A Language for Writing Code Generators

Christopher W. Fraser
AT&T Bell Laboratories
Murray Hill, NJ 07974

Introduction

This paper describes a programming language for writing code generators. The language abbreviates repetitive constructs, simplifies encoding, and assumes responsibility for making the code generator small and fast. As a result, a specification for the VAX takes 126 lines, one for the Motorola 68020 takes 156, and one for the MIPS R3000 takes 75.

Each specification is compiled into a fast, monolithic C program that accepts dags (directed acyclic graphs) annotated with intermediate code, and generates, optimizes, and emits code for the target machine. The code generators are used with a front end for ANSI C. The resulting compilers emit code similar to `pcc1`'s, but they run about twice as fast. The compilers are in use by small research groups at Bell Labs and Princeton University and by classes at Princeton.

The technique described here stands in sharp contrast to recent methods for retargetable code generation, including the author's:

> Most recent systems accept non-procedural machine descriptions and produce tables for a compile-time interpreter. The current system accepts a compact representation of a program and emits a hard-coded code generator. The current system's specifications have a modest procedural aspect, but they are smaller than the specifications required by most high-tech code generators.
>
> Most recent systems use sophisticated techniques to generate their tables, but the current system uses a preprocessor whose operation is largely transparent. It's like parsers: one can look at a recursive descent parser and "see" the grammar behind it, but it is difficult to see any meaningful patterns in an LR table. Transparent operation is not always important, but it helps when things go wrong.
>
> Most recent systems rely on general-purpose algorithms with applications beyond just code generation: Graham-Glanville systems [2, 8] rely on LR parsing, Twig and BURS systems [1, 10] rely on recent advances in pattern matching on trees [3, 9] and systems based on retargetable peephole optimizers [4] rely on symbolic simulation. In contrast, the technique underlying the current system suits code generation and little else.

This project grew out of experience with a system that tracked the operation of a high-tech peephole optimizer and generated a hard-coded code generator from the trace [7]. The current system generates similar code generators directly from a compact document that captures their entropy.

Representation

Programs in the code generation language consist principally of simple rewriting rules. Some rules rewrite intermediate code as naive assembly code. Others peephole-optimize the result.

Currently, the front end and the rule language denote operators in the intermediate code by short strings like `ADDI`, which adds integers. The `ADD` denotes a generic operator, and the `I` denotes the type of the result and, for most operators, the type of the operands as well. The current intermediate code has 43 generic operators, nine type suffixes, and 139 valid combinations thereof. It would be straightforward to adapt the rule language and compiler to a different intermediate code.

The rule language denotes each target machine instruction with an assembler instruction "template".

For correspondence: C. W. Fraser, AT&T Bell Laboratories 2C-464, 600 Mountain Avenue, Murray Hill, NJ, 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0-89791-306-X/89/0006/0238 $1.50

For example, the display below informally represents several VAX instructions:

```
mov{b w l f d} y,z
{add sub mul div}{b w l f d}3 x,y,z
```

A similar sketch might note that the operands include strings like `rn` and `c(rn)`. The rule language formalizes such sketches by replacing variant portions with placeholders of the form `%a`. For example, "`mov%t %y,%z`" denotes a generic move instruction and "`%f%t3 %x,%y,%z`" denotes a generic three-operand arithmetic instruction. `%t` denotes a type suffix (`b`, `w`, `f`, `l`, `d`), `%f` a binary operator (`add`, `sub`, etc), and `%x`, `%y`, and `%z` operand templates. Operand templates are represented similarly. For example, a register operand is represented with the string `r%n`, and a displacement-mode operand with `%c(r%n)`.

The code generators accept dags annotated with intermediate code. The portion of the node structure relevant here is

```
struct node {
  int op;
  int count;
  struct node *kids[MAXKIDS];
  struct symbol *syms[MAXSYMS];
  int ints[MAXINTS];
}
```

`op` identifies the node's operation; the rule language allows the retargeter to treat this field as a string like `ADDI` or "`mov%t %y,%z`", but for efficiency the rule compiler collects these strings in a table and henceforth represents them with their integer table index. `ints[0..MAXINTS-1]` play a similar role in encoding values for placeholders like those for the VAX's binary operators, type suffixes and operand templates above. `count` holds the node's reference count. `kids[0..MAXKIDS-1]` point to the dag nodes that develop the values used by the current node. `syms[0..MAXSYMS-1]` point to the entries in the symbol or constant tables that are used by the current instruction.

