SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(9), 963–988 (SEPTEMBER 1991)

A Code Generation Interface for ANSI C

CHRISTOPHER W. FRASER
AT&T Bell Laboratories, 600 Mountain Avenue 2C-464, Murray Hill, NJ 07974 U.S.A.

AND

DAVID R. HANSON
*Department of Computer Science, Princeton University,
Princeton, NJ 08544 U.S.A.*

SUMMARY

`lcc` *is a retargetable, production compiler for ANSI C; it has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000, and some versions have been in use for over a year and a half. It is smaller and faster than generally available alternatives, and its local code is comparable. This paper describes the interface between the target-independent front end and the target-dependent back ends. The interface consists of shared data structures, a few functions, and a dag language. While this approach couples the front and back ends tightly, it results in efficient, compact compilers. The interface is illustrated by detailing a code generator that emits naive VAX code.*

KEY WORDS intermediate languages code generation compilers ANSI C

1. INTRODUCTION

`lcc` is a retargetable compiler for ANSI C.[1] It has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000. Its local code is comparable with that from other generally available C compilers, but it runs up to twice as fast and is about half the size. `lcc` is in production use at Princeton University and AT&T Bell Laboratories.

This paper describes the interface between the target-independent front end and the target-dependent back ends. Good code-generation interfaces are hard to design. An inadequate interface may force each back end to do work that could otherwise be done once in the front end. Annotating frequently referenced variables for assignment to registers is an example. If the interface is too small, it may not encode enough information to exploit new machines thoroughly. If the interface is too large, the back ends may be needlessly complicated. These competing demands require careful engineering of details, and re-engineering as new targets expose flaws.

This paper reports the results of such experience. The detailed engineering is best presented by showing how is it used in an implementation. The interface is illustrated, there-

0000--0000/00/000000--00$00.00 *Received 00 Month 1900*
Revised 00 Month 1900

fore, by most of a *sample* code generator for the VAX. This code generator emits naive code; i.e., it uses only the ‘RISC subset’ of the VAX instruction set. It is nevertheless complete: when used with a conforming preprocessor and library, the compiler with this code generator passes the conformance section of Version 2.00 of the Plum Hall Validation Suite for ANSI C. The production versions of `lcc` use the interface described here, but use back ends in which instruction selection and optimization are generated automatically from a compact specification.[2] The presentation does omit some trivial and repetitive constructs. Reference 3 gives the complete sample.[*]

The interface consists of a few shared data structures, 18 functions, most of which are very simple, and a 36-operator dag language, which encodes the executable code from a source program. Table I summarizes the interface functions. The dag language corresponds to the ‘intermediate language’ used in other compilers, but it is smaller than typical intermediate languages.

Table I. Interface Functions

Section	Interface Function
5.1	`void address(Symbol p, Symbol q, int n)`
5.2	`void blockbeg(Env *e)`
5.2	`void blockend(Env *e)`
4	`void defaddress(Symbol p)`
4	`void defconst(int ty, Value v)`
4	`void defstring(int len, char *s)`
4	`void defsymbol(Symbol p)`
6.2	`void emit(Node p)`
4	`void export(Symbol p)`
5	`void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls)`
6.1	`Node gen(Node p)`
4	`void global(Symbol p)`
4	`void import(Symbol p)`
5.1	`void local(Symbol p)`
3	`void progbeg(int argc, char *argv[])`
3	`void progend(void)`
4	`void segment(int s)`
4	`void space(int n)`

[*]The `lcc` front end and the sample code generator are available for anonymous `ftp` from `princeton.edu`. The file `README` in the directory `pub/lcc` gives details.

2. BACKGROUND

`lcc`'s interface consists of shared data structures and functions. The front and back ends are tightly coupled in a single program. They are clients of one another. The front end calls on the back end to generate and emit code. The back end calls on the front end to perform output, allocate storage, interrogate types, and manage dags, symbols, and strings.

`lcc`'s interface resembles others that pass structures, like trees, directly to the code generator.[4] This type of interface differs from abstract machines like EM,[5] P-code,[6] or U-code.[7] Abstract machine code resembles assembly or machine language for a fictitious computer.[8] A front end emits a stream of instructions (in a text or compressed binary encoding) to a logically separate back end.

Each approach has strengths and weaknesses. Abstract machine codes permit the front and back ends, and perhaps an optimizer, to run as separate processes. Uni-process compilers are generally faster, but multi-process compilers might run faster on some multi-processor computers. If the compiler is complex, a multi-process compiler might simplify team development.

Abstract machine code captures all required data in a single, monolithic encoding. `lcc`'s shared data structures include interface data in the same structure with data that is private to the front and back ends. This problem, which cannot occur when front and back ends are isolated, is managed by conventions that keep the front and back ends from interfering.

With abstract machines, a front end recodes its data structures into abstract machine code. At a minimum, the back end reads and decodes the abstract machine code. It may also build its own structures. `lcc`'s shared structure avoids this overhead altogether. Its interface involves no encoding, decoding, or, for multi-process compilers, I/O.

These differences can be reduced and even eliminated. For example, EM emitters have been modified to generate machine code directly,[9] trading code quality for faster compilation. And one of the back ends for `lcc` generates quadruples, which form an abstract machine. The production `lcc` compilers, however, emit *final* assembly code *faster* than this back end emits quadruples.

Some compilers use register transfers as an intermediate representation.[10, 11] Register transfers are most useful as a machine-independent representation of machine-specific instructions. In particular, they are invaluable in machine-independent peephole optimizers. `lcc`'s code generators could use register transfers internally (an early one did), but the interface described here is at a higher level, before the introduction of any target-specific data, except for the size and alignment of the basic datatypes, which are required by the front end to compute field offsets and structure sizes. [12] The interface described here is closer to the syntax trees used in References 10 and 11 *before* register transfers are introduced.

3. CONFIGURATION

Target-specific configuration parameters specify the widths and alignments of the basic datatypes and optionally define conditional compilation flags. They are defined in a 'header' file, `config.h`, which is included when a target-specific `lcc` is compiled.

The sample type metrics and conditional compilation flags are defined as follows. Target-specific fields of the shared data structures are defined in Sections 4 and 6.

```
#define VAX

/* type metrics: size,alignment,constants */
#define CHAR_METRICS     1,1,0
#define SHORT_METRICS    2,2,0
#define INT_METRICS      4,4,0
#define FLOAT_METRICS    4,4,1
#define DOUBLE_METRICS   8,4,1
#define POINTER_METRICS  4,4,0
#define STRUCT_ALIGN     1

#define LEFT_TO_RIGHT    /* evaluate args left-to-right */
#define LITTLE_ENDIAN    /* right-to-left bit fields */
#define JUMP_ON_RETURN  0
```

Type metrics give the size and alignment of the type in bytes and a flag. If the flag is `1` constants of that type cannot appear in instructions; the front end generates variables to hold them.

