KENDALL - A BURS Code Generator Generator for Stack Caching Stack-Based VMs

Jakob Sievers
TU Wien

Abstract

Kendall is a code generator generator for stack-based stack caching VMs. Given a description of the target interpreter, it generates BURS tables suitable for controlling table-driven BURS pattern matchers which may be used to select a locally optimal sequence of transitions and instructions for any legal expression sequence of the target interpreter.

1 Introduction

Generation of optimal code is clearly a necessary feature of any serious compiler project. BURS style code generating automata are known for their extreme speed. Unfortunately, existing BURS code generator generators such as burg do not handle the sort of grammars encountered when dealing with finite state machine descriptions of stack caching VMs very well. Therefore, a code generator generator based on BURS theory but tailored to the specifics of a stack caching VM is desirable.

One of the effects of utilizing stack caching is that an interpreter may be viewed as a non-deterministic finite automaton, with multiple stack states and multiple instances of every instruction. The basic idea of BURS based tree parser generators is that the cost of reducing every nonterminal at every possible point in every possible sequence of instructions is calculated in advance. The parsers then use this information to label and reduce expression sequences at compile time and find the shortest path of instructions and stack state transitions from a given start stack state to an end stack state (using all the instructions of the expression sequence) in the graphical representation of the interpreter and hence select an ideal sequence of machine instructions.

Kendall generates either tables for table-driven parsers or hard-coded parsers from a burg-style input grammar describing the target interpreter.

These parsers emit optimal code when run on an expression sequence (a sequence of instructions and a start stack state) and the compiler front end may use them to determine the sequence of instructions and transitions for the interpreter to run.

2 Background

2.1 Stack Caching

Conventionally, a stack-based virtual machine (VM) might represent the stack by a pointer to the uppermost stack item (the top-of-stack) which is stored in a register (the actual contents of the stack are kept in memory). A multiple-state stack caching VM on the other hand utilizes a stack cache in addition to a pointer to the first in-memory stack item. Conceptually, the stack cache consists of a number of registers reserved for holding items off of the top of the stack. This notion gives rise to a set of stack-cache states or stack representations (S1, S2, ..., SN) which differ in the number of items kept in registers (for example, in S1 only the top-of-stack might be kept in a register while in S2 the TOS and the second item might be kept in registers, with the stack pointer pointing to the third item which is kept in memory). Figure 1 shows a hypothetical stack state Si in which three items are kept directly in registers and the stack pointer points to the fourth item (in memory). In general, stack caching vastly reduces the number of loads and stores to and from memory needed.

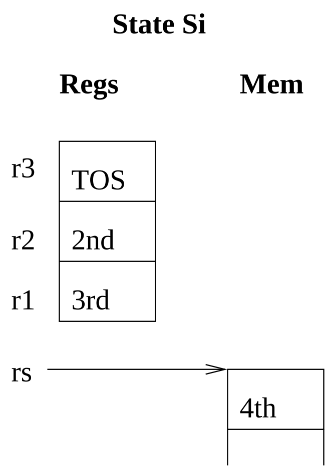

Figure 1: Hypothetical Stack State.

If stack caching is used, the current stack-cache state has to be taken into consideration when implementing VM instructions. Specifically, for any given sequence of instructions, each instruction has to start in the stack state the previous instruction ended in. For example, assuming that in stack state S0 no stack items are kept in registers (aside from the stack pointer) and that in stack state S1 one item is kept in a register, if an instruction requiring two operands (popped off the stack) and pushing one result value back onto the stack were executed in stack state S0, two loads would have to be performed and the interpreter would be in stack state S1 after it finished (the result would be pushed into one of the registers allotted for stack caching). Now, if the next instruction also required two operands, only one load would be necessary. Hence, at least one instance per instruction per stack state is needed and the interpreter may now be viewed as a non-deterministic finite state machine with the states corresponding to stack states, and transitions corresponding to VM instructions.

To reduce the number of VM instruction instances (which is equal to the number of VM instructions times the number of stack states in the naive case), state transitions are introduced (these correspond to epsilon transitions in the automaton): Firstly, rarely used VM instruction instances are removed, keeping only the most frequently used instances of each instruction. Now, if the interpreter is in stack state S_i after executing instruction I_j and wishes to execute instruction I_k for which the S_i instance has been removed, it simply inserts transitions to the start stack state of one of the remaining instances of I_k (S_l, say) before executing I_k. Secondly, as it turns out, frequently the machine code to which certain instruction instances are compiled is a suffix of the machine code to which other instances of the same instruction are compiled. In such cases only the suffix instance is needed (as an example for a suffix instance, consider two instances of an instruction I, I1 and I2, which are compiled into real machine instructions $MI1_1$, ..., $MI1_n$ and $MI2_1$, ..., $MI2_m$ respectively. Now, if for some $o > 0$ the equation $MI1_(n-i) = MI2_(m-i)$ (with i = 1, ..., o) holds, I2 is a suffix of I1 and I1 may be replaced by state transitions from its own start stack state to the start stack state of I2 followed by I2).

