function of every rule by a cost in $\mathbb{N}_0$. Hence in the sequel we consider C as a function from $P \to \mathbb{N}_0$. Then we define C^* as follows.

- If $\tau = (X, a, \epsilon)$ for $a \in \Sigma$ then $C^*(\tau)$ is the minimal cost of a X–derivation tree for a.
- If $\tau = (s, a, s_1 \ldots s_k)$ with $s = a(s_1, \ldots, s_k)$ then $C^*(\tau) := 0$.
- If $\tau = (X, a, s_1 \ldots s_k)$ then $C^*(\tau)$ is the minimum of the values $(\gamma + C(p))$ for rules $p : X' \to a(s_1, \ldots, s_k)$ and minimal cost γ of a X–derivation tree for X'.

By this definition, the cost $C^*(\phi)$ of a X–computation ϕ coincides with the minimal cost of a X–derivation tree represented by ϕ.

More generally, assume $A = (Q, \Sigma, \delta, Q_F)$ is a finite tree automaton and $C : \delta \to \mathbb{N}_0$ is an additive cost measure. Let $A_r = (Q_r, \Sigma, \delta_r, Q_{r,F})$ be the (reduced) subset automaton for A. We modify the above algorithm enumerating the computations of A to return a cheapest computation of A only. The idea is to provide every choice point of the algorithm with information about an optimal choice. Assume ϕ is a B–computation of the subset automaton A_r for some input tree t. To ϕ we attach two tuples $C(\phi) = \langle C(\phi)_q \rangle_{q \in B}$ and $D(\phi) = \langle D(\phi)_q \rangle_{q \in B}$ where $C(\phi)_q$ contains the cost of a cheapest q–computation for t and $D(\phi)_q$ the transition selected at the root of a q–computation with cost $C(\phi)_q$. The tuples $C(\phi/n)$ and $D(\phi/n)$ for all nodes n of ϕ can be computed during a post-order traversal of ϕ.

Now, the modified algorithm traverses the computation ϕ of A_r for t in pre-order. At every node n it selects the corresponding component from $D(\phi/n)$. Thus, it returns a q–computation of A for t with minimal costs.

Computing all the tuples of costs can be rather expensive. Therefore, Pelegri–Llopart succeeded to incorporate the cost computations into the state transitions themselves [23, 24]. He observed that for usual machine grammars $G_m = (N_m, \Sigma, P_m)$, the differences of the minimal costs of X–derivation trees and X'–derivation trees for the same tree t are bounded by some constant. This is due to the presence of MOVE instructions in most physical machines that introduce chain rules between (almost) all nonterminals [24]. This is the reason why it suffices to consider *cost differences* instead of total costs, when constructing a cheapest computation of the non-deterministic automaton $A = (Q, \Sigma, \delta, Q_F)$ from the computation of the subset automaton.

Finitely many cost differences can be compiled into the states of the subset automaton. This results in the subset automaton $A_c = (Q_c, \Sigma, \delta_c, Q_{c,F})$. Assume $\mu \in Q_c$ is a state of A_c. Then μ contains information about reached states q of A together with a cost difference d, i.e., $\mu : B \to \mathbb{N}_0$ maps a subset B of Q to $\mathbb{N}_0$. For $q \in B$, the cost difference $\mu(q)$ describes the difference between the cost of a cheapest q–computation of A and the cost of a cheapest computation.

Definition 7.1 (Subset Construction III)
Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton and $C : \delta \rightarrow \mathbb{N}_0$ a cost function that maps every transition from δ to a cost in $\mathbb{N}_0$. The corresponding *(reduced) subset automaton with integrated costs* is the deterministic finite tree automaton $P_c(A) = (Q_c, \Sigma, \delta_c, Q_{c,F})$ with $Q_{c,F} := \{\mu : B \rightarrow \mathbb{N}_0 \in Q_c \mid B \cap Q_F \neq \emptyset\}$ whose sets of states and transitions are iteratively given by $Q_c := \bigcup_{n\geq 0} Q_c^{(n)}$ and $\delta_c := \bigcup_{n\geq 0} \delta_c^{(n)}$ as follows.

