DRAFT: DO NOT DISTRIBUTE
‘C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation

Dawson R. Engler Wilson C. Hsieh M. Frans Kaashoek

Abstract

Dynamic code generation allows specialized code sequences to be crafted using runtime information. Since this information is by definition not available statically, the use of dynamic code generation can achieve performance inherently beyond that of static code generation. Previous attempts to support dynamic code generation have been low-level, expensive, or machine-dependent. We describe ‘C (*backquote C*), a superset of ANSI C, that allows high-level, efficient, and machine-independent specification of dynamically generated code. Our experiments with a prototype compiler show that ‘C enables excellent performance improvement (in some cases, more than an order of magnitude) and easy specification of dynamically generated code.

1 Introduction

Dynamic code generation (i.e., generation of executable code at *runtime*) allows the use of runtime information to improve code generation. For instance, the propagation of runtime constants may be used to feed optimizations such as strength-reduction, dead-code elimination, and constant folding. As another example, interpreters can compile frequently used code and execute it directly [2]; this technique can improve performance by an order of magnitude, compared to even heavily tuned interpreters [5].

Unfortunately, current dynamic code generation systems are not satisfactory. Programmers must tradeoff among portability, ease of programming (including debugging), and efficiency: efficiency can be had, but only by sacrificing portability, ease of programming or, in the case of the fastest dynamic code generators [9], both. We attack all three of these problems by adding support for dynamic code generation directly to ANSI C: portability and ease of programming are achieved through the use of high-level, machine-independent specifications; efficiency is achieved from static typing, which allows the bulk of dynamic code generation costs to be paid at compile time. We call our language ‘C (backquote-C).

‘C grew out of our previous work with `dcg` [5], an efficient, retargetable dynamic code generation system. ‘C offers several improvements over `dcg`, but retains `dcg`'s portability and flexibility. First, ‘C provides a high-level interface for code specification, whereas `dcg`'s interface is based on the intermediate representation of `lcc` [6]. Second, it provides the opportunity for static analysis, which reduces the cost of dynamic compilation; because it has no compiler support, `dcg` must do runtime analysis. Finally, by making dynamic code generation a first-class capability of a high-level language, both profiling and debugging facilities can be added.

{`engler|wchsieh|kaashoek`}`@lcs.mit.edu`, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 617-253-7436. This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985; the last author is partially supported by an NSF National Young Investigator award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

This paper describes ‘C, a superset of ANSI C that provides support for specifying dynamically generated code. The contributions of this work are twofold: we have designed an efficient, flexible, high-level means of specifying dynamically generated code in ANSI C, a statically-typed language; and we have mostly completed a prototype ‘C compiler that allows us to demonstrate how ‘C simplifies the generation of dynamic code. We describe the ‘C language and library in Section 2. We give two examples programs in Section 3, and show how our prototype implementation performs on these examples in Section 4. We outline a more efficient implementation in Section 5; we discuss related work in Section 6; and we summarize our results in Section 7. The ‘C grammar is given in Appendix A.

2 ‘C Language and Library

Three conflicting goals have driven our design:

1. ‘C must be a clean extension to ANSI C, both syntactically and semantically. Extensions must not affect the syntax and semantics of ANSI C, and should be in the spirit of the ANSI C language.

2. ‘C must allow flexible specification of dynamically generated code. For instance, it must allow the construction of functions whose parameters are unknown (both in number and in type) at static compile-time.

3. ‘C must allow an efficient implementation. The most important effect of this goal is that the majority of code generation costs must be paid at compile-time.

Designing a language that can satisfy all three goals is a difficult problem. For example, one issue that we faced was deciding between a dynamic or a static type system for dynamically generated code. A dynamic type system aids in the flexible specification of dynamically generated code, as it adds a form of polymorphism to the language. However, a static type system is more efficient, since it allows the bulk of instruction selection and optimization to occur at compile-time; in addition, it is also more in the spirit of ANSI C.

