Tree Parsing for Code Selection

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

July 7, 1997

Code Generation

Real machines instead of abstract machines:

- Register machines,
- Limited resources (registers, memory),
- Fixed word size,
- Storage hierarchy,
- Intraprocessor parallelism.

Phases in code generation

code selection: selecting semantically equivalent sequences of machine instructions for programs,

register allocation: exploiting the registers for storing values of variables and temporaries,

instruction scheduling: reordering instruction sequences to exploit intraprocessor parallelism.

Code Selection

Task: Select (best) instruction sequences for a program.

- Control statements – translated as for abstract machines,
- Procedure organisation – same as on abstract machines,
- Expressions, variable and data structure access – many different translations.

Expressions to be translated into (locally) optimal code according to some cost measure.

An Example Architecture, the Motorola 68000

- 8 Data registers,
- 8 Address registers,
- many addressing modes,
- 2–address machine, i.e., two operand locations in each instruction, one is also the result location,

  ADD D1, D2

  adds the contents of registers D1 and D2 and stores the result in D2.
- most instructions are scalable to byte (.B), word (.W), double word (.L) operands.

Addressing Modes

- D_n **Data register direct:** *cont*(D_n).
- A_n **Address register direct:** *cont*(A_n).
- (A_n) **Address register indirect:** *St(cont*(A_n)*)*.
- $d(A_n)$ **Addressregister indirect with address distance:** *St(cont*$(A_n) + d$*)* with 16-Bit-constant d.
- $d(A_n, I_x)$ **Addressregister indirect with Index and Address distance:** $St(cont(A_n)+cont(I_x)+d)$ with A_n used as base register, I_x index register (either address or data register), 8-Bit-distance d.
- x **Absolute short:** *St*(x) with 16-Bit-constant x.
- x **Absolute long:** *St*(x) with 32-Bit-constant x.
- $\#x$ **Immediate:** x.

Execution Times

	Addressing mode	Byte, Word	Double Word
D_n	Data register direct	0	0
A_n	Address register direct	0	0
(A_n)	Address register indirect	4	8
$d(A_n)$	Address register indirect with Address distance	8	12
$d(A_n, I_x)$	Address register indirect with Index and Address distance	10	14
x	Absolute short	8	12
x	Absolute long	12	16
$\#x$	immediate	4	8

Alternative Code Sequences

Load a Byte into the lower quarter of data register D5, the address results from adding base register A1's content to the contents of the lower half of data register D1 and incrementing the result by 8.

The execution time, 14 cycles, consists of
the execution time for the operation proper, 4 cycles, and
the execution time for the addressing, 10 cycles.

MOVE.B	8(A1, D1.W), D5	ADDA	#8, A1	costs: 16	ADDA	D1.W, A1	costs: 8
total costs 8		ADDA	D1.W, A1	costs: 8	MOVE.B	8(A1), D5	costs: 1
		MOVE.B	(A1), D5	costs: 8		total costs 20	
			total costs 32				

Alternative Code Sequences for `b := 2 + a[i]`

`b`, `i` integer variables,

`a: array[1 ..10] of integer`.

`a`, `b`, `i` in the same frame addressed by address register A5,

`b` at relative address 4,

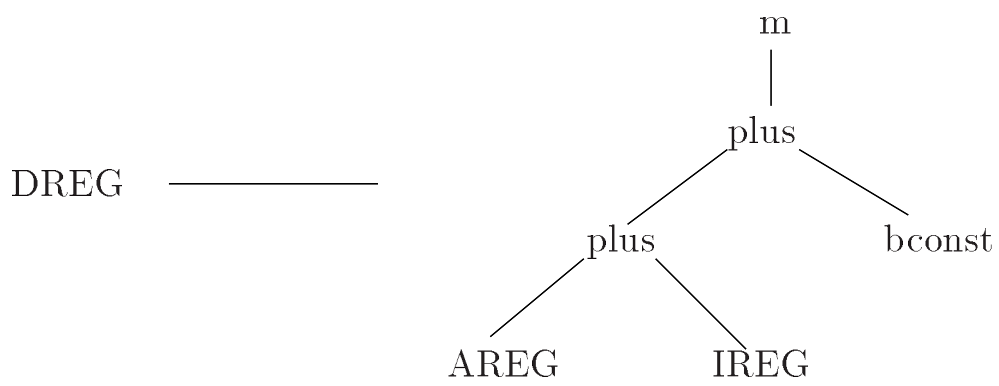
`i` at relative address 6,

`a` at relative address 8.

