'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation

Dawson R. Engler, Wilson C. Hsieh,* and M. Frans Kaashoek
engler@lcs.mit.edu, whsieh@cs.washington.edu, kaashoek@lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Dynamic code generation allows specialized code sequences to be created 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. Despite the growing use of dynamic code generation, no mainstream language provides flexible, portable, and efficient support for it.

We describe 'C (*Tick C*), a superset of ANSI C that allows flexible, high-level, efficient, and machine-independent specification of dynamically generated code. 'C provides many of the performance benefits of pure partial evaluation, but in the context of a complex, statically typed, but widely used language. 'C examples illustrate the ease of specifying dynamically generated code and how it can be put to use. Experiments with a prototype compiler show that 'C enables excellent performance improvement (in some cases, more than an order of magnitude).

Keywords: dynamic code generation, C

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 [6]; this technique can improve performance by an order of magnitude, even compared to heavily tuned interpreters [12].

Unfortunately, current dynamic code generation systems are not satisfactory. Programmers must choose between 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 [23], 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 through static typing, which allows the bulk of dynamic code generation costs to be paid at compile time. The result of our design effort is 'C (Tick C), which is ANSI C augmented with a small number of primitives for dynamic code generation.

'C inherits many of the performance advantages of partial evaluation [8, 19]. 'C differs from languages that support partial evalution in two ways. First, our language extensions and prototype implementation have been done in the context of ANSI C, a complex, statically typed, but very widely used language. Second, it is not a source-to-source translation, but rather gives the programmer powerful, flexible mechanisms for the construction of dynamically generated code. This control allows programmers to use dynamic code generation both for improved efficiency in situations where it would not normally be applicable (e.g., in the context of aliased pointers) and for simplicity (e.g., by using the act of dynamic code generation to simplify an application's implementation).

'C provides support for specifying dynamically generated code through the addition of two type constructors and three unary operators. This paper makes two contributions. The first is a set of efficient, flexible, high-level primitives for ANSI C to specify dynamically generated code that can be statically type checked. The language design has been challenging, because a static type system makes the runtime specification of arbitrary code difficult (e.g., expressing functions whose number and type of arguments are not known at compile time). While the primitives are designed for ANSI C, we expect the primitives can also be added to other statically typed languages. To illustrate how 'C can be used, we provide a range of examples that exploit dynamically generated code. The second contribution is a prototype 'C compiler. This compiler demonstrates that the use of dynamic code generation improves application performance by up to an order of magnitude.

The remainder of this paper is structured as follows. Sec-

*Author's current address: Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195.

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.

To appear in the *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, January 1996, St. Petersburg, FL.

tion 2 describes the ‘C language. Section 3 summarizes the ‘C standard library. Section 4 illustrates the power of ‘C by discussing a number of example programs. Section 5 reports how the prototype implementation performs on two example programs. Section 6 describes some language extensions we are considering. Section 7 relates ‘C to other work. Section 8 summarizes our conclusions. Appendix A lists the base ‘C grammar.

2 ‘C Language

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 either be *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 code is compiled, and *runtime* to mean when dynamic code is executed.

2.1 Design decisions

Three conflicting goals have driven the design of ‘C:

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 and calling 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 satisfies all three goals is a difficult problem. The second goal led us to avoid functional composition for specifying dynamically generated code. Functional composition is conceptually elegant, but it would not give ‘C sufficient expressive power: functional composition disallows, for example, explicit manipulation of variables shared between fragments of dynamic code.

An important 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.

The final major design decision we made was to limit the construction of dynamic code to one function at a time; each invocation of the library function compile, which instantiates dynamic code, ends one function definition. This restriction reduces bookkeeping complications that would otherwise be necessary in the language. Section 6 discusses how it can be removed.

2.2 Language Modifications for ‘C

‘C adds 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 dynamic code to be statically typed, which in turn allows us to select instructions at compile time.