- $Q_c^{(0)} := \emptyset$; $\delta_c^{(0)} := \emptyset$;
- Let $n > 0$. For $a \in \Sigma_k$ and $\mu_j \in Q_c^{(n-1)}$ with domain B_j, $j = 1, \ldots, k$, let $B := \{q \in Q \mid \exists q_j \in B_j : (q, a, q_1 \ldots q_k) \in \delta\}$ where for $q \in B$, $d_q := min\{C(\tau) + \mu(q_1) + \ldots + \mu(q_k) \mid \exists q_j \in B_j : \tau = (q, a, q_1 \ldots q_k) \in \delta\}$. If $B \neq \emptyset$, define the mapping $\mu : B \rightarrow \mathbb{N}_0$ by $\mu(q) := d_q - d$ where $d := min\{d_q \mid q \in B\}$.
  Then μ is added to $Q_c^{(n)}$ and $(\mu, a, \mu_1 \ldots \mu_k)$ is added to $\delta_c^{(n)}$. □

The algorithm to construct a q–computation of A from a computation of the subset automaton can be used to compute cheapest computations when given a computation of the (reduced) subset automaton with integrated costs $A_c = (Q_c, \Sigma, \delta_c, Q_{c,F})$ just by replacing the sets of transitions $\Theta(\tau)_q$ with $\Theta_c(\tau)_q$. Here, for a transition $\tau = (\mu, a, \mu_1 \ldots \mu_k) \in \delta_c$ and q in the domain of μ, $\Theta_c(\tau)_q$ denotes the set of cheapest transitions of A contained in τ with q as successor state. Assume B_j is the domain of μ_j, $j = 1, \ldots, k$. Then $\Theta_c(\tau)_q$ formally is defined by $\Theta_c(\tau)_q := \{\eta = (q, a, q_1 \ldots q_k) \in \delta \mid q_1 \in B_1, \ldots, q_k \in B_k$ such that $C(\eta) + \mu_1(q_1) + \ldots + \mu_k(q_k)$ is minimal$\}$. Let $\phi = \langle a, B\rangle(\phi_1, \ldots, \phi_k)$ be the B–computation of A_c for some input tree t. The algorithm traverses ϕ in pre-order. Let τ be the transition at the root of ϕ. The algorithm chooses a transition $(q, a, q_1 \ldots q_k)$, now from $\Theta_c(\tau)_q$. Then, it recursively determines q_j–computations ψ_j, $j = 1, \ldots, k$ from ϕ_j. As the result it returns $\langle a, q\rangle(\psi_1, \ldots, \psi_k)$. All possible outputs are cheapest q–computations of A.

The advantage of subset automata with integrated costs is that they allow for a much faster construction of a cheapest computation of the original tree automaton. In general, the subset automata of Construction III are larger than the corresponding automata generated by Construction II without cost differences. Also, Construction III terminates only for finite tree automata with cost functions where the differences between cheapest computations for trees t are bounded by a constant.

We give a sufficient condition guaranteeing that the difference between the costs of cheapest computations for every input tree is bounded.

A subset $S \subseteq Q$ is called

- *unavoidable* iff the number of computations ϕ without using states from S is finite;
- *dominating* iff for all trees $t = a(t_1, \ldots, t_k) \in T_\Sigma$, all computations $\phi = \langle a, q\rangle(\phi_1, \ldots, \phi_k)$ of A for t with $\tau = (q, a, q_1, \ldots, q_k)$ being the transition at the root of ϕ, and every $s \in S$ the following holds: whenever there is a s–computation of A for t then also $(s, a, q_1 \ldots q_k) \in \delta$.

In the context of code selection, S corresponds to the set of "universal" registers; "unavoidable" means that every instruction sequence for expressions former or later uses such registers whereas the property of being "dominating" corresponds to the existence of *move*–instructions. Note that for the tree automaton of example 6.1, the set $\{REG\}$ is both unavoidable and dominating. We prove:

Lemma 7.1 *Assume that some set S of states exists for a tree automaton A that is both unavoidable and dominating. Then* for every *additive cost function* $\mathbf{c}$ *the difference of costs of cheapest computations is bounded.*

Proof: We extend the notion of a computation of a tree automaton to computations on input trees containing variables. Assume $V = \{x_1, \ldots, x_k\}$ and $t \in T_\Sigma(V)$. A $(q, p_1 \ldots p_k)$*–computation* ϕ of the finite tree automaton A on t is a tree $\phi \in T_{\Sigma \times Q}(V)$ with the following properties. $\phi = x_j$ whenever $t = x_j$, and $q = p_j$, and $\phi = \langle a, q\rangle(\phi_1, \ldots, \phi_m)$ whenever $t = a(t_1, \ldots, t_m)$, where ϕ_j are $(q_j, p_1 \ldots p_k)$–computations for the subtrees t_j, $j = 1, \ldots, m$ (for suitable states q_j) and $(q, a, q_1 \ldots q_m)$ is a transition from δ.