`MAXKIDS` is a machine-specific constant. The front end needs at most three children per node, but target machines may need more. Once an intermediate code dag has been rewritten as assembly code, a node's children are the instructions that set the registers it reads. After optimization, the VAX compiler uses instructions with up to three operands, each of which may use a base register and an index register, so the VAX code generator runs with `MAXKIDS` set to six.

`MAXSYMS` is also machine-specific. The front end needs at most one symbol per node, for nodes that develop simple addresses and constant values. The VAX compiler, however, may need one symbol for each of its three operands, so the VAX code generator runs with `MAXSYMS` set to three.

`MAXINTS` completes the set of machine-specific array sizes. The code generator may use integers internally to bind some placeholders. The VAX code generator does so for `f`, `t`, and the three operand templates, so it runs with `MAXINTS` set to five. As with the `op` field, the rule language allows the retargeter to treat these fields as if they were strings, but for efficiency the rule compiler collects these strings in a table and henceforth represents them with their integer table index.

Programs in the rule language read, test, and write the fields of such nodes, so these fields comprise the "variables" of the language. By default, the rule language denotes `op` as "`.`", `count` as "`#`", `syms[0..2]` as `S0-S2`, `kids[0..5]` as `K0-K5`, and `ints[0..4]` as `I0-I4`. The retargeter may give them more mnemonic names by preceding the rules with the declarations

`%symnames` *name* ...
`%kidnames` *name* ...
`%intnames` *name* ...

which enumerate, in order, the names to be used. For example, after

```
%symnames yc xc zc
%kidnames yn xn zn yi xi zi
%intnames ym xm zm f t
```

the rules may use `yc` for `S0`, `xc` for `S1`, etc.

Names with a common prefix (like the four that start with `y` above) are like the target-specific structures used in hand-written code generators to represent multi-part operands. For example, in the "structure" `y`, "field" `m` holds the assembler template for the addressing mode, `c` holds the constant part, and `n` and `i` point to the children that develop the base and index registers, respectively. The declarations above impose a machine-specific organization on a machine-independent structure. Confining the machine-specific interpretation to the file of rules simplifies retargeting.

To interpret an assembler template (say, to generate output), placeholders are replaced with the values of the corresponding fields. For example, if `f` denotes `add` and `t` denotes `l` then the template substring `%f%t3` denotes `addl3`. If there is no corresponding field but the placeholder is a prefix of an `intname`, then the placeholder is replaced with the value of that field. For example, none of the declaration lines above defines a field named `x`, but they do define `xm` as an `intname`, so the value of `xm` replaces the placeholder `%x` in "`%f%t3 %x,%y,%z`". When interpreting such a "structure" field, the name of the structure as a whole is used to disambiguate subordinate placeholders. For example, if `xm`

holds $%c, then the %c is replaced with the value of xc, not yc or zc. Placeholders that denote children are replaced with the name of the child's result register. For example, if xm holds r%n, then the %n denotes the name of xn's result register. Registers are assigned after peephole optimization has reduced the demand for registers.

Thus the rule language makes it appear as though the instruction addl3 r6,r7,r8 is represented as a node with the following fields:

```
.  = "%f%t3 %x,%y,%z"
f  = "add"
t  = "l"
xm = "r%n"
xn =  address of node that develops  r6
ym = "r%n"
yn =  address of node that develops  r7
zm = "r%c"
zc = "8"  after register allocation
```

The rule compiler implements this representation by assigning appropriate values to the node's op, ints, kids and syms fields. For instance, if "%f%t3 %x,%y,%z" occupies position 722 in the table of intermediate codes and assembler templates, then the rule compiler implements the first line above by assigning 722 to the op field of the node that represents the addl3. If the string "add" has been recoded as the integer 1, then the rule compiler implements the second line above by assigning 1 to the node's ints[3], which the declarations above allocated to hold f.

Code Generation Rules

The rule language has two basic operators: "==" tests and "=" assigns. Most rules are short, and the operations are collected on one line. The double-column formatting here requires line breaks, so indentation flags material normally combined with the previous line. The following rule generates naive code for integer addition:

```
.=="ADDI"
  .="%f%t3 %x,%y,%z"
  f="add"
  t="l"
  xm="r%n"
  ym="r%n"
  zm="r%c"
  yn=K0
  xn=K1
```

It verifies the presence of the ADDI opcode and then rewrites the node in place by simply assigning the fields as outlined above. Only code generation rules use the Kn notation. They do so to communicate with the front end, which uses the first positions in kids to indicate each node's children.