Both the size and alignment for characters must be `1`. To simplify the back end, unsigned and long integers are assumed to have metrics identical to integer, and long doubles are assumed to have metrics identical to double. The front end correctly treats all of these types as distinct, however. `POINTER_METRICS` apply to all pointer types, and pointers must fit in unsigned integers. The alignment of a structure is the maximum of the alignments of its fields and `STRUCT_ALIGN`; it is needed to conform with the layouts required by some target systems.

If `JUMP_ON_RETURN` is non-zero, the front end generates a jump to a generated label for each `return` statement and defines this label at the end of each function where the return sequence must be emitted. Similar action is taken if the compiler is asked to generate data for a debugger, because some debuggers assume a common exit point. Since the VAX does returns with one instruction, its `JUMP_ON_RETURN` is `0`.

By default, the front end generates code that evaluates function arguments from right to left; defining `LEFT_TO_RIGHT` yields the opposite order. A sequence of bit fields is laid out from left to right in one or more unsigned values; defining `LITTLE_ENDIAN` yields a right-to-left layout. The standard permits either argument evaluation or bit-field layout order.

The front end contains a few target-specific operations. These are protected by conditional compilation on `VAX`, `MIPS`, `MC`, etc. Thus, `VAX` is defined above. During initialization, the front end calls `progbeg` with those program arguments that it does not recognize, e.g., target-specific options. At the end of compilation, the front end calls `progend` so the back end can finalize its output.

4. SYMBOLS

The front and back ends use symbol table entries to represent variables, constants, and labels. They are represented by pointers to the following structure.

```
typedef struct symbol *Symbol;
struct symbol {        /* symbol table entries: */
   Xsymbol x;             /* back end’s type extension */
   char *name;            /* name */
   unsigned char scope;   /* scope level */
   unsigned char class;   /* storage class */
   unsigned defined:1;    /* 1 if defined */
   unsigned temporary:1;  /* 1 if a temporary */
   unsigned generated:1;  /* 1 if a generated identifier */
   unsigned addressed:1;  /* 1 if its address is taken */
   Type type;             /* data type */
   union {
      int label;          /* labels: label value */
      struct {            /* constants: */
         Value v;            /* value */
         Symbol loc;         /* out-of-line location */
      } c;
      int seg;            /* globals statics: def segment */
   } u;
};
```

The `scope`, `class`, and `type` fields give the symbol’s scope level, its storage class, and its type, respectively. Most of the 1-bit fields flag self-explanatory attributes for each symbol; fields relevant only to the front end are elided above.

Scope values classify symbols as constants, labels, or variables. For labels, constants, and some variables, a field of the union `u` supplies additional data.

Labels have a `scope` equal to the enumeration constant `LABELS`, and `u.label` is the numeric value of the label. The `name` of a label is the string representation of `u.label`. Labels have no `type` or `class`.

Constants have a scope equal to `CONSTANTS`, a `class` equal to `STATIC`, and a `name` equal to the string representation of the constant. The actual value of the constant is stored in the `u.c.v` field, which is defined by

```
typedef union value {   /* constant values: */
   char sc;                /* SIGNED/plain char */
   unsigned char uc;       /* UNSIGNED CHAR */
   short ss;               /* SIGNED SHORT */
   unsigned short us;      /* UNSIGNED SHORT */
   int i;                  /* INT */
```

```
        unsigned int u;         /* UNSIGNED INT */
        float f;                /* FLOAT */
        double d;               /* DOUBLE */
        void *p;                /* POINTER to anything */
    } Value;
```

If a variable is generated to hold the constant, u.c.loc points to the symbol table entry for that variable.

Variables have a scope equal to GLOBAL, PARAM, or LOCAL$+k$ for nesting level k. class is STATIC, AUTO, EXTERN, or REGISTER. The name of most variables is the name used in the source code. For temporaries and other generated variables, name is a digit sequence. temporary and generated are set for temporaries, and generated is set for other generated variables, e.g., those that hold constants. For global and static variables, u.seg gives the logical segment in which the variable is defined.

The type field for constants and variables points to a structure that describes types. The size and align fields of this structure give, respectively, the size and alignment constraints of the type in bytes.

The x field is an ‘extension’ in which the back end stores target-specific data for the symbol. The sample requires only one field:

```
    typedef struct {
        char *name;    /* name for back end */
        int offset;    /* frame offset */
    } Xsymbol;
```

p->name identifies the symbol to the front end, but the back end may need to emit a different ‘name’. For example, the ‘name’ for locals is an offset from a frame pointer. p->x.name is the back end’s name for the symbol.

Whenever the front end defines a new symbol for a constant, label, global, or static, it calls defsymbol to give the back end an opportunity to initialize its Xsymbol fields. It is often defined as a macro, e.g., for the sample defsymbol assigns the name field to the x.name field. For parameters and locals, the Xsymbol fields are initialized by function and local, respectively, and symbols that represent address expressions are initialized by address.

global emits code to define a global variable. The front end will follow the call to global with any appropriate calls to the data initialization functions. global handles the necessary alignment adjustments and the actual definition. The sample definition for global y is simply _y: preceded by an alignment directive, if necessary:

```
    void global(Symbol p) {
        switch (p->type->align) {
        case 2: print(".align 1; "); break;
        case 4: print(".align 2; "); break;
        case 8: print(".align 3; "); break;
        }
        print("%s:", p->x.name);
     }
```

The underscore could be prefixed to y’s x.name when it is initialized instead of being included in the back end’s output formats. print is a front-end function similar to the standard printf.

defconst(ty, v) emits the scalar v. ty is one of the values listed in the first seven lines of Table II and indicates which field of v is to be emitted. These codes and the codes B and V are also used for the dag operators, which are described in Section 6.

Table II. Type Suffixes

ty	v field	type
C	v.uc	character
S	v.us	short
I	v.i	int
U	v.u	unsigned
P	v.p	any pointer type
F	v.f	float
D	v.d	double
B		structure or block
V		void

The sample defconst is

```
void defconst(int ty, Value v) {
    switch (ty) {
    case C: print(".byte %d\n",   v.uc); break;
    case S: print(".word %d\n",   v.us); break;
    case I: print(".long %d\n",   v.i ); break;
    case U: print(".long 0x%x\n", v.u ); break;
    case P: print(".long 0x%x\n", v.p ); break;
    case F:
        print(".long 0x%x\n", ((unsigned *) &v.f)[0]);
        break;
    case D:
        print(".long 0x%x,0x%x\n", ((unsigned *) &v.d)[0],
            ((unsigned *) &v.d)[1]);
        break;
    }
}
```

In the production compilers, `defconst` accommodates cross-compilation, correcting for different representations and byte orders.