Assume $n = \#Q$, L is the maximal rank of a symbol of the input alphabet, Δ denotes the maximal cost of a transition of A, i.e., $\Delta = \max\{\mathbf{c}(\tau) | \tau \in \delta\}$.

Consider some tree $t \in T_\Sigma$, and let ϕ and ϕ' denote the cheapest computation and the cheapest q–computation respectively of A for t. If $q \in S$ then $\mathbf{c}(\phi') - \mathbf{c}(\phi) \leq \Delta$. Otherwise, we decompose $t = t_0[t_1, \ldots, t_m]$, $\phi = \phi_0[\phi_1, \ldots, \phi_m]$ and $\phi' = \phi'_0[\phi'_1, \ldots, \phi'_m]$ such that

1. For all $i \geq 0$, ϕ_i and ϕ'_i are computations of A for t_i;
2. No variable occurs in t_0 (and in ϕ_0 and ϕ'_0) more than once;
3. ϕ'_0 is a $(q, s_1 \ldots s_m)$–computation without using states from S, but where $s_1, \ldots, s_m \in S$.

Since S is unavoidable, $depth(\phi'_0) = depth(t_0) < n$. Hence, $m \leq L^{n-1}$, and $\mathbf{c}(\phi'_0) < \Delta \cdot 2 \cdot L^{n-1}$.

Since S is dominating, $\mathbf{c}(\phi'_j) - \mathbf{c}(\phi_j) \leq \Delta$ for $j > 0$. Therefore, we can conclude:

$$\begin{aligned}
\mathbf{c}(\phi') - \mathbf{c}(\phi) &\leq \mathbf{c}(\phi'_0) + \sum_{j=1}^{m} [\mathbf{c}(\phi'_j) - \mathbf{c}(\phi_j)] \\
&\leq 2 \cdot \Delta \cdot L^{n-1} + \Delta \cdot L^{n-1} \\
&= 3 \cdot \Delta \cdot L^{n-1}
\end{aligned}$$

This implies the assertion. □

Having constructed the subset automaton for A, the sufficient condition provided by Lemma 7.1 can be tested rather efficiently. Note however, that it is *independent* of the cost function $\mathbf{c}$. Therefore, it does not detect boundedness implied by the specific choice of the cost function.

Example 7.1 Let G_c be the grammar from example 4.2 with following costs:

$$
\begin{array}{lllll}
P_c = \{ & addmc: & REG & \rightarrow & plus(m(const), REG) & cost: 3, \\
 & addm: & REG & \rightarrow & plus(m(REG), REG) & cost: 3, \\
 & add: & REG & \rightarrow & plus(REG, REG) & cost: 2, \\
 & ldmc: & REG & \rightarrow & m(const) & cost: 2, \\
 & ldm: & REG & \rightarrow & m(REG) & cost: 2, \\
 & ldc: & REG & \rightarrow & const & cost: 1, \\
 & ld: & REG & \rightarrow & REG & cost: 1\}
\end{array}
$$

The subset automaton $A_c = (Q_c, \Sigma, \delta_c, Q_{F,c})$ according to Subset Construction III has the set of states $Q_c = \{\mu_1, \mu_2, \mu_3, \mu_4\}$ where

$$
\begin{array}{llll}
\mu_1: & REG & \rightarrow & 0 \\
\mu_2: & const & \rightarrow & 0 \\
 & REG & \rightarrow & 1 \\
\mu_3: & m(REG) & \rightarrow & 0 \\
 & REG & \rightarrow & 2 \\
\mu_4: & m(const) & \rightarrow & 0 \\
 & REG & \rightarrow & 2 \\
 & m(REG) & \rightarrow & 1
\end{array}
$$

The next section deals with efficient implementation techniques for tree automata. For simplicity, we will only consider subset constructions without integrated costs.

8 Implementation of Deterministic Finite Tree Automata

Assume $A = (Q, \Sigma, \delta, Q_F)$ is a deterministic tree automaton. As a first approach, we represent the set δ_a of transitions for an operator a of rank k as a k–dimensional array M_a. $M_a[q_1, \ldots, q_k] = \delta(a, q_1 \ldots q_k)$ whenever δ is defined for these arguments and $M_a[q_1, \ldots, q_k] = \perp$ otherwise, where $\perp$ is a special error symbol.