Static stack caching requires the compiler front end to keep track of stack states and record the VM instruction instances the interpreter has to execute.

2.2 BURS

Code generators based on Bottom-Up Rewrite System (BURS) theory consist of two automata: A labeler and a reducer. When parsing a code tree, the labeler first executes a bottom-up walk of the tree, assigning each node in the tree a label (a BURS state) as it does so. The reducer then performs a top-down traversion of the tree, reducing it to a goal nonterminal, and emitting code in the process. Since kendall only supports unary operators, all expressions trees may be viewed simply as sequences of expressions; in the sections that follow, the term "sequence" will often be used rather than the term "tree". The tree-specific terminology translates to sequences in a very straight-forward fashion, e.g. a sequences' "leaf" is the last element in the sequence. Both reducer and labeler are usually table-driven, turning the problem of generating code generators into the problem of generating tables to control the code generators. BURS code generators can be extremely fast because all dynamic programming is done when the automaton is built (i.e., at table generation time).

The generator generators use tree pattern matching and dynamic programming to generate BURS tables for table-driven parsers. The input is a set of rules describing possible tree patterns with associated replacement symbols, costs and actions:

$$\text{ReplacementSymbol} \quad \Longrightarrow \quad \text{LinearizedTreePattern} \quad \text{Cost} \quad \text{Action}$$

- The set of all rules is known as the *grammar*.
- Replacement symbols are also known as *nonterminals*.
- Tree patterns are made up of operators and nonterminals.
- Rules of the form

  $$\text{Nonterminal} \quad \Longrightarrow \quad \text{Nonterminal} \quad \text{Cost} \quad \text{Action}$$

  are known as *chain rules*.
- Rules of the form

  $$\text{Nonterminal} \quad \Longrightarrow \quad \text{Operator(Nonterminal)} \quad \text{Cost} \quad \text{Action}$$

  are known as *base rules*.

As an example of a tree grammar, consider table 5. Looking at rule number zero, one can see that it reduces the nonterminal "s0" from the nonterminal "s2" at a cost of one. Rule number zero is a chain rule. Rule number one on the other hand has as its RHS the linearized tree pattern "istore(s1)" and it reduces the nonterminal "s0" at a cost of two. Rule two is a base rule.

The central data structure of the BURS code generator generator is the *BURS state* which encodes all the information needed to label and reduce trees, i.e. the rules necessary to reduce every nonterminal with associated costs. Sample BURS state:

$$
\begin{array}{lllll}
\{ & Nonterminal_1 & \Rightarrow & Rule_i & \# c_1 \\
 & Nonterminal_2 & \Rightarrow & Rule_j & \# c_2 \\
 & \quad ... & & & \\
 & Nonterminal_n & \Rightarrow & Rule_k & \# c_n \quad \}
\end{array}
$$

BURS states are also known as *item sets*.

To generate all necessary BURS states (i.e. the BURS states needed to label every possible sequence in such a way that the reducer can find a least cost parse), every combination of

- Goal nonterminal and node label
- Operator and operand BURS state

must be considered. A precise description of the necessary algorithms may be found in Proebsting's 1995 paper [5], but the basic idea is to try every possible operator/BURS state combination for every goal nonterminal. To do this efficiently, BURS realizes dynamic programming via the concept of representer states. Representer states are operator specific: They retain only those nonterminals which may actually be matched by the operator (that is to say, the grammar contains at least one rule with a right-hand side of "operator(nonterminal)"). Dynamic programming, then, is implemented as follows: Since many different BURS states usually map to the same representer state, the representer states identify common subproblems and at the beginning of each iteration of the table generation algorithm (for each new operator/BURS state pair), the current BURS state is projected to a representer state for the current operator and if the representer state already exists in the global table of representer states (op-reps), no further work is needed, the transition table (a table giving the BURS state resulting from every combination of operator/operand BURS state) is updated, and the next iteration begun.

2.3 Code Generation

As mentioned above, static stack caching requires the front end to keep track of the current stack state, selecting instructions and inserting transitions as required. The question now arises how to map the stack states, instructions and transitions of the interpreter automaton to the rules, chain-rules and nonterminals of BURS code generators, so that a BURS code generator generator may be used to create them. An answer is provided by Ertl and Gregg [2]:

Stack states	$\longrightarrow$	Nonterminals
VM instructions	$\longrightarrow$	Unary operators
VM instruction instances	$\longrightarrow$	Base rules
Stack state transitions	$\longrightarrow$	Chain rules

The base rules have the shape"$n_i \Rightarrow op(n_j)$', where n_i and n_j are nonterminals (that is, the RHS consists of the unary operator for that instruction and the stack state of the instance).

