A ‘C Tutorial

Dawson R. Engler and Massimiliano Poletto
engler@lcs.mit.edu, maxp@lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Last modified: June 14, 1997

Abstract

This paper describes ‘C (*Tick C*), a superset of ANSI C that allows high-level, efficient, and machine-independent specification of dynamically generated code.

Keywords: dynamic code generation, C

1 Introduction

‘C is a superset of the ANSI C language that allows high-level, efficient, and portable specification of dynamically generated code. This paper is intended to serve both as a reference to the language, and as a source of ideas and techniques on how to use ‘C and dynamic code generation effectively.

The discussion assumes a working knowlege of the C language. The classic C text is “K & R” [8]; a good reference is [5]. Section 2 describes the language in detail. Section 3 provides several example uses that help illustrate the language’s features.

2 ‘C Language

‘C extends ANSI C with two type constructors, two unary operators, and some special forms. These extensions do not change the semantics of the core language, and generally preserve the “spirit” of C.

‘C allows programmers to easily specify pieces of dynamic code. These dynamic code specifications may be *composed* to build larger specifications, or *instantiated* (compiled at run time) to dynamically produce executable code.

A time line of this process is as follows:

1. At edit time, programmers write ‘C code.

2. At *static compile time*, a ‘C compiler processes the written program and emits both (1) traditional static code and (2) code-generating code that will create the specified dynamic code at run time.

3. At run time, when control reaches the dynamic code specifications written by the programmer, these specifications are created and built up into larger specifications. This phase is called *specification time*.

4. After specification time, the dynamic code specifications may be passed to the ‘C compile special form, which dynamically compiles them to machine code. We call this phase *instantiation time*. Compile returns a function pointer to the generated code, which programmers can then call as they would any other function.

The rest of this section describes in detail the dynamic code generation extensions introduced in ‘C. The syntax of the language is defined in Appendix A.

2.1 The ‘ Operator

At the core of ‘C is the ‘ (backquote, or "tick") operator, which serves to specify dynamic code. ‘ applied to an expression or compound statement indicates that code corresponding to that expression or statement should be generated at run time. Some simple usages of backquote are as follows:

```
/* Specification of dynamic code for the expression "4": results
   in dynamic generation of code that moves 4 into a register */
‘4

/* Specification of dynamic code for a call to printf;
   j must be declared in an enclosing scope */
‘printf("%d", j)

/* Specification of dynamic code for a compound statement */
‘{
    int i;
    for (i = 0; i < 10; i++)
        printf("%d\n", i);
}
```

Since ‘C currently disallows the dynamic generation of code-generating code, ‘ does not nest.

Dynamic code is lexically scoped: variables in enclosing static code can be captured by free variables in dynamic code. The use of such a variable after its scope has been exited is undefined. Furthermore, like C, ‘C is statically typed. Lexical scoping and static typing allow type-checking and some instruction selection to occur at static compile time, decreasing dynamic code generation overhead.

The use of several C constructs is restricted within backquote expressions. In particular, a break, continue, case, or goto statement cannot be used to transfer control outside the enclosing backquote expression. For instance, the destination label of a goto statement must be contained in the same backquote expression that contains the goto. ‘C, however, provides other means for transferring control between backquote expressions. These are discussed in Section 2.4. The limitation on goto and other control transfer statements enables ‘C compilers to statically determine whether a control flow change is legal. The use of return is not restricted in this way, because dynamic code is always implicitly inside a function.

A ‘ expression can be composed with the values of other ‘ expressions to create larger pieces of dynamic code. This is discussed further in Sections 2.2.1 and 2.2.3. In addition, a ‘ expression specifying a dynamic compound statement can be dynamically compiled using the compile special form, which is described in more detail in Section 2.4. compile returns a function pointer, which can then be invoked like any other function pointer. The following code fragment dynamically constructs, instantiates, and invokes a function that prints "Hello World":

```
int main(void) {
    /* Generate function to print greeting */
    void (*f)() = compile(‘{ printf("hello, world\n"); }, void);
    /* Invoke the function */
    (*f)();
    return 0;
}
```

2.2 Type Constructors

‘C introduces two new type constructors, cspec and vspec, that are used to declare dynamic code and variable specifications. A cspec or vspec has an associated *evaluation type*, which is the type of the dynamic value of the specification. The evaluation type is analogous to the type to which a pointer points. It allows dynamic code to be statically typed, which in turn enables the ‘C compiler to type-check code composition and perform most instruction selection for dynamic code statically.

2.2.1 cspec Types

cspec (short for *code specification*) is the type of a dynamic code specification; the evaluation type of the cspec is the type of the dynamic value of the code. For example, the type of the expression ‘4 is int cspec.

Applying ‘ to a statement or compound statement yields an expression of type void cspec. In particular, if the dynamic statement or compound statement contains a return statement, the type of the return value does not affect the type of the ‘ expression. Since all type-checking is performed statically, it is currently possible to compose ‘ expressions to dynamically create a function with multiple (possibly incompatible) return types. We are considering ways of solving this problem.

The type void cspec is the type of a generic cspec type (analogous to the use of void * as a generic pointer).

The code generated by ‘ may include implicit casts used to reconcile the result type of ‘ with its use; the standard conversion rules of ANSI C apply.

Some simple uses of cspec follow:

```
int cspec expr1 = ‘4;            /* Code specification for expression "4" */
float x;
float cspec expr2 = ‘(x + 4.);  /* Capture free variable x: its value will be
                                   bound when the dynamic code is executed */

/* All dynamic statements have type void cspec, regardless of
   whether the resulting code will return a value */
void cspec stmt = ‘{ printf("hello, world\n"); return 0; };
```

2.2.2 vspec Types

vspec (*variable specification*) is the type of a dynamically generated lvalue, a variable whose storage class (whether it should reside in a register or on the stack, and in what location exactly) is determined dynamically. The evaluation type of the vspec is the type of the lvalue. Objects of type vspec may be created by invoking the special forms param and local. param is used to create a parameter for the function currently under construction; local is used to reserve space in its activation record (or allocate a register if possible). See Section 2.4 for more details.

void vspec is used as a generic vspec type.

In general, an object of type vspec is automatically treated as a variable of the vspec’s evaluation type when it appears inside a cspec. A vspec inside a ‘ expression can thus be used like a traditional C variable, both as an lvalue and an rvalue. For example, the following function specifies dynamic code that will take a single integer argument, add one to it, and return the result:

```
void cspec plus1(void) {
    /* Param takes the index and type of the argument to be generated */
    int vspec i = param(0, int);
    return ‘{ return i + 1; };
}
```

Please refer to Section 2.2.3 for the exact details regarding the use of vspecs within cspecs.

vspecs allow us to construct functions that take a run-time-determined number of arguments; this power is necessary in applications such as the compiling interpreter described in Section 3.8.