Assume that the input tree t is given as a node labeled ordered rooted tree. The state at a node n with label $a \in \Sigma_k$ is determined as $M_a[q_1, \ldots, q_k]$ where $q_1, \ldots, q_k$ are the states at the children of n. The states are computed during a post-order traversal of t. Therefore, the computation time for such a "run" of the tree automaton consists of the time spent for the tree traversal, which is proportional to the size of t, together with an indexed array access for every node in t.

In most existing processors, the time for such an array access is proportional to the number of indices. Since every subtree of t takes part in indexed accesses only once the total time is linear in the size of t and independent of the ranks of operators.

Example 8.1 The reduced subset automaton $A_r = (Q_r, \Sigma, \delta_r, Q_{F,r})$ from example 6.1 has the set of transitions δ_r (represented as tables):

$\delta_{r,const} =$ q_2

$\delta_{r,REG} =$ q_1

$\delta_{r,m} =$

child			
q_1	q_2	q_3	q_4
q_3	q_4	q_3	q_3

$\delta_{r,plus} =$

		right child			
		q_1	q_2	q_3	q_4
	q_1	q_1	q_1	q_1	q_1
left	q_2	q_1	q_1	q_1	q_1
child	q_3	q_1	q_1	q_1	q_1
	q_4	q_1	q_1	q_1	q_1

The representation of δ as a set of arrays is usually rather storage space consuming because the size of an array M_a for an operator $a \in \Sigma_k$ is proportional to $|Q|^k$, i.e., exponential in the rank of a no matter how many (few) defined transitions the implemented automaton has for a. In most of the cases occurring in practice the necessary storage space can significantly be reduced by standard table compression methods.

An alternative method to represent the transition functions δ_a is provided by *decision trees* (see figure 2). Let Q and D be finite sets and $H : Q^k \to D$ a partial function. A *decision tree* for H is a leaf labeled tree of height k whose set of vertices V is given by $V = V_0 \cup \ldots \cup V_k$ where

$$V_j := \{q_1 \ldots q_j \mid \exists q_{j+1}, \ldots, q_k \in Q : H(q_1 \ldots q_j q_{j+1} \ldots q_k) \text{ is defined } \}.$$

Here, nodes $q_1 \ldots q_{j-1}$ and $q_1 \ldots q_{j-1}q'$ are connected by a directed edge (of level j) with label q'. Furthermore, the leaves $b = q_1 \ldots q_k$ are labeled with $H(q_1 \ldots q_k)$.

For an operator a of rank k, we can represent δ_a by the decision tree for the function H_a that is given by $H_a(q_1 \ldots q_k) := \delta(a, q_1 \ldots q_k)$. The nodes of the decision tree exactly represent the prefixes of state sequences occurring in δ_a. The state at a node n with label $a \in \Sigma_k$ in the input tree for whose sons we have already computed the states $q_1, \ldots, q_k$ is obtained by following the path in the decision tree for δ_a whose edges successively are labeled with $q_1, \ldots, q_k$. The label at the final node on this path yields the desired result.

In case of the above presented tree parser generator the generated deterministic finite tree automaton is of the form $P_r(A_{G,X})$ for some regular tree grammar G and some nonterminal X of G. Here, we are not only interested in the state (of the subset automaton) at some node n but also in the set of possible transitions of the tree automaton $A_{G,X}$ at n. Accordingly, we choose H_a "more informative", i.e., as

$$H_a(B_1 \ldots B_k) := \langle \delta(a, B_1 \ldots B_k), \Theta \rangle$$

where $\Theta := \{(q, a, q_1 \ldots q_k) \in \delta_G \mid q_j \in B_j\}$.

If we represent the tree automaton by decision trees then the required storage space for the automaton is proportional to the size of δ. However, if the transition function δ of the deterministic finite tree automaton A is *total* (which is always the case for automata obtained by Subset Construction I) then the decision trees have the same size as the arrays.