Rules may be abbreviated by substituting constant strings for their placeholders. Thus the rule above would normally be expressed as

```
.=="ADDI"
  .="addl3 r%n,r%n,r%c"
  yn=K0
  xn=K1
```

The rule compiler is given a list of the valid assembler instruction templates and addressing strings, so it can dismantle the "addl3 r%n,r%n,r%c" above and internally produce the initial, expanded version of this rule.

Comparisons involving intermediate codes may separately test the generic operator and the type suffix, by testing op (not to be confused with the op field in dag nodes) and type. For example, the rule above is equivalent to

```
op=="ADD"
  type=="I"
  .="addl3 r%n,r%n,r%c"
  yn=K0
  xn=K1
```

This feature collaborates with another to offer additional abbreviations. A line of the form

%name old=new ...

defines a set of translations. For example,

```
%ty C=b D=d F=f I=l P=l S=w U=l V=l
```

declares the translation of the intermediate code's type suffixes (C, D, etc, for the C types char, double, etc) to the corresponding VAX type suffixes (b for bytes, d for doubles, etc). The declaration of a translation set does nothing by itself, but when @*name* appears as a comparand in a rule, the rule compiler automatically replicates the rule, once for each pair in the translation set. For example, in

```
op=="ADD"
  type==@ty
  .="add%t3 r%n,r%n,r%c"
  t=@1
  yn=K0
  xn=K1
```

the @ty tells the rule compiler to replicate the rule eight times, once for each pair in the translation set ty. Each *old* value replaces the @ty and the corresponding *new* value replaces the @1. Thus the rule above generates

```
op=="ADD"
  type=="C"
  .="add%t3 r%n,r%n,r%c"
  t="b"
```

```
  yn=K0
  xn=K1
op=="ADD"
  type=="D"
  .="add%t3 r%n,r%n,r%c"
  t="d"
  yn=K0
  xn=K1
...
```

The rule compiler is given a table of the valid opcodes, so it can ignore invalid combinations like `ADDV`, which would add voids.

Rules may expand multiple translation sets. The `@`*digit* is a positional parameter: `@1` refers to the *new* component of the rule's first translation set, `@2` refers to the second, etc. Thus the lines

```
%bin ADD=add BOR=bis BXOR=xor DIV=div
     LSH=ash MOD=mod MUL=mul SUB=sub
%ty C=b D=d F=f I=l P=l S=w U=l V=l
op==@bin
  type==@ty
  .="%f%t3 r%n,r%n,r%c"
  f=@1
  t=@2
  yn=K0
  xn=K1
```

collaborate to enumerate 64 variations on a single rule. 32 of them test valid intermediate opcodes. The rest are automatically discarded. The variations for `ash` and `mod` represent fictitious instructions: the real VAX `ash` opcode does not use the suffix `3`, and there is no `mod` instruction at all. The actual instructions used, however, benefit from the same peephole optimizations that benefit all binary instructions, so it is convenient to temporarily grant `ash` and `mod` first-class citizenship. Later rules will map them onto real instructions just before output. The retargeter is free to use fictitious instructions so long as they are removed before code is emitted.

Special cases may also be handled by preceding a general rule with one tailored to the exceptions. For example, the VAX code generator implements unsigned division and modulus by calling the routines `udiv` and `urem`, so the general rule above is preceded with a special one for these two operations:

```
.=={DIVU=udiv MODU=urem}
  (funcop(a,0,1))
  .="calls $2,%c"
  yc=@1
```

This rule introduces two new syntactic forms. First, the notation {*old*=*new* ...} gives an anonymous, in-line translation set; it is equivalent to `@temp`, where `temp` is a translation set of the given pairs. Second, the notation (*expr*) executes an arbitrary C expression. In the expression, `a` denotes a pointer to the current node. The particular call here removes `kids[0..1]` from the current node and hangs them underneath "argument" nodes; this is necessary because the compiler treats arguments not as children of the call but as separate nodes in the forest of dags. The remaining assignments rewrite the node as a call to the appropriate routine. Such escapes into arbitrary C are rare but provided because it is impractical for any code generation language to anticipate every need.

