Code Generation by Tree Walking

Amey Karkare

karkare@cse.iitk.ac.in

April 10, 2019
Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.

Normalization of costs
Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.
Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.
- Only cases taken into account are different patterns matching a node.
Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.
- Only cases taken into account are different patterns matching a node.
- Normalization of costs
An example expression tree and an example machine:

- Terminal: goal, reg, reg, addr, int
- Non-terminal: Fetch, Plus

<table>
<thead>
<tr>
<th>Rule No</th>
<th>Pattern</th>
<th>Rule Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal &lt;- reg</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>reg &lt;- Reg</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>reg &lt;- int</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>reg &lt;- Fetch</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>reg &lt;- Plus</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>addr &lt;- reg</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>addr &lt;- int</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>addr &lt;- Plus</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>int &lt;- Int</td>
<td>0</td>
</tr>
</tbody>
</table>
The tree can be covered in more than one ways

Cost = 6

Cost = 7
The tree can be covered in more than one ways

We are finally interested in the least cost tree.
The tree can be covered in more than one ways

We are finally interested in the least cost tree.

We also want to do some pre-processing before we get any tree,
How is this done? Given a tree,
How is this done? Given a tree,
- traverse the tree bottom up. With the help of a *transition table*, annotate each node of the tree with a state.

```
state 1: reg <- Reg, 0
goal <- reg, 0
addr <- reg, 0
```
```
state 2: reg <- int, 1
goal <- reg, 1
addr <- reg, 1
int <- Int, 0
```
```
state 3: reg <- Plus(reg, reg), 2
goal <- reg, 2
addr <- reg, 2
```
```
state 4: reg <- Fetch(addr), 4
goal <- reg, 4
addr <- reg, 4
```
```
state 5: reg <- Fetch(addr), 6
goal <- reg, 6
addr <- reg, 6
```

```
reg <- int, 0
goal <- reg, 0
addr <- reg, 0
```
```
reg <- Reg, 0
```
```
reg <- Plus(reg, reg), 2
```
```
reg <- Fetch(addr), 4
```
```
reg <- Fetch(addr), 6
```
```
reg <- int, 1
goal <- reg, 1
addr <- reg, 1
int <- Int, 0
```
**State:** Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
Code Generation by Tree Walking

- **State:** Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
- **Transition table:** Gives
Code Generation by Tree Walking

- **State:** Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
- **Transition table:** Gives
  - state corresponding to leaf nodes (0-ary terminals).
State: Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.

Transition table: Gives

- state corresponding to leaf nodes (0-ary terminals).
- given the states of children, gives state of interior nodes (n-ary terminals).
A second top-down pass determines the instructions to be used at each node assuming that the root is to be evaluated in goal.

```
state 1:  reg ← Reg, 0
          goal ← reg, 0
          addr ← reg, 0

state 2:  reg ← int, 1
          goal ← reg, 1
          addr ← reg, 1

state 3:  reg ← Plus(reg, reg), 2
          goal ← reg, 2
          addr ← reg, 2

state 4:  reg ← Fetch(addr), 4
          goal ← reg, 4
          addr ← reg, 4

state 5:  reg ← Fetch(addr), 6
          goal ← reg, 6
          addr ← reg, 6
```
Precomputing the Transition Table

- For 0-ary terminals

\[
\begin{align*}
   \text{Int} & \rightarrow \text{reg}, \ 1 \\
   \text{reg} & \rightarrow \text{int}, \ 1 \\
   \text{int} & \rightarrow \text{Int}, \ 0 \\
   \text{addr} & \rightarrow \text{reg}, \ 1 \\
\end{align*}
\]

- Chain rule
- Covering rule
- Chain rule
- Chain rule

Cost of reducing \text{Int} to goal is cost of reducing \text{Int} to int (0) + cost of reducing int to reg (1) + cost of reducing reg to goal (0) +
Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost **covering rules**. A covering rule can cover the terminal with its pattern.
Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost covering rules. A covering rule can cover the terminal with its pattern.
  - Find least cost chain rules. A chain rule is of the form:
    
    \[
    \text{Int} \leftarrow \text{goal} \leftarrow \text{reg}, \ 1 \\
    \text{reg} \leftarrow \text{int}, \ 1 \\
    \text{int} \leftarrow \text{Int}, \ 0 \\
    \text{addr} \leftarrow \text{reg}, \ 1
    \]
  