`defconst` cannot leave the encoding of floating-point constants to the assembler, because the assembler cannot cope with the effect of casts; few assemblers can. For example, the correct initialization for

```
double x = (float)0.3;
```

is `.long 0x999a3f99,0x0`. The more natural directive `.double 0.3` is wrong; it initializes `x` to the equivalent of

```
.long 0x99993f99,0x0999a9999
```

because it cannot represent the effect of the cast.

`defstring(len, s)` emits code to initialize a string of length `len` to the characters in `s`. The front end converts escape sequences, like `\n`, into the corresponding ASCII characters. `defaddress(p)` emits the address denoted by `p` and `space(n)` emits code to allocate `n` zero bytes.

Modules import identifiers defined elsewhere, and they export identifiers for use elsewhere. The front end identifies an exported or imported symbol by calling `export` or `import`.

The front end announces a segment change by calling `segment(s)` where `s` is one of `CODE`, `BSS`, `DATA`, and `LIT`. The front end emits executable code into the `CODE` segment, uninitialized variables into the `BSS` segment, initialized variables into the `DATA` segment, and constants into the `LIT` segment. `segment` maps the logical segments onto the segments provided by the target machine. The `CODE` and `LIT` segments can be mapped to read-only segments; the others must be mapped to read/write segments.

5. FUNCTIONS

The front end completely consumes each function before passing any part of the function to the back end. This organization permits certain optimizations. For example, only by processing complete functions can the front end identify the locals and parameters whose address is not taken; only these variables may be assigned to registers.

The front end accumulates functions into private data structures. At the end of each function, it calls `function` to generate and emit code. The typical form of `function` is

```
void function(Symbol f, Symbol caller[],
   Symbol callee[], int ncalls) {
   ...initialize
   gencode(caller, callee);
   ...emit prologue
   emitcode();
   ...emit epilogue
}
```

gencode is a front-end routine that traverses the front end's private structures and passes each dag to the back end's gen (see Section 6.1), which selects code, allocates registers, annotates the dag to record its choices, and returns a dag pointer. emitcode is a front-end routine that traverses the private structures again and passes each of the pointers from gen to emit (see Section 6.2), which emits the code.

This organization offers the back end flexibility in generating function prologue and epilogue code. Before calling gencode, function initializes the Xsymbol fields of the function's parameters, as described below, and does other per-function initializations, if necessary. After calling gencode, the size of the activation record, or *frame*, the number of registers used, etc. are known; this information is usually needed to emit the prologue. After calling emitcode to emit the code for the body of the function, function emits the epilogue.

The argument f to function is the pointer to the symbol table entry for the current function, and ncalls is the number of calls it makes. ncalls is useful on targets like the SPARC where 'leaf' functions get special treatment.

caller and callee are zero-terminated arrays of pointers to symbol table entries. The symbols in caller are the function parameters as passed by a caller; those in callee are the parameters as seen within the function. For most functions, the symbols in each array are the same, but they can differ in both class and type. For example, in

```
    foo(x) float x; { ... }
```

the ANSI standard specifies that actual arguments to foo be passed as doubles, but within foo, x is a float. Thus, caller[0]->type is double and callee[0]->type is float. And in

```
    int strlen(register char *s) { ... }
```

caller[0]->class is AUTO and callee[0]->class is REGISTER. Even without register declarations, the front end assigns REGISTER class to frequently referenced parameters and sets callee's class accordingly. This assignment is made only when there are no explicit register *locals* to avoid interfering with the programmer's intentions.

caller and callee are passed to gencode. If caller[i]->type is not equal to callee[i]->type or if caller[i]->class is not equal to callee[i]->class, then gencode generates an assignment of caller[i] to callee[i]. If the types are not equal, this assignment may include a conversion; for example, the assignment to x in foo truncates a double to a float. For REGISTER parameters, function must assign a register and initialize the x.name accordingly, or change the callee's class to AUTO.

function could also change callee[i]->class from AUTO to REGISTER if it wished to assign a register to that parameter. On the MIPS, for example, some of the parameters are passed in registers, so function assigns those registers to the corresponding callees in leaf functions. If, however, callee[i]->addressed is set, the address of the parameter is taken in the function body, and it must be stored in memory on most machines.

Initialization of the Xsymbol fields of the symbols in caller and callee depends on the frame layout, which is target specific. Figure 1 shows the layout of the VAX frame;

following the standard reference,[13] the stack grows towards lower addresses and towards the top of the page.

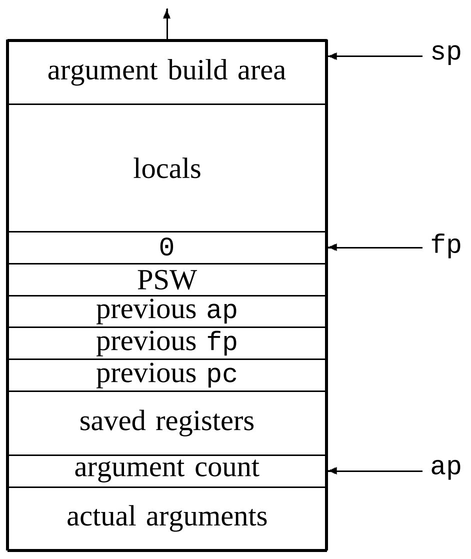

Figure 1. VAX Frame Layout

Arguments are referenced by displacement-mode addressing with positive offsets from register `ap`, so the first argument is at address `4(ap)`. Locals are referenced via negative offsets from `fp`, e.g., the first local is at `-4(fp)`. The ‘argument build area’ is used to store arguments to functions that are called by the current function. The front end ‘un-nests’ calls as required by back ends for targets that pass some arguments in registers. The argument build area can thus be used for all calls and must be large enough to hold the largest argument list. When a function is called, the caller’s argument build area becomes the callee’s ‘actual arguments’.

Typical VAX calling sequences can handle nested calls, so using an argument build area is not strictly necessary. But other targets, such as the MIPS, require this approach, so it is used here to illustrate the technique. This approach also has the advantage that stack overflow can occur only at function entry, which is useful on targets that require explicit prologue code to detect stack overflow.