3 Implementation

3.1 Overview

Kendall implements the code generation scheme outlined above. Its input is a set of normal-form rules describing the NFA representation of a stack caching interpreter. It generates either tables for table driven reducers/labelers or hard-coded reducers/labelers. The tables are output either in Scheme reader friendly format (using print) or in a more Unix-style format of one record per line with whitespace separated fields. The hardwired automata use a template identical to that of the table-driven automata with lookup calls replaced by cond statements. Example implementations of the table-driven automata are provided.

3.2 Data Structures

BURS rules are represented by proper lists:

```
(rule-number lhs rhs cost action)
```

Where rule-number and cost are integers and lhs, rhs and action are strings. BURS states are alists of (nonterminal . rcp) pairs where rcp is a Rule-Cost pair, a simple define-record structure. The various tables are alists of alists (one alist per dimension).

3.3 Algorithm

Kendall's main algorithm is a slight variation of the standard BURS algorithms. It is presented in figure 2 in C-like pseudocode.

Close-rules() is a procedure which inserts a number of additional rules into the rule set to simplify some aspects of sequence parsing and table generation. Specifically, some rules are inserted for a new dummy nonterminal 'kendall' which can be reduced from any other nonterminal at zero cost:

```
kendall() {
    close-rules();
    compute-leaf-states(nonterminals);
    while (worklist) {
        state = pop(worklist);
        for-each (op in operators) {
            pstate   = project(op, state);
            ps-index = op-reps-lookup(op, pstate);
            result   = make-state();
            if (!ps-index) {
                op-reps-install(op, pstate);

                for-each (r in oprules) { //rules with rhs == op
                    nt   = rhs-arg(r);
                    cost = cost(r) + state-cost(pstate, nt);

                    if (cost < state-cost(result, lhs(r)))
                        state-entry-set(result, lhs(r),
                                        make-rcp(number(r), cost));
                }
                normalize-costs(result);
                rs-index = state-lookup(result);
                if (!rs-index) {
                    closure(result);
                    rs-index = state-install(result);
                    append(worklist, rs-index);
                }

                transition-table-install(op, ps-index, rs-index);
            }
            op-map-install(op, state, ps-index);
        }
    }
    make-rules-table();
}
```

Figure 2: Kendall's main table generation algorithm.

$$\text{kendall} \quad \Longrightarrow \quad Nonterm_i \quad 0$$

The kendall rules are used in the reducer whenever the caller does not provide an explicit goal nonterminal. In such a case, kendall deduces the nonterminal in which the instruction used for the operator of the root node ends (on the assumption that the root node is always labeled with some operator and that the cheapest rule will be the one corresponding to a user provided rule for that operator) by checking which rule would be applied if kendall was the goal nonterminal (since there aren't any user provided rules for kendall and only chain-rules for kendall are added, the kendall entries are always added during closure and since the cost of deriving kendall from any other nonterminal is zero (i.e. equal), the cheapest nonterminal is picked).

This nonterminal isn't really necessary and it could be replaced by a BURS state table lookup (to find the cheapest nonterminal in the root node's label's item set). As neither the "kendall" rules nor the nonterminal itself are relevant to any other part of the table generation and parsing process, they have been omitted from the examples below.

Also, a 'stay' rule is inserted for every nonterminal, which reduces the nonterminal from itself at zero cost (these could be replaced by a modification of compute-leaf-states):

$$Nonterm_i \quad \Longrightarrow \quad Nonterm_i \quad 0$$

The use of these will be illustrated below. Normalize-costs simply subtracts from the pre-normalization cost of every item in a BURS state the minimum cost for that BURS state. The other auxiliary procedures (closure(), project(), and compute-leaf-states()) are listed in the appendix as they will be referred to in the next section. Make-state() initializes the BURS state so that all costs are equal to infinity.

A sample labeler/reducer combination is distributed with kendall. They are written in Scheme but may be called from C by including kendall.h and calling match() on the root of the target sequence. The file ctest.c shows how to do this (it also includes a constructor/destructor for the node type). Figures 3 and 4 may be referred to for details.

```
label(node) {
    chld = node-child(node);
    op   = node-op(node);
    if (chld == #f) { // Leaf node
        node-label-set(node, leaf-label-lookup(op)); //op is a nonterm
    } else {
        label(chld);  // recur
        node-label-set(node,
                       tt-lookup(op,
                                 state-to-repstate(op,
                                                   node-label(chld))));
    }
}
```

Figure 3: Labeler

3.4 Examples

Some examples of BURS state generation and sequence parsing shall now be examined. The input grammar (description of the target interpreter) used is given in figure 5 (in an actual input file, each rule would be rendered as “ruleno:lhs:rhs:cost:action”, with one entry per line for easy parsing).
The rules are in normal-form, meaning that only rules of the form “n $\Rightarrow$ m” and “$n_0 \Rightarrow op(n_1, ..., n_k)$” are permitted, that is, rules of the form “n $\Rightarrow$ op(..., op(...), ...)” are disallowed. Note that rules zero, two, three, seven, and eight are chain-rules and that the action strings verbally describe movements around the graphical representation of the automaton (see figure 7). Close-rules() adds the rules shown in figure 6.