    --- chain rule
  
    --- chain rule
  
    --- covering rule
  
    --- chain rule

- Cost of reducing Int to goal is cost of reducing Int to int (0) + cost of reducing int to reg (1) + cost of reducing reg to goal (0) +...
Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost *covering rules*. A covering rule can cover the terminal with its pattern.
  - Find least cost *chain rules*. A chain rule is of the form $\text{nonterminal} \leftarrow \text{nonterminal}$.

\[
\begin{align*}
\text{Int} & \rightarrow \text{goal} \leftarrow \text{reg}, 1 & \text{-- chain rule} \\
\text{reg} & \rightarrow \text{int}, 1 & \text{-- chain rule} \\
\text{int} & \rightarrow \text{Int}, 0 & \text{-- covering rule} \\
\text{addr} & \rightarrow \text{reg}, 1 & \text{-- chain rule}
\end{align*}
\]

- Cost of reducing Int to goal is
  \[
  \text{cost of reducing Int to int (0) + cost of reducing int to reg (1) + cost of reducing reg to goal (0) +}
  \]
Precomputing the Transition Table

- n-ary terminals
Precomputing the Transition Table

- n-ary terminals
  - If both children of Plus are in state 2, in which state would Plus be?
Precomputing the Transition Table

- n-ary terminals
  - If both children of Plus are in state 2, in which state would Plus be?

- The rule reg ← Plus(reg, reg) gives

  - state 2: reg ← int, 1
    goal ← reg, 1
    addr ← reg, 1
    int ← Int, 0
  - state 2: reg ← Plus(reg, reg), 2
    addr ← reg, 2
  - goal ← reg, 2
    reg ← int, 1
The rule \( \text{reg} \leftarrow \text{Plus}(\text{reg}, \text{int}) \) gives

\[
\begin{align*}
\text{state 2:} & \quad \text{reg} \leftarrow \text{int}, 1 \\
& \quad \text{goal} \leftarrow \text{reg}, 1 \\
& \quad \text{addr} \leftarrow \text{reg}, 1 \\
& \quad \text{int} \leftarrow \text{Int}, 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{goal} & \leftarrow \text{reg}, 3 \\
\text{reg} & \leftarrow \text{Plus}(\text{reg}, \text{int}), 3 \\
\text{addr} & \leftarrow \text{reg}, 3 \\
\end{align*}
\]
Precomputing the Transition Table

The rule \( \text{reg} \leftarrow \text{Plus} (\text{reg}, \text{int}) \) gives

\[
\begin{align*}
\text{state 2:} & \quad \text{reg} \leftarrow \text{int}, 1 \\
& \quad \text{goal} \leftarrow \text{reg}, 1 \\
& \quad \text{addr} \leftarrow \text{reg}, 1 \\
& \quad \text{int} \leftarrow \text{Int}, 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{Plus} \\
& \quad \text{reg} \\
& \quad \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{state 2:} & \quad \text{reg} \leftarrow \text{Plus} (\text{reg}, \text{int}), 3 \\
& \quad \text{goal} \leftarrow \text{reg}, 3 \\
& \quad \text{addr} \leftarrow \text{reg}, 3 \\
\end{align*}
\]

Conclusion: If the leaves of Plus are both in state 2, then Plus will be in

\[
\begin{align*}
\text{state 6:} & \quad \text{goal} \leftarrow \text{reg}, 2 \\
& \quad \text{reg} \leftarrow \text{Plus} (\text{reg}, \text{reg}), 2 \\
& \quad \text{addr} \leftarrow \text{reg}, 2 \\
\end{align*}
\]
Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs
  (state1, state1), (state1, state2), (state2, state6)
Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs (state1, state1), (state1, state2), (state2, state6)
- ...
Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs (state1, state1), (state1, state2), (state2, state6)
- ...
- Will this process always terminate?
Precomputing the Transition Table