We chose to use a static type system for ‘C. The resulting loss in flexibility is minor, especially since ANSI C does not provide any mechanisms for polymorphism. The gain in performance should be large: the information provided by a static type system allows us to push more dynamic code generation costs to compile-time. As a result, we believe an aggressive compiler for ‘C should achieve performance close to that of hand-crafted dynamic code generators.

2.1 Language Modifications for ‘C

We use the term *dynamic code* to mean dynamically generated code; we use *static code* to mean all other code. In ‘C dynamic code is *specified* at runtime; these specifications can then be either *composed* to build larger specifications or *instantiated* (compiled at runtime) to produce executable code. We use the term *compile-time* to mean static compile-time, *specification-time* to mean when code is being specified and composed, *instantiation-time* to mean when dynamic is compiled, and *runtime* to mean when dynamic code is executed. Dynamic code is constructed one function at a time; each invocation of the library function `compile` ends one function definition. This restriction reduces bookkeeping complications that would otherwise be necessary in the language.

We have added two type constructors and three unary operators to ANSI C. The two new type constructors, `cspec` and `vspec`, are both postfix-declared types (similar to pointers). A `cspec` or `vspec` has an associated *evaluation type*, which is analogous to the type to which a pointer points; the evaluation type allows to dynamic code to be statically typed, which in turn allows us to select instructions at compile-time.

The new unary operators, `` ` ``, `@`, and `$`, have the same precedence as the standard unary prefix operators.

2.1.1 The ` Operator

Dynamically generated code is specified using the ` (backquote) operator. [1] ` can be applied to an expression or a compound statement. However, backquote expressions may not nest (i.e., we do not allow the specification of dynamic code that in turn may specify dynamic code). Some simple usages of backquote are:

```
`4          /* specification of dynamic code for the integer 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); }
```

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 leads to undefined behavior; in other words, only downward funargs are legal. Lexical scoping allows type-checking and instruction selection to occur at compile-time.

The use of several C constructs is restricted within backquote expressions. In particular, `break`, `continue`, `case` and `goto` cannot be used to transfer control outside of the expression. For example, the destination label of a `goto` statement must be contained in the backquote expression. This restriction is present so that a `C compiler can statically determine that a control flow change is legal. The use of `return` is not similarly restricted because dynamic code is always associated with a function.

2.1.2 `cspec` Types

The type of a dynamic code specification is a `cspec` type (for *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`. By statically typing dynamic code specifications, we can type-check code composition (described in Section 2.1.3) statically.

Applying ` to a compound statement yields a result of type `void cspec`. We also use the type `void cspec` as 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 promotion rules of ANSI C apply.

`cspec`s can be compiled using the `compile` function. `compile` returns a `void` function pointer, which can then be cast to the appropriate type. The following code fragment dynamically constructs and instantiates a traditional greeting:

[1] This usage of ` is based on Lisp's usage of ` for specifying list templates.

```
void cspec c = `{ printf("hello world\n"); };
/* Compile and call the result; the "v" indicates that the return type is void. */
compile(c, "v")();
```

2.1.3 The `@` Operator

The `@` operator allows dynamic code specifications to be combined into larger specifications. `@` can only be applied inside a backquote; legal operands must be `cspec`'s or `vspec`'s (`vspec`'s are described in Section 2.1.4), and are evaluated at specification-time. `@` "dereferences" `cspec`'s and `vspec`'s: it returns an object whose type is the evaluation type of `@`'s operand. The returned object is incorporated into the `cspec` in which the `@` occurs. For example, in the following fragment, `c` is the additive composition of two `cspec`'s:

```
/* Compose c1 and c2; evaluation yields '9'. */
int cspec c1 = `4, c2 = `5, c = `(@c1 + @c2);
```

Statements can be composed through concatenation:

```
/* Concatenate two null statements. */
void cspec s1 = `{}, s2 = `{}, s = `{ @s1; @s2; };
```

2.1.4 `vspec` Types

An object of type `vspec` (for *variable specification*) represents a dynamically generated lvalue; its evaluation type is the type of the lvalue. `Vspec`'s allow lvalues to be statically typed, so that we can perform instruction selection at compile-time. Objects of type `vspec` may be initialized by calling the 'C library functions `param` or `local`. `Param` is used to create a parameter for the function under construction; `local` is used to reserve space in its activation record (or allocate a register if possible).