MOVE	6(A5), D1	costs 12	MOVE.L	A5, A1	costs 4
ADD	D1, D1	costs 4	ADDA.L	#6, A1	costs 12
MOVE	8(A5,D1), D2	costs 14	MOVE	(A1), D1	costs 8
ADDQ	#2, D2	costs 4	MULU	#2, D1	costs 44
MOVE	D2, 4(A5)	costs 12	MOVE.L	A5, A2	costs 4
	total costs 46		ADDA.L	#8, A2	costs 12
			ADDA.L	D1, A2	costs 8
			MOVE	(A2), D2	costs 8
			ADDQ	#2, D2	costs 4
			MOVE.L	A5, A3	costs 4
			ADDA.L	#4, A3	costs 12
			MOVE	D2, (A3)	costs 8
				total costs 128	

Machine Description

- Input to Code Selector Generator,
- Regular Tree Grammar, Terminals from the program representation, non–terminals represent machine resources,
- Often ambiguous,
- Each rule has associated costs,
- Factorization of addressing modes reduces size.



Generated Code Selector

- Parses intermediate representations (IR) of programs,
- Computes derivations according to “machine grammar”, each corresponding to one instruction sequence,
- Has to select cheapest derivation, corresponding to (locally) cheapest code sequence
- May compute costs in states or use dynamic programming.

Tree Languages

- **Alphabet with arity** is a finite set Σ of operators together with a function $\rho : \Sigma \rightarrow \mathbb{N}_0$, **arity**.

- $\Sigma_k = \{a \in \Sigma \mid \rho(a) = k\}$.

- The **homogeneous tree language** over Σ is the following inductively defined set $T(\Sigma)$:
  - $a \in T(\Sigma)$ for all $a \in \Sigma_0$;
  - Are $b_1, \ldots, b_k$ in $T(\Sigma)$ and is $f \in \Sigma_k$, so is $f(b_1, \ldots, b_k) \in T(\Sigma)$.

Example:
$\Sigma = \{a, cons, nil\}$,
$\rho(a) = \rho(nil) = 0$, $\rho(cons) = 2$.
Some trees over Σ:
a, $cons(nil,\ nil)$, $cons(cons(a, nil), nil)$.

Patterns, Substitutions

V infinite set of variables (arity 0).

- $p \in T(\Sigma \cup V)$ is called a **pattern** over Σ,
- p is **linear** if no variable occurs twice in p.
- A **Substitution** Θ maps variables to patterns, $\Theta : V \rightarrow T(\Sigma \cup V)$.
- Θ extended to $\Theta : T(\Sigma \cup V) \rightarrow T(\Sigma \cup V)$ by
  $t\Theta = x\Theta$, if $t = x \in V$ and
  $t\Theta = a(t_1\Theta, \ldots, t_k\Theta)$, if $t = a(t_1, \ldots, t_k)$.

Let $V = \{X\}$.
X, $cons(nil, X)$, $cons(X, nil)$ are patterns over Σ.

Regular Tree Grammars

Regular Tree Grammar (RTG)

$G = (N, \Sigma, P, S)$ consists of

- N, finite set of **non–terminals**,
- Σ, finite alphabet (with arity) of **terminals** (operators labeling nodes)
- P, finite set of **rules** $X \rightarrow s$ where $X \in N$ and $s \in T(\Sigma \cup N)$,
- $S \in N$, the start symbol.

Notions:

- $p : X \rightarrow Y$ **chain rule**,
- p has **type** $(X_1, \ldots, X_k) \rightarrow X$, if j-th occurrence of a non–terminal in s (counted from the left) is X_j.
- $\tilde{s}$ results from s by replacing non–terminal X_j by variable x_j.

Why "Regular"?

- Path words form a regular word language,
- Regular tree languages are closed under union, intersection, and complement,
- Emptiness and therefore containment are decidable.

Example: Lists

$G_1 = (N_1, \Sigma, P_1, L)$;
$\Sigma = \{a, cons, nil\}$
where $\rho(a) = \rho(nil) = 0, \rho(cons) = 2$,
$N_1 = \{E, L\}$ und

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

$L(TG_1)$ is the language of linear lists of a's including the empty list,
i.e. $L(G_1) = \{nil, cons(a, nil), cons(a, cons(a, nil)), \ldots\}$.

