Tree Parsing for Code Selection

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

3. Januar 2010
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.
Complexity

Many subproblems in the compiler backend are complex:

Early results:
Bruno & Sethi [1976]: generation of optimal code for straight-line programs and 1-register machine is NP-complete
Garey & Johnson [1979]: Instruction scheduling, even for very simple target machines, is NP-hard.

What makes the difference in code generation?

input: straight-line programs w/o common subexpressions
machine model: register constraints, e.g., interchangeable registers or not, operations on register pairs or not

Common subexpressions need directed acyclic graphs (DAGs).
Code generation for expression trees has efficient solutions.
Phase Ordering Problem

Issues:

- Software Complexity
- Result Quality
- Order in Serialization
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 (without common subexpressions) to be translated into (locally) optimal code according to some cost measure.
An Example CISC 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,

\[
\text{ADD D1, D2}
\]

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: $\text{cont}(D_n)$.
- $A_n$ Address register direct: $\text{cont}(A_n)$.
- $(A_n)$ Address register indirect: $\text{St(\text{cont}(A_n))}$.
- $d(A_n)$ Address register indirect with address distance: $\text{St(\text{cont}(A_n) + d)}$ with 16-Bit-constant $d$.
- $d(A_n, I_x)$ Address register indirect with Index and Address distance: $\text{St(\text{cont}(A_n) + \text{cont}(I_x) + d)}$ with $A_n$ used as base register, $I_x$ index register (either address or data register), 8-Bit-distance $d$.
- $x$ Absolute short: $\text{St}(x)$ with 16-Bit-constant $x$.
- $x$ Absolute long: $\text{St}(x)$ with 32-Bit-constant $x$.
- $\#x$ Immediate: $x$. 
## Tree Parsing for Code Selection

### Execution Times

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Byte, Word</th>
<th>Double Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_n$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(A_n)$</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>$d(A_n)$</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>$d(A_n, I_x)$</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>$x$</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>$x$</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>$#x$</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
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
total costs 8
```

```
ADDA #8, A1 costs: 16
ADDA D1.W, A1 costs: 8
MOVE.B (A1), D5 costs: 8
```

```
ADDA D1.W, A1 costs: 8
MOVE.B 8(A1), D5 costs: 12
```

```
total costs 32
```

```
total costs 20
```
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, Relative addresses: $b \mapsto 4, i \mapsto 6, a \mapsto 8$.

The code for addressing $a[2]$ computes:

$$A5 + 8 + \text{value}(i) \times 2$$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE 6(A5), D1</td>
<td>MOVes $6$ to D1</td>
<td>12</td>
</tr>
<tr>
<td>ADD D1, D1</td>
<td>ADds D1</td>
<td>4</td>
</tr>
<tr>
<td>MOVE 8(A5,D1), D2</td>
<td>MOVes $8$ to D2</td>
<td>14</td>
</tr>
<tr>
<td>ADDQ #2, D2</td>
<td>ADAdd Q #2 to D2</td>
<td>4</td>
</tr>
<tr>
<td>MOVE D2, 4(A5)</td>
<td>MOVes $D2 + 4A5$</td>
<td>12</td>
</tr>
</tbody>
</table>

Total costs $46$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE.L A5, A1</td>
<td>MOVES L $A5, A1$</td>
<td>4</td>
</tr>
<tr>
<td>ADDA.L #6, A1</td>
<td>ADD with #6 to A1</td>
<td>12</td>
</tr>
<tr>
<td>MOVE (A1), D1</td>
<td>MOVES (A1) to D1</td>
<td>8</td>
</tr>
<tr>
<td>MULU #2, D1</td>
<td>MULU Q #2 to D1</td>
<td>44</td>
</tr>
<tr>
<td>MOVE.L A5, A2</td>
<td>MOVES L $A5, A2$</td>
<td>4</td>
</tr>
<tr>
<td>ADDA.L #8, A2</td>
<td>ADD with #8 to A2</td>
<td>12</td>
</tr>
<tr>
<td>ADDA.L D1, A2</td>
<td>ADD with D1 to A2</td>
<td>8</td>
</tr>
<tr>
<td>MOVE (A2), D2</td>
<td>MOVES (A2) to D2</td>
<td>8</td>
</tr>
<tr>
<td>ADDQ #2, D2</td>
<td>ADAdd Q #2 to D2</td>
<td>4</td>
</tr>
<tr>
<td>MOVE.L A5, A3</td>
<td>MOVES L $A5, A3$</td>
<td>4</td>
</tr>
<tr>
<td>ADDA.L #4, A3</td>
<td>ADD with #4 to A3</td>
<td>12</td>
</tr>
<tr>
<td>MOVE D2, (A3)</td>
<td>MOVES D2 to (A3)</td>
<td>8</td>
</tr>
</tbody>
</table>

Total costs $128$
An Example RISC Architecture, the MIPS

- RISC microprocessor architecture developed by John L. Hennessy at Stanford University in 1981
- no interlocked pipeline stages
- Load/Store-Architecture (R3000)
- 32 registers
- $2^{30}$ memory words = $2^{32}$ bytes
- Still used: Playstation Portable, PS2, etc.
Instruction Set (MIPS R3000)

Arithmetic:
- add $1, $2, $3
- sub $1, $2, $3
- addi $1, $2, CONST

Data Transfer:
- lw $1, CONST($2)
- sw $1, CONST($2)

Cond. Branch:  beq $1, $2, CONST

Unconditional Jumps:
- j CONST
- jr $1
- jal CONST

Logical operations:  Bitwise Shift, etc.

Pseudoinstructions:  Translated into real instructions before assembly:
- bgtz Label (branch greater than), etc.
Example Code