The `@` operator is used to incorporate `vspec`'s into `cspec`'s; a `vspec` (like a variable in ANSI C) can be used both as an lvalue and an rvalue. The following function returns code that takes a single integer argument, adds one to it, and returns the result:

```
void cspec plus1(void) {
    /* The arguments to param are the argument index and its type. */
    int vspec i = (int vspec) param(0, "i");
    return `{ return @i + 1; };
}
```

`Vspec`'s allow us to construct functions that take a runtime-determined number of arguments; this power is necessary in applications such as our compiling interpreter (described in Section 3.2). 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 `vspec`'s we can easily construct the desired function:[2]

```
/* Construct function to sum n integer arguments. */
void cspec construct_sum(int n) {
```

[2] As written, this function does not work in our implementation: due to limitations of `dcg`, our compiler target, parameter allocation must be factored into a separate loop.

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

2.1.5 The $ Operator

The `$` operator is used to specialize dynamic code. `$` may be applied to expressions within dynamic code that are not of type `cspec` or `vspec`. The operand of `$` is evaluated at specification-time, and the resulting value is incorporated as a *runtime constant*. This is illustrated in the code fragment below.

```
/* Demonstrate use of $. */
int cspec c1, c2;
void cspec c;
int x = 1;

/* Bind x as a runtime constant; i.e., its value at this point. */
c1 = `$x;
/* Bind x at runtime; i.e., use value in x when the code for c2 is run. */
c2 = `x;
c = `{ printf("$x = %d, x = %d\n", @c1, @c2); };
x = 14;
/* Compile and run: will print "$x = 1, x = 14". */
compile(c, "v")();
```

2.2 The ‘C Standard Library

We provide a small library in order to minimize changes to ANSI C; for example, we do not want to change ANSI C to allow the construction of function calls at runtime.

2.2.1 Type Specifications

A number of functions in the ‘C standard library expect types (e.g., `local`, `param`, `push` and `compile`). This type information specified through a rudimentary string specification (similar to that of `printf`). Built-in types are specified by the first letter(s) of their type. For example, `unsigned short` is specified by "`us`". All pointers are represented by "`*`": operationally, they are treated as `void *` pointers for purposes of storage and register allocation. Aggregates (arrays and structures) are specified by a single "`A`"; their size must be given as an additional argument. Finally, the character "`r`" can be appended to the type strings to indicate that the allocated object will not have its address taken. While information about addressing could be derived at runtime, doing so quickly would add complexity to the code generator.

2.2.2 Library Description

`void (*compile(void cspec code, char *type-spec [, int type-size]))(void)`

`compile` generates machine code from `code`. `type-spec` gives the return type of generated function; `type-size` is an optional argument used to give the size of aggregate types. In the future, `compile` will take a number of flags relating to optimization, debugging and profiling.

`void vspec local(char *type-spec [, int type-size])` `local` returns a `vspec` to access a local variable of type `type-spec`, and reserves space in the activation record of the function currently being specified.

`void vspec param(int param-num, char *type-spec [, int type-size])`
: `param` returns a `vspec` to access a parameter of type `type-spec` and number `param-num`.

`void cspec push(void cspec args, char *type-spec, void cspec code-spec [, int type-size])`
: `push` returns a `cspec` that pushes `code-spec` onto the call stack as an argument.

`void cspec jump(void cspec target)`
: `jump` returns the cspec of a jump to `target`. This is useful for constructing "hard-coded" finite-state machines.