The sample version of `function` is

```
static int framesize;     /* size of activation record */
static int offset;        /* current frame offset */
static int argbuildsize;  /* size of argument build area */

void function(Symbol f, Symbol caller[],
   Symbol callee[], int ncalls) {
   int i;
```

```
        offset = 4;
        for (i = 0; caller[i] && callee[i]; i++) {
           offset = roundup(offset, caller[i]->type->align);
           callee[i]->x.offset = caller[i]->x.offset = offset;
           callee[i]->x.name = caller[i]->x.name = stringf("%d(ap)", offset);
           offset += caller[i]->type->size;
           callee[i]->class = AUTO;
        }
        usedmask = argbuildsize = framesize = offset = 0;
        gencode(caller, callee);
        print("%s:.word 0x%x\n", f->x.name, usedmask&0 x3f);
        framesize += 4*nregs + argbuildsize;
        print("subl2 $%d,sp\n", framesize);
        if (isstruct(freturn(f->type)))
           print("movl r1,-4(fp)\n");
        emitcode();
        if (glevel > 1)
           print("ret\n");
     }
```

The VAX `calls` instruction saves the general registers specified by the entry mask, `ap`, `fp`, and the return address, `pc`, as shown in the frame figure above. `function` computes the size of the locals and argument build area, given by `framesize` and `argbuildsize`, respectively.

The first part of `function` initializes the `x.name` field of each `caller` and `callee` symbol to the appropriate offset. The running `offset` is rounded up to the alignment for each argument. `roundup(n,m)` is a front-end macro that returns `n` rounded up to the next multiple of `m`. Depending on the type metrics, the size of an argument may not be a multiple of longwords (e.g., 3-byte structures), but the front end ensures that the minimum alignment for each `caller[i]` is that for integers, which keeps the VAX stack longword-aligned. `stringd(n)` is a front-end function that returns the string representation of the integer `n`. The appropriate base register, `ap` or `fp`, appears in instruction templates instead of appearing in `x.name`.

This version of `function` does not support register declarations, so each `callee`'s storage `class` is set to `AUTO`.

During code generation, `argbuildsize` is increased as code for each argument is generated (see Section 6.1), `offset` is adjusted in response to the definition of locals and block boundaries, and `framesize` records `offset`'s maximum. Before calling `gencode`, `function` clears `usedmask`, which records the registers used in the function body, and clears `argbuildsize`, `framesize`, and `offset`.

After `gencode` returns, `usedmask` and `framesize` hold the information needed to generate the prologue. `framesize` is adjusted to include the argument build area and space for saving all of the registers. This space, which is in addition to that specified by the register save mask, is used to save registers across those instructions that destroy fixed registers, like the character copy instruction `movc3`. The `subl2` instruction allocates the remainder of the frame.

The last instruction of the prologue is emitted only for functions that return structures. For a function type `ty`, `freturn(ty)` gives the type of the value returned by the function, and `isstruct(ty)` is true if `ty` is a structure or union type. Section 6 explains returning structures.

5.1 Locals

During the execution of `gencode`, the front end calls `local` to announce locals and temporaries. `local` must initialize `p`'s `Xsymbol` fields. That is, it must set `p->x.name` to a stack offset or register number, depending on the availability of registers and on `p`'s `type` and `class`.

For each block, the front end first announces locals with explicit register declarations, in order of declaration, to permit programmer control of register assignment. Then it announces the rest, starting with those that appear to be most frequently referenced. It assigns `REGISTER` class to even these locals *if* their address is never taken and if their estimated frequency of use exceeds two, which approximates the cost of saving and restoring a register. This announcement order and `class` override collaborate to put the most promising locals in registers even if no registers were declared. As with parameters, `local` could assign a register to `p` and change `p->class` from `AUTO` to `REGISTER`, but it should do so only if `p->addressed` is not set.

If `p->class` is `REGISTER`, `local` can decline to allocate a register by setting `p->class` to `AUTO` and initializing `p->x.name` to the appropriate frame address. This choice is illustrated by the sample version:

```
void local(Symbol p) {
   offset = roundup(offset + p->type->size, p->type->align);
   offset = roundup(offset, 4);
   p->x.offset = -offset;
   p->x.name = stringf("%d(fp)", -offset);
   p->class = AUTO;
}
```

The second `roundup` keeps `offset` and hence the stack aligned on longwords.

The front end folds constant expressions, but it cannot fold those with symbolic operands completely. These are announced by calling `address(q, p, n)` to initialize `q`'s `Xsymbol` fields to represent an address of the form $x + n$, where x is the address represented by `p`. The sample `address` is

```
void address(Symbol q, Symbol p, int n) {
   if (p->scope == GLOBAL || p->class == STATIC || p->class == EXTERN)
```

```
         q->x.name = stringf("%s%s%d", p->x.name, n >= 0 ? "+" : "", n);
      else {
         q->x.offset = p->x.offset + n;
         q->x.name = stringf("%d(%s)", q->x.offset,
            p->scope == PARAM ? "ap" : "fp");
      }
   }
```

which sets `q->x.name` to `p->x.name` concatenated with `+n` or `-n`. `stringf` returns a pointer to a string formatted as specified by its `print`-style arguments. For example, in

```
struct node { struct node *link; int count; } a;
f() { int b[10]; b[4] = a.count; ... }
```

suppose `a` and `b` point to the symbols for `a` and `b`, respectively. `a->x.name` is set to `"a"` by `defsymbol`, and `b->x.name` is set to `"-40"` by `local`. `address(q1,a,4)` is called with `q1` representing the address of `a.count`, and `q1->x.name` is set to `"a+4"`. Likewise, `address(q2,b,16)` sets `q2->x.name` to `"-40+16"`, which is the address of `b[4]`.

5.2 Compound Statements

Source-language blocks bracket the lifetime of locals. `gencode` announces the beginning and end of a block by calling `blockbeg(e)` and `blockend(e)`, respectively. The argument `e` points to a target-specific `Env` structure, which holds the data necessary to reuse registers and frame space associated with the block. The sample `Env` is

```
typedef struct {
   unsigned rmask;
   int offset;
} Env;
```

The fields save the values of `rmask` and `offset` at the beginning of a block so that they can be restored on the end of the block. The sample `blockbeg` and `blockend` are thus

```
void blockbeg(Env *e) {
   e->rmask = rmask;
   e->offset = offset;
}

void blockend(Env *e) {
   if (offset > framesize)
      framesize = offset;
   offset = e->offset;
   rmask = e->rmask;
}
```

blockend also updates framesize if the locals for the current block require more space than previous blocks. The sample could do without the rmask field, but if its local assigned registers to locals, it would need the field to release those registers.

Temporaries --- locals with temporary set --- to which local assigned registers live only for the expressions in which they are used. They are announced by local as usual, but are used only in the dags passed to next call on gen (see Section 6.1). gen can thus release all registers assigned to temporaries.