Naive VAX code generation requires 27 rules and nine translation sets. The compiler can be bootstrapped with just these plus perhaps ten more rules to correct fictitious instructions and perform the most crucial optimizations. Less orthogonal targets require a few more rules. For example, the 68020 uses different instruction templates for floating point and integer arithmetic, so the 68020 rules replace `ty` with two smaller translation sets, which are used by two separate, but similar rules.

Optimization Rules

It is easy to generate naive code, and it has been shown that thorough peephole optimization can yield good instruction selection even when the original code generator is confined to a RISC subset of the target machine [4]. So the code generation rules are augmented with highly factored rules for peephole optimization.

Optimization rules are written in the same language as code generation rules, though the idioms differ somewhat. Code generation rules match intermediate code and yield target code, but optimization rules match target code and yield (better) target code. Most code generation rules examine only one node because naive code generation requires little contextual analysis. Most optimization rules examine two nodes, and rewrite one of them so that it no longer needs the other.

Consider the VAX code fragment:

```
movl 4(r5),r6
movl $1,(r6)
```

If this is the only use of `r6`, the fragment should be replaced with an indirect store:

```
movl $1,*4(r5)
```

An important part of this optimization is the translation of an operand template from `%c(r%n)` to `*%c(r%n)`. This translation uses a translation set:

```
%toInd "%c"="*%c" "%c(r%n)"="*%c(r%n)" ...
```

The redundancy above may be avoided. A translation set's name may be followed by a "replacement pattern",

borrowed from the substitution command of the UNIX text editor **ed**. When the *new* half of the pair is omitted, it is derived automatically by substituting the *old* half for any ampersands in the replacement pattern. For instance, the declaration

```
%toInd/*&/ "%c" "%c(r%n)" ...
```

is equivalent to the longer one above. Any *new* half overrides the replacement pattern. For example, appending `"r%n"="(r%n)"` to the set above records that an indirect register reference is written `(r%n)`, not `*r%n`. For additional abbreviation, the declaration of one translation set may enumerate another. For example, the VAX rules include the translation set

```
%addr FORMAL="%c(ap)"
      GLOBAL="_%c"
      LOCAL="%c(fp)"
```

which supports the translation of the intermediate codes that develop the addresses of simple variables. When enumerated in the definition of another translation set like `toInd`

```
%toInd/*&/ @addr ...
```

the rule compiler discards the old half of the `addr` translations and acts as if the declaration had been

```
%toInd/*&/ "%c(ap)" "_%c" "%c(fp)" ...
```

Thus translation sets are generally built up in stages. Even though an operand template like `%c(fp)` appears in several versions (eg, with and without indirection and indexing), it is usually possible to type it only once, into a basic translation set like `addr`, which is then included as a unit into larger translation sets like `toInd`.

The rule that uses this translation set is

```
.=="mov%t %y,(r%n)"
  zn.=="mov%t %y,%z"
  /#==1
  /ym==@toInd
  zm=@1
  zc=/yc
  zn=/yn
```

The first condition asks if the current instruction is a move with an indirect target. The second asks if the child that prepares the target address is another move. In the rule language, composite variables like `zn.` are formed by concatenating simple variables like `zn` and "."; `zn` denotes a child and "." denotes an opcode, so `zn.` denotes a child's opcode. Simple concatenation would be a liability if complex composite variables were needed, but peephole optimizations don't need them. The third condition above, `/#==1`, asks if the child's reference count is one. Once a condition has examined one field of a child, the pseudo-variable "/" abbreviates the name of that child, so after testing `zn.`, `/#` is equivalent to `zn#`. This shorthand saves little space, but it helps reduce errors. The last condition above, `/ym==@toInd`, asks if the child's source operand template is one of the modes that has an indirect version. (Many don't, like those that already involve indirection.) As with code generation rules, the rule compiler implements `@toInd` by replicating the rule once for each element of the translation set.

The three assignments above are straightforward. The `zm=@1` changes the target operand template to use the translation of the child's source template. The `zc=/yc` and `zn=/yn` hoist up the fields that are used by both the old and new templates; in the example instruction

```
movl 4(r5),r6
```

from the head of this section, the `zc=/yc` pulls up the `4` and the `zn=/yn` pulls up the address of the child that develops `r5`. This last assignment overwrites the last pointer to the child, and thus effectively deletes it.