Table generation begins with the computation of leaf states (“compute-leaf-states(leaves)”, line three in figure 2) which results in BURS states one, two and three for nonterminals s2, s1 and s0 respectively. We shall go through the process step-by-step for nonterminal s1:

The “leaf-rules” for s1 are rule number eight and rule number 14 (one of the newly added “stay” rules). After the inner for-each loop, the BURS state looks as follows:

$$\begin{array}{lllll} \{ & s0 & \Rightarrow & - & \# \infty \\ & s1 & \Rightarrow & 14 & \# 0 \\ & s2 & \Rightarrow & 8 & \# 1 \quad \} \end{array}$$

Cost normalization has no effect since the minimum cost is zero. The BURS state is now closed by applying chain-rules, specifically, rule number zero which derives s0 from s2:

$$\begin{array}{lllll} \{ & s0 & \Rightarrow & 0 & \# 2 \\ & s1 & \Rightarrow & 14 & \# 0 \\ & s2 & \Rightarrow & 8 & \# 1 \quad \} \end{array}$$

```
reduce(root, &optional goal) {
    if (!goal) {
        goal = rhs(rule-lookup(rules-table-lookup("kendall",
                       node-label(root))));
    }
    result = reverse(reduce-tree(root, goal));
    for-each (r in result) {
        fire();
    }
}

reduce-tree(node, nonterm) {
    state = state-reverse-lookup(node-label(node));
    op    = node-op(node);
    chld  = node-child(node);
    rlist = reduce-node(state, op, nonterm); //
    nnt   = get-nnt(rlist); // next nonterm
    if (chld == #f) { //leaf, done
        return(rlist);
    } else {
       append(rlist,reduce-tree(chld, nnt));
    }
}

reduce-node(state, op, nonterm) {
    loop(nt) {
        rule     = rule-lookup(state-rule(state,nt));
        rule-rhs = rhs-op(rule);
        case {
            (op == nonterm) '()

            (op == rule-rhs) cons(number(rule), '())

            default cons(number(rule), loop(rule-rhs))
        }
    }
    loop(nonterm); //initiate
}
```

Figure 4: Reducer.

RuleNo	LHS		RHS	Cost	Action
0	s0	$\Longrightarrow$	s2	1	S2 TO S0 VIA EPS
1	s0	$\Longrightarrow$	istore(s1)	2	S1 TO S0 VIA ISTORE
2	s1	$\Longrightarrow$	s0	1	S0 TO S1 VIA EPS
3	s1	$\Longrightarrow$	s2	1	S2 TO S1 VIA EPS
4	s1	$\Longrightarrow$	iload(s0)	2	S0 TO S1 VIA ILOAD
5	s1	$\Longrightarrow$	istore(s2)	2	S2 TO S1 VIA ISTORE
6	s1	$\Longrightarrow$	iadd(s2)	2	S2 TO S1 VIA IADD
7	s2	$\Longrightarrow$	s0	1	S0 TO S2 VIA EPS
8	s2	$\Longrightarrow$	s1	1	S1 TO S2 VIA EPS
9	s2	$\Longrightarrow$	iload(s1)	2	S1 TO S2 VIA ILOAD

Figure 5: Sample Grammar.

RuleNo	LHS		RHS	Cost	Action
10	kendall	$\Longrightarrow$	s2	0	KENDALL
11	kendall	$\Longrightarrow$	s1	0	KENDALL
12	kendall	$\Longrightarrow$	s0	0	KENDALL
13	s2	$\Longrightarrow$	s2	0	STAY
14	s1	$\Longrightarrow$	s1	0	STAY
15	s0	$\Longrightarrow$	s0	0	STAY

Figure 6: Additional rules.

Notice the cost of two which reflects the fact that to reduce s0, two rules have to be applied at a cost of one each: Firstly, s2 has to be reduced via rule eight and from that rule zero can be used to reduce s0. Looking at table 1, it can be seen that global BURS state number two has just been computed. This BURS state may be informally read as follows: A node labeled with BURS state number two can be reduced to s0 at cost two via rules eight and zero, to s1 via rule 14 at zero cost and to s2 at cost one via rule eight.