6. DAGS

Executable code is specified by dags. A function body is a sequence of forests of dags, each of which is passed to the back end via gen. Dag nodes, or simply nodes, are defined by

```
    typedef struct node *Node;
```

The kids point to the operand nodes. Some operators also take symbol table pointers as operands; these appear in the syms array. The default and minimum allowable value for both MAXKIDS and MAXSYMS is 2; larger values can be defined for the back end's convenience in the configuration file. count holds the number of references to this node from kids in other nodes. link points to the root of the next dag in the forest.

The x field is the back end's 'extension' to nodes. The configuration defines the type Xnode to hold the per-node data that the back end needs to generate code. The sample Xnode is

```
    typedef struct {
       unsigned visited:1; /* 1 if dag has been linearized */
       int reg;            /* register number */
       unsigned rmask;     /* unshifted register mask */
       unsigned busy;      /* busy regs */
       int argoffset;      /* ARG: argument offset */
       Node next;          /* next node on emit list */
    } Xnode;
```

Section 6.1 describes the fields.

The op field holds an operator. The last character of each is a type suffix from Table II. For example, the generic operator ADD has the variants ADDI, ADDU, ADDP, ADDF, and ADDD.

Table III lists each generic operator, its valid type suffixes, and the number of kids and syms that it uses; multiple values for kids indicate type-specific variations, which are detailed below. For most operators, the type suffix denotes the type of operation to perform and the type of the result. The operators for assignment, comparison, arguments, and some calls return no results; their type suffixes denote the type of operation to perform.

The leaf operators yield the address of a variable or the value of a constant. syms[0] identifies the variable or constant. The unary operators accept and yield a number, except

Table III. Node Operators

syms	kids	operator	type suffixes	operation
1	0	ADDRF	P	address of a parameter
1	0	ADDRG	P	address of a global
1	0	ADDRL	P	address of a local
1	0	CNST	CSIUPFD	constant
	1	BCOM	U	bitwise complement
	1	CVC	IU	convert from `char`
	1	CVD	I F	convert from `double`
	1	CVF	D	convert from `float`
	1	CVI	CS U D	convert from `int`
	1	CVP	U	convert from pointer
	1	CVS	IU	convert from `short`
	1	CVU	CSI P	convert from `unsigned`
	1	INDIR	CSI PFDB	fetch
	1	NEG	I FD	negation
	2	ADD	IUPFD	addition
	2	BAND	U	bitwise AND
	2	BOR	U	bitwise inclusive OR
	2	BXOR	U	bitwise exclusive OR
	2	DIV	IU FD	division
	2	LSH	IU	left shift
	2	MOD	IU	modulus
	2	MUL	IU FD	multiplication
	2	RSH	IU	right shift
	2	SUB	IUPFD	subtraction
0 1	2	ASGN	CSI PFDB	assignment
1	2	EQ	IU FD	jump if equal
1	2	GE	IU FD	jump if greater than or equal
1	2	GT	IU FD	jump if greater than
1	2	LE	IU FD	jump if less than or equal
1	2	LT	IU FD	jump if less than
1	2	NE	IU FD	jump if not equal
2	1	ARG	I PFDB	argument
	1 2	CALL	I FDBV	function call
	0 1	RET	I FD V	return from function
	1	JUMP	V	unconditional jump
1	0	LABEL	V	label definition

for `INDIR`, which accepts an address and yields the value at that address. The binary operators accept two numbers and yield one. Exceptions are `ADDP`, in which the first integer operand is added to the second pointer operand, and `SUBP`, which subtracts the second integer operand from the first pointer operand.

The type suffix for a conversion operator denotes the type of the result. For example, `CVUI` converts an unsigned (`U`) to an signed integer (`I`). Conversions between unsigned and short or character are unsigned conversions; those between integer and short or character are signed conversions. For example, `CVSU` converts an unsigned short to an unsigned while `CVSI` converts an signed short to a signed integer.

To minimize the number of operators, the front end composes conversions to form those not in the table. For example, it converts a short to a float by first converting it to an int and then a double. The 16 conversion operators are represented by arrows in Figure 2. Composed conversions follow the path from the source type to the destination type.

$$\begin{tikzcd}
 & \mathtt{C} \arrow[d, leftrightarrow] & \mathtt{C} \arrow[d, leftrightarrow] & \\
\mathtt{D} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \mathtt{I} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \mathtt{U} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \mathtt{P} \\
\mathtt{F} & \mathtt{S} & \mathtt{S} &
\end{tikzcd}$$

Figure 2. Conversions

There is no `CVUD`; conversion of an unsigned `u` to a double is done by the equivalent of the expression

```
(int)u >= 0 ? (double)(int)u : (double)(int)u + x + 1
```

where x is the maximum value for an unsigned. Likewise, there is no `CVDU`; conversion of a double `d` to an unsigned is done by the equivalent of

```
d >= x+1 ? (unsigned)(int)(d-(x+1)) + x+1 : (unsigned)(int)d
```

where x is the maximum value for a signed integer.

`ASGN` stores the value of `kids[1]` in the cell addressed by `kids[0]`. `ASGNB` implements structure assignment and other 'block moves' (e.g., initialization of automatic arrays). For `ASGNB`, `syms[0]` points to a symbol for an integer constant that gives the number of bytes to be moved.

For the comparisons, `syms[0]` points to a symbol table entry for the label to jump to if the comparison is true. `JUMPV` is an unconditional jump to the address computed by `kids[0]`. `LABEL` defines the label given by `syms[0]` and is otherwise a no-op.

Function calls have a `CALL` node preceded by zero or more `ARG` nodes. The front end 'un-nests' function calls, so `ARG` nodes are always associated with the next `CALL` node in the forest. `ARG` nodes establish the value computed by `kids[0]` as the next argument.

`syms[0]` and `syms[1]` point to symbol table entries for integer constants that give the size of the argument and its alignment, respectively.

In `CALL` nodes, `kids[0]` computes the address of the callee. `CALLB` calls functions that return structures; `kids[1]` computes the address of a temporary local variable to hold the returned value. There is no `RETB`; the front end uses a `RETV` preceded by an `ASGNB` to the structure addressed by the first local. The `CALLB` code and the function prologue must collaborate to store the `CALLB`'s `kids[1]` into the callee's first local. `function`, `local`, and the code emitted below for `CALLB` illustrate such collaboration. `CALLB` nodes have a `count` of `0` because the front end references the temporary wherever the returned value is referenced.

In `RET` nodes, `kids[0]` computes the value returned, except for `RETV` nodes, which are childless.