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

2.2.1 The ‘ Operator

Dynamically generated code is specified using the ‘ (backquote) operator, which is based on Lisp’s usage of ‘ for specifying list templates. ‘ can be applied to an expression or a compound statement. However, backquote expressions may not nest: ‘C does not allow the specification of dynamic code that in turn may specify dynamic code. Some simple usages of backquote are as follows:

```
/* specification of dynamic code for the integer 4 */
‘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, a break, continue, case, or goto statement cannot be used to transfer control outside its containing backquote 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 implicitly inside a function.

2.2.2 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, a compiler can type-check code composition (described in Section 2.2.3) statically.

Applying ‘ to a compound statement yields a result of type void cspec. The type void cspec is the type for 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.

Cspecs can be compiled using the compile function, which is part of the ‘C standard library. The library is described in more detail in Section 3. 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:

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

2.2.3 The @ Operator

The @ operator allows dynamic code specifications to be combined into larger specifications. @ can only be applied inside a backquote expression; legal operands must be cspecs or vspecs (vspecs are described in Section 2.2.4), and are evaluated at specification time. @ “dereferences” cspecs and vspecs: it returns an object whose type is the evaluation type of @’s operand. (@ on cspec and vspec types is analogous to the * operator on pointer types.) 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 cspecs:

```
/* Compose c1 and c2. Evaluation of c yields "9". */
int cspec c1 = ‘4, cspec c2 = ‘5;
int cspec c = ‘(@c1 + @c2);
```

Statements can be composed through concatenation:

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

Applying @ inside a backquote expression to a function that returns a cspec or a vspec causes the function to be called at specification time and its result to be incorporated into the backquote expression.

2.2.4 vspec Types

A variable with a vspec (for *variable specification*) type represents a dynamically generated lvalue; its evaluation type is the type of the lvalue. Vspecs allow lvalues to be statically typed, so instruction selection can be performed at compile time. Objects of type vspec may be initialized by calling the ‘C library functions param and 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). void vspec is used as a generic vspec type.

@ is used to incorporate vspecs into cspecs. An incorporated vspec (like a variable in ANSI C) can be used both as an lvalue and an rvalue inside a backquote expression. The following function returns code that takes a single integer argument, adds one to it, and returns the result:

```
void cspec plus1(void) {
  /* param takes an argument index and its type. */
  int vspec i = (int vspec) param(0, TC_I);
  return ‘{ return @i + 1; };}
```

Vspecs allow us to construct functions that take a runtime-determined number of arguments; this power is necessary in applications such as the compiling interpreter described in Section 5.2. 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 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 = (int vspec) param(i, TC_I);
    /* Add param 'v' to current sum. */
    c = ‘(@c + @v);
  }
  return ‘{ return @c; };
}
```