Consider computation of the state at Fetch, with reg in the state shown.

\[
\begin{aligned}
\text{Fetch} \\
\text{reg} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{reg} &\leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 0 \\
\text{goal} &\leftarrow \text{reg}, \ 0 \\
\text{addr} &\leftarrow \text{reg}, \ 0 \\
\text{reg} &\leftarrow \text{Fetch}(\text{addr}), \ 2 \\
\text{goal} &\leftarrow \text{reg}, \ 2 \\
\text{addr} &\leftarrow \text{reg}, \ 2 \\
\text{reg} &\leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 0 \\
\text{goal} &\leftarrow \text{reg}, \ 0 \\
\text{addr} &\leftarrow \text{reg}, \ 0 \\
\text{reg} &\leftarrow \text{Fetch}(\text{addr}), \ 4 \\
\text{goal} &\leftarrow \text{reg}, \ 4 \\
\text{addr} &\leftarrow \text{reg}, \ 4 \\
\end{aligned}
\]

and so on...
Precomputing the Transition Table

Consider computation of the state at Fetch, with reg in the state shown.

Successive computation of the states for Fetch yield:
Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost.
Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost.

- After relativization, the state on the left changes to the state on the right:

```
reg ← Plus(reg, reg), 2
goal ← reg, 2
addr ← reg, 2

reg ← Plus(reg, reg), 0
goal ← reg, 0
addr ← reg, 0
```

- Does this make the resulting transition table different? Obviously not.

- Does this necessarily lead to a finite number of states?
The solution is to relativize the costs in a state with respect to the item with the cheapest cost.

After relativization, the state on the left changes to the state on the right:

\[
\begin{align*}
\text{reg} &\leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 2 \\
\text{goal} &\leftarrow \text{reg}, \ 2 \\
\text{addr} &\leftarrow \text{reg}, \ 2
\end{align*}
\]

\[
\begin{align*}
\text{reg} &\leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 0 \\
\text{goal} &\leftarrow \text{reg}, \ 0 \\
\text{addr} &\leftarrow \text{reg}, \ 0
\end{align*}
\]

Does this make the resulting transition table different? Obviously not.
Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost.

- After relativization, the state on the left changes to the state on the right:

  \[
  \begin{align*}
  \text{reg} & \leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 2 \\
  \text{goal} & \leftarrow \text{reg}, \ 2 \\
  \text{addr} & \leftarrow \text{reg}, \ 2
  \end{align*}
  \quad
  \begin{align*}
  \text{reg} & \leftarrow \text{Plus}(\text{reg}, \text{reg}), \ 0 \\
  \text{goal} & \leftarrow \text{reg}, \ 0 \\
  \text{addr} & \leftarrow \text{reg}, \ 0
  \end{align*}
  \]

- Does this make the resulting transition table different? Obviously not.

- Does this necessarily lead to a finite number of states?
Relativization of states

Consider a machine with only these two instructions involving Fetch.

\[
\text{reg} \xleftarrow{\text{Fetch}} \text{reg} \quad \text{cost} = 1
\]

\[
\text{int} \xleftarrow{\text{Fetch}} \text{int} \quad \text{cost} = 3
\]

Consider a state in which the \( \text{reg} \) ← . . . item is 2 cheaper than the \( \text{int} \) ← . . . item.

Transits to a state in which the \( \text{reg} \) ← . . . item is 4 cheaper than the \( \text{int} \) ← . . . item.

Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.

Typical instruction sets do not lead to divergence.
Relativization of states

Consider a machine with only these two instructions involving Fetch.

- `reg ← Fetch`
  - `reg`
  - `cost = 1`

- `int ← Fetch`
  - `int`
  - `cost = 3`

Consider a state in which the `reg ← ... item` is 2 cheaper than the `int ← ... item`.

Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.