Let's assume for a moment that no stay rules had been inserted. Leaf state computation would now give BURS state 2':

$$\begin{array}{lcccc} \{ \; \text{s0} & \Rightarrow & 0 & \#\,1 & \\ \phantom{\{ \;}\text{s1} & \Rightarrow & 3 & \#\,1 & \\ \phantom{\{ \;}\text{s2} & \Rightarrow & 8 & \#\,0 & \} \end{array}$$

The rules for nonterminals s0 and s2 are the same as before but since rule 14 wouldn't exist, s1 would have to be reduced via rules eight and three at cost one (cost-normalization would have reduced the cost of rule eight to zero). This is clearly illogical, especially taking into account the graphical representation of the interpreter: To get from s1 to s1 we go first to s2 and

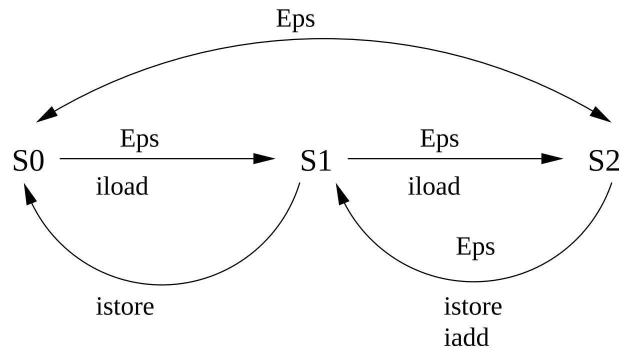

Figure 7: Target Interpreter.

then back to s1.

BURS states 1, 2, and 3 are special in that they are "leaf-states", used as the base case by the recursive BURS labeler which will, upon encountering a leaf node, label it directly without a transition table lookup, and errors made here propagate upwards.

Having filled the worklist with some leaf states, we are now ready to enter the while loop (line four in figure 2). In order to simplify tracing the execution of the algorithm, three "breakpoints" are set after which the value of the current (new) BURS state will be given: Just after project() (line seven) the value of *pstate* will be shown, after normalize-costs() (line 19), the value of *result* will be given and the last breakpoint is just after closure() where the final value of *result* (as it will be entered into the global BURS states table) will be provided.

The first BURS state popped off of the worklist is *BURS state number one*:

$$\begin{array}{llllll} \{ & s0 & \Rightarrow & 0 & \#\,1 & \\ & s1 & \Rightarrow & 3 & \#\,1 & \\ & s2 & \Rightarrow & 13 & \#\,0 & \} \end{array}$$

The first operator is *iadd*. There's only one rule for iadd, rule number six, which makes state computation very simple. The representer state resulting from project(iadd, 1) (which, incidentally, is iadd's only representer state) looks as follows:

$$\begin{array}{llll} \{ & s0 & \#\,\infty & \\ & s1 & \#\,\infty & \\ & s2 & \#\,0 & \} \end{array}$$

Two things to notice here are that representer states carry only cost information (rule numbers are not relevant for labeling) and that cost normalization reduced the cost to zero. After going through the for-each loop and subsequent cost normalization, result looks as follows (again, only rule number six is an element of oprules for iadd):

$$\{ \begin{array}{lcll} s0 & \Rightarrow & - & \# \infty \\ s1 & \Rightarrow & 6 & \# 0 \\ s2 & \Rightarrow & - & \# \infty \end{array} \}$$

One might think of this new BURS state as the BURS state resulting from iadd receiving a node labeled with BURS state one as its argument. Finally, the BURS state is closed to get

$$\{ \begin{array}{lcll} s0 & \Rightarrow & 0 & \# 2 \\ s1 & \Rightarrow & 6 & \# 0 \\ s2 & \Rightarrow & 8 & \# 1 \end{array} \}$$

This corresponds to BURS state number four in table 1. The outer for-each loop is now re-iterated with *op* equal to *iload*. Projection results in:

$$\{ \begin{array}{ll} s0 & \# 0 \\ s1 & \# 0 \\ s2 & \# \infty \end{array} \}$$

As can be seen from table 4, this is iloads's representer state number one. The inner for-each loop is run twice this time, giving

$$\{ \begin{array}{lcll} s0 & \Rightarrow & - & \# \infty \\ s1 & \Rightarrow & 4 & \# 0 \\ s2 & \Rightarrow & 9 & \# 0 \end{array} \}$$

Finally, closure produces

$$\{ \begin{array}{lcll} s0 & \Rightarrow & 0 & \# 1 \\ s1 & \Rightarrow & 4 & \# 0 \\ s2 & \Rightarrow & 9 & \# 0 \end{array} \}$$

and that is BURS state five. The same procedure is then carried out with istore. During the second run of the outermost while loop, already dynamic programming kicks in. *State* is now *BURS state number two.*

$$\{ \begin{array}{lcll} s0 & \Rightarrow & 0 & \# 2 \\ s1 & \Rightarrow & 14 & \# 0 \\ s2 & \Rightarrow & 8 & \# 1 \end{array} \}$$

Firstly, for operator *iadd* the same representer state as before is projected:

$$\{ \quad \begin{array}{ll} \text{s0} & \# \infty \\ \text{s1} & \# \infty \\ \text{s2} & \# 0 \end{array} \quad \}$$