2.2.3 Automatic Conversion to Evaluation Types

‘C disallows dynamic generation of code which dynamically generates code. As a result, it is never necessary to refer to an object of type cspec or vspec within a ‘ expression. In addition, it is illegal to declare objects of type vspec or cspec within the body of a ‘ expression.

With one exception (see below), an object of type vspec appearing inside a ‘ expression is treated as an object of its corresponding evaluation type. Similarly, objects of type cspec appearing inside a ‘ expression are usually considered to have their corresponding evaluation type, and denote code having that type. It is thus possible to easily build up larger cspecs from smaller ones:

```
int cspec a = ‘4; int cspec b = ‘5;
int cspec sum = ‘(a+b);
int cspec sumofsum = ‘(sum+sum);
```

Compiling sumofsum results in dynamic code equivalent to (4+5)+(4+5).

Statements (and compound statements) are considered to have type void; an object of type void cspec inside a ‘ expression thus cannot be used inside an expression, but can appear alone (as an expression statement) to denote code composition:

```
void cspec hello = ‘{ printf("hello "); };
void cspec world = ‘{ printf("world\n"); };
void cspec greeting = ‘{ hello; world; };
```

Dynamically compiling greeting results in code equivalent to:

```
printf("hello ");
printf("world\n");
```

There is one exception to this automatic conversion from cspec and vspec types to their evaluation types. If the return value of a function or special form has type cspec or vspec, then calling that function within a ‘ expression implicitly unquotes its arguments. Therefore, the arguments are considered to have their full type, not their evaluation type. This rule is necessary because functions that return cspecs or vspecs and are called within ‘ expressions must be invoked at specification time, before any code is generated, and not at the run time of the dynamic code.

For instance, this code is incorrect:

```
int cspec f(int j);          /* This prototype is wrong */
                             /* Should be int cspec f(int cspec j) */
void main() {
    int vspec v;
    /* ... */
    void cspec c = ‘{ f(v); }; /* v here is an int vspec, not int */
}
```

In this case, f returns an object of type cspec, so v, its argument, is not automatically converted from int vspec to int. As a result, f must have the prototype int cspec f(int vspec j).

Variables declared within the scope of dynamic code have their storage class determined dynamically, so the same unquoting rule that applies to vspecs and cspecs also applies to locals in the body of a cspec. In other words, a variable of type T local to a backquote expression takes on type vspec T when unquoted by a call to a function that returns a vspec or cspec. The example below illustrates this behavior. It is analogous to the int vspec example above.

```
int cspec f(int j);          /* This prototype is wrong */

void main() {
    void cspec c = ‘{ int v; f(v); };
}
```

In short, any cspec or vspec object in a backquote expression is converted to its evaluation type, unless it is unquoted by a call to a function with cspec or vspec return type. The following code is another example of this unquoting mechanism.

```
int cspec f1(int cspec c); int cspec f2(int vspec v);

void main() {
    int vspec v;
    /* ... */
    void cspec a = `{ int x = f1(
                            f2(
                              v /* v is an int vspec */
                             )  /* f2(v) is a cspec (unquoted by f1) */
                           );     /* f1(...) is converted to int */
                    };
    void cspec b = `{ int x = f2(
                            v   /* v is an int vspec */
                           );    /* f2(v) is converted to int */
                   };
}
```

In the specification for b, f2(v) is automatically converted to its evaluation type. This is not the case in the specification for a, because there the call to f2 appears in the argument list of a call to f1. Since f1 returns a cspec, it unquotes its arguments, preventing them from being automatically converted to their evaluation types.

2.3 The $ Operator

The $ operator allows *run-time constants* to be incorporated into dynamic code. $ evaluates its operand at specification time; the resulting value is incorporated as a run-time constant into the containing cspec. $ may only appear inside a ` expression, and it may not be unquoted. It may be applied to any object not of type cspec or vspec. The use of $ is illustrated in the code fragment below.

```
int cspec c1, cspec c2;
void cspec c;
int x = 1;

/* Bind x as a run−time constant with value "1" */
c1 = `$x;
/* Bind x at run−time when the code for c2 is run */
c2 = `x;
c = `{ printf("$x = %d, x = %d\n", c1, c2); };
x = 14;
/* Compile and run: will print "$x = 1, x = 14". */
(*compile(c, void))();
```

Use of $ enables specialization of code based on run-time constants. An example of this is the following program, which specializes multiplication of a matrix by an integer:

```
/* Return a cspec corresponding to multiplication of matrix m by constant c */
void cspec mkscale(int **m, int n, int s) {
    /* Construct code that loops over matrix m (of size n x n), multiplying
       each of its elements by s. */
    return `{
        int i,j;
        /* $n can be encoded directly in the loop termination check */
        for (i = 0; i < $n; i++) {
            int *v = ($m)[i];
```

```
            for (j = 0; j < $n; j++)
                /* Multiplication by '$s' can be strength-reduced at
                   instantiation time */
                v[j] = v[j] * $s;
        }
    }
};
```

Remember that applying $ to a cspec or vspec object is illegal. The following code, for example is incorrect:

```
int cspec x = '1;
int vspec v;
void cspec c = '{ int y = $x; };   /* Error */
void cspec d = '{ int y = $v; };   /* Error */
```

2.4 Special Forms

Much of the functionality of dynamic code generation in 'C is provided by a small set of special forms which manipulate cspecs and vspecs. Many of these special forms take types as arguments, similarly to the sizeof operator. Unlike sizeof, however, their result types sometimes vary depending on this input: effectively, they are polymorphic. We describe each special form below:

T (*compile(void cspec *code*, *T*))()

compile generates machine code from *code*, returning a pointer to a function returning type *T*. In the future, compile may take a number of flags related to optimization, debugging and profiling.

void decompile(*T*)

decompile takes as argument a function pointer (of an arbitrary function pointer type *T*) returned by compile, and destroys the corresponding function. In this way, it is possible to reclaim the memory consumed by dynamic code when the code is no longer necessary.

T vspec local(*T*)

local returns an object of type *T* vspec denoting a dynamic local variable of type *T*. Space for this variable may be reserved in the activation record of the function currently being specified, or the variable may be allocated to a register. Note that *T* may include one of two storage class specifiers, auto and register. If auto is specified, the dynamic variable will be allocated on the stack. Otherwise, the compiler will attempt to place it in a register. Other storage class specifiers (extern, static, typedef) may not appear in the argument to local.