Example: Machine Grammar

- $G_m = (N_m, \Sigma, P_m, REG)$;

- $\Sigma = \{const, m, plus, REG\}$
  where $\rho(const) = 0; \rho(m) = 1, \rho(plus) = 2$,

- $N_m = \{REG\}$

- $P_m = \{$

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

G_m describes a subset of an instruction set of a simple processor, rules are marked with names of instructions.

The first three instructions add

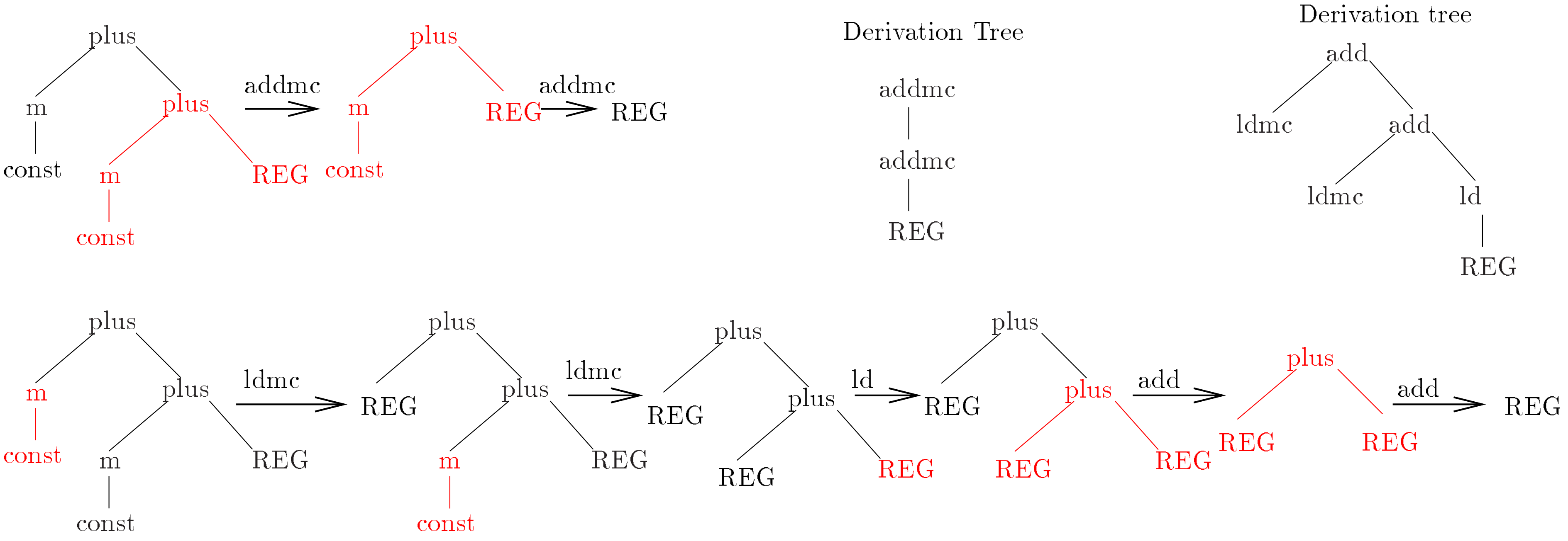
- the contents of a memory cell, whose address is given by a constant,
- the contents of a memory cell, whose address is in a register, resp.,
- the contents of a register

to the contents of a register and put the result into a register.
The last three rules describe load instructions, which load

- the contents of a memory cell, whose address is given by a constant,
- a constant, resp.,
- the contents of a register

into a register.

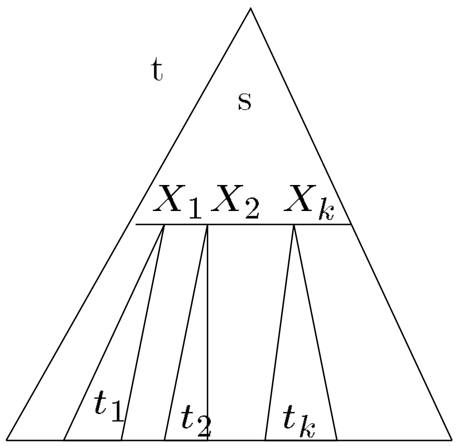
Example Derivations

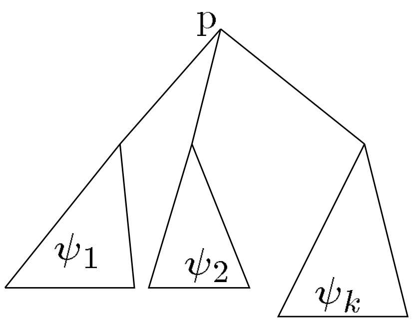


Derivation Tree

An X–**derivation tree** for tree $t \in T(\Sigma \cup N)$ according to tree grammar G is a tree $\psi \in T(P \cup N)$, such that

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





The **generated language**
$L(TG) = \{t \in T(\Sigma) \mid \exists \psi \in T(P \cup N) : \psi$ is S-derivation tree for $t\}$.