```assembly
if (x <= 0)
    y = x + 1;
el
    x = y + x;
```

```
bgtz $1 el
    addi $2, $1, 1
    j end
el:
    addi $1, $2, $1
end:
```

Assuming
```
x in $1 and y in $2
```
Looking for a Description Mechanism

Several compilation subtasks

- can be formally described and
- their implementation can be automatically generated.

Examples:

<table>
<thead>
<tr>
<th>compilation subtask</th>
<th>description formalism</th>
<th>acceptor</th>
<th>desired output</th>
<th>algorithmic aspects</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical analysis</td>
<td>regular expressions</td>
<td>finite automata</td>
<td>final states</td>
<td>r.e. (\mapsto) nfa, nfa (\mapsto) dfa, minimization (determin.)</td>
<td>equivalences, closure properties, decidabilities non-equiv. of det. and non-det. pda, undecidabilities</td>
</tr>
<tr>
<td>syntax analysis</td>
<td>context-free grammars</td>
<td>pushdown automata</td>
<td>syntax trees, derivations</td>
<td>parser generation</td>
<td></td>
</tr>
<tr>
<td>compilation subtask</td>
<td>description formalism</td>
<td>acceptor</td>
<td>desired output</td>
<td>algorithmic aspects</td>
<td>properties</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------</td>
<td>----------</td>
<td>----------------</td>
<td>---------------------</td>
<td>------------</td>
</tr>
<tr>
<td>lexical analysis</td>
<td>regular expressions</td>
<td>finite automata</td>
<td>final states</td>
<td>r.e. $\mapsto$ nfa, nfa $\mapsto$ dfa, minimization</td>
<td>equivalences, closure properties, decidabilities</td>
</tr>
<tr>
<td>syntax analysis</td>
<td>context-free grammars</td>
<td>pushdown automata</td>
<td>syntax trees, derivations</td>
<td>(determin.) parser generation</td>
<td>non-equiv. of det. and non-det. pda, undecidabilities</td>
</tr>
<tr>
<td>code selection</td>
<td>regular tree grammars</td>
<td>finite tree automata</td>
<td>derivations</td>
<td>rtg $\mapsto$ fta, fta $\mapsto$ budding</td>
<td>closure properties, decidabilities</td>
</tr>
</tbody>
</table>
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.