T vspec param(*T*, int *param-num*)

param returns an object of type *T* vspec that denotes an argument of the function currently being specified. *param-num* is the parameter's position in the function's argument list, whereas *T* denotes its evaluation type. In the future, param will take an additional flag, to indicate whether or not the user has a prototype for the dynamic function. If so, the type of the argument in the prototype is assumed to match *T*, and dynamic code is generated to handle an argument of type *T*. Otherwise, the dynamic code is made to implement whatever C argument conversions are applicable to *T*.

void cspec push_init(void)

push_init returns a cspec corresponding to a new (initially empty) dynamic argument list for a function call. This allows the programmer to create function calls which pass a dynamically determined number of arguments to the invoked function. See Section 3.4 for an example of this.

void push(void cspec *args*, *T* cspec *next-arg*)

push adds the code specification for the next argument, *next-arg*, to the dynamically-generated list of arguments *args*. *args* must have previously been initialized by push_init. *T* may not be void. Again, an example of this mechanism appears in Section 3.4.

void cspec label()

label returns a void cspec that may be used as the destination of a jump command. The parentheses are optional.

void cspec jump(void cspec *target*)

jump returns the cspec of a jump to *target*. *Target* may be any object of type void cspec, including a cspec returned by label. Jump and label are useful for constructing "hard-coded" finite state machines, and (although at the expense of some syntactic baggage) provide an alternative to the constraint on goto statements described in Section 2.1. As for label, the parentheses are optional.

T self(*T*, *other-args...*)

self allows recursive calls in dynamic code. *T* denotes the return type of the function that is being dynamically generated. Invoking self results in a call to the function containing the invocation, with *other-args* passed as the arguments. self is just like any other function call, except that the return type of the current function is unknown, so it must be specified as the first argument.

2.5 Discussion

'C is a strict superset of ANSI C. There are, however, necessary departures from the spirit of C at some points. For example, the memory required to represent a cspec is the responsibility of the 'C run-time system. Hence, cspecs are objects whose allocation and manipulation is not controlled by the programmer. This is a marked departure from ANSI C, in which dynamic allocation of objects is entirely controlled by the programmer.

In addition, 'C performs some run-time checks that are not present in ANSI C. Specifically, the compiler must guard against conflicting parameter definitions for dynamically generated functions. Furthermore, as described in Section 2.2.1, it is possible to compose cspecs into a function containing return statements with incompatible return types. For the sake of efficiency, 'C currently does not require a check at instantiation time to ensure that return types are compatible.

3 Applications

Section 2 provided a succint reference description of 'C. The goal of this section is to illustrate 'C by example, giving sample code for various applications of the language. Among the uses of 'C illustrated below are dynamic code construction and the creation of executable data structures, specialization, use of run-time constants, dynamic call construction, function composition, fast small language compilation, and compiling interpreters.

3.1 Code Construction

'C provides programmers with a general set of mechanisms to build code at run time. These mechanisms can be used to perform unusual operations, such as custom-fitting a piece of code to a data structure or creating an executable data structure.

3.1.1 Vector Dot Product

Matrix and vector manipulations such as dot product are a fruitful realm for dynamic code generation. They often involve a large number of operations on values which change relatively infrequently. Matrices may have run-time characteristics (*i.e.*., large numbers of zeros and small integers) that can improve performance of matrix operations, but cannot be exploited by static compilation techniques. In addition, sparse matrix techniques are only efficient for matrices with a high degree of sparseness.

In the context of matrix multiplication the use of dynamic code generation allows us to exploit zeros and small integers by crafting locally optimized code based on run-time values. Because code for each row is specified once and then used *n* times (once for each column), the costs of code generation are easily recouped. Consider the following C code to compute the dot product:

```
void dot(int *a, int *b, int n) {
    int sum, k;
    for (sum = k = 0; k < n; k++)
        sum += a[k]*b[k];
    return sum;
}
```

At run time several optimizations can be employed. For example, the programmer can directly eliminate multiplication by zero. The necessary ‘C code is given below:

```
/* Construct code to multiply arbitrary vector
   by a given row  */
void cspec mkdot(int row[], int n) {
    int k;

    /* Pointer to the col to multiply row by */
    int * vspec col = param(int *, 0);

    /* Specification for the sum of products; initally 0 */
    int cspec sum = ‘0;
    for (k = 0; k < n; k++) {
        /* Only generate code for non-zero multiplications */
        if (row[k]) {
            /* Hardwire the index used to load col; $row[k] yields
               a runtime constant that allows the multiplication by
               col[$k] to be strength-reduced; the result is added
               to the accumulated sum.  */
            sum = ‘(sum + col[$k] * $row[k]);
        }
    }
    return ‘{ return sum; };
}
```

The dot product written in ‘C can perform substantially better than its static C counterpart. The ‘C code omits code generation for non-zero multiplications. In addition, the ‘C compiler can encode values as immediates in arithmetic instructions and can take advantage of the runtime constant `$row[k]` to strength-reduce multiplication (replacing it with a sequence of cheaper operations such as shifts and adds).

3.1.2 Binary Search

In addition to enabling the customization of algorithms to input sets with particular characteristics, ‘C allows the programmer to create code from some input data structure, effectively implementing executable data structures.

Consider the following piece of C code, which implements an iterative binary search on a sorted integer array:

```
/* array from 0..n-1, return position or -1 */
int search(int x[], int n, int key) {
    int l, u, r;

    for(l = 0, u = n-1, r = n / 2; l <= u ; r/=2 ) {
        int p = u - r;
        if(x[p] == key)
            return p;
        else if(x[p] < key)
            l = p+1;
        else
            u = p-1;
    }
    return -1;
}
```

A tail-recursive implementation more clearly expresses the nature of the search, allowing us to easily view the algorithm as traversing a binary tree:

```
/* Recursively generate code to perform binary search */
int bin(int *x, int key, int l, int u, int r) {
    int p;

    if(l > u)
        return -1;
    p = u - r;

    if(x[p] == key)
        return p;
    else if(x[p] < key)
        bin(x, key, p+1, u, r/2);
    else
        bin(x, key, l, p-1, r/2);
}
```

Both of these implementations access the information in the data structure by referencing into the input array, and both involve overhead due to looping.