To accommodate typical calling conventions, character and short integer arguments are always promoted to the corresponding integer type even in the presence of a prototype. To adhere to the standard, the promoted values are converted back to the intended type upon entry to the function. The front end accomplishes this conversion by specifying the intended types for the callee. For example, the body for

```
f(char c) { f(c); }
```

becomes two forests, which are shown in Figure 3. In Figure 3, the linearized form for each dag is listed to the left of its graphical form; the `syms` column lists the `x.name` fields. The first forest holds one dag, which converts the actual argument to the intended type. The second forest holds two dags. The first (nodes 1--4) promotes `c` to pass it as an integer, and the second (nodes 5 & 6) calls `f`.

Unsigned variants of `ASGN`, `INDIR`, `ARG`, `CALL`, and `RET` were omitted as unnecessary. Signed and unsigned integers have the same size, so the corresponding signed operator is used instead. Likewise, there is no `CALLP` or `RETP`. A pointer is returned by using `CVPU` and `RETI`. A pointer-valued function is called by using `CALLI` and `CVUP`.

In Table III, the operators from `ASGN` on are used for their side-effects. They always appear as roots in the forest of dags, and they appear in the order in which they must be executed. Except for `CALLD`, `CALLF` and `CALLI`, their reference counts are always zero. Even these `CALL` nodes have zero reference counts when their values are unused.

6.1 Generating Code

The front end calls `gen` to select code. It passes `gen` a forest of dags. For example,

```
int i, *p; f() { i = *p++; }
```

yields the forest shown in Figure 4. This forest consists of three dags, rooted at nodes 2, 5, and 8. The `INDIRP` node, which fetches the value of `p`, comes before node 5, which changes `p`, so the original value of `p` is available for subsequent use by node 7, which fetches the integer pointed to by that value.

```
#  op     count kids   syms
1. ADDRFP   1          4
2. ADDRFP   1          4
3. INDIRI   1   2
4. CVIC     1   3
5. ASGNC    0   1 4

1. ADDRFP   1          4
2. INDIRC   1   1
3. CVCI     1   2
4. ARGI     0   3      4 4
5. ADDRGP   1          f
6. CALLI    0   5      4
```

Figure 3. Forest for `f(char c) { f(c); }`

```
#  op     count kids   syms
1. ADDRGP   2           p
2. INDIRP   2   1
3. CNSTI    1           4
4. ADDP     1   2 3
5. ASGNP    0   1 4
6. ADDRGP   1           i
7. INDIRI   1   2
8. ASGNI    0   6 7
```

Figure 4. Forest for `i = *p++`

`gen` traverses the forest and selects code, but it emits nothing because it may be necessary to determine, for example, the registers needed before the function prologue can be emitted. So `gen` merely annotates the nodes to identify the code selected, and it returns a pointer that is ultimately passed to the back end's `emit` to actually output the code. Once the front end calls `gen`, it does not inspect the contents of the nodes again, so `gen` may modify them freely.

The sample code generator emits naive code, so `gen` concerns itself mainly with register allocation. The sample `gen`

```
Node gen(Node p) {
   Node head, *last;

   for (last = &head; p; p = p->link)
```

```
            last = linearize(p, last, 0);
        for (p = head; p; p = p->x.next) {
            ralloc(p);
            if (p->count == 0 && sets(p))
                putreg(p);
        }
        return head;
    }
```

linearizes each dag in the forest, in execution order, and then allocates registers for each node. If the node sets a register, but no subsequent node references it, which is indicated by a reference `count` of `0`, `gen` releases the register. Finally, it returns the linearized node list for traversal by `emit`.

`gen` linearizes the dags before allocating registers to simplify the insertion of spills and reloads.[14] The back end's function

```
    static Node *linearize(Node p, Node *last, Node next)
```

linearizes the dag at `p` and inserts it into the growing list of nodes between `*last` and `next`.

`ralloc` handles register allocation for a single node.

```
    static void ralloc(Node p) {
        switch (generic(p->op)) {
        case ARG:
            argoffset = roundup(argoffset, p->syms[1]->u.c.v.i);
            p->x.argoffset = argoffset;
            argoffset += p->syms[0]->u.c.v.i;
            if (argoffset > argbuildsize)
                argbuildsize = roundup(argoffset, 4);
            break;
        case CALL:
            argoffset = 0;
            break;
        }
        putreg(p->kids[0]);
        putreg(p->kids[1]);
        p->x.busy = rmask;
        if (needsreg(p))
            getreg(p);
    }
```

`generic` strips the type suffix from an operator and thus simplifies the switch. `ralloc` calls `putreg` to release the registers allocated for the node's children and then it calls getreg to allocate a register for the node itself if it needs one. It sets the node's x.busy to record the busy registers; emit needs this information for a few operators. The register state is encoded in rmask; rmask&(1<<r) is 1 if register r is busy, for r from 0 to 11. rmask is initialized to 1.

CALL and ARG nodes require extra steps. ARG nodes produce no result; instead, they store the value computed by kids[0] into the next location in the argument build area. syms[0] and syms[1] point to constant symbols that give the size and alignment of the argument. argoffset is the running offset into the argument build area. argoffset is rounded up to the appropriate alignment boundary, saved in the node's x.argoffset for use in emit, and incremented by the size of the argument. argbuildsize tracks the maximum argoffset needed by the current function. The case for CALL nodes clears argoffset for the next set of ARG nodes.

getreg accepts a node and allocates a register for it:

```
static void getreg(Node p) {
   int r, m = optype(p->op) == D ? 3 : 1;

   for (r = 0; r < nregs; r++)
      if ((rmask&(m<<r)) == 0) {
         p->x.rmask = m;
         p->x.reg = r;
         rmask |= sets(p);
         usedmask |= sets(p);
         return;
      }
   r = spillee(p, m);
   spill(r, m, p);
   getreg(p);
}
```

optype(op) returns the type suffix of operator op. m is set to the mask 01_2 if the result of p->op needs an ordinary register and 11_2 if it needs a double register. getreg loops over the registers. If it finds one that's free, it sets the node's x.reg field to the register allocated and the x.rmask field to the mask. It also updates usedmask, which is used to generate the prologue; sets(p) returns p->x.rmask<<p->x.reg. If no registers are free, getreg spills a register and calls itself recursively to try again. Reference 14 describes spills.

putreg releases registers:

```
static void putreg(Node p) {
   if (p && --p->count <= 0)
      rmask &= s̃ ets(p);
}
```

Dags can use result registers multiple times, so `putreg` decrements the reference count and frees the register only when the last reference is removed.