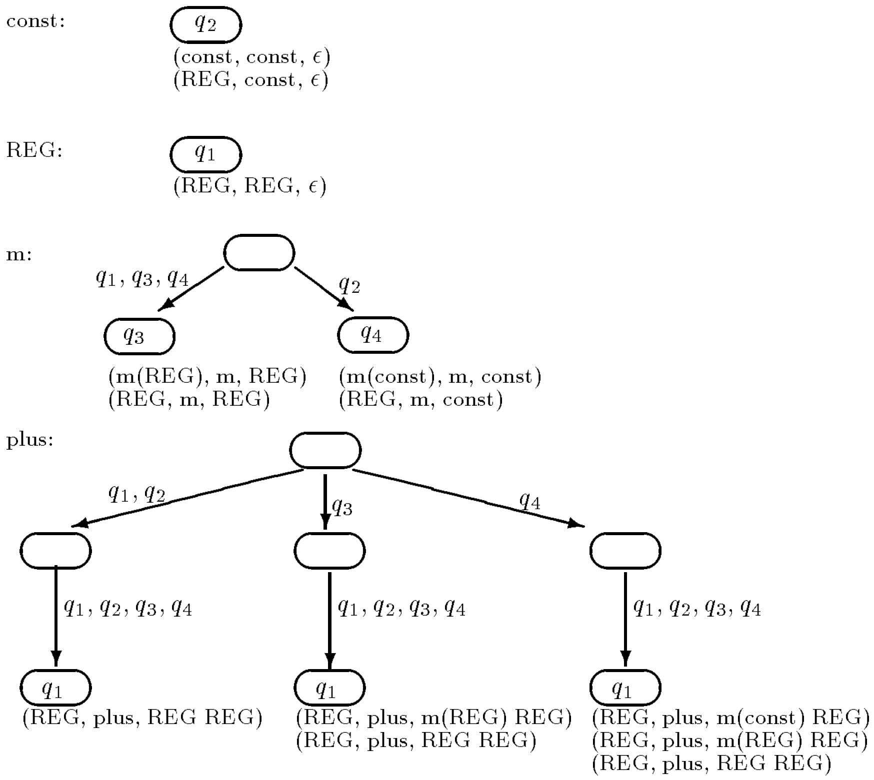


Figure 2: Compressed decision trees for the grammar of example 4.2

Decision trees can (possibly) more economically be represented by identifying isomorphic subtrees. Such a graph is called a *compressed decision tree* or *decision graph*. Especially, we can represent a decision tree by its subtree graph. A (compressed) decision tree T_a again can be represented by a 2-dimensional array N_a whose first component is indexed with the inner nodes v of T_a and whose second component is indexed with labels q of edges. The entry $N_a[v,q]$ contains the successor node of v in T_a when following the edge labeled with q, if such a node exists, and $\perp$ otherwise.

In the worst case, arrays N_a are up to a linear factor of the same sizes as arrays M_a. In practice however, arrays N_a are significantly smaller. Again, the arrays N_a can be subject to all kinds of further table compression methods. For example [2] consider various decision tree and table compression methods for tree automata.

Uncompressed decision trees often do not fit into main storage. Hence, one is interested in *directly* producing compressed decision trees during the generation phase. This leads to Subset Construction IV.

Let $A = (Q, \Sigma, \delta, Q_F)$ be a (non-deterministic) finite tree automaton and $a \in \Sigma_k$. The idea of Subset Construction IV is to generate a decision graph for the set of transitions $(\delta_r)_a$ of the reduced subset automaton for a such that the nodes are sets of transitions of A tagged with the level of the node. The root is the complete δ_a tagged 0. Edges are labeled with sets of states. An edge with label B leads from a node $(v, j-1)$ to a node (v', j) if and only if v' consists of all transitions of v that have a j-th argument $q \in B$. A leaf b is additionally labeled with the set of all states that occur as left sides in transitions in b. The generation of this graph can be combined with the Subset Construction II.

Definition 8.1 (Subset Construction IV)
Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton. For $n \geq 0$ we define a set of states $Q_s^{(n)}$ and graphs $T_a^{(n)}$, $a \in \Sigma$, as follows.

$Q_s^{(0)} := \emptyset$ and $T_a^{(0)}$, $a \in \Sigma$, are empty graphs.

Let $n > 0$ and $a \in \Sigma_k$. Then $T_a^{(n)}$ has a set of vertices $V = V_0 \cup \ldots \cup V_k$ and a set of edges E, which are defined as follows.