Using ‘C, it is possible to write the following code:

```
/* Recursively generate code to perform binary search */
void cspec gen(int *x, int vspec key, int l, int u, int r) {
    int p;

    if(l > u)
        return ` { return -1; };
    p = u - r;
    return ` {
        if($(x[p]) == key)
            return $p;
        else if($(x[p]) < key)
            gen(x, key, p+1, u, r/2);
        else
            gen(x, key, l, p-1, r/2);
```

```
    };
}
```

The structure of this code is very similar to that of the recursive binary search. By simply adding a few dynamic code generation primitives to the original algorithm, we have obtained a function which returns a code specification of the algorithm tailored to a given input set. We can easily obtain a usable function pointer from this specification, as follows:

```
typedef int (*ip)(int key);
ip mksearch(int n, int *x) {
    int vspec key = param(int, 0); /* One argument: the key to search for */
    return (ip)compile(gen(x, key, 0, n−1, n/2), int);
}
```

In the resulting code, the values from the input vector are hard-wired into the instruction stream, and the loop is unrolled into a "binary tree" of conditionals. As a result, table lookup involves neither loads from memory nor looping overhead. For small input vectors (on the order of 30 elements), this results in lookup performance superior even to that of a hash table.

3.1.3 Exponentiation

Another example in which it is beneficial to create code tailored to a given input set comes from work in computer graphics[4], where it is sometimes necessary to apply an exponent function to a large data set. Consider the following C program, which computes $base^{exp}$ by repeated multiplication and squaring:

```
double pow(double base, int exp)
{
    int t, bit = 1;
    int square = base;
    double result = 1.0;

    while (bit <= exp) {
        if (bit & exp)
            result *= square;
        bit = bit << 1;
        square *= square;
    }
    return result;
}
```

It is possible to do a lot better, however. Given a fixed exponent, we can unroll the main loop and the enclosed conditional, performing only the necessary multiplications:

```
typedef double (*dptr)();
dptr mkpow(int exp)
{
    /* 1 argument: the base */
    double vspec base = param(double, 0);
    /* 1 local: the running product */
    double vspec result = local(register double);

    /* Initialize the running product */
    void cspec squares = (1&exp) ? '{ result=base; } : '{ result=1.; };
    int bit = 2;
```

```
    /* Multiply some more, if necessary */
    while (bit <= exp) {
        squares = `{ squares; base *= base; };
        if (bit & exp)
            squares = `{ squares; result *= base; };
        bit = bit << 1;
    }

    /* Compile a function which returns the result */
    return compile(`{ squares; return result; }, double);
}
```

This function composes code specifications corresponding to the sequence of multiplications necessary for the given exponent value, invokes compile, and returns a function pointer to the resulting customized exponent function.

3.2 Specialization

'C makes it easy to specialize often-used routines to the kinds of inputs which they will be processing. We present two examples, a data-swapping routine customized to the size of its input, and a fast copy routine. These optimizations are similar in spirit to those in Section 3.1 on code construction. We include them in a separate section because they are somewhat less closely tied to their input data sets: rather than customizing a procedure to one particular subset of inputs (similarly to partial evaluation), they specialize a generic routine for improved performance on a broad class of inputs (depending, for example, on the inputs' size).

3.2.1 Swap

It is often necessary to swap the contents of two regions of memory. Most sorting algorithms (in particular, algorithms which sort in place), for example, move data around in this way. As long as the data being manipulated is no larger than a machine word, this process is quite efficient. However, when manipulating larger regions (for example, C structs), the code is often inefficient. One way to copy the regions is through a series of memory copy operations, such as these,

```
memcpy(tmp, src, size);
memcpy(src, dst, size);
memcpy(dst, tmp, size);
```

where src and dst point to source and destination memory regions respectively. This incurs large overhead in the library calls to the copy routine. Another method is to implement a loop which iterates as many times as necessary to cover the entire region to be swapped, swapping one word at a time. This incurs unnecessary loop overhead. 'C provides a solution by allowing us to create a customized swap routine specialized to the size of the region being swapped. The following code is an example, simplified to handle only the case where the size of the region is a multiple of sizeof(long):

```
typedef void (*fp)(void *, void *);

/* Generate swap routine; this is a toy example, so we assume
   size is a multiple of 4 */
fp mk_swap(int size) {
    /* Arg 0: source */
    long * vspec src = param(long *, 0);
    /* Arg 1: destination */
    long * vspec dst = param(long *, 1);
    /* Local for swaps */
    long vspec tmp = local(long);
```

```
    /* Code to be built up, initially empty;
    void cspec s = `{}
    int i;

    /* Generate a swap loop */
    for(i = 0; i <  size/sizeof(long); i++)
        s = `{
            s;
            tmp = src[$i];
            src[$i] = dst[$i];
            dst[$i] = tmp;
        };
    return (fp)compile(s, void);
}
```

This routine returns a pointer to a function which contains only assignments, swapping the region of the given size without resorting to looping or multiple calls to memcpy. The size of the generated code will usually be rather small and the performance benefits over traditional methods quite large, making this a very profitable optimization.

3.2.2 Copy

Copying an arbitrarily sized region of memory is another common operation. An important application of this is computer graphics [10]. Similarly to the previous code for swapping regions of memory, the following routine returns a function customized for copying a memory region of a given size:

```
typedef void (*fp)(void *, void *);

/* Generate a copy of a scan line; a real implementation would
   specialize based on alignment, size and shape.
   "n" is size of scan line;
   "unrollx" indicates how many times to unroll the copy loop
*/
fp mk_copy(int n, int unrollx) {
    int i, j;

    /* Arg 0: source */
    unsigned * vspec src = param(unsigned *, 0);
    /* Arg 1: destination */
    unsigned * vspec dst = param(unsigned *, 1);
    /* Local: loop counter */
    int vspec k = local(int);
    /* Code to be built up, initially empty;
    cspec void copy = `{}, unrollbody = `{};

    /* Unroll the remainder copy */
    for(i = 0; i < n % unrollx; i++)
        copy = `{ copy; dst[$i] = src[$i]; };

    /* Unroll copy loop unrollx times */
    if(n >= unrollx) {

        /* Generate a sequence of assignment statements to put in
           the body of the unrolled loop; number of assignments
```

```
        depends on how much we need to unroll loop */
      for (j = 0; j < unrollx; j ++)
          unrollbody = `{ unrollbody; dst[k+$j] = src[k+$j]; };

      /* Compose remainder copy with the main unrolled loop */
      copy = `{
          copy;                              /* remainder */
          for(k = $i; k < $n; k += $unrollx)
              unrollbody;                    /* unrolled loop */
      };
  }

  /* Compile this code and return a function pointer */
  return (fp)compile(copy, void);
}
```

The procedure mk_copy takes two arguments: the size of the regions to be copied, and the number of times that the inner copying loop should be unrolled. It creates a code specification for a function which takes two arguments, pointers to source and destination regions. It then specifies "prologue" code to copy regions which would not be copied by the unrolled loop (if n mod $unrollx \neq 0$), and generates the body of the unrolled loop. Finally, it composes these two code specifications, invokes compile, and returns a pointer to the resulting customized copy routine.