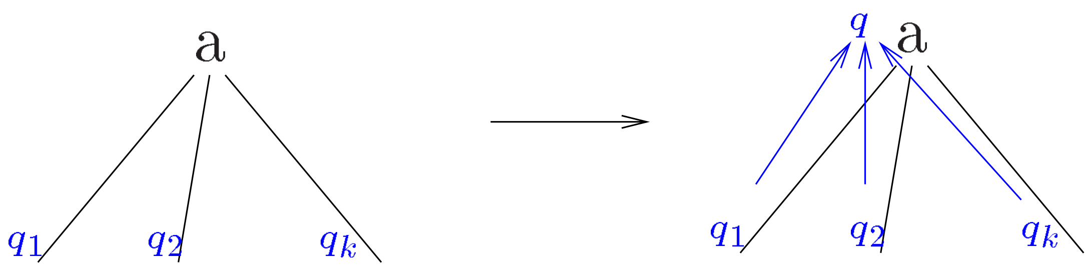
The Tree Analysis Problem

- An instance of the **tree analysis problem** consists of an RTG G and a tree t.
- A solution consists of the set of all derivation trees of t according to G,
- A **Tree Analyzer** for G solves the tree analysis problem for G and all its trees,
- A **Tree Analyzer Generator** generates a tree analyzer for each RTG.

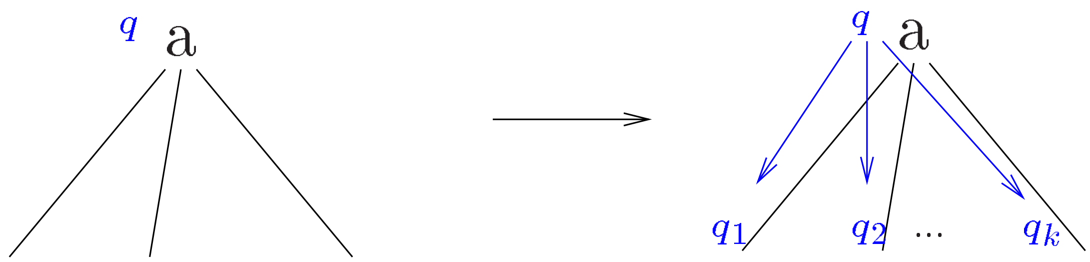
Finite Tree Automata, Intuition

- Generalization of finite word automata to trees,
- Transitions $(q, a, q_1, \ldots, q_k)$, where
  $a \in \Sigma_k$, q state at node n labeled a,
  $q_1, \ldots, q_k$ state at children of n,
- Traversal strategies,

  bottom up:



  top down:



Finite Tree Automata, Definition

Finite tree automaton (FTA)
$A = (Q, \Sigma, \delta, Q_F)$, where

- Q, finite set of **states**,
- $Q_F \subseteq Q$, **final states**,
- Σ, input alphabet (with arity),
- $\delta \subseteq \bigcup_{j \geq 0} Q \times \Sigma_j \times Q^j$, **transition relation**.
- A is **top down deterministic**, if
  - exactly one final state, and
  - at most one transition $(q, a, q_1 \ldots, q_k) \in \delta$ for all a and q.
- A is **bottom up deterministic**, if at most one transition $(q, a, q_1 \ldots q_k) \in \delta$ for all a and all $q_1, \ldots, q_k$.
  In this case, we write δ as partial function:
  $\delta : \bigcup_{j \geq 0} \Sigma_j \times Q^j \rightarrow Q$

Computation

- A annotates the nodes with states;
  hence new alphabet $\Sigma \times Q = \{\langle a, q\rangle \mid a \in \Sigma,\ q \in Q\}$,
  where $\rho(\langle a, q\rangle) = \rho(a)$.