Hence, the transition table can be updated without further calculations. We carry on using the next operator, *iload* which produces a new representer state and hence a new BURS state as well:

$$\{ \quad \begin{array}{ll} \text{s0} & \# 2 \\ \text{s1} & \# 0 \\ \text{s2} & \# \infty \end{array} \quad \}$$

Using this representer state, the resulting BURS state initially looks like so:

$$\{ \quad \begin{array}{llll} \text{s0} & \Rightarrow & \text{-} & \# \infty \\ \text{s1} & \Rightarrow & 4 & \# 2 \\ \text{s2} & \Rightarrow & 9 & \# 0 \end{array} \quad \}$$

Closure() turns this into:

$$\{ \quad \begin{array}{llll} \text{s0} & \Rightarrow & 0 & \# 1 \\ \text{s1} & \Rightarrow & 3 & \# 1 \\ \text{s2} & \Rightarrow & 9 & \# 0 \end{array} \quad \}$$

That is, BURS state number seven. It should be observed that closure() found a cheaper derivation via a chain-rule application for nonterminal s1 in addition to the rule for nonterminal s0 for which no direct iload rule exists. This process continues until no combination of operator/BURS state yields new BURS states: The first "round" consists of computing the BURS states resulting from each of the operators receiving each of the leaf states (which were computed separately) as an argument; in the second round, the BURS states resulting from every operator receiving every BURS state from round one as an argument are computed and then used in round three and so on until no new BURS states are found.

Again, suppose we didn't have stay rules and BURS state 2' rather than BURS state 2 was used as the argument to istore. The end result would now be:

$$\{ \quad \begin{array}{llll} \text{s0} & \Rightarrow & 1 & \# 1 \\ \text{s1} & \Rightarrow & 5 & \# 0 \\ \text{s2} & \Rightarrow & 8 & \# 1 \end{array} \quad \}$$

StateNo	*s*0	*s*1	*s*2
1	(0, 1)	(3, 1)	(13, 0)
2	(0, 2)	(14, 0)	(8, 1)
3	(15, 0)	(2, 1)	(7, 1)
4	(0, 2)	(6, 0)	(8, 1)
5	(0, 1)	(4, 0)	(9, 0)
6	(1, 1)	(5, 0)	(8, 1)
7	(0, 1)	(3, 1)	(9, 0)
8	(1, 0)	(5, 1)	(7, 1)
9	(0, 2)	(4, 0)	(9, 1)
10	(1, 0)	(5, 0)	(7, 1)
11	(0, 1)	(4, 1)	(9, 0)

Table 1: States.

Let's call this BURS state 7'. In this item set, s1 is cheaper to reduce than s0.

Once the worklist is exhausted and the algorithm finishes, the tables look as follows.

Table 1 contains the global list of BURS states in tabularized form, with rows corresponding to item sets (each entry states[s][nt] with BURS state number s and nonterminal nt gives the rule-cost pair (n,m) which contains the number (n) of the rule needed to reduce nonterminal nt in BURS state s and the cost (m) of doing so).

Op	1	2	3	4
iadd	4			
iload	5	7	9	11
istore	6	8	10	

Table 2: Transition Table - tt-lookup(Op, Repstate) -¿ BURS State.

Table 2, the transition table describes the mapping $TT : Op \times Repstate \rightarrow State$; it is used by the labeler to deduce the label of a node from its operator and the label of its child (if the labeler is at a node with node.op = iload and child.label = 3, for example, it will label the current node "9").

Table 3 maps BURS states to operator specific representer states. This table is also used by the labeler which has to deal both with real BURS states as node labels and representer states as indexes into the transition table (for example, all BURS states map to the same representer state ("1") with respect to operator iadd; istore on the other hand has three different representer states).

Table 4 give each operator's repstates in a format similar to that of table

Op	1	2	3	4	5	6	7	8	9	10	11
iadd	1	1	1	1	1	1	1	1	1	1	1
iload	1	2	3	2	4	4	1	3	2	1	1
istore	1	2	3	2	3	2	1	3	2	2	1

Table 3: Op-map - state-to-repstate(Op, State) -¿ Repstate.

Op	*StateNo*	*s*0	*s*1	*s*2
iadd	1	inf	inf	0
iload	1	0	0	inf
	2	2	0	inf
	3	0	1	inf
	4	1	0	inf
istore	1	inf	1	0
	2	inf	0	1
	3	inf	0	0

Table 4: Op-reps.

1, except that representer states do not encode any information about rule numbers (which are irrelevant for the labeler) - they carry only cost information.

The BURS tree parsing mechanism shall now be examined by way of an example. The expression sequence for which code is to be generated is given in figure 8, and might have resulted from the expression a = b + c. It may be read as "iload, iload, iadd, istore, starting in s1".

```
istore (8)
 |
iadd   (4)
 |
iload  (5)
 |
iload  (7)
 |
 s1    (2)
```

Figure 8: Example sequence.