Typical instruction sets do not lead to divergence.
Relativization of states

- Consider a machine with only these two instructions involving Fetch.

\[
\begin{align*}
\text{reg} & \xleftarrow{\text{Fetch}} \text{cost} = 1 \\
\text{int} & \xleftarrow{\text{Fetch}} \text{cost} = 3
\end{align*}
\]

- Consider a state in which the reg \(\leftarrow\) \ldots item is 2 cheaper than the int \(\leftarrow\) \ldots item.

- Transits to a state in which the reg \(\leftarrow\) \ldots item is 4 cheaper than the int \(\leftarrow\) \ldots item.
Relativization of states

- Consider a machine with only these two instructions involving Fetch.

  \[ \text{reg} \xleftarrow{\text{Fetch}} \]
  \[ \text{int} \xleftarrow{\text{Fetch}} \]

  cost = 1

  cost = 3

- Consider a state in which the reg $\leftarrow \ldots$ item is 2 cheaper than the int $\leftarrow \ldots$ item.

- Transits to a state in which the reg $\leftarrow \ldots$ item is 4 cheaper than the int $\leftarrow \ldots$ item.

- ...
Relativization of states

- Consider a machine with only these two instructions involving Fetch.
  
  \[
  \begin{align*}
  \text{reg} & \xleftarrow{\text{Fetch}} \text{reg} \quad \text{cost} = 1 \\
  \text{int} & \xleftarrow{\text{Fetch}} \text{int} \quad \text{cost} = 3
  \end{align*}
  \]

- Consider a state in which the \text{reg} \xleftarrow{\ldots} item is 2 cheaper than the \text{int} \xleftarrow{\ldots} item.

- Transits to a state in which the \text{reg} \xleftarrow{\ldots} item is 4 cheaper than the \text{int} \xleftarrow{\ldots} item.

- Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.
Relativization of states

- Consider a machine with only these two instructions involving Fetch.
  - \texttt{reg} ← Fetch, cost = 1
  - \texttt{int} ← Fetch, cost = 3

- Consider a state in which the \texttt{reg} ← \ldots item is 2 cheaper than the \texttt{int} ← \ldots item.

- Transits to a state in which the \texttt{reg} ← \ldots item is 4 cheaper than the \texttt{int} ← \ldots item.

- Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.

- Typical instruction sets do not lead to divergence.
Relativization of states

- Naively generated transition tables are very large.
Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situations.
Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situations.

- Two major optimizations
Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situations.

- Two major optimizations
  - State reduction by projecting out irrelevant items.
Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situations.

- Two major optimizations
  - State reduction by projecting out irrelevant items
  - State reduction by triangle trimming.
Consider a machine in which the only instructions involving `Plus` are:

- **State 1:**
  - Goal: `reg`, 1
  - Register: `reg`, 1
  - Address: `reg`, 1
  - Integer: `int`, 0

- **State 2:**
  - Goal: `reg`, 0
  - Register: `Reg`, 0
  - Address: `reg`, 0
  - Integer: `Int`, 0
State Reduction by Projecting out Irrelevant Items

Consider a machine in which the only instructions involving Plus are:

\[
\text{Plus} \quad \text{Plus} \\
\text{reg} \quad \text{reg} \quad \text{reg} \quad \text{int}
\]

Also assume that there are two states:

- **state 1:**
  - goal ← reg, 0
  - reg ← Reg, 0
  - addr ← reg, 0

- **state 2:**
  - goal ← reg, 1
  - reg ← int, 1
  - addr ← reg, 1
  - int ← Int, 0
State Reduction by Projecting out Irrelevant Items

The normal transition table for Plus:

<table>
<thead>
<tr>
<th></th>
<th>first argument</th>
<th>state 1</th>
<th>state 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td>state 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>state 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

second argument
State Reduction by Projecting out Irrelevant Items

Since the first argument of Plus is a reg, we can project the int ← ... item out of both the states. The resulting transition table for Plus is:

<table>
<thead>
<tr>
<th>first argument</th>
<th>state 1</th>
<th>state 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>state 1</td>
<td>state 1</td>
<td>state 2</td>
</tr>
</tbody>
</table>
State Reduction by Triangle Trimming