2.2.5 The $ Operator

The $ operator allows runtime constants to be incorporated into dynamic code. $ evaluates its operand at specification time; the resulting value is incorporated as a *runtime constant* into the containing cspec. $ may be applied to any expression within dynamic code that is 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 runtime constant with value "1" */
c1 = ‘$x;
/* Bind x at runtime when the code for c2 is run */
c2 = ‘x;
c = ‘{ printf("$x = %d, x = %d\n", @c1, @c2); };
x = 14;
/* Compile and run: will print "$x = 1, x = 14". */
compile(c, TC_V)();
```

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, 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 controlled by the programmer 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. The return type of dynamically constructed functions is specified dynamically as well; there seems to be no good way to do it statically. As a result, checking and promotion of return types must be done at runtime.

3 The ‘C Standard Library

We have designed and implemented a small library in order to minimize changes to ANSI C. For example, we have avoided changing ANSI C to allow the construction of function calls at runtime. The library provides the following functions:

void (∗compile(void cspec *code*, int *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 that relate to optimization, debugging and profiling.

void vspec local(int *type-spec* [, int *type-size*])

> local returns a vspec that represents 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*, int *type-spec* [, int *type-size*])

> param returns a vspec that represents a parameter of type *type-spec* and number *param-num*.

void cspec arg(int *arg-num*, void cspec *args*, int *type-spec*, void cspec *code-spec* [, int *type-size*])
void cspec push(void cspec *args*, int *type-spec*, void cspec *code-spec* [, int *type-size*])

> arg returns a cspec that represents code to move *code-spec* into the correct position for argument *arg-num*. push performs the same action as arg, except that the argument number is implicit.

void cspec jump(void cspec *target*)

> jump returns the cspec of a jump to *target*. This function is useful for constructing “hardcoded” finite state machines.

void (∗self(void))(void)

> self returns a pointer to the function that the next invocation of compile will return. This function allows the construction of recursive dynamic procedures.

3.1 Type Specifications

A number of functions in the ‘C standard library expect types as arguments: local, param, arg, push, and compile. This type information is specified using enumerated types. Built-in types are specified by the first letter(s) of their type prefixed by TC_. For example, unsigned short is specified by TC_US. All pointers are represented by TC_P: operationally, they are treated as void ∗ pointers for purposes of storage and register allocation. Aggregates (arrays and structures) are specified by TC_B; their size must be given as an additional argument. Finally, the flag TC_REGISTER can be bitwise-ored with the type 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.

3.2 Runtime-Constructed Function Calls

Function calls can be constructed “on the fly” by using the library functions arg and push. Consider the function sum, which is described in Section 2.2.4. To call sum one constructs the argument list at runtime by using push. ‘C allows a void cspec (which represents the 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, TC_V, ‘$arg_vec[i]);
  }
  return ‘sum(@args);
}
```

4 Examples

This section gives program examples to illustrate ‘C’s power. These programs exhibit how flexibly dynamic code can be specified in ‘C, as well as some of the different uses to which dynamic code generation can be put.

4.1 Runtime constants

Dynamic code generation can be used to hardwire infrequently changing runtime values into the instruction stream, so that the values need not be loaded from memory. An example is a generic hash function, where the table size is determined at runtime, and where the function uses a runtime 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 ∗/
  /∗ vector of pointers to hash table entries ∗/
  struct hte ∗∗hte;
};

/∗ 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 runtime 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(0, TC_I);
  void cspec hc;

  hc = ‘{
    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 (hptr) compile(hc, TC_P);
}
```

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 much faster than the equivalent C version, since a ‘C compiler can exploit the runtime constants hte, scatter, and norm both by hardcoding them directly in the instruction stream and by strength reducing the multiplication and division to shifts and adds. Such optimizations are increasingly profitable on modern architectures, where cache misses are very expensive and division and multiplication are frequently provided in only software [18, 31]. This examples also illustrates that turning a C function into ‘C requires few changes. The cost of using the ‘C generated function is an indirect jump on a function pointer.

4.2 Matrix Dot Product

Matrix manipulations such as dot product are a fruitful realm for dynamic code generation. Matrices often have runtime characteristics (e.g., large numbers of zeros and small integers) that 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 multiplcation the use of dynamic code generation allows us to exploit zeros and small integers by crafting locally optimized code based on runtime 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 compute_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 runtime several optimizations can be employed. For example, the programmer can directly eliminate multiplication by zero. The resulting ‘C code is given below.

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

  /∗ pointer to the col to multiply row by ∗/
  col = (int ∗ vspec) param(0, TC_P);

  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 reduced in
         strength.
         This 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 the one programmed in C. 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 replace multiplication with shifts and adds.

4.3 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 comparisions. 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 [5] illustrates ‘C’s advantages for parameterized functions. The function newton takes as arguments the allowed number of iterations, a tolerance, an initial estimate, and two pointers to functions that return cspecs to compute a function and its derivative. The cspecs returned by these functions are incorporated directly into the dynamically generated code, which eliminates function call overhead and allows inter-cspec optimization to occur at instantiation time.

```
/* pointer to a function that takes a vspec
   as an argument and returns a cspec. */
typedef double cspec (*dptr)(double vspec);

/* Dynamically create a newton-raphson routine
   specialized to the given function and derivative.
   In a real implementation we would memoize the
   function and, if it was used heaviliy enough,
   unroll the loop to the maximum number of iterations.