- ϕ q-**computation** of A on tree $t = a(t_1, \ldots, t_m)$:
  a tree $\langle a, q\rangle(\phi_1, \ldots, \phi_m) \in T(\Sigma \times Q)$, where
  ϕ_j are q_j-computations for the t_j, $j = 1, \ldots, m$,
  $(q, a, q_1 \ldots q_m)$ is a transition.

- Is $q \in Q_F$, then ϕ is **accepting**.

- The language $L(TA)$ consists of the trees with accepting computations.

- A state resp. transition is **superfluous** if it does not occur in any accepting computation.

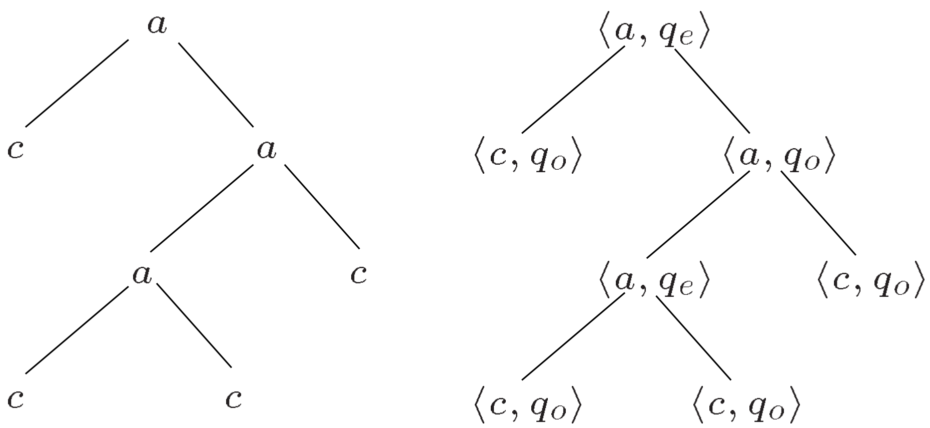
Example Computation

DFTA $A_b = (Q_b, \Sigma_b, \delta_b, Q_{F,b})$ with
states $Q_b = \{q_e, q_o\}$,
alphabet $\Sigma_{b,0} = \{c\}$ and $\Sigma_{b,2} = \{a\}$,
final states $Q_{F,b} = \{q_e\}$
transitions: $\delta_b = \{\ (q_o, c)$

$$\begin{array}{l} (q_e, a, q_o, q_o) \\ (q_o, a, q_e, q_o) \\ (q_o, a, q_o, q_e) \\ (q_e, a, q_e, q_e)\} \end{array}$$

Accepts trees with even number of c's.

Tree and q_e–computation



Determinism – Non–determinism

- Bottom up NFTAs and Top down NFTAs are equivalent,
- Bottom up DFTAs and Top down DFTAs are not equivalent;
  example language cannot be recognized by top down DFTA.
- NFTAs are equivalent to bottom up DFTAs (powerset construction).

(Bottom up) DFTA:

- At most one computation for each tree,
- At most one state at each node,
- δ extended to a partial function $\delta : T(\Sigma) \rightarrow Q$ by:
  $\delta(t) = \delta(a, \delta(t_1) \ldots \delta(t_k))$, if $t = a(t_1, \ldots, t_k)$.
- $\delta(t) = q$ iff there is a q–computation for t.

Generating Tree Parsers

The generation (and the explanation) process:

Input: G

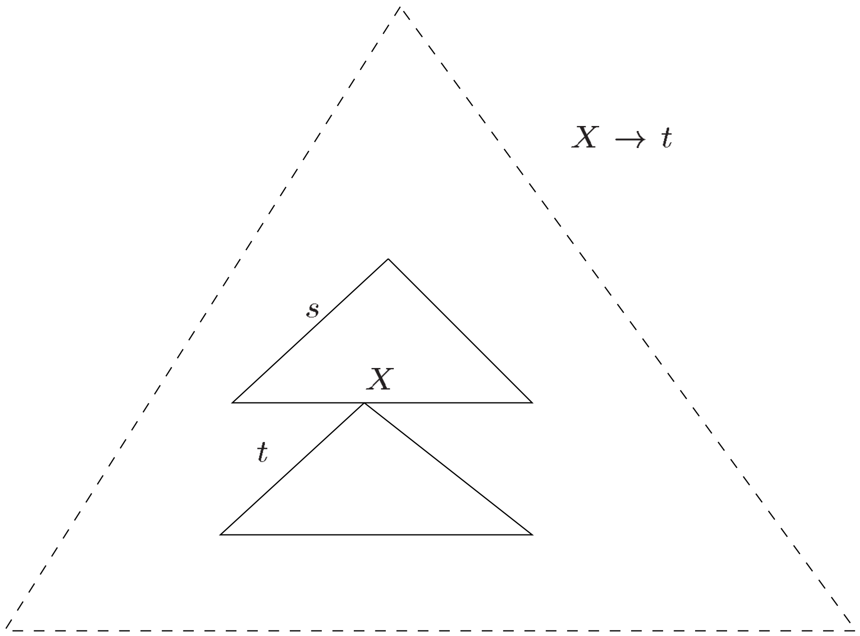
1. Generate NFTA A_G,
2. Apply powerset construction to obtain DFTA $P(G)$.