- $V_0 := \{(\delta_a, 0)\}$.
- Assume $j > 0$ and V_{j-1} has already been defined. For every $(v, j-1) \in V_{j-1}$ and $B \in Q_s^{(n-1)}$ consider the set $v' := \{(q, a, q_1 \ldots q_k) \in v \mid q_j \in B\}$. If $v' \neq \emptyset$ then we add (v', j) to the set V_j and an edge (v, v'), with label B, to E.
- Every node $(v, k) \in V_k$ is labeled with the set $\{q \in Q \mid \exists q_1, \ldots q_k \in Q : (q, a, q_1 \ldots q_k) \in v\}$.

The set $Q_s^{(n)}$ is the set of all labels of leaves (b, k) of graphs $T_a^{(n)}$, $a \in \Sigma$. □

$T_a^{(n)}$ is a subgraph of $T_a^{(n+1)}$, and $T_a^{(n)} = T_a^{(n+1)}$ provided $Q_s^{(n)} = Q_s^{(n+1)}$. By induction over n one proves:

Lemma 8.1 *Assume $A = (Q, \Sigma, \delta, Q_F)$ is a finite tree automaton, and $Q_r^{(n)}$ and $\delta_r^{(n)}$ are the n-th approximations to the sets of states and transitions, respectively, of the reduced subset automaton according to Subset Construction II. Then for all $n \geq 0$:*

- $Q_s^{(n)} = Q_r^{(n)}$;
- *The graph obtained from $T_a^{(n)}$ by removing all nodes of which no labeled leaf can be reached is a decision graph for $(\delta_r^{(n)})_a$.* □

Therefore, define $T_a := T_a^{(n)}$ for the first n with $Q_s^{(n)} = Q_s^{(n+1)}$. Assume A itself has no superfluous transitions. It turns out that then T_a does not contain useless nodes, i.e., it is a compressed decision tree for $(\delta_r)_a$.

For our generated tree parser every leaf b should not only contain the state of the subset automaton but also the set of the transitions of A that correspond to the path to (b, k), namely b itself! Provided the finite tree automaton A did not contain superfluous transitions, the generated decision graph equipped with this extended labeling H_a is *minimal*.

Lemma 8.2 *Assume $A = (Q, \Sigma, \delta, Q_F)$ is a finite tree automaton without superfluous transitions. Then for all $a \in \Sigma$:*

- *T_a is a decision graph for δ_a;*
- *T_a is isomorphic to the subtree graph of the decision tree for H_a.*

Proof: We only prove the second statement. Assume T_a were not isomorphic to the subtree graph of the decision tree for H_a. Then two nodes $(v, j) \neq (v', j)$ exist from which the same set of leaves are reachable. However, if every transition of A is used in some computation then the set of transitions of a node in T_a is precisely the union of the sets of transitions of the leaves reachable from it. Hence, $v = v'$ in contradiction to our assumption. □

Construction IV generalizes the method (implicitly) described in [18], which for the special case of tree pattern matchers directly generates compressed decision trees. [27] use a similar construction to derive tree parsers.

The Construction IV is optimal in that it returns decision graphs with a minimal number of nodes. However, Chase in [Ch87] observed that many states of the generated subset automaton are equivalent w.r.t. transitions. Again, let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton and $a \in \Sigma_k$. For $j = 1, \ldots, k$ we define the set $Q_{a,j} := \{q_j \mid (q, a, q_1 \ldots q_k) \in \delta\}$. The (a, j)*–relevant part* of a set $B \subseteq Q$ is the set $B \cap Q_{a,j}$. Within the decision graph T_a of Construction IV, sets of states with identical (a, j)-relevant parts lead from the same nodes $(v, j-1)$ to the same nodes (v', j).

The fifth subset construction returns compressed decision trees T'_a whose sets of vertices coincide with those of the T_a, but whose edges of level j are only labeled with (a, j)-relevant parts. Thus, the decision graphs themselves can be represented much more succinctly. The price one has to pay is to maintain separate tables that, for all $a \in \Sigma_k$, $j \in \{1, \ldots, k\}$, and occurring sets of states B, contain the (a, j)-relevant parts of B. Determining the state at a node n with label $a \in \Sigma_k$ of the input tree is now done in two steps. First for the states $B_1, \ldots, B_k$ at the children of n, we successively have to look up the (a, j)-relevant parts B'_j. Then, the path with edge labeling $B'_1, \ldots, B'_k$ in the decision graph for a yields the result.

As in Subset Construction IV, we construct the modified decision graphs by "need".