`void (*self(void))(void)`
: `self` returns a pointer to the function that the next invocation of `compile` will return. This allows the construction of recursive calls.

2.2.3 Runtime-constructed function calls

Function calls can be constructed "on the fly" by using the library function `push`; it takes a `cspec` of the argument list constructed so far, the type of the next argument, a `cspec` of that argument, and possibly the size of the argument.

Consider our function `sum`, which sums a runtime-determined number of integer arguments. To call `sum` we construct the argument list at runtime by using `push`. In addition, we allow a `void cspec` (which we assume represents an argument list) to be incorporated as a single argument in a call: `` `sum(@args) `` specifies code that calls `sum` using the argument list specified by `args`:

```
int cspec construct_call(int nargs, int *arg_vec) {
    void cspec args = `{};      /* initialization */
    int i;
    /* For each argument in arg_vec, */
    for(i=0; i < nargs; i++)
        /* create cspec that pushes it on call stack. */
        args = push(args, "i", `$arg_vec[i]);
    return `sum(@args);
}
```

2.3 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 runtime system. Hence, `cspec`'s 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 controlled by the programmer (e.g., by using `malloc`).

In addition, there is some runtime checking in 'C that is not present in ANSI C: for example, the compiler must guard against conflicting parameter definitions. Currently, the return type of the dynamically constructed functions is specified dynamically as well (there seems to be no good way to do it statically); this requires that checking and promotion of return types must be done at runtime. This is inconvenient, but localized.

Other design decisions will be described in more detail in the full paper. In addition, we will discuss deallocation of 'C objects.

```
/* Construct code to scale matrix m of size n by constant 's'. */
void cspec mkscale(int s, int n, int **m) {
   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 */
            v[j] = v[j] * $s;
      }
   };
}
```

Figure 1: ‘C code for scaling a matrix by a runtime constant

3 Examples

To illustrate how ‘C makes dynamic code generation easy to use, we present two examples.

3.1 Matrix Scaling

Scaling a matrix by a runtime constant allows ample opportunity for speedup from the use of dynamic code generation. For instance, multiplication can be strength-reduced to shifts and adds; division can be strength-reduced to multiplication (and then to shifts and adds). Additionally, loop bounds can be encoded in branch checks as constants, which can alleviate register pressure. The ‘C code for expressing matrix scaling by a runtime constant is shown in Figure 1. This example can be viewed as programmer-expressed partial evaluation.

3.2 Compiling Interpreter

Interpreters can use dynamic code generation technology to improve performance by compiling and then directly executing frequently interpreted pieces of code [2, 4]. To show that ‘C can be used to do this easily and efficiently we present a recursive-descent compiling interpreter that accepts a subset of C that we call Tiny C [5]. Tiny C has only an integer type; it supports most of C's relational and arithmetic operations (`/`, `-`, <, etc.) and provides `if` statements, `while` loops, and function calls as control constructs.

A subset of the parser is shown in Figure 2. What should be noted is the degree to which the flexibility of ‘C is exercised: functions having an arbitrary number of parameters and local variables can be created, and code is specified and composed in a diffuse fashion. Without the flexibility afforded by ‘C, this example would be difficult to write and inefficient in its generated code. In addition, our experience has been that specifying dynamically generated code in ‘C is easier then constructing an efficient interpreter.

4 Implementation

We have completed most of a prototype ‘C compiler that emits ANSI C code augmented with calls to `dcg`'s [5] dynamic code generation primitives. Our compiler correctly generates code for a subset of legal ‘C; it compiles all of the examples in this paper except one, which is due to a restriction of `dcg`.

Our prototype allows us to demonstrate the ease of using 'C to specify dynamic code generation. In addition, despite our lack of optimization and `dcg`'s rudimentary optimization (it does not perform instruction scheduling nor peephole optimization) we still achieve good performance. In Section 5 we describe a future implementation that will give better performance than our prototype.