3.3 Run-time constants

Dynamic code generation can be used to hard-wire infrequently changing run-time values into the instruction stream, so that the values need not be loaded from memory. This is another instance of specialization, but often requires less thought than the code construction examples presented in Section 3.2. An example is a generic hash function, where the table size is determined at run time, and where the function uses a run-time value to help its hash. Consider the following C code:

```
/* Hash table entry structure */
struct hte {
    int val;            /* Key that entry is associated with */
    struct hte *next;   /* Pointer to next entry */
    /* ... */
};

/* Hash table structure */
struct ht {
    int scatter;        /* Value used to scatter keys */
    int norm;           /* Value used to normalize */
    struct hte **hte;   /* Vector of pointers to hash table entries */
};

/* A hash table pointer (ht) stores a pointer to entries in its
   associated table of hash entries (hte) along with a scatter value
   and size; hash returns a pointer to the hash table entry that
   matches val (or null, if there is no match) */
struct hte *hash(struct ht *ht, int val) {
    struct hte *hte;

    hte = ht->hte[(val * ht->scatter) / ht->norm];
    while (hte && hte->val != val)
        hte = hte->next;
```

```
    return hte;
}
```

The C function has three values that can be treated as run-time constants: ht→hte, ht→scatter, and ht→norm. The following ‘C code specializes the function for these values:

```
/* Type of the function generated by mk_hash: takes a value as input
   and produces a (possibly null) pointer to a hash table entry */
typedef struct hte *(*hptr)(int val);

/* Construct a hash function with the size, scatter, and hash table
   pointer hard–coded (the function must be regenerated when any of
   these values are changed) */
hptr mk_hash(struct ht *ht) {
    int vspec val = param(int, 0);
    void cspec code;

    code = ‘{
        struct hte *hte;
        hte = ($ht–>hte)[(val * $ht–>scatter) / $ht–>norm];
        while (hte && hte–>val != val)
            hte = hte–>next;
        return hte;
    };

    /* Compile and return the result */
    return compile(code, struct hte *);
}
```

This function can be specialized even further. For instance, an application could select from several different hash functions, depending on the characteristics of the input stream.

The ‘C version can be considerably faster than the equivalent C version, since a ‘C compiler can exploit the run-time constants hte, scatter, and norm both by hard-coding them directly in the instruction stream and by strength reducing the multiplication and division operations. Such optimizations are increasingly profitable on modern architectures, where cache misses are very expensive and division and multiplication are frequently provided in only software [6, 11]. This examples also illustrates that turning a C function into ‘C requires few changes. The cost of using the function generated with ‘C is an indirect jump on a function pointer.

3.4 Dynamic Function Call Construction

An important feature of ‘C is that it allows programmers to generate functions (and calls to functions) for which the number and types of arguments are not known at compile time. This feature does not just enable performance improvement, but provides previously unavailable functionality: ANSI C gives no mechanisms for constructing function calls dynamically.

3.4.1 A Simple Example

Function calls can be constructed dynamically by using the special forms push_init and push (see Section 2.4). Consider a function sum that takes a runtime-determined number of integer arguments, sums them, and returns the result. While it is possible to construct such a function using C’s variable argument facilities, or by requiring that the arguments be marshaled in an integer vector, such solutions are clumsy and inefficient. With vspecs one can easily construct the desired function:

```
/* Construct cspec to sum n integer arguments. */
```

```
void cspec construct_sum(int n) {
    int i, cspec c = `0;
    for (i = 0; i < n; i++) {
        /* Create a parameter */
        int vspec v = param(int, i);
        /* Add param 'v' to current sum. */
        c = `(c + v);
    }
    return `{ return c; };
}
```

To call sum, one initializes an "arument stack," denoted by an object of type void cspec, by invoking push_init. One then dynamically adds arguments to this list by calling push, as follows:

```
int cspec construct_call(int nargs, int *arg_vec) {
    void cspec args = push_init();      /* Initialize argument list */
    int i;
    for (i = 0; i < nargs; i++)         /* For each arg in arg_vec...   */
        push(args, `$arg_vec[i]);       /* push it onto the args stack */
    return `sum(args);
}
```

'C allows a void cspec (which represents the argument list) to be incorporated as a single argument in a call: 'sum(args) thus specifies code that calls sum using the argument list denoted by args.

3.4.2 Marshaling and Unmarshaling Byte Vectors

An important application of the 'C facilities for dynamic function call construction is the generation of code to marshal and unmarshal arguments stored in a byte vector. These operations are frequently performed to support remote procedure call [1]. Generating specialized code for the most active functions results in substantial performance benefits [12].

We present two functions, marshal and unmarshal, that dynamically generate marshaling and unmarshaling code (respectively) given a printf-style format string specifying the types of arguments.

The generation of marshaling code relies on 'C's ability to specify arbitrary numbers of incoming parameters. The simplified code fragment below generates a marshaling function for arguments with a particular set of types (in this example, INT, POINTER, and DOUBLE):

```
typedef union {
    int i;
    double d;
    void *p;
} type_t;

/* The types of arguments that mk_marshal can marshal */
typedef enum { INTEGER, DOUBLE, POINTER } type_t;

void cspec mk_marshal(type_t *types, int nargs) {

    /* Allocate the vector into which we will marshal arguments */
    type_t * vspec m = local(sizeof(type_t) * $nargs, type_t *);

    /* Initialize the code sequence */
    void cspec s = `{};
    int i;
```

```
    /* Add code which marshals parameters into the result vector */
    for (i = 0; i < nargs; i++) {
        switch(types[i]) {
        case INTEGER:  s = `{ s; m[$i].i = param($i, int); };          break;
        case DOUBLE:   s = `{ s; m[$i].d = param($i, double); };   break;
        case POINTER:  s = `{ s; m[$i].p = param($i, void *); };   break;
        }
    }

    /* Return code which marshals parameters and returns the result vector */
    return `{ s; return m; };
}
```

The code works as follows. First, it allocates storage on the stack for a byte vector large enough to hold the arguments specified by the type format vector. Then, for every type in the type vector, it (1) creates a vspec that refers to the corresponding parameter, (2) constructs code to store the parameter's value into the byte vector at the current offset, and (3) adds the size of the type to the offset. Finally, it specifies code to call a function pointer with the marshaled arguments as an argument. After dynamic code generation the function that has been constructed will store all of its parameters at fixed, non-overlapping offsets into a stack-allocated memory block. Since all type and offset computations have been done during specification time, the generated code will be efficient. Further performance gains could be achieved if the code were to manage endianness, alignment, and so forth.