```
DREG  _________
   |
   |  plus
   |  |
   |  plus  bconst
   |  |
   | AREG  IREG
m
```
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 \( \Sigma \) of operators together with a function \( \rho : \Sigma \to \mathbb{N}_0 \), **arity**.

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

- The **homogeneous tree language** over \( \Sigma \) 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:
\( \Sigma = \{ a, \text{cons}, \text{nil} \} \),
\( \rho(a) = \rho(\text{nil}) = 0, \rho(\text{cons}) = 2 \).
Some trees over \( \Sigma \):
\( a, \text{cons}(\text{nil}, \text{nil}), \text{cons}(\text{cons}(a, \text{nil}), \text{nil}) \).
Patterns, Substitutions

$V$ infinite set of variables (arity 0).

- $p \in T(\Sigma \cup V)$ is called a **pattern** over $\Sigma$,
- $p$ is **linear** if no variable occurs twice in $p$.
- A **Substitution** $\Theta$ maps variables to patterns, $\Theta : V \rightarrow T(\Sigma \cup V)$.
- $\Theta$ 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 $V = \{X\}$.
$X, \text{cons}(\text{nil}, X), \text{cons}(X, \text{nil})$ are patterns over $\Sigma$. 
Regular Tree Grammars

Regular Tree Grammar (RTG) \( G = (N, \Sigma, P, S) \) consists of

- \( N \), finite set of non–terminals,
- \( \Sigma \), 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,
- \( p : X \rightarrow s \) 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 $\rho(a) = \rho(nil) = 0, \rho(cons) = 2$
- $N_1 = \{ E, L \}$ and
- $P_1 = \{ L \rightarrow nil, \\
  L \rightarrow cons(E, L), \\
  E \rightarrow a \}$

$L(TG_1)$ 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

- $G_m = (N_m, \Sigma, P_m, \text{REG})$
- $\Sigma = \{\text{const}, m, \text{plus}, \text{REG}\}$
  where $\rho(\text{const}) = 0; \rho(m) = 1, \rho(\text{plus}) = 2$,
- $N_m = \{\text{REG}\}$
- $P_m = \{ \begin{align*}
  \text{addmc} & : \text{REG} &\rightarrow & \text{plus}(m(\text{const}), \text{REG}), \\
  \text{addm} & : \text{REG} &\rightarrow & \text{plus}(m(\text{REG}), \text{REG}), \\
  \text{add} & : \text{REG} &\rightarrow & \text{plus}(\text{REG}, \text{REG}), \\
  \text{ldmc} & : \text{REG} &\rightarrow & m(\text{const}), \\
  \text{ldc} & : \text{REG} &\rightarrow & \text{const}, \\
  \text{ld} & : \text{REG} &\rightarrow & \text{REG}\} \end{align*} $
$G_m$ 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 instructions **load** into a register:

- the contents of a memory cell whose address is given by a constant,
- a constant, and
- the contents of a register, resp.
Example Derivations
Derivation Tree

An \textbf{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

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

\[ L(TG) = \{ t \in T(\Sigma) \mid \exists \psi \in T(P \cup N) : \psi \text{ is } S\text{-derivation tree for } t \} \]
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 \(n\),
- Non-deterministic automaton “guesses” computations in any order (like a puzzle).
Traversal strategies,

bottom up:

```
q_1  q_2  q_k
```

```
q  a
```

top down:

```
q  a
```

```
q_1  q_2  q_k
```

```
q_1  q_2  ...  q_k
```
Finite Tree Automaton (FTA)

\[ A = (Q, \Sigma, \delta, Q_F), \]

- \( Q \), finite set of states,
- \( Q_F \subseteq Q \), final states,
- \( \Sigma \), 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 \( \delta \) 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 $\rho(\langle a, q \rangle) = \rho(a)$.

- $q$-computation $\phi$ 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 $\phi_j$ are $q_j$-computations for the $t_j, j = 1, \ldots, m$, $(q, a, q_1 \ldots q_m)$ is a transition.

- If $q \in Q_F$, then $\phi$ 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: $\delta_b = \{(q_o, c)$
\[(q_e, a, q_o, q_o)\]
\[(q_o, a, q_e, q_o)\]
\[(q_o, a, q_o, q_e)\]
\[(q_e, a, q_e, q_e)\]\}
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,
- \( \delta \) extended to a partial function \( \delta : T(\Sigma) \rightarrow Q \) by:
  \[ \delta(t) = \delta(a, \delta(t_1) \ldots \delta(t_k)), \text{ if } t = a(t_1, \ldots, t_k). \]
- \( \delta(t) = q \) 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}, \text{Definition} \\
A_{G} = ( Q_{G}, \Sigma, \delta_{G}, \{S\} ), \text{ where} \\
\begin{itemize}
  \item $Q_{G} = N \cup \{s' \mid \exists (X \rightarrow s) \in P, \text{ where } s' \text{ is proper subtree of } s\}$. 
  \item Transition relation $\delta_{G}$: transitions of the forms 
    \begin{align*}
    &\{(s, a, s_{1} \ldots s_{k}) \mid s = a(s_{1}, \ldots, s_{k}) \in Q_{G}\} \text{ and} \\
    &\{(X, a, s_{1} \ldots s_{k}) \mid \exists (X \rightarrow s) \in P \text{ and } s = a(s_{1}, \ldots, s_{k})\}.
    \end{align*}
\end{itemize}
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 $\delta$.

$$
\delta_G := \{(s, a, s_1 \ldots s_k) \mid s = a(s_1, \ldots, s_k) \in Q_G\} \cup (\text{proper transition}) \\
{(X, a, s_1 \ldots s_k) \mid \exists (X' \rightarrow s) \in P : \exists X-\text{derivation tree for } X' \text{ and } s = a(s_1, \ldots, s_k)} (\text{reduction}) \\
(\text{chain rules})
$$
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

$$Q_{G_m} = \{\text{const}, \text{REG}, m(\text{const}), m(\text{REG})\}$$

and the transitions

$$\delta_{G_m} = \{(\text{const}, \text{const}, \epsilon)\}
\{(\text{REG}, \text{const}, \epsilon)\}
\{(\text{REG}, \text{REG}, \epsilon)\}
\{(m(\text{const}), m, \text{const})\}
\{(\text{REG}, m, \text{const})\}
\{(m(\text{REG}), m, \text{REG})\}
\{(\text{REG}, \text{plus}, m(\text{const}) \text{ REG})\}
\{(\text{REG}, \text{plus}, m(\text{REG}) \text{ REG})\}
\{(\text{REG}, \text{plus}, \text{REG} \text{ REG})\}$$
Example Computation of $A_{G_m}$
Properties

G RTG and $t$ input tree.

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

In particular: $L(G) = L(A_G)$. 
Principle of the Powerset Construction

**Finite Word Automata:**

\[ \exists \text{ old states } q_1, q_2 \text{ and word } w \text{ such that } (q_0, w) \vdash^*_M (q_1, \varepsilon) \text{ and } (q_0, w) \vdash^*_M (q_2, \varepsilon) \]

\[ \implies \exists \text{ new state } Q \text{ such that } q_1, q_2 \in Q \text{ and } \delta^*_d(q_d, w) = Q \]

**Finite Tree Automata:**

\[ \exists \text{ old states } q_1, q_2 \text{ and tree } t \text{ such that } \exists q_1 - \text{ and } q_2 - \text{ computations for } t \]

\[ \implies \exists \text{ new state } B \text{ such that } q_1, q_2 \in B \text{ and } \delta_p(t) = B \]
Word automata

new state $P$

new state $Q$ such that $\delta(P, a) = Q$
Tree automata

new state $B$ with $(B, a, B_1 \ldots B_k) \in \delta_p$
Powerset Construction

Powerset automaton $P(A)$ is built iteratively, $Q_p^{(n)}$ and $\delta_p^{(n)}$ occur in computations on trees of height $\leq n - 1$. 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 state set $Q_{G_m} = \{ q_1, q_2, q_3, q_4 \}$ where

$q_1 = \{ \text{REG} \}$
$q_2 = \{ \text{const, REG} \}$
$q_3 = \{ m(\text{REG}) \}$
$q_4 = \{ m(\text{const}), \text{REG, m(REG)} \}$

and transition function $\delta_{G_m}$:

<table>
<thead>
<tr>
<th>state</th>
<th>operator</th>
<th>children state(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>REG</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>const</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$m$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$m$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>plus</td>
<td>$q_1$ $q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>plus</td>
<td>$q_4$ $q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>plus</td>
<td>$q_3$ $q_1$</td>
</tr>
</tbody>
</table>
Properties

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

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

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

- Rules have cost functions, i.e. costs of the instruction,
- Translated into cost functions for the transitions of the NFTA,
- Deterministic bottom up automaton constructs cheapest derivations.
Rule $p$ of type $(X_1, \ldots, X_k) \rightarrow X$ gets $k$-place function $C(p) : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$

$C$ extended to derivation trees $\psi$.
- $\psi = X \in N$, then $C(\psi) := 0$.
- $\psi = p(\psi_1, \ldots, \psi_k)$, then $C(\psi) := C(p)(C(\psi_1), \ldots, C(\psi_k))$.

$C$ is monotone, if for all $p \in P$, $C(p)$ is monotone,

$C$ is additive, if for all $p \in P$, $C(p)$ has the form $C(p) = c_p + x_1 + \cdots + x_k$, $c_p \in \mathbb{N}_0$. 
From cost annotation $C$ of grammar $G$ to cost annotation $C^*$ of automaton $A_G$.

- Assume an additive cost measure. Costs can be described by a constant, i.e. $C$ is a function from $P \rightarrow \mathbb{N}_0$.
- Define $C^*$ as
  - For $\tau = (s, a, s_1 \ldots s_k)$ where $s = a(s_1, \ldots, s_k)$, $C^*(\tau) := 0$.
  - For $\tau = (X, a, s_1 \ldots s_k)$, let $C^*(\tau)$ be the minimal costs of an $X$-derivation tree for $a(s_1 \ldots s_k)$.
- Extend cost function of automaton to cost function for computations.
Extracting Cheapest Derivations

Extract cheapest computations of $A$ from computations of $A_r$ as follows:

1. Tabulate for each node $\langle a, B \rangle$ of a computation $\phi$ of the powerset automaton $P(A_r)$ the costs $c_q$ and the transitions $d_q$ for all $q \in B$.

2. $c_q$ are the costs of a cheapest $q$-computation of a given tree $t$, and $d_q$ are the chosen transitions of $A$. 
Assume that the set of cheapest $X$-derivations has differences bounded by a constant (realistic).

Integrate the (finitely many) cost differences into the states of the subset automaton.

Computed cost for state $q$ is the difference between the cheapest $q$-computation and the cheapest computation.