In the rest of this section we present performance results for the code generated by our prototype compiler. We ran our compiler on the examples in Section 3. Our experiments were conducted on a SPARC 10 system that does integer divide and multiply in software. Times were measured using the Unix system call `getrusage` and include both "user" and "system" time. The times given are the median time of three trials. Static compilation was done using gcc versions 2.5.8.

4.1 Matrix Scaling

We compare a static matrix scaling routine against the 'C routine given above. We run two experiments, one for division, the other for multiplication. Multiplication is done on a matrix of type `int`; division on a matrix of type `short`. The experimental times given in Figure 3 measure the summation of the time required to scale 1024x1024 matrix by the integers 1 through 1024; in the 'C implementation we include the time to generate the code at runtime.

The performance of multiplying a 1024x1024 integer matrix by a runtime constant improved by a factor of 3. The performance of dividing a 1024x1024 matrix of type `short` by a runtime constant improved by a factor of 35%.[3] More dramatic improvements would be possible with a more sophisticated factorization scheme for strength-reducing division.

4.2 Compiling Interpreter

A recursive Fibonacci program is used to measure the performance of four implementations of Tiny C (which was described in Section 3.2).

Tree-interpreter: We wrote a simple interpreter that translates Tiny C into abstract syntax trees, which it then recursively evaluates.

gcc -O2: We compile Tiny C using gcc with optimization level "-O2". This gives an upper bound on the quality of local code.

'C: We run the 'C compiling interpreter described in Section 3.2.

Our test computes the 30$^{\text{th}}$ Fibonacci number; Figure 4 summarizes the results. The code that is generated using our simple backend is fairly efficient: its performance is 85% of gcc's performance. Since these numbers include the cost of dynamic code generation, these numbers are lower bounds on the performance of our code. Comparing our results to the interpreter, we see that using dynamic code generation is 50 times faster than the evaluator. From a more global perspective, this same technique can give order of magnitude improvements in the performance of operating system extension languages such as packet filters [1, 10].

[3] Our matrix scaling code is approximately a factor of 2-3 slower than hand-optimized `dcg`; this is because as we emit naive `dcg` IR, and we do no global optimization.

```
/* Set of helper parsing functions */
/* Remove token from the input stream; returns type. */
int gettok(void);
/* Put a token back into the input stream. */
void puttok(void);
/* Result of compare of tok to next input symbol. */
int look(int tok);
/* Consumes expected token or gives parse error. */
void expect(int tok);
/* Pointer to current token. */
char *cur_tok;

/* Symbol table functions */
/* Associate v with name; error to insert duplicate. */
void insert_sym(char *name, int vspec v);
/* Return the vspec associated with a given name. */
int vspec lookup(char *name);
/* Return function pointer associated with name. */
int (*fptr_lookup(char *name))();

/* parse unary-expressions and '(' expr ')' */
int cspec expr(void) {
    switch(gettok()) {
     case CNST: 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(cur_tok)) ;
                  else
                    return expr1(fcall());
     default:        parse_err("bogus expr");
    }
}

/* name '(' { arg { ',' arg }* }? ')' */
void cspec fcall(void) {
    int (*ip)() = fptr_lookup(cur_tok);
    void cspec args = `{};
    gettok();       /* consume '(' */
    /* epsilon ')' */
    if (look(')')) {
     gettok();      /* consume ')' */
     return `($ip)();
    }
    /* arg { ',' arg }*  ')' */
    while (1) {
     /* get argument list */
     args = push(args, "i", (void cspec) `@expr());
     if (look(','))       gettok();
     else if (look(')')) break;
     else                 parse_err("malformed arg list");
    }
    gettok(); /* consume ')' */
    return `($ip)(@args);
}

/* Consume zero or more declarations:  "int ID {, ID}* ';' " */
void declare(void) {
    /* no more declarations */
    if(! look(INT)) return;
    gettok();
    while(gettok() == ID) {
     insert(cur_tok);
     switch(gettok()) {
      case ',': break;                /* another declaration */
      case ';': declare(); return; /* start next decl_seq */
      default: parse_err("malformed declaration");
     }
    }
    parse_err("expecting ID");
}

/* 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 ';':       pushtok(); return e;
     default: parse_err("bogus expr1");
    }
}

/* simple iteration and control flow statements */
void cspec stmt(void) {
    int cspec e;
    void cspec s, s1, s2;
    switch(gettok()) {
     case RETURN:  /* return expr ';' */
        s = `{ return @expr(); };
        expect(';');
        return s;
     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))
              return `{ if (@e) @s1; };
         gettok(); s2 = stmt();
         return `{ if (@e) @s1; else @s2; };
     case '{': /* '{' stmt* '}' */
          push_scope(); declare();
          s = `{};
          while (! look('}')) s = `{ @s; @stmt(); };
          expect('}'); pop_scope();
          return s;
     case ';':
         return `{};
     case ID: /* ID '=' expr ';' */
        {
          int vspec lvalue = lookup(cur_tok);
          expect('='); e = expr(); expect(';');
          return `{ @lvalue = @e; };
        }
     default: /* expression statements not allowed */
        parse_err("expecting statement");
    }
}
```