6.2 Emitting Code

`emit` emits the linearized forest. The sample walks down the list and switches on the opcode to identify the code to emit:

```
void emit(Node p) {
   for (; p; p = p->x.next) {
      Node a = p->kids[0], b = p->kids[1];
      int r = p->x.reg;
      switch (p->op) {
      case CNSTC: case CNSTI: case CNSTP:
      case CNSTS: case CNSTU:
         print("movl $%s,r%d\n", p->syms[0]->x.name, r);
         break;
      case ADDRGP: case ADDRFP: case ADDRLP:
         print("moval %s,r%d\n", p->syms[0]->x.name, r);
         break;
      ...
      }
   }
}
```

The individual cases emit naive code for a single operator. The `CNST` cases above emit a VAX instruction that loads a constant into a register. `defsymbol` stored the constant string in `p->syms[0]->x.name`, and `ralloc` stored the result register name `p->x.reg`. The `ADDR` cases above emit a VAX instruction that moves the address of a variable into the result register. The back end's `asmname` returns an assembler operand for the variable, e.g., for global `x` it returns `"_x"` and for the first argument it returns `"4(ap)"`.

Most of the unary operators share a common pattern, e.g.,

```
case BCOMU: print("mcoml r%d,r%d\n",  a->x.reg, r); break;
case CVCI:  print("cvtbl r%d,r%d\n",  a->x.reg, r); break;
case CVCU:  print("movzbl r%d,r%d\n", a->x.reg, r); break;
...
```

Most of the binary operators and all of the comparisons are handled similarly:

```
case ADDI: case ADDP: case ADDU:
   print("add%c3 r%d,r%d,r%d\n", suffix(p),
      b->x.reg, a->x.reg, r); break;
...
case GEU:
   print("cmpl r%d,r%d; jgequ L%s\n",
      a->x.reg, b->x.reg, p->syms[0]->x.name); break;
case GED:  case GEF:  case GEI:
   print("cmp%c r%d,r%d; jgeq L%s\n", suffix(p),
      a->x.reg, b->x.reg, p->syms[0]->x.name); break;
...
```

The macro `suffix` gives the VAX type suffix character. The cases

```
   case INDIRC: case INDIRD: ...
      print("mov%c (r%d),r%d\n", suffix(p),
         a->x.reg, r); break;
   case ASGNC:  case ASGND:  ...
      print("mov%c r%d,(r%d)\n", suffix(p),
         b->x.reg, a->x.reg); break;
```

emit code to indirectly load and store memory cells. The cases

```
   case ARGD: case ARGF: case ARGI: case ARGP:
      print("mov%c r%d,%d(sp)\n", suffix(p),
         a->x.reg, p->x.argoffset);
      break;
   case CALLD: case CALLF: case CALLI: case CALLV:
      save(p->x.busy&0x3e);
      print("calls $0,(r%d)\n", a->x.reg);
      if (p->op != CALLV)
         print("mov%c r0,r%d\n", suffix(p), r);
      restore(p->x.busy&0x3e);
      break;
   case RETD: case RETF: case RETI:
      print("mov%c r%d,r0; ret\n", suffix(p), a->x.reg);
      break;
   case RETV:
      print("ret\n");
      break;
```

emit code to move an argument onto the stack, to call a procedure, and to return a value. The `ARG` cases use `x.argoffset`, into which `ralloc` stored the stack offset for the argument. Procedures may destroy registers 1--5, so the `CALL` cases use

```
   static void save(unsigned mask) {
      int i;

      for (i = 1; i < nregs; i++)
         if (mask&(1<<i))
```

```
            print("movl r%d,%d(fp)\n", i,
               4*i - framesize + argbuildsize);
    }
```

to emit code to save any of those registers that are busy; `restore` is similar. The `RET` cases merely copy any return value into register 0 and return.

Several binary operators require special handling. For example, no instructions directly implement unsigned right shift or division, so `emit` uses a field-extraction instruction for the first and a library call for the second, and `a%b` is computed by `a-(a/b)*b`:

```
    case RSHU:
       print("subl3 r%d,$32,r0; extzv r%d,r0,r%d,r%d\n",
          b->x.reg, b->x.reg, a->x.reg, r);
       break;
    case DIVU:
       save(p->x.busy&0x3e);
       print("pushl r%d; pushl r%d; calls $2,udiv; movl r0,r%d\n",
          b->x.reg, a->x.reg, r);
       restore(p->x.busy&0x3e);
       break;
    case MODI:
       print("divl3 r%d,r%d,r0; mull2 r%d,r0; subl3 r0,r%d,r%d\n",
          b->x.reg, a->x.reg, b->x.reg, a->x.reg, r);
       break;
```

Unsigned modulus is handled similarly.

Only the structure instructions remain:

```
    case INDIRB:
       print("moval (r%d),r%d\n", a->x.reg, r);
       break;
    case ASGNB:
       save(p->x.busy&0x3f);
       print("movc3 $%s,(r%d),(r%d)\n", p->syms[0]->x.name,
          b->x.reg, a->x.reg);
       restore(p->x.busy&0x3f);
       break;
    case ARGB:
       save(p->x.busy&0x3f);
       print("movc3 $%s,(r%d),%d(sp)\n", p->syms[0]->x.name,
          a->x.reg, p->x.argoffset);
       restore(p->x.busy&0x3f);
       break;
```

The scalar `INDIR`s and `ASGN`s load and store values directly, but structures won't fit in registers, so their instructions manipulate *addresses* instead. `ASGNB` uses a block move instruction, which needs the size from `p->syms[0]->x.name`; it also destroys registers 0--5, so `emit` arranges to save and restore their values. `ARGB` operates similarly, but copies the structure into the stack instead. Finally,

```
case CALLB:
   save(p->x.busy&0x3e);
   if (a->x.reg == 1) {
      print("movl r1,r0\n");
      a->x.reg = 0;
   }
   if (b->x.reg != 1)
      print("movl r%d,r1\n", b->x.reg);
   print("calls $0,(r%d)\n", a->x.reg);
   restore(p->x.busy&0x3e);
   break;
```

is like an ordinary call, but it also passes the address at which to store the return value in register 1; if the address of the *function* is already in register 1, it is first moved out of the way into register 0, which is not otherwise allocated.

7. DISCUSSION

`lcc`'s code generation interface is compact because it omits the inessential and makes simplifying assumptions. These omissions and assumptions do, however, limit the interface's applicability to other languages and machines.

The datatype assumptions detailed in Section 3, e.g., that unsigneds, integers, and long integers all have the same size, make it possible to have only 9 type suffixes and 111 type-specific operators. Relaxing these assumptions would increase this vocabulary; e.g., adding a suffix for long doubles would also add at least 19 more operators.

The interface assumes that all pointer types have the same representation, which precludes its use for word-addressed machines. Differentiating between character and word pointers would add another suffix and at least 13 more operators.