- Assume that a state $s$ has two items $n_1 \leftarrow \ldots$ and $n'_1 \leftarrow \ldots$. Under what conditions can we say that $n_1 \leftarrow \ldots$ is subsumed by $n'_1 \leftarrow \ldots$ and thus can be removed from $s$. 
State Reduction by Triangle Trimming

- Assume that a state $s$ has two items $nt \leftarrow \ldots$ and $nt' \leftarrow \ldots$. Under what conditions can we say that $nt \leftarrow \ldots$ is subsumed by $nt' \leftarrow \ldots$ and thus can be removed from $s$.

- Assume that the state has been used in the context of the operator $op$ at the argument position shown.

```
  nti ← op
    /
   /\  /
  nt1 / * /
  \   /  /
  nt2 /   /
    \   /
     \  /
      \(*
       (state s) nt
```

$nt_1$, $nt_2$, $nt_3$, and $nt'$ refer to the substates in the triangle diagram.
The general situation under which $nt \leftarrow \ldots$ is subsumed by $nt' \leftarrow \ldots$ is:

The cost of the rule $nt_i \leftarrow \ldots$ and the black chain reductions should be less than the rule $nt_j \leftarrow \ldots$ and the red chain reductions.

Further this should be true in all contexts in which $s$ can be used.
State Reduction by Triangle Trimming

- The general situation under which $nt \leftarrow \ldots$ is subsumed by $nt' \leftarrow \ldots$ is:

- The cost of the rule $nt_i \leftarrow \ldots$ and the black chain reductions should be less than the rule $nt_j \leftarrow \ldots$ and the red chain reductions.
State Reduction by Triangle Trimming

- The general situation under which \( \text{nt} \leftarrow \ldots \) is subsumed by \( \text{nt}' \leftarrow \ldots \) is:

\[
\begin{align*}
\text{nti} & \quad \text{op} \\
\text{nt1} & \quad \ast \quad \text{nt} \quad \ast \\
\text{nt1}' & \quad \ast \quad \text{nt}' \quad (s) \\
\ast & \quad \ast \quad \ast \quad \ast \\
\text{ntj} & \quad \text{op} \\
\text{nt2} & \quad \ast \quad \text{nt2}' \\
\ast & \quad \ast \\
\text{nt3} & \quad \ast \quad \text{nt3}'
\end{align*}
\]

- The cost of the rule \( \text{nti} \leftarrow \ldots \) and the black chain reductions should be less than the rule \( \text{ntj} \leftarrow \ldots \) and the red chain reductions.

- Further this should be true in all contexts in which \( s \) can be used.
BURG – A Code Generation Tool

- Bottom Up Rewriting based code Generator

```
%{
  #define NODEPTR_TYPE treepointer
  #define OP_LABEL(p) ((p)->op)
  #define LEFT_CHILD(p) ((p)->left)
  #define RIGHT_CHILD(p) ((p)->right)
  #define STATE_LABEL(p) ((p)->state_label)
%
%start goal
%%

con: Constant                 = 1 (0);
con: Four                     = 2 (0);
addr: con                     = 3 (0);
addr: Plus(con, reg)          = 4 (0);
addr: Plus(con, Mul(Four, reg)) = 5 (0);
reg: Fetch(addr)              = 6 (1);
reg: Assign(addr, reg)         = 7 (1);
goal: reg                     = 8 (0);
```

BURG’s name for node type user’s name for node type
cost rule number

<table>
<thead>
<tr>
<th>terminals</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>con:</td>
<td></td>
</tr>
<tr>
<td>con: Four</td>
<td></td>
</tr>
<tr>
<td>addr:</td>
<td></td>
</tr>
<tr>
<td>addr: con</td>
<td></td>
</tr>
<tr>
<td>addr: Plus(con, reg)</td>
<td></td>
</tr>
<tr>
<td>addr: Plus(con, Mul(Four, reg))</td>
<td></td>
</tr>
<tr>
<td>reg: Fetch(addr)</td>
<td></td>
</tr>
<tr>
<td>reg: Assign(addr, reg)</td>
<td></td>
</tr>
<tr>
<td>goal: reg</td>
<td></td>
</tr>
</tbody>
</table>
BURG – A Code Generation Tool