Figure 2: A Subset of the Tiny-C Recursive-Descent Parser

Description	‘C	Statically compiled C code
Multiplication	390	1100
Division	570	770

Figure 3: Matrix scaling routines; times are in seconds.

‘C	gcc -O2	Tree-Interpreter
2.1	1.8	102

Figure 4: Calculation of the 30th Fibonacci number; times are in seconds.

5 Future Work

We are developing a ‘C compiler that will generate code using templates [9]; this implementation will take full advantage of our static type system, which allows instruction selection to occur at compile-time. Templates are highly specialized code emitters where the instructions have been chosen statically; any holes in the instructions (e.g., runtime constants and addresses of variables) are filled at runtime. Since templates allow the bulk of code generation analysis to be done at compile-time, they can emit code very quickly. For example, register allocation can be done at statically through symbolic register specifications; software register renaming is used at runtime to select the actual registers.

To implement template-driven code generation, our compiler will capture the *closure* of each `cspec`. A closure consists of a pointer to a template and to the collection values within the `cspec` that cannot be determined at compile-time: operands of `$` and `@`, and addresses of free variables in the code specification. A closure is later used at specification-time to create the specified code. We expect the use of templates to improve the speed of dynamic code generation by an order of magnitude [9].

Issues in the design of a template-driven code generator will be discussed in the final paper; we hope to have measurements for a prototype implementation.

6 Related Work

Dynamic code generation has a long history. It has been used to increase the performance of operating systems [12], windowing operations [11], dynamically typed languages [2, 4], simulators [15] and matrix manipulations [5]. Partial evaluation [3] is related to dynamic code generation, but it has traditionally been a source-to-source transformation that specializes programs to particular inputs.

Many languages, such as most Lisp dialects [14, 13] and Perl [16], provide an “eval” operation that allows code to be generated dynamically. This approach is extremely flexible but, unfortunately, comes at a high price: since these languages are dynamically typed, little code generation cost can be pushed to compile-time.

Many of the language design issues involved in ‘C also appear in designing macro languages, such as Weise and Crew’s work [17]. The difference is that macro languages allow programmers to specify code templates that are compiled statically, whereas dynamic code templates are compiled at runtime. Interestingly, although not surprisingly, the syntax we chose turned out to be similar to that used by Weise and Crew.