The labeler first descends to the leaf node, s1, and labels it by looking up its leaf-state, BURS state number two. To label the second node from the bottom, iload, it looks up state-to-repstate(iload, 2) which gives repstate number two and then, tt-lookup(iload, 2) which gives BURS state number seven. The labeler continues in this fashion up the sequence. In figure 8, the labels are given in parentheses.

If the reducer is called with no goal nonterminal, it assumes that the nonterminal to which the root node can most cheaply be reduced will suffice. It finds this nonterminal by looking up (istore, kendall) which gives s0.

So, the first call to reduce-node() is reduce-node(8, istore, s0). Looking at table 1, it can be seen that for BURS state number eight, nonterminal s0 the RCP entry is (1,0) i.e. rule number one. Rule number one's RHS is istore(s1) which fits the node's operator. Thusly, the first element of *rlist* is 1.

Moving on to the next node: This time the call is reduce-node(4,iadd,s1) (s1 is the previous rules' right hand sides' argument). We again check the states table to find out that the rule we're looking for is rule number six, the RHS operator of which is iadd so were done with this node. Rlist is now (1 6).

Rule sixes RHS argument is s2 so the next call becomes reduce-node(5,iload,s2), adding '9' to rlist.

The following call is more interesting: It is reduce-node(7,iload,s1), and in BURS state number seven the rule for s1 is rule number three but the RHS of rule 3 is s2, not iload so we have to call loop(s2). The rule for s2 in BURS state 7 is rule 9 which has an RHS of iload so the endresult is (3, 9).

The final call to reduce-node is reduce-node(2,s1,s1). Since s1 doesn't have to be reduced to itself, the first case fires and the empty list is returned.

The final list of rules is (1 6 9 3 9) and the actions are fired in reverse order:

```
S1 TO S2 VIA ILOAD
S2 TO S1 VIA EPS
S1 TO S2 VIA ILOAD
S2 TO S1 VIA IADD
S1 TO S0 VIA ISTORE
```

the reader may wish to trace the path described by these instructions in figure 7.

We will go through one more example to demonstrate the significance of stay rules. Suppose that we didn't have stay rules and that we were given the expression sequence depicted in figure 9. Clearly, the cheapest derivation consists simply of an application of rule one, however, rather than being labeled with BURS states 2 and 8 as it would be using the tables given above, it would be labeled with the BURS states 2' and 7' which we calculated earlier.

```
istore (7')
 |
s1     (2')
```

Figure 9: Another example sequence.

Recalling BURS state 7', we see that if we run the reducer on this sequence without providing a goal nonterminal, it picks the cheapest nonterminal (in this case s1) for the first call to reduce-node(). The first call to reduce-node is reduce-node(7', istore, s1) now we get rule number five followed by a call reduce-node(2', s2, s1) which gives rule number eight.

```
S1 TO S2 VIA EPS
S2 TO S1 VIA ISTORE
```

Two, rather than one actions are fired! Generally, this problem occurs when

- The expression sequences' leaf is nonterminal nt_j.
- The first non-leaf node's operator is *operator*.
- No explicit goal nonterminal is provided.
- There are multiple instances of an instruction (rules with the same RHS operator) such that

  $$nt_i \implies \text{operator}(nt_j) \quad \# \text{ n}$$

  and

  $$nt_l \implies \text{operator}(nt_k) \quad \# \text{ m}$$

  as well as at least two chain-rules:

  $$nt_k \implies nt_j \quad \# \text{ o}$$

  and

  $$nt_j \implies nt_k \quad \# \text{ p}$$

and $(n + o + p) > (m + o)$ (without stay rule, nt_j's leaf state records a cost of (o+p) for reducing nt_j from itself).

In systems supporting nullary operators, kendall's stay rules could be replaced by introducing a nullary operator "$Nonterm_i$" for every nonterminal "$nonterm_i$", along with rules to reduce "$nonterm_i$" from "$Nonterm_i$" at zero cost ("$nonterm_i \Rightarrow Nonterm_i\#$ 0"). In expression sequences, the nullary operator for the desired start stack state would be used as the leaf of the sequence rather than using the nonterminal directly.

3.5 Improvements

Performance and error checking are the areas which could benefit most from improvements. Kendall does not implement any of the optimization techniques described by Proebsting [5]. The alist-based internal representation of tables isn't very efficient. The use of hash tables or vectors would probably be desirable. In places, recursion is non-tail-recursive. Kendall does not use an efficient encoding for the output tables and neither do the hardwired parsers utilize any special optimizations. No syntax checking is done on the input grammar. Kendall doesn't scale very well; see the conclusion for details.