- **Bottom Up Rewriting based code Generator**

- **Sample BURG input.**

```c
%{
  #define NODEPTR_TYPE treepointer
  #define OP_LABEL(p) ((p)->op)
  #define LEFT_CHILD(p) ((p) -> left)
  #define RIGHT_CHILD(p) ((p) -> right)
  #define STATE_LABEL(p) ((p) -> state_label)
%
%start goal
%term Assign=1 Constant=2 Fetch=3 Four=4
%term Mul=5 Plus=6

%%

con: Constant = 1 (0);
con: Four = 2 (0);
 addr: con = 3 (0);
 addr: Plus(con, reg) = 4 (0);
 addr: Plus(con, Mul(Four, reg)) = 5 (0);
 reg: Fetch(addr) = 6 (1);
 reg: Assign(addr, reg) = 7 (1);
 goal: reg = 8 (0);
```
BURG – A Code Generation Tool

- Two traversals over the subject tree.

  - Labeling traversal.
    - Done entirely by generated function `burg(label(NODEPTR TYPE p))`.
    - Labels the subject tree with states (represented by integers).

  - Rule selection traversal.
    - Done by a wrapper function `reduce(NODEPTR TYPE p, int goalInt)` written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
    - Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
- Labeling traversal done entirely by generated function `burg
label(NODEPTR TYPE p)`.
- Labels the subject tree with states (represented by integers).
- Rule selection traversal done by a wrapper function `reduce(NODEPTR
TYPE p, int goalInt)` written by user around BURG generated functions.
- Starts with the root of the subject tree and the non-terminal goal.
- At each node selects a rule for evaluating the node.
- Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      
      ```c
      burg_label(NODEPTR_TYPE p).
      ```
  - Rule selection traversal.
    - Done by a wrapper function
      ```c
      reduce(NODEPTR_TYPE p, int goalInt)
      ```
      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
    - Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      \texttt{burg\_label(NODEPTR\_TYPE p)}.
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      \texttt{reduce(NODEPTR\_TYPE p, int goalInt)} written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
    - Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      \[
      \text{burg\_label}(\text{NODEPTR\_TYPE}\ p).
      \]
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      \[ \text{burg}\_\text{label}(\text{NODEPTR}\_\text{TYPE} \ p) \].
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      \[ \text{reduce}(\text{NODEPTR}\_\text{TYPE} \ p, \text{int} \ \text{goalInt}) \]
      written by user around BURG generated functions.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      
      ```c
      burg_label(NODEPTR_TYPE p).
      ```
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      
      ```c
      reduce(NODEPTR_TYPE p, int goalInt)
      ```
      
      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      \[
      \text{burg}_\text{label}(\text{NODEPTR}_{\text{TYPE}} \ p). \\
      \]
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      \[
      \text{reduce}(\text{NODEPTR}_{\text{TYPE}} \ p, \text{int} \ \text{goalInt}) \\
      \]
      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      \[
      \text{burg\_label(NODEPTR\_TYPE p)}.
      \]
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      \[
      \text{reduce(NODEPTR\_TYPE p, int goalInt)}
      \]
      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
    - Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.
BURG – A Code Generation Tool

Here is an outline of a code-generator produced with the help of BURG. Constructs in red are BURG generated.

```c
parse(NODEPTR_TYPE p) {
    burg_label(p) /* label the tree */
    reduce(p, 1) /* and reduce it, goal = 1*/
}

reduce(NODEPTR_TYPE p, int goalint) {
    int ruleno = burg_rule(STATE_LABEL(p), goalint);
    short *nts = burg_nts[ruleno];
    NODEPTR_TYPE kids[10];
    int i;
    /* ... do something with this node... */
    /* process the children of this node */
    burg_kids(p, ruleno, kids);
    for (i = 0; nts[i]; i++)
        reduce(kids[i], nts[i]);
}
```