Dynamic Code Generation with the E-Code Language

Greg Eisenhauer
eisen@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

August 3, 2005 – E-Code Version 1.0

1 Introduction

E-Code is a higher-level language for dynamic code generation. E-Code specifically targets the dynamic generation of small code segments whose simplicity does not merit the complexity of a large interpreted environment. To that end, E-code is as simple and restricted a language as possible within the bounds of its target environment. E-Code's dynamic code generation capabilities are based on DRISC, a Georgia Tech package that provides a programatic on-the-fly code generation through a virtual RISC instruction set. DRISC also has a "virtual register" mode in which it performs simple register allocation and assignment. E-Code consists primarily of a lexer, parser, semanticizer and code generator.

This paper describes the E-Code language and the subroutines that support its conversion into native code.

2 The Basics

2.1 The E-Code Language

E-Code may be extended as future needs warrant, but currently it is a subset of C. Currently it supports the C operators, function calls, `for` loops, `if` statements and `return` statements.

In terms of basic types, it supports C-style declarations of integer and floating point types. That is, the type specifiers `char`, `short`, `int`, `long`, `signed`, `unsigned`, `float` and `double` may be mixed as in C. `void` is also supported as a return type. E-Code does not currently support pointers, though the type `string` is introduced as a special case with limited support.

E-Code supports the C operators !,+,-,*,/,%,⟩,⟨,⟨=,⟩=,==,!=,&&,‖,= with the same precedence rules as in C. Paranthesis can be used for grouping in expressions. As in C, assignment yields a value which can be used in an expression. C's pre/post increment/decrement operators are not included in E-Code. Type promotions and conversions also work as in C.

As in C, semicolons are used to end statments and brackets are used to group them. Variable declarations must preceed executable statements. So, a simple E-Code segment would be:

```
{
    int j = 0;
    long k = 10;
    double d = 3.14;

    return d * (j + k);
}
```

2.2 Generating Code

The subroutine which translates E-Code to native code is `ecl_code_gen()`. The sample program below illustrates a very simple use of `ecl_code_gen()` using the previous E-Code segment.

```
#include "ecl.h"

char code_string[] = "\
{\n\
    int j = 4;\n\
    long k = 10;\n\
    short l = 3;\n\
\n\
    return l * (j + k);\n\
}";

main()
{
    ecl_parse_context context = new_ecl_parse_context();
    ecl_code gen_code;
    long (*func)();

    gen_code = ecl_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    printf("generated code returns %ld\n", func());
    ecl_free_parse_context(context);
    ecl_code_free(gen_code);
}
```

When executed, this program should print "`generated code returns 42.`" Note that code generation creates a function in malloc'd memory and returns a pointer to that function. That pointer is cast into the appropriate function pointer type and stored in the `gen_code` variable. It is the programs responsibility to free the function memory when it is no longer needed. This demonstrates basic code generation capability, but the functions generated are not particularly useful. The next sections will extend these basic capabilities.

3 Parameters and Structured Types

3.1 Simple Parameters

The simplest extensions to the subroutine generated above involve adding parameters of atomic data types. As an example, we'll add an integer parameter "i". To make use of this parame-

ter we'll modify the return expression in the `code_string` to `l * (j + k + i)`. The subroutine `ecl_add_param()` is used to add the parameter to the generation context. It's parameters are the name of the parameter, it's data type (as a string), the parameter number (starting at zero) and the `ecl_parse_context` variable. To create a function with a prototype like `long f(int i)` the code becomes:

```
    ecl_parse_context context = new_ecl_parse_context();
    ecl_code gen_code;
    long (*func)(int i);

    ecl_add_param("i", "int", 0, context);
    gen_code = ecl_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    printf("generated code returns %ld\n", func(15));
    ecl_free_parse_context(context);
    ecl_code_free(gen_code);
}
```

When executed, this program prints ``generated code returns 87.'' Additional parameters of atomic data types can be added in a similar manner.

3.2 Structured Types

Adding structured and array parameters to the subroutine is slightly more complex. In particular, E-Code tries to avoid making assumptions about the alighment and layout decisions that might be made by whatever compiler is in use. In order to avoid this, structured E-Code types must be described in the manner of PBIO[1], specifying the name, type, size and offset for each field. For convenience, the PBIO declarations relating to `IOField` lists are repeated in `ecl.h`. A detailed discussion of their use can be found in [1], but an example structure declaration and its associated `IOField` declaration is below:

```
typedef struct test {
    int i;
    int j;
    long k;
    short l;
} test_struct, *test_struct_p;

ecl_field_elem struct_fields[] = {
    {"i", "integer", sizeof(int), IOOffset(test_struct_p, i)},
    {"j", "integer", sizeof(int), IOOffset(test_struct_p, j)},
    {"k", "integer", sizeof(long), IOOffset(test_struct_p, k)},
    {"l", "integer", sizeof(short), IOOffset(test_struct_p, l)},
    {(void*)0, (void*)0, 0, 0}};
```

The `ecl_add_struct_type()` routine is used to add this structured type definition to the parse context. If we define a single parameter "input" as this structured type, the new return value in `code_string` is `input.l * (input.j + input.k + input.i)` and the program body is:

```
main() {
    ecl_parse_context context = new_ecl_parse_context();
    test_struct str;
    ecl_code gen_code;
    long (*func)(test_struct *s);

    ecl_add_struct_type("struct_type", struct_fields, context);
    ecl_add_param("input", "struct_type", 0, context);

    gen_code = ecl_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    str.i = 15;
    str.j = 4;
    str.k = 10;
    str.l = 3;
    printf("generated code returns %ld\n", func(&str));
        ecl_code_free(gen_code);
        ecl_free_parse_context(context);
}
```

Note that the structured type parameter is passed by reference to the generated routine. However in `code_string` fields are still referenced with the '.' operator instead of '-⟩' (which is nor present in E-Code).

4 External Context and Function Calls

One aspect of external context not yet addressed is E-Code's ability to generate calls to external functions and subroutines. Like all external entities, subroutines must be defined to E-Code before they are available for reference. To make a subroutine accessable in E-code requires defining both the subroutine's profile (return and parameter types) and its address. To specify the subroutine profile, E-code parses C-style subroutine and function declarations, such as:

```
int printf(string format, ...);
void external_proc(double value);
```

This is done through the subroutine `ecl_parse_for_context()`.

However, to associate addresses with symbols requires a somewhat different mechanism. To accomplish this, E-code allows a list of external symbols to be associated with the `ecl_parse_context()` value. This list is simply a null-terminated sequence of $\langle symbolname, symbolvalue \rangle$ pairs of type `ecl_extern_entry`. The symbol name should match the name used for the subroutine declaration. For example, the following code sequence makes the subroutine "printf" accessable in E-code:

```
extern int printf();
static ecl_extern_entry externs[] =
{
    {"printf", (void*)printf},
    {NULL, NULL}
};

static char extern_string[] = "int printf(string format, ...);";

{
    ecl_parse_context context = new_ecl_parse_context();
    ecl_assoc_externs(context, externs);
    ecl_parse_for_context(extern_string, context);
    ...
```

References

[1] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu).*

[2] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, (24), 1998.