Dynamic generation of unmarshaling code is equally profitable. The process relies on our mechanisms for constructing calls to arbitrary functions at run time. The ability to invoke arbitrary functions is not just useful for efficiency: it also provides useful functionality. For example, in Tcl [9] the run-time system can make upcalls into an application. However, because Tcl cannot dynamically create code to call an arbitrary function, it marshals all of the upcall arguments into a single byte vector, and forces applications to explicitly unmarshal them. If systems such as Tcl used `C to construct upcalls, clients would be able to write their code as normal C routines, which would increase the ease of expression and decrease the chance for errors.

The code below generates the unmarshaling function to work with the marshaling code given above:

```
typedef int (*fptr)(); /* Type of the function we will be calling */
void cspec mk_unmarshal(type_t *types, int nargs) {

    /* Argument 0 is the function we must invoke */
    fptr vspec fp = param(fptr, 0);

    /* Argument 1 is the vector we are unmarshaling */
    type_t * vspec m = param(type_t *, 1);

    /* Initialize the dynamic argument list */
    void cspec args = push_init();
    int i;

    /* Build up the dynamic argument list */
    for(i = 0; i < nargs; i++) {
        switch(types[i]) {
        case INTEGER:  push(args, `m[$i].i); break;
        case DOUBLE:   push(args, `m[$i].d); break;
        case POINTER:  push(args, `m[$i].p); break;
        }
    }

    /* Return code spec to call the given function with unmarshaled args */
    return `{ fp(args); };
```

```
}
```

The generated code takes a function pointer as its first argument and a byte vector of marshaled arguments as its second. It unmarshals the values in the byte vector into their appropriate parameter positions, and then invokes the function pointer. The code is generated as follows. First, the parameter variables for the generated function's incoming arguments are created. Second, the argument list is initialized. Then, for every type in the type vector, mk_unmarshal (1) creates a cspec to index into the byte vector at a fixed offset, (2) pushes this cspec into its correct parameter position, and (3) adds the size of the unmarshaled element to the current offset. Finally, the call to the vspec function pointer with the constructed argument list is specified, and the result is compiled.

3.5 Parameterized Functions

Many library routines are parameterized via function pointers. For instance, the standard C library provides quicksort and heapsort, which accept user-defined routines for performing comparisons. Many mathematical libraries also provide support for solving generic functions in the same way. Unfortunately, indirect function calls can eliminate many potential optimizations, since the function cannot be integrated with the library code. By using ‘C to compose cspecs instead of function pointers, library functions can be parameterized easily and efficiently, since cspecs can be integrated directly into ‘C code without extra function calls.

The ‘C code for Newton's method [2] illustrates ‘C's advantages for parameterized functions. The function newton takes as arguments the maximum allowed number of iterations, a tolerance, an initial estimate, and two pointers to functions that return cspecs to evaluate a function and its derivative. Using ‘C's features for code composition, the cspecs returned by these functions are incorporated directly into the dynamically generated code, eliminating function call overhead and allowing inter-cspec optimization to occur at instantiation time.

```
/* Pointer to a function from double vspec to double cspec */
typedef double cspec (*dptr)(double vspec);

/* Dynamically create a Newton-Raphson routine:
   "n" is the number of allowed iterations;
   "tol" is the maximum allowed tolerance; "p0" is the initial estimate;
   "f" is the function to solve and "fprime" is its derivative */
double newton(int n, double tol, double usr_p0, dptr f, dptr fprime) {
    void cspec cs = `{
        int i;
        double p, p0;

        p0 = usr_p0;
        for (i = 0; i < $n; i++) {

            /* Incorporate the cspecs returned by f and fprime and
               use them to calculate the next point in the series */
            p = p0 - f(p0) / fprime(p0);

            /* When (if?) we converge sufficiently, return result */
            if (abs(p - p0) < tol)
                return p;

            /* Seed the next iteration. */
            p0 = p;
        }
        error("method failed after %d iterations\n", i);
    };

  /* Compile, call, and return the result. */
```

```
  return (*compile(cs,double))();
}
```

Note that in the calls f(p0) and fprime(p0), p0 is sent as a vspec argument at instantiation time. The example below illustrates how to use newton to find the root of the function $f(x) = (x + 1)^2$:

```
/* Function that constructs a cspec to compute f(x) = (x+1)^2 */
double cspec f(double vspec x) {
    return '((x + 1.0) * (x + 1.0));
}

/* Function that constructs a cspec to calculate f'(x) = 2(x+1) */
double cspec fprime(double vspec x) {
    return '(2.0 * (x + 1.0));
}

/* Call newton to solve an equation */
void use_newton(void) {
    printf("Root is %f\n", newton(100, .000001, 10., f, fprime));
}
```

3.6 Function Composition

In a manner similar to function parameterization, 'C also allows modular function *composition*. Inexpensive function composition has many applications; an important one is the optimization of networking code.

The modular composition of different protocol layers has long been a goal in the networking community [3]. Unfortunately, each protocol layer frequently has data-touching operations associated with it (e.g., computing a checksum, byte-swapping, etc.). As a result, as data moves through each layer, data can be touched multiple times, which is expensive [3].

'C can be used to construct a network subsystem that solves this problem by dynamically integrating protocol data operations into a single pass over memory (e.g., by integrating encryption and compression into a single copy operation). A simple implementation of such a system divides each data manipulation state into *pipes* that each consume a single input and produce a single output. These pipes can then be composed and incorporated into a data copying loop. The design includes the ability to specify prologue and epilogue code that is executed, respectively, before and after the data copying loop. This allows the pipes to manipulate state and make end-to-end checks, such as ensuring that a checksum is valid after it has been computed.