Definition 8.2 (Subset Construction V)
Let $A = (Q, \Sigma, \delta, Q_F)$ be a finite tree automaton. For $n \geq 0$ we define a set of states $Q_s^{(n)}$ graphs $T_a^{'(n)}$, $a \in \Sigma_k$, together with sets $R_{a,j}^{(n)} = \{B \cap Q_{a,j} \mid B \in Q_s^{(n)}\} \backslash \{\emptyset\}$ for $1 \leq j \leq k$ as follows.

$Q_s^{(0)} := \emptyset$ and $T_a^{'(0)}$, $a \in \Sigma$ are empty graphs.

Let $n > 0$ and $a \in \Sigma_k$.

$T_a^{'(n)}$ has a set of vertices $V = V_0 \cup \ldots \cup V_k$ and a set of edges E, which are defined as follows.

- $V_0 := \{(\delta_a, 0)\}$.

- Assume $j > 0$ and V_{j-1} has already been defined. For every $(v, j-1) \in V_{j-1}$ and $B \in R_{a,j}^{(n-1)}$ consider the set $v' := \{(q, a, q_1 \dots q_k) \in v \mid q_j \in B\}$. If $v' \neq \emptyset$ then we add (v', j) to the set V_j and an edge $((v, j-1), (v', j))$, with label B, to E.

- Every node $(v, k) \in V_k$ is labeled with the set $\{q \in Q \mid \exists q_1, \dots q_k \in Q : (q, a, q_1 \dots q_k) \in v\}$.

The set $Q_s^{(n)}$ is the set of all labels of leaves (b, k) in $T_a^{'(n)}$, $a \in \Sigma$. □

Chase uses the idea of an equivalence relation on states to generate compressed tables for pattern matching [3]. C. Peper observes that in case of operators of arity[6] 2, the sets of states corresponding to the labels on edges between two nodes of the compressed decision tree in Construction IV coincide with Chase's equivalence classes [25]. For greater arities, our Construction V generalizes the methods both of Chase and Kron.

9 Practical Experience

The presented subset constructions have been implemented in an experimental generator [25]. Given an annotated regular tree grammar it produces tables and a driver program that selects code for an IR tree. As an example, the processor NSC32000 was described by a grammar [19]. This grammar has 761 rules with 49 nonterminals and 173 terminals. The tree parser generated according Subset Construction II has 791 states. The arrays M for this parser would have 52938937 entries. The representation as compressed decision trees, i.e., as arrays N_a according to Subset Construction V needs 13241 entries. Applying table compression methods as described in [2] we can further reduce the needed storage space to 3771 entries. The generation time including table compression on a SPARC station II was 10.7 seconds.

The compressed parser tables according to Subset Construction V with integrated costs (processor cycles) for the grammars `vax.bwl` and `mot.bwl` that describe integer (byte, word, long) subsets of the VAX and the Motorola 68000 need 7317 and 61572 entries.

For detailed numbers on generation times and table sizes of code selectors we refer the reader to [3, 16, 23].

10 Conclusion

The present paper systematically presents the basic theory behind code selection. We exhibit the connections both of tree pattern matching and tree parsing algorithms to finite tree automata. We present general methods for the implementation of tree automata. Especially, we generalize the method of Kron to arbitrary tree automata. This makes it possible to write efficient generators of efficient code selectors.

Our results are also of interest for other areas. Both tree pattern matching and tree parsing are useful for checking the syntactic applicability of tree

[6] In most existing systems the arity of operators is restricted to at most 2.

transformation rules [7, 22]. Especially tree pattern matching is used for the implementation of term rewriting systems [13].

Acknowledgements

We would like to thank Reinhold Heckmann and the unknown referees for their helpful comments on this paper, and Todd A. Proebsting for supplying his machine grammars.