4 Related Work

The most well known implementations of BURS style code generator generators are burg and iburg. Burg is described by Fraser, Henry, and Proebsting [4], and iburg by Fraser, Hanson and, Proebsting [3]. The most important difference between iburg and kendall (and burg for that matter) is that iburg's code generators use dynamic programming at compile time (not at compile compile time like kendall), "IBURG produces slower code generators, but does so very quickly". Iburg is otherwise compatible with burg, and both burg and iburg are geared towards generating generators for more general types of grammars. They support nullary, unary and binary operators whereas kendall supports only unary operators (expression trees containing only nonterminals and unary operators have degenerated to lists).

Another, commercial, code generator generator is BEG, described by Emmelmann, Schröer, and Landwehr [1]. In addition to generating code generators, modules for a number of other tasks such as register allocation are available. BEG, like iburg, uses dynamic programming at compile time.

5 Conclusion

To gauge kendall's behavior when confronted with large input grammars, two sets of scalability tests were run using subsets of a description of the GNU Forth VM as the input. The grammar describing the GForth VM consists of 9 stack states and 352 instructions, with 1177 rules altogether. Details of the test setup (machine, etc.) may be found in the file time/TIMINGS.

For the first group of tests, the size of the input grammar was reduced by including only a subset of the available stack states. The first column in table 5 shows the number of stack states retained (an entry of 1 means that only stack state number zero was kept, an entry of two that only stack states number zero and one were kept and so on), the second column shows the number of rules remaining in the grammar after removing all rules related to one of the removed stack states, and the fourth column gives the runtime of kendall on this grammar as reported by the UNIX time command. Column number three gives the number of BURS states generated for this input (this number does not include the leaf states).

No. of Stack States	No. of Rules	No. of BURS States	Time
1	353	354	27.083
2	383	403	32.793
3	481	795	1:17.07
4	595	3682	13:07.57
5	708	-	1h+

Table 5: Benchmarks - Part 1.

In the second set of tests, in addition to removing some of the stack states, only the first 100 instructions were used. This reduced the size of the grammar even further, and the results are given in table 6.

No. of States	No. of Rules	No. of BURS States	Time
1	100	101	1.141
2	107	107	1.379
3	156	321	4.018
4	209	1395	26.659
5	265	5378	3:26.69
6	323	18287	42:00.01
7	383	-	1h+

Table 6: Benchmarks - Part 2.

It seems clear that the number of stack states has a greater influence on performance than the number of rules. This is likely due to the increased size of item sets (BURS states) resulting from an increase in the number of nonterminals. Also, a larger number of nonterminals means a larger number of distinct BURS states as fewer BURS states will map to the same representer state or be identical after cost normalization. In addition to the reasons mentioned above, the fact that Scheme uses garbage collection rather than explicit memory management probably slows kendall down somewhat since large amounts of memory are allocated and almost immediately released (BURS states for which the representer state already exists). In fact, Proebsting [5] mentions that burg uses a specialized memory management scheme with many specific optimizations.

Overall, it may be said that kendall handles grammars with up to about five stack states reasonably well but is too slow to be of much practical use for larger grammars.

6 Acknowledgments

I would like to thank A. Ertl for guidance, comments, etc. and F. Winkelmann for CHICKEN Scheme, a nice implementation of a great language.

7 Availability

Kendall sources are available from:

`ftp.quux.baz/foo`

kendall-XX.tgz contains the Scheme source files. kendall-XX-dist.tgz contains C source files compiled by the CHICKEN Scheme-to-C compiler along with the CHICKEN runtime. They were tested with gcc 2.95.

References

[1] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG - A generator for efficient back ends. In *PLDI*, pages 227–237, 1989.

[2] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.

[3] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[4] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[5] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

Appendix

Compute-leaf-states() computes the initial set of BURS states, one for each nonterminal. Project() computes the representer state of a BURS state with respect to a given operator. Closure() closes an item set/BURS state by applying chain-rules.

```
compute-leaf-states(leaves) {
    for-each (leaf in leaves) {
        state = make-state();
        for-each (r in leaf-rules) { // leaf-rules: rules with rhs == leaf
            if (cost(r) < state-cost(state, lhs(r)))
                state-entry-set(state, lhs(r), make-rcp(number(r), cost(r)));
        }
        normalize-cost(state);
        closure(state);
        n = state-install(state);
        append(worklist, n);
    }
}


project(op,st) {
    pstate = make-state();
    for-each (nt in nonterminals) {
        if (find-rule(op,nt))  // rule with rhs = op(nt)
            state-cost-set(pstate, nt, state-cost(state,nt));

    }
    normalize-costs(pstate);
    return(pstate);
}


closure(state, changes) {
    if (changes) {
        changes = #f;
        for-each (r in chain-rules) {
            cost = cost(r) + state-cost(state, rhs(r));
            if (cost < state-cost(state, lhs(r))) {
                state-entry-set(state, lhs(r), make-rcp(number(r), cost));
                changes = #t;
            }
        }
        closure(state, changes);
    }
}
```