For instance, the following pipe specification can be used to do byte-swapping. Since a byte swapper does not need to maintain any state, there is no need to specify initial and final code: we simply need to specify the "consumer" routine which manipulates the data.

```
/* Example pipe: construct a cspec for code that, given an input,
   produces a byte-swapped output */
unsigned cspec byteswap(unsigned vspec input) {
    return '((input << 24) |
             ((input & 0xff00) << 8) |
             ((input >> 8) & 0xff00) |
             ((input >> 24) & 0xff));
}
/* Byteswap maintains no state and so needs no initial or final code */
void cspec byteswap_initial(void) { return '{}; }
void cspec byteswap_final(void) { return '{}; }
```

To construct the integrated data-copying routine, the initial, consumer, and final code specifications of each pipe are composed with the corresponding specifications of the pipe's neighbors. The composed initial code is placed at the beginning of the resulting routine, the consumer code is inserted in a loop and composed with code which provides it with input and stores its output, and the final code is placed at the end of the routine. A simplified code fragment is provided below.

```
/* Pointer to function that returns a void cspec */
typedef void cspec (*vptr)();

/* Pointer to function that returns an unsigned cspec */
typedef unsigned cspec (*uptr)(unsigned cspec);

/* Pipe structure: contains pointers to functions that return cspecs that
   specify the initialization, pipe, and finalization code for each pipe */
struct pipe {
     vptr initial;          /* initial code */
     uptr pipe;             /* pipe */
     vptr final;            /* final code */
};

/* Return cspec that results from composing the given vector of
   pipe–tuples. Creates a function that given an input pointer, an
   output pointer and the number of words, will apply the pipe
   expression to each word as it copies the data. */
void cspec compose(struct pipe *plist, int n) {
     struct pipe *p;

     /* Arg 0: size of input */
     int vspec nwords = param(int, 0);
     /* Arg 1: pointer to memory that provides pipe input */
     unsigned * vspec input = param(unsigned *, 1);
     /* Arg 2: pointer memory that accepts pipe output */
     unsigned * vspec output = param(unsigned *, 2);
     /* Local: loop counter */
     int vspec i = local(int);

     /* Initial prologue and epilogue code */
     void cspec initial_stmts = ‘{};
     void cspec final_stmts = ‘{};
     /* Base pipe input */
     unsigned cspec pipes = ‘input[i];

     /* Compose all stages together */
     for (p = &plist[0]; p < &plist[n]; p++) {

          /* Compose initial statements */
          initial_stmts = ‘{ initial_stmts; p–>initial(); };

          /* Compose pipes: the previous pipe's output is the next pipe's
             input. In the generated code, pipes are *not* function calls,
             but simply arithmetic expressions occurring in a given order */
          pipes =‘p–>pipe(pipes);
```

```
        /* Compose final statements */
        final_stmts = `{ final_stmts; p->final(); };
    }

    /* Create a function with initial statements first, consumer statements
       second, and final statements last. */
    return `{
        initial_stmts;
        /* loop over input, piping through consumer. */
        for (i = 0; i < nwords; i++)
            output[i] = pipes;
        final_stmts;
    };
}
```

In a mature implementation, we could further improve performance by unrolling the data-copying loop. Additionally, pipes would take inputs and outputs of different sizes (or "gauges") that the composition function would have to reconcile.

3.7 Small Language Compilation

There are myriads of small, primitive languages that are both performance-critical and amenable to dynamic compilation. The small query languages used to query databases are well-known targets for dynamic code generation [7]; since databases are large, dynamically compiled queries will usually be applied many times, easily amortizing the cost of dynamic code generation.

We provide a toy example below. The function mk_query takes a vector of queries. Each query specifies (1) a database record field (*i.e.*, CHILDREN or INCOME), (2) a value to compare this field to, and (3) the operation to use in the comparison (*i.e.*, $<$, $>$, etc.). Given a query vector, mk_query dynamically creates a query function which takes a database record as an argument and checks whether that record satisfies all of the constraints in the query vector. This check is implemented simply as an expression which computes the conjunction of the given constraints. The query function never references the query vector, since all the values and comparison operations in the vector have been hard-coded into the dynamic code's instruction stream.

```
/* Constants to indicate query type */
typedef enum { INCOME, CHILDREN /* ... */ } query;
/* Constants to indicate comparison operation for query */
typedef enum { LT, LE, GT, GE, NE, EQ } bool_op;

/* Query */
struct query {
    query record_field;        /* Which field to use */
    unsigned val;              /* Value to compare to */
    bool_op bool_op;           /* Operation used to compare */
};

/* Simple database record */
struct record {
    int income;
    int children;
    /* ... */
};

/* Type of the function generated by mk_query: takes a pointer to a
   database record as its sole argument and returns 0 or 1 depending
```

```
   on whether the record matched the query */
typedef int (*iptr)(struct record *r);

/* Compile the given query by constructing a boolean expression built
   up from its specified predicates.
   A predicate is a tuple (field, val, bool_op):
       "field" indicates a record field;
       "val" is the value to compare it to;
       "bool_op" is the operation used in the comparision (<, >, etc.)  */
iptr mk_query(struct query *q, int n) {
    int cspec field, cspec expr, i;

    /* R is a pointer to the record to compare to. */
    struct record * vspec r = param(struct record *, 0);

    /* Initialize the boolean expression */
    expr = `1;

    /* Build the rest of the boolean expression */
    for (i = 0; i < n; i++) {

        /* Load the appropriate field value */
        switch (q[i].record_field) {
        case INCOME:   field = `(r->income); break;
        case CHILDREN: field = `(r->children); break;
            /* ... */
        }

        /* Compare the field value to the runtime constant in
           q[i] using the given boolean operation; concatenate this
           comparison to the rest of the boolean expression */
        switch (q[i].bool_op) {
        case LT:
            expr = `(expr && field < $q[i].val);
            break;
        case EQ:
            expr = `(expr && field == $q[i].val);
            break;
        case LE:
            expr = `(expr && field <= $q[i].val);
            break;
            /* ... */
        }
    }
    return (iptr)compile(`{ return expr; }, int);
}
```

This code first specifies one incoming argument, the database record to be compared. It then "seeds" the boolean expression: since we are building a conjunction, the beginning value is 1. The loop then traverses the query vector, building up the dynamic code for the conjunction according to the fields, values, and comparison operations specified in the vector. When the specification for the boolean expression is finally constructed, mk_query compiles it and returns a function pointer. This can then be applied to database entries to efficiently determine whether they match the given constraints.

3.8 Compiling Interpreters

Compiling interpreters are important pieces of technology: they combine the flexibility of an interpreted programming environment with the performance of compiled code. For a given piece of code, the user of a compiling interpreter pays a one-time cost of compilation, which can be roughly comparable to that of interpretation. Every subsequent use of that piece of code will employ the compiled version, greatly improving performance over a traditional interpreter.