References

[1] Aho AV, Ganapathi M: *Efficient Tree Pattern Matching: An Aid to Code Generation.* In: Proc. of the 12th ACM Symp. on Principles of Programming Languages, 1985, pp 334-340

[2] Börstler J, Möncke U, Wilhelm R: *Table Compression for Tree Automata.* In: ACM Transactions on Programming Languages and Systems, Vol. 13, No. 3, July 1991, pp 295-314

[3] Chase DR: *An improvement to bottom-up tree pattern matching.* In: Proc. of 14th ACM Symposium on Principles of Programming Languages, 1987, pp 168-177

[4] Dauchet M, Deruyver A: *Compilation of Ground Term Rewriting Systems and Applications.* In Dershowitz (ed) Proceedings of the Conference: Rewriting Techniques and Applications, Springer, 1989, LNCS 355, pp 556-558

[5] Downey PJ, Sethi R, Tarjan ER: *Variations on the common subexpression problem.* In: JACM 27 , 1980, pp 758-771

[6] Emmelmann H: *Code Selection by Regularly Controlled Term Rewriting.* In: Giegerich, Graham (eds): Code Generation– Concepts, Tools, Techniques, Springer, 1991, Workshops in Computing Series, pp3-29

[7] Ferdinand C: *Pattern Matching in a Functional Transformation Language using Treeparsing.* In: Deransart, Małuszyński (eds): Proceedings of the Workshop: Programming Language Implementation and Logic Programming 90, Springer, 1990, LNCS 456, pp 358-371

[8] Gecseg F, Steinby M: *Tree Automata.* Akademiai Kiado, Budapest, 1984

[9] Giegerich R: *Code Selection by Inversion of Order-sorted Derivors.* In: Theoretical Computer Science 73, 1990, pp 177-211

[10] Giegerich R, Schmal K: *Code Selection Techniques: Pattern Matching, Tree Parsing, and Inversion of Derivors.* In: Ganzinger (ed) Proc. ESOP 88, Springer, 1988, LNCS 300, pp 247-268

[11] Glanville RS: *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers.* Ph.D. Thesis, Univ. of California, Berkeley, 1977

[12] Glanville RS, Graham SL: *A new Method for Compiler Code Generation.* In: Proc. of the 5th ACM Symp. on Principles of Programming Languages, 1978, pp. 231-240

[13] Gräf A: *Left-to-Right Tree Pattern Matching.* In Book (ed) Proceedings of the Conference: Rewriting Techniques and Applications, Springer, 1991, LNCS 488, pp 323-334

[14] Henry RR: *Graham–Glanville Code Generators.* Ph.D. Thesis, Univ. of California, Berkeley, 1984

[15] Henry RR, Damron PC: *Algorithms for Table-Driven Code Generators Using Tree-Pattern Matching.* Technical Report # 89-02-03, University of Washington, Seattle, 1989

[16] Henry RR, Damron PC: *Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree-Pattern Matcher.* Technical Report # 89-02-04, University of Washington, Seattle, 1989

[17] Hoffmann DM, O'Donnell MJ: *Pattern Matching in Trees.* In: JACM 29,1, 1982, pp 68-95

[18] Kron H: *Tree Templates and Subtree Transformational Grammars.* Ph.D. Thesis, Univ. of California, Santa Cruz, 1975

[19] Mathis N: *Weiterentwicklung eines Codeselektorgenerators und Anwendung auf den NSC32000.* Diplomarbeit, Universität des Saarlandes, 1990

[20] Möncke U: *Simulating Automata for Weighted Tree Reductions.* Technischer Bericht Nr. A10/87, Universität des Saarlandes, 1987

[21] Nivat M, Podelski A (eds): *Tree Automata and Languages.* Elsevier Science Publishers, 1992, Amsterdam, the Netherlands

[22] Möncke U, Weisgerber B, Wilhelm R: *Generative support for transformational programming.* In: ESPRIT: Status Report of Continuing Work, Elsevier Sc., Brussels, 1986

[23] Pelegri-Llopart E: *Rewrite Systems, Pattern Matching, and Code Generation.* Ph.D. Thesis, Univ. of California, Berkeley, 1988

[24] Pelegri-Llopart E, Graham SL: *Optimal Code Generation for Expression Trees: An Application of BURS Theory.* In: Proc. of the 15th ACM Symposium on Principles of Programming Languages, 1988, San Diego, CA, pp 294-308

[25] Peper C: *Generische Baumanalysatoren-Generatoren.* Diplomarbeit, Universität des Saarlandes, 1993

[26] Ripken K: *Formale Beschreibungen von Maschinen, Implementierungen und optimierender Maschinencode–Erzeugung aus attributierten Programmgraphen.* Dissertation, TU München, 1977

[27] Weisgerber B, Wilhelm R: *Two tree pattern matchers for code selection.* In: Hammer (ed) Proceedings of the Workshop: Compiler Compilers and High Speed Compilation, Springer, 1988, LNCS 371, pp 215-229