The rule above does not change the opcode. The current instruction remains a move after the optimization. It simply uses a different target operand template, so the peephole optimization is implemented by changing that template and leaving the instruction template alone.

The Generated Code Generator

Most of the rules are compiled into a monolithic routine called `rewrite`, which accepts a pointer to a dag and rewrites the dag in place with naive and then optimized target code. The retargeter may regard `rewrite` as a long if-then-else chain that implements each expanded rule in order, but the rule compiler makes five transformations that arrange a much faster and much smaller equivalent:

1. When adjacent rules start with the same condition, the rule compiler factors out the common part and tests it only once. The indirection rule above benefits because expanding `toInd` yields many copies of the rule, and they all share several common leading tests.
2. When adjacent rules start by comparing one field with a series of constants, the comparisons are implemented with a C switch. The indirection rule benefits again because, once the rule compiler has stripped the common prefix from the replicated rules, the remaining conditions compare `/ym` with a series of constants.
3. When adjacent cases in a switch perform the same action, the rule compiler arranges for them to share

code. This transformation benefits some of the code generation rules above: they specify the same actions for integer, pointer and unsigned additions, so these three cases label the same code.

4. When transformation 3 yields a `switch(x)` with multiple case labels but only one action, the rule compiler replaces it with

   `if (t[x])` *action*

   It arranges for array `t` to record which values of x require the action and which fell through the old switch. The rule compiler knows the range of x, which is generally small enough that the table can cover the range and eliminate the switch's implicit range check, so the resulting program is faster. This transformation benefits rules that ask if an addressing template is in a certain class, but then perform a common action for all members of the class.

5. When all actions in a `switch(x)` differ by only a single constant in a common position, the rule compiler replaces it with

   `if (s=t[x])` *action*

   where the action has been edited to use `s` instead of the constant. Note that the "=" above denotes assignment, not an equality test. This transformation benefits the indirection rule, the replicated copies of which differ only by the value tested in `/ym==@toInd` and assigned in `zm=@1`, so they can be implemented by testing and assigning a value from a table.

A portion of the resulting `rewrite` appears below.

```
rewrite(register struct node *a) {
  register struct node *b;
  switch (a->op) {
  ...
  case 309: L309:   /* ADDI */
  case 310: L310:   /* ADDU */
  case 311: L311:   /* ADDP */
    setreg(a, sregs);
    rewrite(a->kids[0]);
    rewrite(a->kids[1]);
    a->ints[3] = 1;  /* add */
    a->ints[4] = 41;  /* l */
    a->ints[1] = 37;  /* r%n */
    a->ints[0] = 37;  /* r%n */
    a->ints[2] = 43;  /* r%c */
    goto L722;   /* %f%t3 %x,%y,%z */
  ...
  case 720: L720:   /* mov%t %y,%z */
    switch (a->ints[2]) { ... }
    switch (a->ints[0]) { ... }
    a->op = 720;
    break;
  ...
  }
}
```

When `rewrite` is entered, the opcode will denote intermediate code unless the node has been already rewritten as the result of previous references, so the switch usually goes to a case like the one for `ADDI` above, which was generated from the expanded rule for `ADDI`:

```
.=="ADDI"
  .="%f%t3 %x,%y,%z"
  f="add"
  t="l"
  xm="r%n"
  ym="r%n"
  zm="r%c"
  yn=K0
  xn=K1
```

The first three lines are the result of two declarations not previously shown. One states that the default register set for integer operations is called `sregs`, so this case uses the macro `setreg` to record this fact for the table-driven register allocator, which runs after `rewrite` finishes with the dag. The other states that addition is binary, so this case thus includes two recursive calls on `rewrite` to process the node's children.

The next five statements implement the assignments to `f`, `t`, `xm`, `ym`, and `zm`. The rule's assignments `yn=K0` and `xn=K1` are omitted because `yn` and `xn` already occupy `K0` and `K1`; the layout was chosen because these assignments were particularly common, but it may be possible to choose such efficient layouts automatically.

The rule's assignment to "." is implemented by jumping to the case that optimizes the template assigned. The value of `a->op` is read only at the head of `rewrite`, so there is no need to keep it up-to-date until the node is completely optimized and control leaves the switch, via a `break` like the one shown above.