Later: Consider variant with costs.

How A_G Works

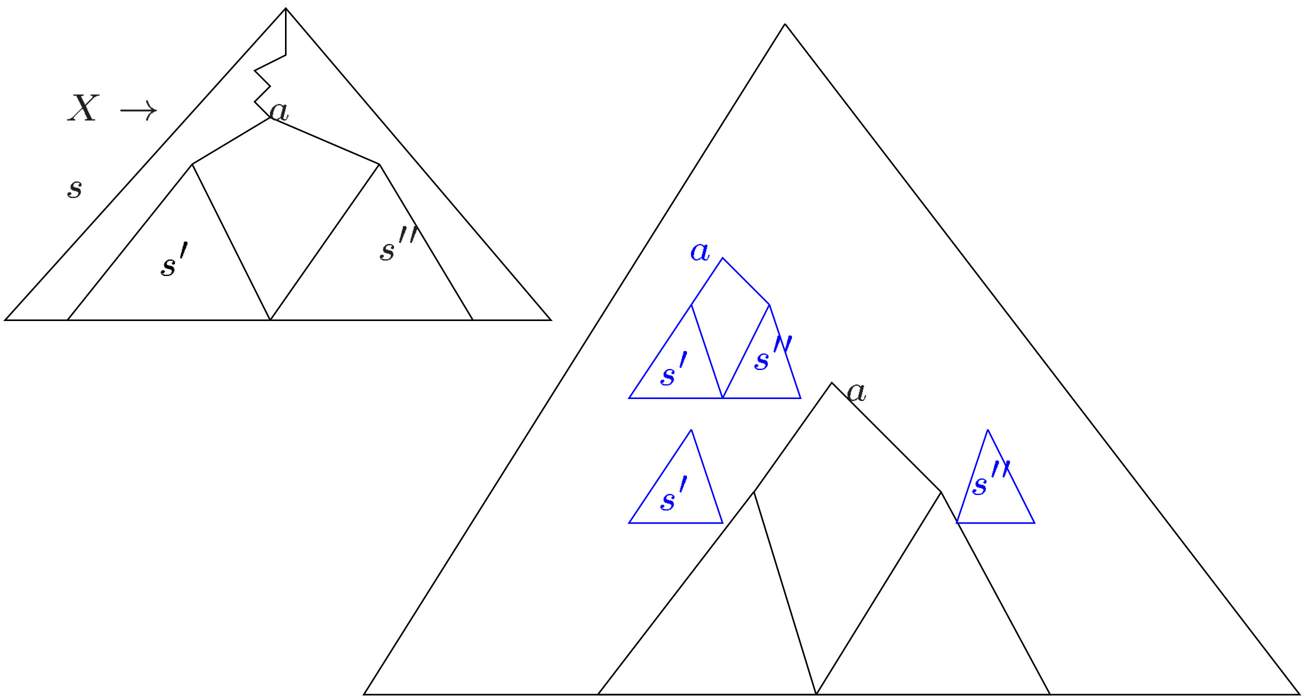
A_G

- tries to cover the given tree with right sides of productions (like a puzzle),
- does reductions to check whether neighbouring rules fit.



States and transitions of A_G

- States: "Tree Items", subtrees of right sides, interpretation: what has been analyzed so far.
- Transitions: analyze next generation of a right side,



- What about "complete items", i.e. full right sides s?
  do reduction in the same step,
  i.e., new state is X, not s.

A_G, Definition

$A_G = (Q_G, \Sigma, \delta_G, \{S\})$, where

- $Q_G = N \cup \{s' \mid \exists (X \to s) \in P,$
  where s' is proper subpattern of $s\}$.