   "n" is the number of allowed iterations; "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 cspec returned by f and
         fprime and use them to calculate the next
         point in the (hopefully) convergent series. */
      p = p0 - @f(p0) / @fprime(p0);
      /* When we converge to a given tolerance,
         return the result.  */
      if (abs(p - p0) < tol)
        return p;
      p0 = p; /* Seed the next iteration. */
    }
    error("method failed after %d iterations\n", i);
  };
  /* Compile, call, and return the result. */
  return ((dptr)compile(cs, TC_D))();
}
```

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 the derivative of f
   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) {
  double root;
  root = newton(100, .000001, 10., f, fprime);
  printf("root is %f\n", root);
}
```

4.4 Small Language Compilation

There are myriads of small, primitive languages that are both time-critical and amenable to dynamic compilation. The small query languages used to interogate data bases are well-known targets for dynamic code generation [21]; because databases are large, dynamically compiled queries will be applied many times.

We provide a small example below. The function mkquery takes a vector of queries, where each query specifies a database record field (such as CHILDREN or INCOME), a value to compare this field to, and the comparison function to use. The compiler creates code fragments to access each field and then generates code to compare it to the given value.

```
typedef enum { INCOME, CHILDREN /* ... */ } 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 mkquery.  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 made up of (field, val, bool_op) where
   "field" indicates a record field, "val" is the
   value to compare it to, and "bool_op" is the
   operation used in the comparision (<, >, etc.).  */
iptr mkquery(struct query *q, int n) {
  int cspec field, cspec expr, i;
  struct record * vspec r;

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

```
  expr = `1;
  for (i = 0; i < n; i++) {
    /* first 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.
     */
    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; }, TC_I);
}
```

4.5 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 [7]. Unfortunately, each protocol layer frequently has data-touching operations associated with it (e.g., checksumming, byte-swapping, etc.). As a result, as data moves through each layer, data can be touched multiple times, which is expensive [7].

‘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 would be to divide 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. To allow pipes to manipulate state, they are allowed to specify initial and final code to call, which would be used by other applications to initialize variables and detect errors. For instance, a checksum routine would have final code to check whether the checksum was valid.

The following pipe can be used to do byte-swapping. Since a byte swapper does not maintain any state, there is no need to specify initial and final code.

```
/* Example pipe: construct a cspec 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 does not maintain any state and so does
    not need initial and final statements. */
void cspec byteswap_initial(void) { return `{}; }
void cspec byteswap_final(void) { return `{}; }
```

To construct the integrated data copying routine, the initial, consumer, and final statements of each pipe are composed with the code of its neighbors, respectively. The composed initial statements are placed at the beginning of the routine, the consumer statements are placed in a loop to provide them with input and store their output, and the final code is placed at the end of the routine. A simplified code fragment is provided below. In a mature implementation the loop would be unrolled. Additionally, pipes would take different size inputs and outputs (or “gauges”) that the composition function would have to reconcile.

```
/* 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 as
    it copies the data.  */
void cspec compose(struct pipe *plist, int n) {
  struct pipe *p;
  unsigned cspec result, cspec pipes;
  unsigned * vspec input, * vspec output;
  int vspec i, vspec nwords;
  void cspec initial_stmts = `{};
  void cspec final_stmts = `{};

  nwords = (int vspec) param(0, TC_I);
  /* pointer to memory that provides pipe input */
  input = (unsigned * vspec) param(1, TC_P);
  /* pointer memory that accepts pipe output */
  output = (unsigned * vspec) param(2, TC_P);
  i = (int vspec) local(TC_I);

  /* Compose all stages in parallel */
  pipes = `(@input)[@i];         /* base pipe input */
  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.  Note: in the generated
        code, pipes are *not* function calls, but simply
```