The optimization cases are typified by case 720 above, which improves move instructions. The two subordinate switches examine the integer codes for the operand templates assigned to `zm` and `ym`, which are represented by the values stored in `a->ints[2]` and `a->ints[0]`, respectively. The first of these switches includes the case that implements the indirection rule above, which is reproduced below in a leading comment:

```
/*
.=="mov%t %y,(r%n)"
  zn.=="mov%t %y,%z"
  /#==1
  /ym==@toInd
  zm=@1
  zc=/yc
  zn=/yn
*/
```

```
case 20:   /* (r%n) */
  b = a->kids[2];
  switch (b->op) {
  case 720:   /* mov%t %y,%z */
    if (
    b->count == 1
    && (s=T4[b->ints[0]])
    ) {
      a->ints[2] = s;
      a->syms[2] = b->syms[0];
      a->kids[2] = b->kids[0];
      goto L720;   /* mov%t %y,%z */
    }
    break;
  ...
```

Merely arriving at case 20 above ensures the rule's first condition is met. The second is checked by fetching a pointer to the child and switching on its opcode. The rule compiler uses a switch because there are other rules that combine an indirect store with other children, so there are other cases following case 720 above. Arriving at case 720 above ensures that the rule's second condition is met, and the rule's last two conditions are tested explicitly. The assignment to `s` above asks if the operand template has an indirect version and fetches the index of that version. The other assignments implement those from the rule, and the `goto` jumps to the case that improves the resulting opcode. In this instance, the rule changed an operand, not the opcode, so control returns to the current outermost case label.

The complete optimization case for move instructions is about 300 lines, and it is one of the biggest. Some of the switches have only two or three cases, so the compiler implements them with condition chains. Even so, heavy use of nested `if` statements means that even this large case identifies and makes a typical peephole optimization in perhaps 8 comparisons (two VAX instructions each), 5 assignments (one each), and a jump.

A few rules need to examine register assignments. These rules are segregated from the others by placing them after the directive `%final` in the file of rules. They are compiled into a separate routine called `final` that looks like `rewrite`, but that runs after `rewrite` and the register allocator complete. The rules that correct fictitious instructions are also generally placed in `final`, but none do more than simple one-for-one edits. For instance, the rule

```
.=="ash%t3 %x,%y,%z" .="ashl %x,%y,%z"
```

corrects the over-generalization of `ash` instructions. `final` also implements output, so the rule compiler turns this rule into

```
case 722: L722:   /* %f%t3 %x,%y,%z */
  switch (a->ints[3]) {
  case 5:   /* ash */
    goto L741;   /* ashl %x,%y,%z */
  ...
case 741: L741:   /* ashl %x,%y,%z */
  a->op = 741;
  bp = bp;
  *bp++ = 'a';
  *bp++ = 's';
  *bp++ = 'h';
  *bp++ = 'l';
  *bp++ = ' ';
  bp = emitstruct(bp, a, 1);
  ...
```

The first case recognizes the invalid shift and jumps to the second case, which deposits a valid shift in the output buffer. The rule compiler also writes the routine `emitstruct`, which emits the operand pseudo-structures `x`, `y`, and `z` in much the same way that `final` emits instruction templates above: with a switch and direct deposit into the output buffer. Strings like `ashl` were originally emitted using a loop, but unrolling the loop as shown above added less than 5kb to the compiler and reduced compile times by 5%.

Discussion

The sample rules above demonstrate all principal features of the rule language. The rule compiler is written in the Icon programming language and takes about 1000 lines. The VAX specification currently takes 126 lines: 22 for translation sets, 27 for the rules that implement naive code generation, 43 for `rewrite`'s optimization rules, 16 for `final`'s, and the rest for various declarations.

The rule compiler turns this specification into a 4000-line C program, which compiles into less than 17kb of code and 8kb of data. `final` and `emitop` take about 5kb; `rewrite` takes about 11kb. They are compiled with about 8000 lines of machine-independent code, mostly for the front end, and 500 lines of system-specific emitters for data definition, function prologues, and other items that require no instruction selection. Almost half of the system-specific code emits symbol tables for the debugger.