Leone and Lee [8] describe the use of programmer-supplied hints to perform compile-time specialization in a primitive functional language (e.g., data-structures are not mutable, and its only heap allocated data structures are pointers and integers). They achieve low code generation costs through templates. In contrast to the rudimentary control provided by hints, ‘C gives the programmer powerful, flexible mechanisms for the construction of dynamically generated code: it is difficult to see how the compiler in Section 3.2 could be easily or efficiently realized using their system. Additionally, our language extensions and prototype implementation have been done in the context of ANSI C, a complex non-functional language.

Many Unix systems provide utilities to dynamically *link* object files to an executing process. Thus, a retargetable dynamic code generation system could emit C code to a file, spawn a process to compile and assemble this code, and then dynamically link in the result. Preliminary tests on `gcc` indicate that the compile and assembly phases alone require approximately 30,000 cycles per instruction generated. Our current implementation of ‘C is two orders of magnitude faster than this.

7 Conclusions

Dynamic code generation should be efficient and portable; specifying dynamically generated code should be flexible and simple. We have described ‘C, a superset of ANSI C, that satisfies both of these constraints; we have also paid close attention to preserving the semantics and spirit of ANSI C. We have given two examples of ‘C programs that demonstrate the expressiveness of our language; our prototype compiler demonstrates that we can achieve excellent performance (the use of dynamic code generation can improve performance by up to an order of magnitude), even with a low level of optimization. The reliance on static type-checking reduces the cost of runtime compilation: code generation operations such as instruction selection are performed at compile-time. Furthermore, since types are known statically, the compiler can optimize dynamic code as well as static code.

By making dynamic code generation a language facility, programs that generate dynamic code are more portable, easier to write, and easier to debug. In addition, we make dynamic code generation more useful than in partial evaluation or other automatic specialization systems. For example, ‘C (with some support for linkage) could be used by fast compilers as a portable means of emitting efficient machine code. Finally, while our language design has taken place in the context of ANSI C, we believe the mechanisms used to specify dynamically generated code can also be mapped onto other statically typed languages.

Acknowledgments

Andrew Myers and Raymie Stata provided substantial feedback on our work. We thank Sanjay Ghemawat, David Kranz, Kevin Lew, and Massimiliano Poletto for valuable comments. Deborah Wallach provided us with our base “C to C” compiler.

References

[1] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. Spin - an extensible microkernel for application-specific operating system services. TR 94-03-03, Univ. of Washington, February 1994.

[2] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI ’89*, pages 146–160, Portland, OR, June 1989.

[3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of 20th POPL*, pages 493–501, Charleston, SC, January 1993.

[4] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of 11th POPL*, pages 297–302, Salt Lake City, UT, January 1984.

[5] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[6] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Princeton University, Dept. of Computer Science, Princeton, New Jersey, July 1990.

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

[8] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Copenhagen, Denmark, June 1994.

[9] H. Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1992.

[10] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th SOSP*, pages 39–51, Austin, TX, November 1987.

[11] 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.

[12] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[13] J. Rees, W. Clinger (editors), et al. Revised^4 report on the algorithmic language Scheme. AIM 848b, MIT AI Lab, November 1992.

[14] G.L. Steele Jr. *Common Lisp*. Digital Press, second edition, 1990.

[15] J.E. Veenstra and R.J. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Modeling and Simulation of Computers and Telecommunications Systems*, 1994.

[16] D. Wall. *The Perl Programming Language*. Prentice Hall Software Series, 1994.

[17] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of PLDI '93*, pages 156–165, Albuquerque, NM, June 1993.

A ‘C Grammar

The grammar for ‘C consists of the grammar specified in Harbison and Steele's C reference manual [7] with the following additions:

unary-expression :
- *backquote-expression*
- *at-expression*
- *dollar-expression*

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

at-expression :
- *@ unary-expression*

dollar-expression :
- *$ unary-expression*

pointer :
- $cspec\ type - qualifier - list_{opt}$
- $vspec\ type - qualifier - list_{opt}$
- $cspec\ type - qualifier - list_{opt}\ pointer$
- $vspec\ type - qualifier - list_{opt}\ pointer$