The operator repertoire omits some operators whose effect can be synthesized from simpler ones. For example, bit fields are accessed with shifting and masking instead of specific bit-field operators, which may complicate thorough exploitation of machines with bit-field instructions. The front end special-cases one-bit fields and generates efficient masking dags, which often yields better code than code that uses bit-field instructions.

The front end implements switch statements completely. It generates a binary search of dense branch tables; inline comparisons replace small tables.[15] It fabricates the tables and indirect jumps using the functions `global` and `defaddress` and the `JUMPV` operator. This decision prevents back ends from using larger 'case' instructions, which usually combine a bounds check and an indirect branch through a table, but these instructions are increasingly rare, and some don't suit C anyway. For example, the branch table for the VAX's `casel` instruction is limited to 16-bit offsets.

The interface has gone through several revisions and has been simplified each time by moving functionality into the front end or by pruning the interface vocabulary. For example, earlier versions included a separate interface function to lay out branch tables. And there were more operators, e.g., `SWTCH`, which identified switch expressions, and `CVUD`. Each revision made writing a back end simpler.

Nonetheless, blemishes remain. They complicate writing a back end or at least invite bugs. For example, `ASGN`, `CALL`, and `RET` have type-specific variants that take different numbers of operands. This variability complicates decisions that otherwise could be made by inspecting only the generic operation. Operators that always generate trivial target code are another example. A few operators generate nothing on *some* targets, but some, like `CVUI` and `CVIU`, generate nothing on *all* current or conceivable targets. In the sample, these operators generate vacuous register-to-register moves. In production back ends, however, such moves must be either avoided or eliminated. Similarly, the 'narrowing' conversions `CV{UI}`×`{CS}` are vacuous on all targets and might well be omitted.

There are also several conventions that, if not obeyed, can cause subtle bugs. An example is the interaction between `local`, `function`, and `CALLB` for functions that return structures; failure to obey the convention in at least three different places in the back end results in bad code. The front end *could* deal with such functions completely, but doing so would exclude some established calling sequences. The tradeoff for generating compatible code is a more complex code generation interface.

Sharing data structures between the front end and the code generator is manageable because there are few such structures. A disadvantage of this approach, however, is that the structures can be more complex than they might be in other designs, which comprises simplicity. For example, symbols represent all identifiers across the interface. Symbols have many fields, but some are relevant only to the front end and access to them can be regulated only by convention. Some symbols use only a few of the fields; labels, for example, use only the `name` and `u.label` fields.

C shares the blame for this complexity: specifying all of the possibilities requires a type system richer than C's. Some of the complexity might be avoided by defining separate structures, e.g., one for each kind of symbol and another for private front-end data, but doing so increases the data structure vocabulary and hence complexity. Type systems with inheritance simplify defining variants of a structure without also complicating uses of those variants. Oberon has perhaps the minimum inheritance machinery required. [16]

The interface was designed for use in a monolithic compiler; it complicates separating the front and back ends into separate programs. Some of the interaction is two-way; the upcalls from `function` to `gencode` and `emitcode` are examples. These upcalls permit the front end to generate conversion code required at function entry; if the back end is made a separate program, it may have to implement such conversions. The back end examines few fields in the source-language type representation; it uses front-end functions like `isstruct` to query types. If the back end is made a separate program, type data must be transmitted to answer such queries.

The interface is designed to support only code generation; it has no direct support for building a flow graph and other structures that facilitate global optimization. More elaborate versions of `function` and `gen` could collaborate to build the relevant structures, perform optimizations, and invoke the simpler `gen`. The front end's `gencode` and `emitcode` traverse an approximation of a flow graph; current work on a machine-independent optimizer edits this graph while preserving the current interface.

To date, the interface has been used only for ANSI C. But it has little that is specific to C, and it could be used for similar languages and perhaps for languages with features like nested procedures, objects, and exception handling. Existing compilers for some object-oriented languages with these features, such as C++, Modula-3, and Eiffel, generate C, so, in principle, the interface could be used for these languages.

ACKNOWLEDGEMENTS

Suggestions from Todd Proebsting improved `ralloc`.

REFERENCES

1. *American National Standard for Information Systems, Programming Language C ANSI X3.159–1989*, American National Standards Institute, Inc., New York, 1990.
2. C. W. Fraser, ‘A language for writing code generators’, *SIGPLAN Notices*, *24*, 238--245 (1989).
3. C. W. Fraser and D. R. Hanson, ‘A code generation interface for ANSI C’, *Research Report CS-TR-270-90*, Princeton University, Department of Computer Science, Princeton, NJ, July 1990.
4. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, ‘Code generation using tree matching and dynamic programming,’ *ACM Trans. on Programming Languages and Systems*, *11*, 491--516 (1989).
5. A. S. Tanenbaum, H. van Staveren and J. W. Stevenson, ‘Using peephole optimization on intermediate code’, *ACM Trans. on Programming Languages and Systems*, *4*, 21--36 (1982).
6. K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi, ‘Pascal-P Implementation Notes’, *Pascal — The Language and its Implementation*, D. W. Barron, ed., 83--123, John Wiley & Sons, 1981.
7. D. R. Perkins and R. L. Sites, ‘Machine-independent Pascal code optimization’, *SIGPLAN Notices*, *14*, 201--207 (1979).
8. M. C. Newey, P. C. Poole, and W. M. Waite, ‘Abstract machine modelling to produce portable software --- A review and evaluation’, *Software—Practice & Experience*, *2*, 107--136 (1972).
9. A. S. Tanenbaum, M. Frans Kaashoek, K. G. Langendoen and C. J. H. Jacobs, ‘The design of very fast portable compilers’, *SIGPLAN Notices*, *24*, 125--131 (1989).
10. J. W. Davidson and C. W. Fraser, ‘Code selection through object code optimization’, *ACM Trans. on Programming Languages and Systems*, *6*, 505--526 (1988).
11. R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Cambridge, MA, 1990.
12. M. E. Benitez, P. Chan, and J. W. Davidson, A. M. Holler, S. Meloy, and V. Santhanam, ‘ANDF: Finally an UNCOL after 30 years’, *Technical Report TR-91-05*, University of Virginia, Department of Computer Science, Charlottesville, VA, March 1991.
13. T. E. Leonard, ed., *VAX Architecture Reference Manual*, Digital Press, Bedford, MA, 1987.
14. C. W. Fraser and D. R. Hanson, ‘Simple register spilling in a retargetable compiler’, *Software—Practice & Experience*, submitted.
15. R. L. Bernstein, ‘Producing good code for the case statement’, *Software—Practice & Experience*, *15*, 1021--1024 (1985).
16. N. Wirth, ‘The programming language Oberon’, *Software—Practice & Experience*, *18*, 670--690 (1988).