A simple experiment was run to demonstrate compilation rates and code quality. The compiler was compiled five different ways: with itself (`lcc`), with the standard 4.3bsd C compiler (`cc` with and without `-O`), and with the GNU C compiler (`gcc` with and without `-O`). The compile times appear below, with the sizes of the resulting code segments. These times include preprocessing, assembly, and linking.

compiler	*user time*	*system time*	*size*
`cc`	64.9	33.5	100k
`cc -O`	87.1	41.9	90k
`gcc`	63.9	37.7	104k
`gcc -O`	100.7	48.2	83k
`lcc`	38.9	26.2	100k

`lcc` compiled the fastest. Its object code was not the smallest, but it was quite fast. The resulting binaries were each used five times to compile the largest front end module, which is almost 1000 lines before preprocessing and about 1600 afterwards. The mean times appear below.

code from	*user time*	*system time*
`cc`	2.36	0.86
`cc -O`	2.22	0.88
`gcc`	2.28	0.98
`gcc -O`	1.78	0.74
`lcc`	1.80	0.78

Thus `gcc -O` produced the fastest code for `lcc`, but it implements many more global optimizations than `lcc`. `lcc`'s code for itself was a close second.

The first set of figures above includes preprocessing, assembly, and linking. To isolate the performance of the compiler, the large compiler module was preprocessed and then run through `cc`, `gcc`, and `lcc` but not the assembler or linker:

compiler	*user time*	*system time*
`cc`	4.66	1.70
`gcc`	4.10	1.86
`lcc`	1.80	0.78

The timings for `cc` and `gcc` omit `-O`, which would make them slower still. Different benchmarks yield different results, but most show that `lcc` compiles fast and emits competitive code. The compiler spends about 20% of its time generating, optimizing, and emitting code, so the speed of the front end contributes importantly to the compilation rate. The front end also performs machine-independent analysis and transformations that contribute to the quality of the emitted code.

On-going work seeks new target machines and more efficient instruction encodings. Also, the rules could be simpler [5, 6]. With their `vpcc` compiler, Davidson and Whalley have shown that successful retargets can be based on a few very simple rewriting rules, though their technique currently requires compile-time string matching [6]. On-going work seeks a code generator as fast as `rewrite` from rules as simple as `vpcc`'s.

Acknowledgments

Dave Hanson wrote the front end and many of the VAX-specific emitters. He also simplified the job of the back end with countless edits to the interface and was a constant source of constructive criticism during the development of the rule language and its compiler. David Gay, Eleftherios Koutsofios, Kriton Kyrimis and Howard Trickey cheerfully isolated many bugs for us.

References

1. A. V. Aho and M. Ganapathi, Efficient Tree Pattern Matching: An Aid to Code Generation, *Conf. Rec. 12th ACM Symp. on Prin. of Programming Languages*, Jan. 1985, 334–340.

2. P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick, and E. Pelegri-Llopart, Experience with a Graham-Glanville Code Generator, *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984, 13–24.

3. D. R. Chase, An Improvement to Bottom-up Tree Pattern Matching, *Conf. Rec. 14th ACM Symp. on Prin. of Programming Languages*, Jan. 1987, 168–177.

4. J. W. Davidson and C. W. Fraser, Code Selection Through Object Code Optimization, *ACM Trans. Prog. Lang. and Systems* **6**, 4 (Oct. 1984) 505–526.

5. J. W. Davidson and C. W. Fraser, Automatic Inference and Fast Interpretation of Peephole Optimization Rules, *Software—Practice & Experience* **17**, 11 (Nov. 1987) 801–812.

6. J. W. Davidson and D. B. Whalley, Quick Compilers Using Peephole Optimization, *Software—Practice & Experience* **19**, 1 (Jan. 1989) 79–97.

7. C. W. Fraser and A. L. Wendt, Automatic Generation of Fast Optimizing Code Generators, *Proceedings of the SIGPLAN '88 Symposium on Compiler Construction, SIGPLAN Notices* **23**, 7 (July 1988) 79–84.

8. M. Ganapathi and C. N. Fischer, Affix Grammar Driven Code Generation, *ACM Trans. Prog. Lang. and Systems* **7**, 4 (Oct. 1985) 560–599.

9. C. Hoffmann and M. J. O'Donnell, Pattern matching in trees, *J. ACM* **29**, 1 (Jan. 1982) 68–95.

10. E. Pelegri-Llopart and S. L. Graham, Optimal Code Generation for Expression Trees: An Application of BURS Theory, *Conf. Rec. 15th ACM Symp. on Prin. of Programming Languages*, Jan. 1988, 294–308.