- Transition relation δ_G: transitions of the forms
  $\{(s, a, s_1 \ldots s_k) \mid s = a(s_1, \ldots, s_k) \in Q_G\}$ and
  $\{(X, a, s_1 \ldots s_k) \mid \exists (X \to s) \in P$ and
  $s = a(s_1, \ldots, s_k)\}$.

Problem with chain rules:

- A_G would have to "step on the spot" doing chain reductions.
  However, A_G has to consume at least one terminal per step,

- Chain reductions are precomputed and integrated into δ.

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

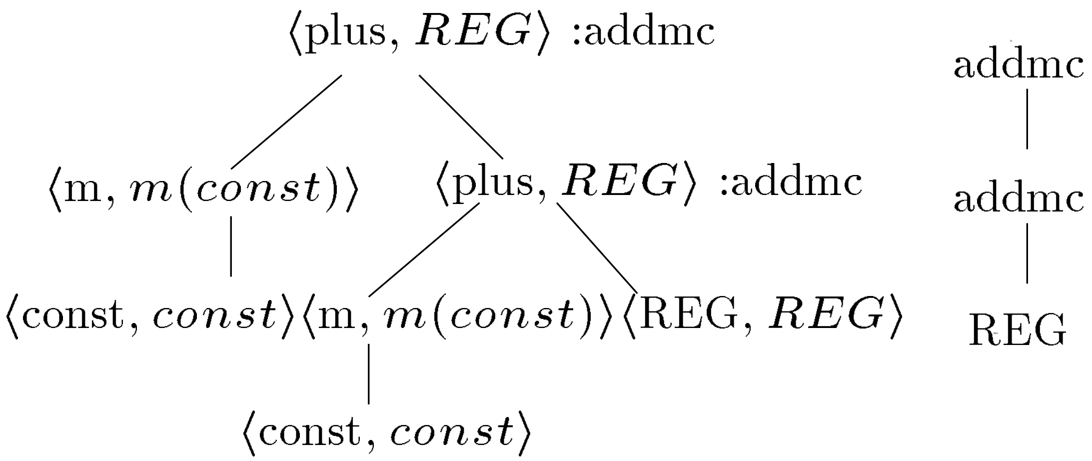
Example A_{G_m}

$A_{G_m} = (Q_{G_m}, \Sigma_{G_m}, \delta_{G_m}, Q_{F,G_m})$ for G_m has the state set
$Q_{G_m} = \{const, REG, m(const), m(REG)\}$

and the transitions

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

Example Computation of A_{G_m}



Properties

G RTG and t input tree.

- Exists X–derivation tree for t according to G iff exists an X–computation for t in A_G.

  In particular: $L(G) = L(A_G)$.

Principle of the Powerset Construction

Finite Word Automata:
new state $s = \{q \in Q \mid \exists w : (q_0, w) \vdash_M^* (q, \varepsilon)\ \}$

Finite Tree Automata:
new state $B = \{q \in Q \mid \exists t : \exists q -$ computation for $t\}$