Compiling interpreters are also useful for languages used for writing "shippable" code, such as Java. The user may not want to pay the cost of compiling in its entirety every piece of code that she downloads. At the same time, simply interpreting the bytecode does not provide adequate performance. A compiling interpreter is the ideal solution in this case, providing an "on-demand," or "lazy," compilation model.

By its very nature, ‘C makes writing compiling interpreters both easy and efficient. We present here a simple example:

```
/* Multi-character tokens of our toy language */
enum { WHILE, IF, ELSE, ID, CONST, LE, GE, NE, EQ };

int expect();           /* Consume the given token from the input stream,
                           or fail if not found */
int cspec expr();        /* Parse unary expressions
int cspec expr1();       /* Parse binary expressions */

int gettok();           /* Consume a token from the input stream */
int puttok();           /* Put a token back on the input stream */
int look();             /* Peek at the next token without consuming it */
int cur_tok;            /* Current token */

int vspec lookup_sym(); /* Given a token, return corresponding vspec */
int (*lookup_fptr())(); /* Given a token, return corresp. function pointer */

void cspec stmt() {
    int cspec e = ‘0;
    void cspec s = ‘{}, s1 = ‘{}, s2 = ‘{};

    switch(gettok()) {
    case WHILE:              /* 'while' '(' expr ')' stmt */
        expect('('); e = expr(); expect(')'); s = stmt();
        return ‘{ while(e) s; };
    case IF:                 /* 'if' '(' expr ')' stmt { 'else' stmt }? */
        expect('('); e = expr(); expect(')'); s1 = stmt();
        if(look(ELSE)) {
            gettok();        /* Consume token */
            s2 = stmt();
            return ‘{ if(e) s1; else s2; };
        } else
            return ‘{ if(e) s1; };
    case '{':                /* ‘{' stmt* ‘}' */
        while(!look('}'))
            s = ‘{ s; stmt(); };
        return s;
    case ';': return ‘{};
    case ID:                 /* ID ‘=' expr ‘;' */ {
        int vspec lvalue = lookup_sym(cur_tok);
        expect('='); e = expr(); expect(';');
        return ‘{ lvalue = e; };
```

```
    }
    default: parse_err("expecting statement");
    }
}

/* Parse unary-expressions and '(' expr ')' */
int cspec expr() {
    switch(gettok()) {
    case CONST:     return expr1(`$atoi(cur_tok));
    case '+':       return expr();
    case '-':       return `-expr();
    case '!':       return `!expr();
    case '(':       { int cspec e = expr(); expect(')'); return e; }
    case ID :       /* ID or function call */
        if(!look('(')) return expr1(`lookup_sym(cur_tok)) ;
        else {
            int (*ip)() = lookup_fptr(cur_tok);
            void cspec args = push_init();
            gettok();       /* Consume '(' */
            /* epsilon ')' */
            if(look(')')) {
                gettok();   /* Consume ')' */
                return `$ip();
            }
            while(1) {      /* arg { ',' arg }*  ')' */
                args = push(args, `expr());
                if(look(',')) gettok();
                else if(look(')')) break;
                else parse_err("malformed arg list");
            }
            gettok();       /* consume '(' */
            return `$(*ip)(args);
        }
    default: parse_err("bogus expr");
    }
}

/* Parse binary expressions */
int cspec expr1(int cspec e) {
    switch(gettok()) {
    case '+':  return `(e +  expr());
    case '-':  return `(e -  expr());
    case '*':  return `(e *  expr());
    case '/':  return `(e /  expr());
    case '<':  return `(e <  expr());
    case LE:   return `(e <= expr());
    case '>':  return `(e >  expr());
    case GE:   return `(e >= expr());
    case NE:   return `(e != expr());
    case EQ:   return `(e == expr());
    case ')':
    case ';':  puttok(); return e;
    default: parse_err("bogus expr");
    }
```

```
}
```

4 Conclusion

This tutorial has presented the ‘C language as it is currently defined. ‘C, however, is slowly but continuously evolving, as we acquire experience with it and find ways of making it more useful and easier to use.

For example, the original language definition included an explicit operator for incorporating cspecs into the bodies of other cspecs. After using the language, however, we realized that this was both verbose and unnecessary, and so developed the automatic type conversion mechanism in Section 2.2.3. Some of the special forms also gradually evolved with use.

Similar changes will probably continue to occur. In particular, there are two directions in which we expect to make changes. First, we plan to create a mechanism similar to LISP's lambda for naming the arguments of dynamic functions. This should be considerably more readable than the current param mechanism, and could supplant it in all cases but those requiring functions with dynamic numbers of arguments.

The second likely change is a differentiation between dynamic expressions and dynamic statements. As described in Section 2, the evaluation type of a ‘ expression containing a C expression is simply the type of that expression, whereas the evaluation type of a ‘ expression containing a compound statement is always void, even when the compound statement contains a non-void return statement. Consequently, it is possible to dynamically compose a function with two return statements that have incompatible types. The static compiler cannot in general detect this, the run-time system ignores the issue for efficiency purposes, and the user is unaware of the problem until strange bugs start occurring. Differentiating expressions and statements, and making the evaluation type of a dynamic compound statement correspond to the type of the return statements, if any, that it contains, would solve this problem.

In the meantime, though, ‘C remains as described in this document. It is implemented by version 1.0 of the tcc compiler.

References

[1] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] R.L. Burden and J.D. Faires. *Numerical Methods*. PWS-kent Publishing Company, Boston, MA, fourth edition, 1989.

[3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, September 1990.

[4] S. Draves. Lightweight languages for interactive graphics. Thesis proposal, June 1995.

[5] S.P. Harbison and G.L. Steele Jr. *C, A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.

[6] SPARC International. *The SPARC Architecture Manual Verson 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[7] D. Keppel, S.J. Eggers, and R.R. Henry. Evaluating runtime-compiled value-specific optimizations. TR 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Ed.* Prentice Hall Software Series, 1988.

[9] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[10] R. Pike, B.N. Locanthi, and J.F. Reiser. Hardware/software trade-offs for bitmap graphics on the Blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[11] R. L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2), February 1993.

[12] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

A ‘C Grammar

The grammar for ‘C consists of the grammar specified in Harbison and Steele's C reference manual [5] with the following additions:

unary-expression :
  backquote-expression
  dollar-expression

backquote-expression :
  ‘ *unary-expression*
  ‘ *compound-statement*

dollar-expression :
  $ *unary-expression*

pointer :
  cspec type-qualifier-list$_{opt}$
  vspec type-qualifier-list$_{opt}$
  cspec type-qualifier-list$_{opt}$ *pointer*
  vspec type-qualifier-list$_{opt}$ *pointer*