```
        arithmetic expressions that occur in a specified
        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;
  };
}
```

Note that ' and @ have the same precedences as unary minus. As a result, the expression '@p→pipe(pipes) results in the function pointed to by p→pipe being called and its result being used as the value of the ' expression at specification time.

4.6 Marshaling and Unmarshaling

Another example use of 'C is the construction of code to marshal and unmarshal arguments stored in a byte vector. These operations are frequently performed to support remote procedure call [4]. By generating specialized code for the most active functions it is possible to gain substantial performance benefits [33].

The generation of marshaling code relies on 'C's ability to specify arbitrary numbers of incoming parameters. Figure 1 gives a simplified code fragment that generates a marshaling function for a particular type set (in this example, INT, POINTER, and DOUBLE). 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 creates a vspec that points to the current parameter; it constructes code to store the parameter's value into the byte vector at the current offset; and it 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. At the end of 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, etc.

The generation of unmarshaling code is equally profitable. Dynamic generation of unmarshaling routines relies on our mechanisms for constructing calls (to arbitrary functions) at runtime. The ability to invoke arbitrary functions is not just useful for efficiency: it is also useful for functionality. For example, in Tcl [25] the runtime 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.

```
/* Type of the function called by the code mk_marshal
    generates. It takes a pointer to memory
    where the caller's arguments have been marshaled. */
typedef void (*vptr)(char *buf);
/* Type of the function generated by mk_marshal: takes
    an unspecified number of arguments. */
typedef void (*mptr)();

/* The types of arguments mk_marshal can marshal. */
typedef enum { DONE, INTEGER, DOUBLE, POINTER, /* ... */ }
  types_t;

/* Generate a function at runtime to marshal arguments
    into a byte vector and call vp with them. */
mptr mk_marshal(vptr vp, types_t *types) {
  char * vspec m;
  void cspec s;
  int i, offset;
  vptr vp;

  /* Allocate a buffer on the stack to hold arguments. */
  m = (char * vspec) local(TC_B, compute_size(types));
  s = '{};   /* Initialize marshalling statements. */

  for (i = offset = 0; types[i] != DONE; i++) {
    switch(types[i]) {
     case INTEGER: {
      /* create a vspec for the ith parameter */
      int vspec v = (int vspec) param(i, TC_I);

      /* Concatenate the code to marshal an integer
          into @m. The code uses the offset to compute
          the location in m. */
      s = '{ @s; * (int *) &(@m)[$offset] = @v; };
      /* add the size of the argument to the current
         buffer offset */
      offset += sizeof(int);
      break;
     }
     case DOUBLE: {
      /* create a vspec for the ith parameter */
      double vspec v = (double vspec) param(i, TC_D);
      /* concatenate the code to marshal a double */
      s = '{ @s; * (double *) &(@m)[$offset] = @v; };
      offset += sizeof(double);
      break;
     }
     case POINTER: {
      /* create a vspec for the ith parameter */
      void * vspec v = (void * vspec) param(i, TC_P);
      /* concatenate the code to marshal a pointer */
      s = '{ @s; * (void **) &(@m)[$offset] = @v; };
      offset += sizeof(void *);
      break;
     }
    /* ... */
    }
  }

  /* after the arguments are marshaled, call the passed
     function with the marshaled data. */
  return (mptr) compile('{ @s; ($vp)(@m); }, TC_V);
}
```

Figure 1: 'C code to generate a function to marshal a byte-vector containing the given types.

```
/* Type of the function generated by mk_marshal.  It
    takes a pointer to memory where the marshaled
    arguments are stored. */
typedef void (*vptr)(char *buf);

/* Generate a function to unmarshal a byte vector into
    arguments and call a function.  For simplicity, we
    ignore alignment and endian issues.  */
vptr mk_unmarshal(type_t *types) {
  vptr vspec vp;
  char * vspec m;
  void cspec args;
  int offset, i;