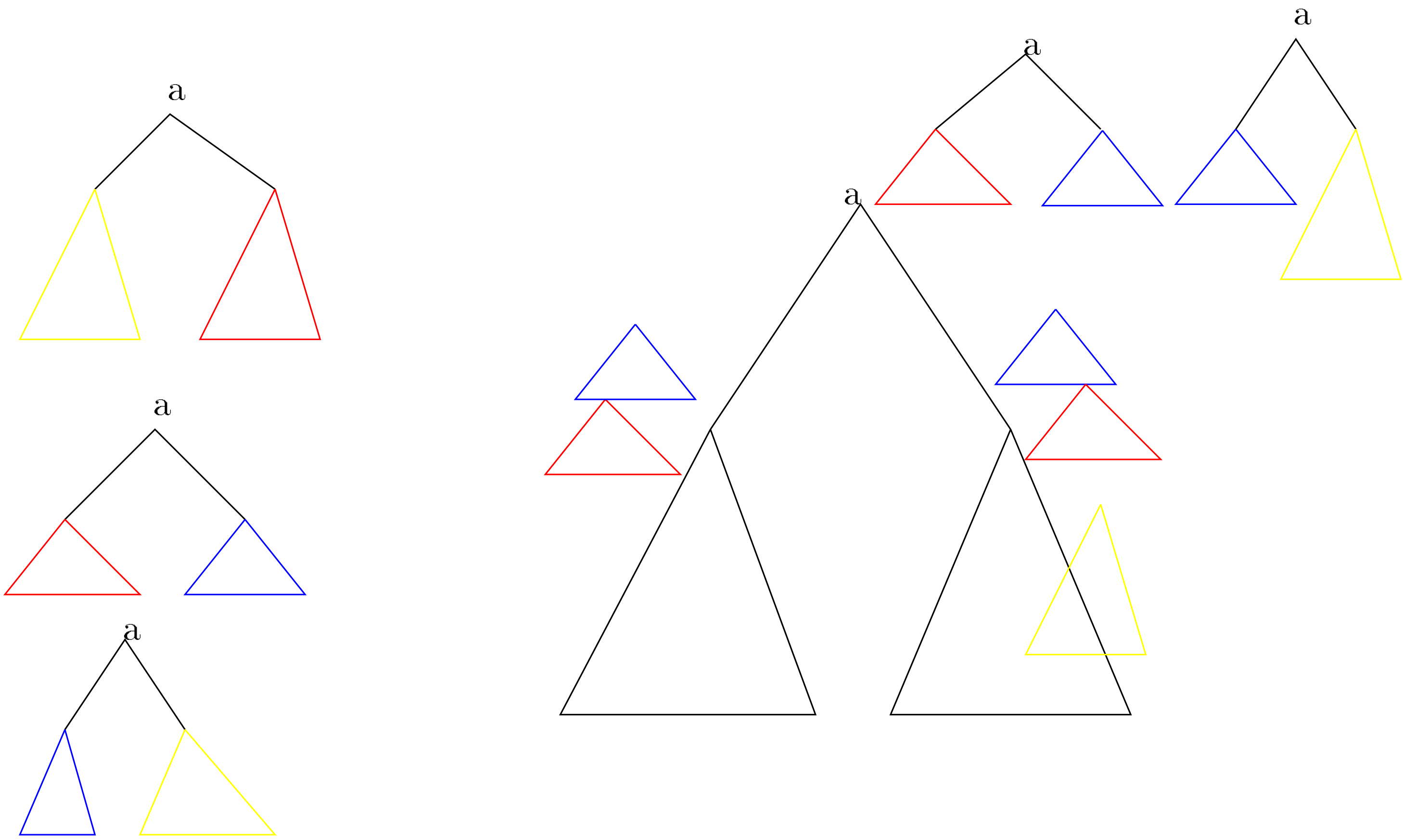
Powerset automaton $P(A)$ is built iteratively, $Q_p^{(n)}$ and $\delta_p^{(n)}$ occur in computations on trees of height $\leq n - 1$.

Powerset Construction

Let $A = (Q, \Sigma, \delta, Q_F)$ be a NFTA.
Its **powerset automaton** is the DFTA
$P(A) = (Q_p, \Sigma, \delta_p, Q_{p,F})$ where

- $Q_p = 2^Q$,
- $Q_{p,F} := \{B \in Q_p \mid B \cap Q_F \neq \emptyset\}$,
- states and transitions are computed in the iteration:
  $Q_p := \bigcup_{n \geq 0} Q_p^{(n)}$ and $\delta_p := \bigcup_{n \geq 0} \delta_p^{(n)}$, where:
  - $Q_p^{(0)} = \emptyset$;
  - Be $n > 0$. For $a \in \Sigma_k$ and $B_1, \ldots, B_k \in Q_p^{(n-1)}$ let $B := \{q \in Q \mid \exists q_1 \in B_1, \ldots, q_k \in B_k : (q, a, q_1 \ldots q_k) \in \delta\}$.
    Is $B \neq \emptyset$,
    then $B \in Q_p^{(n)}$ and $(B, a, B_1 \ldots B_k) \in \delta_p^{(n)}$.

The Powerset Construction on Tree Parsers



Example

The powerset automaton for A_{G_m} has the state set $Q_{G_m} = \{q_1, q_2, q_3, q_4\}$ where

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

Properties

1. For each $t \in T(\Sigma)$:
   - Is $\delta_p(t)$ defined, then $\delta_p(t) = \{q \mid \exists q\text{–computation on } t\}$.
   - Is $\delta_p(t)$ undefined, then there is no $q \in Q$ with a q–computation of A for t.
   - $\delta(t) \cap N = \{X' \in N \mid \exists X'\text{–derivation tree for } t\}$.

2. $L(A) = L(P_p(A))$.

3. For each state $B \in Q_r$ there exists a tree t, such that $\delta_p(t) = B$.