  /* Function pointer: sent as first argument.  */
  vp = (vptr vspec) param(0, TC_P);
  /* Byte vector: sent as second argument. */
  m = (char * vspec) param(1, TC_P);
  /* Dynamically constructed argument list. */
  args = `{};

  for(i = offset = 0; types[i] != DONE; i++) {
    switch(types[i]) {
    case INTEGER:
      /* code to extract an int */
      push(args, TC_I, `* (int *) &(@m)[$offset]);
      offset += sizeof(int);
      break;
    case DOUBLE:
      /* code to extract a double */
      push(args, TC_D, `* (double *) &(@m)[$offset]);
      offset += sizeof(double);
      break;
    case POINTER:
      /* code to extract a pointer */
      push(args, TC_P, `* (void **) &(@m)[$offset]);
      offset += sizeof(void *);
      break;
    /* ... */
    }
  }
  /* Call the given function pointer with
      the unmarshaled parameters. */
  return (vptr) compile(`{ @vp(@args); }, TC_V);
}
```

Figure 2: ‘C code to generate a function to unmarshal a byte-vector containing the given types.

Figure 2 gives the code that generates the unmarshaling function to work with the marshaling code specified in Figure 1. The generated code will take 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 functions incoming arguments are created. Second, the argument list is initialized. Then, for every type in the type vector the function performs the following sequence of actions: it creates a cspec to index into the byte vector at a fixed offset; it pushes this cspec into its correct parameter position; and it 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.

5 Performance

We have implemented a prototype ‘C compiler that emits ANSI C code augmented with calls to DCG's [13] dynamic code generation primitives. The compiler parses, semantically checks, and generates code for ‘C. It generates code correctly for most of the examples in this paper. Our prototype demonstrates that despite our lack of optimization and DCG's rudimentary optimization (it does not perform instruction scheduling nor peephole optimization), the generated code still achieves good performance.

We are developing a full ‘C compiler that will generate fast code using templates and VCODE. VCODE [11] is a retargetable, extensible, very fast dynamic code generation system. It is a portable assembly language that generates specialized code on the fly; the cost for dynamic code generation is about ten instructions per generated instruction. 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 [23]. Since templates, combined with ‘C's static type system, allow the bulk of code generation analysis to be done at compile time, we will emit code very quickly. We expect the use of templates and VCODE to improve the speed of dynamic code generation by an order of magnitude.

The rest of this section presents performance results using the prototype compiler for two ‘C programs that are based on the examples used in [13]. The 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 version 2.5.8.

5.1 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 reduced in strength to shifts and adds [2]; division can be reduced in strength to multiplication, and then to shifts and adds [15]. 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 3. We compare its performance to that of a static matrix scaling routine. We run two experiments, one for division, the other for multiplication. Multiplication is done

```
/* Construct code to scale matrix m of
   size n x n by runtime 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 at runtime */
        v[j] = v[j] * $s;
    }
  };
}
```

Figure 3: ‘C code for scaling a matrix by a runtime constant

on a matrix that contains ints; division on a matrix that contains shorts. The experimental times given in Figure 5 measure the summation of the time required to scale a 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 matrix of ints by a runtime constant improved by a factor of 3. The performance of dividing a 1024x1024 matrix of shorts by a runtime constant improved by 35%. More dramatic improvements would be possible with a more sophisticated factorization scheme for reducing division in strength. The ‘C matrix scaling code is approximately a factor of 2–3 slower than hand-optimized DCG code [13], because the prototype compiler emits naive DCG IR, and does not perform global optimizations.

5.2 Compiling Interpreter

Interpreters can use dynamic code generation technology to improve performance by compiling and then directly executing frequently interpreted pieces of code [6, 10]. 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, called Tiny C [13]. 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 4. 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. In addition, our experience has been that specifying dynamically generated code in ‘C is easier than constructing an efficient interpreter.

A recursive Fibonacci program is used to measure the performance of three implementations of Tiny C.

‘C: The ‘C compiling interpreter for Tiny C.

gcc –O2: Tiny C using gcc with optimization level “–O2”. This gives an upper bound on the quality of local code.

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

Figure 5: Matrix scaling routines; times are in seconds.

‘C	gcc –O2	Tree-Interpreter
2.1	1.8	102

Figure 6: Calculation of the 30th Fibonacci number; times are in seconds.

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

Figure 6 summarizes the results for computing the 30th Fibonacci number. The code that is generated using ‘C’s 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 ‘C code. Comparing the ‘C 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 [24].

The ‘C results are within 10% of hand-generated DCG code. The simple IR generation that the ‘C compiler performs does not lower the performance as much as in the matrix scaling example, because the dynamic code fragments are so simple.

6 Language Extensions

Language design is an iterative process. This section describes extensions that we are considering for ‘C. The ‘C prototype compiler does not implement them, since they were designed after the prototype was written. We intend to have a publicly available compiler that implements the full ‘C language in the near future.

6.1 Partial Evaluation

We have designed support for dynamic partial evaluation in ‘C. While partial evaluation does not change the power of ‘C, it is useful syntactic sugar. Incorporating partial evaluation was complicated by the requirement that we want to support specialization of functions with respect to different arguments at different times. For instance, a programmer may want to specialize the following simple function with respect to x at one call-site and y at another:

```
int foo(int x, int y, short z) { return x + y + z; }
```

In a statically-compiled language such as ‘C, code generation of every function template happens at compile time. Therefore, to prevent the code explosion that would result from specializing every function for every possible combination of arguments, ‘C provides *partial signatures* to statically specify the possible permutations of arguments with which a function will be specialized. Partial signatures (modeled on function

```
/* 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);
  char *cur_tok;          /* Pointer to current token. */

/* 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 declarations */
void declare(void) {
  if(! look(INT))      return; /* no more declarations */
  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");
}

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

/* 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");
  }
}

/* Parse function calls */
void cspec fcall(void) {
  int (*ip)() = fptr_lookup(cur_tok);
  void cspec args = `{};
  gettok();        /* consume `(' */
  if (look(')')) {
    gettok();      /* no args; consume `)' */
    return `($ip)();
  }
  while (1) {
    /* get argument list */
    args = push(args, "i", (void cspec) `@expr());
    if (look(','))        gettok();
    else if (look(')'))   break;
    else                  parse_err("malformed arg list");
  }
  gettok(); /* consume ')' */
  return `($ip)(@args);
}

/* 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 4: A Subset of the Tiny-C Recursive-Descent Parser

prototypes) are used to indicate which arguments can be specialized in a function. A partial signature is a function prototype prefixed with the partial keyword; it contains the bound type specifier before each argument that can be specialized. There can be multiple partial signatures for a given function; each unique partial signature must be in the same scope as the function definition. For example, the following partial signatures allow specialization with respect to the first parameter x or the second parameter y:

```
/* Evaluate foo with respect to x. */
partial int foo(bound int x, int y, short z);
/* Evaluate foo with respect to y. */
partial int foo(int x, bound int y, short z);
```

Partial evaluation of a function is performed using the unary prefix operator eval. eval takes a function and its arguments, and returns a function pointer with the return type of the function and with parameter types corresponding to the non-evaluated parameters. The unbound keyword is used as a placeholder to indicate which arguments are not being provided during partial evaluation. Attempting to use eval with an operand whose type signature does not correspond to any partial signature in scope is an error. For instance, the following are valid specializations of the function foo, whose partial signatures are provided above:

```
int (*ip1)(int, short);
int (*ip2)(int, short);

/* create a partial evaluation
   with the first parameter fixed */
ip1 = eval foo(4, unbound, unbound);
/* create a partial evaluation
   with second parameter fixed */
ip2 = eval foo(unbound, 5, unbound);
```

We expect that partial evaluation of statically specified functions will be a common usage, which is why we added it to ‘C.

6.2 Other Modifications

In the interests of full generality, we are exploring support to allow multiple functions to be generated simultaneously. Every vspec would have to be explicitly associated with a context, and the compiler and runtime would have to ensure two conditions. First, all vspecs used in a cspec would have to be from the same context. Second, all cspecs that are composed would have to contain only vspecs from the same context.

One of the most unfortunate features of ‘C is the use of manually supplied types in the runtime system. We are experimenting with alternative approaches to provide a cleaner, less error-prone mechanism.

We are also experimenting with mechanisms to allow code fragments to be parameterized. Parameterization aids the modular composition of cspecs, since the internal names of a cspec can be hidden. We will most likely introduce a lambda unary operator to create nameless functions. While this functionality can be simulated in the original ‘C language, the syntactic sugar of lambda can remove awkwardness in some situations.

7 Related Work

Dynamic code generation has a long history. It has been used to increase the performance of operating systems [3, 12, 27, 28], windowing operations [26], dynamically typed languages [6, 10, 17], simulators [30, 34] and matrix manipulations [13].

‘C grew out of our previous work with DCG [13], 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 [14]. 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, because we have made dynamic code generation a first-class capability of a high-level language, both profiling and debugging facilities can be added.

Many languages, such as most Lisp dialects [29, 32], Tcl [25], and Perl [35], 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 [36]. 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 perhaps not surprisingly, the syntax we chose turned out to be similar to that used by Weise and Crew.

Massalin et al. briefly note that they are designing a language for code synthesis, Lambda-C [28]. They do not discuss design or implementation issues other than to note that “type-checking of synthesized code is non-trivial.”

Leone and Lee [22] use programmer-supplied hints to perform compile-time specialization in a primitive functional language: their data structures are not mutable, and the 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 5.2 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.

Several other projects address the higher-level issue of automatic compiler support for detecting runtime constants [1, 9]. They use programmer annotations to indicate some runtime constants; the compiler computes what variables are derived runtime constants.

Keppel addressed some issues relevant to retargeting dynamic code generation in [20]. He developed a portable system for modifying instruction spaces on a variety of machines. His system dealt with the difficulties presented by caches and operating system restrictions, but it did not address how to select and emit actual binary instructions. Keppel, Eggers, and Henry [21] demonstrated that dynamic code generation could be effective for several different applications.

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 prototype implementation of ‘C is two orders of magnitude faster.

8 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 and also preserves the semantics and spirit of ANSI C. Examples of ‘C programs demonstrate the expressiveness of the language and illustrate how dynamic code generation can be used. The ‘C prototype compiler demonstrates that ‘C programs can achieve excellent performance (the use of dynamic code generation can improve performance by up to an order of magnitude), even with little optimization. The reliance on static type checking reduces the cost of runtime compilation: code generation operations such as instruction selection can be 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, ‘C’s dynamic code generation facilitiy is more flexible 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 the language design has taken place in the context of ANSI C, we expect the mechanisms used to specify dynamically generated code can also be mapped onto other statically typed languages.

9 Acknowledgments

Massimiliano Poletto helped design the partial evaluation mechanics; both he and Eddie Kohler were involved in the discussion of lambda. Andrew Myers and Raymie Stata provided us with 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. Eddie Kohler helped immensely with the typesetting of this document.

References

[1] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. Submitted for publication, October 1995.

[2] R. Bernstein. Multiplication by integer constants. *Software—Practice and Experience*, 16(7):641–652, July 1986.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

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

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

[6] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989.

[7] 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.

[8] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages*, pages 493–501, Charleston, SC, January 1993.

[9] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1996.

[10] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.

[11] D. R. Engler. VCODE: a very fast, retargetable, and extensible dynamic code generation substrate. Technical Memorandum MIT/LCS/TM534, MIT, July 1995.

[12] D.R. Engler, M.F. Kaashoek, and J. O’Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[13] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 263–272, October 1994.

[14] 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.

[15] T. Granlund and P.L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the SIGPLAN ’94 Conference on Programming Language Design and Implementation*, pages 61–72, June 1994.

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

[17] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN ’94 Conference on Programming Language Design and Implementation*, pages 326–335, Orlando, Florida, June 1994.

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

[19] N. D. Jones, P. Sestoft, and H. Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[20] D. Keppel. A portable interface for on-the-fly instruction space modification. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, April 1991.

[21] 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.

[22] 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.

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

[24] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

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

[26] 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.

[27] C. Pu, T. Autry, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commerical operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

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

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

[30] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 1995. To appear.

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

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

[33] 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.

[34] 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.

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

[36] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 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 [16] 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*