Compilers tutorial VI: Code generation

Once the *analysis* stages are complete, we are ready for the *synthesis* stage: code generation. The compiler transforms the abstract syntax tree (AST) into a linear representation, by traversing the AST and emitting some code at every node. At the core of LLVM there is the *intermediate representation (IR)* language, which is our target language for code generation.

Simple functions in LLVM IR

Functions are specified using the `define` keyword, which has the following simplified syntax: `define <ResultType> @<FunctionName> ([argument list])` followed by the function body. Consider the following example:

```
define i32 @avg(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  %2 = sdiv i32 %1, 2
  ret i32 %2
}

define i32 @main() {
  %1 = call i32 @avg(i32 1, i32 5)
  ret i32 %1
}
```

First, we define the `avg` function. The function adds `%a` to `%b` and stores the result in temporary register `%1`. Then, it divides `%1` by two and stores the result in temporary `%2`, which is the return value. The `main` function calls `avg` with arguments 1 and 5, then storing the result in `%1`, which is the return value. With this code in a file named `avg.ll`, we compile and run the program by entering:

```
$ llc avg.ll -o avg.s
$ clang avg.s -o avg
$ ./avg
$ echo $?           [print the return code of ./avg]
```

Alternatively, we can use the LLVM interpreter to achieve the same result:

```
$ lli avg.ll
$ echo $?           [print the return code of lli]
```

Types

The LLVM IR is a strongly typed assembly language. Primitive types include integers like `i32` and `i64` (32 and 64 bits, respectively). The `i1` type is used for booleans. The types `float` and `double` represent floating point numbers (32 and 64 bits, respectively).

Pointers are also allowed (just like in C) so we can use pointers to primitive types such as `i32*` and `double*`, pointers to pointers such as `i8**`, pointers to functions such as `i32(i32)*`, and pointers to structures.

Arrays are also supported, using the syntax `[<N> x <Type>]`, to represent sequences of elements with a specific type. For example, `[5 x i8]` specifies an array of 5 bytes, and `[20 x i32]` specifies an array of 20 integers.

Operations

In the example above, the `avg` function executes some operations. The first line adds the two operands and stores the result in temporary `%1`. The addition operation has the syntax `<result> = add <Type> <op1>, <op2>` where `<Type>` is the data type of the operation (both operands must have the same type). Moreover, `<op1>` and `<op2>` are the two operands for which the sum is calculated: local variables start with `%` while global variables start with `@`.

Most instructions return a value that is typically assigned to a temporary register such as `%1`, `%2`, or `%tmp`. All LLVM IR instructions are in static single assignment form (SSA), meaning that each register may only be assigned a value once. Furthermore, any register must be assigned a value before being used. To simplify the code, we often use the `alloca` instruction to explicitly allocate memory on the stack frame for mutable variables (shown in the example below).

Calls

The `call` instruction is used for simple function calls, with explicit arguments, and the `ret` instruction is used for returning control flow (and often a value) back to the caller. Examples can be found above, in the `main` and `avg` functions, as well as below, in the `factorial` function.

Declaration of functions

The `declare` keyword allows us to declare functions without specifying their implementation (i.e., without a function body). This allows us to divide big programs into modules that will be combined together by the linker. It also allows us to declare external functions such as those provided by the standard C library (`printf`, `puts`, etc.).

Suppose we want an LLVM IR program to call external functions `_read` and `_write` that are defined in the `io.c` source file. The LLVM IR program must simply include the following declarations:

```
declare i32 @_read(i32)
declare i32 @_write(i32)
```

It then becomes possible to call those functions just like any local function, for example using the instruction `%1 = call i32 @_write(i32 123)` to write number 123 to the standard output.

Control flow

The `factorial` example below is the result of compiling the following program:

```
factorial(integer n) = if n then n * factorial(n-1) else 1
```

This example combines operations, calls, etc., with a new element: control flow.

```
define i32 @factorial(i32 %n) {
  %1 = alloca i32
  %2 = add i32 %n, 0
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %L1then, label %L1else
L1then:
  %4 = add i32 %n, 0
  %5 = add i32 %n, 0
  %6 = add i32 1, 0
  %7 = sub i32 %5, %6
  %8 = call i32 @factorial(i32 %7)
  %9 = mul i32 %4, %8
  store i32 %9, i32* %1
  br label %L1end
L1else:
  %10 = add i32 1, 0
  store i32 %10, i32* %1
  br label %L1end
L1end:
  %11 = load i32, i32* %1
  ret i32 %11
}
```

The `factorial` function is divided into 4 blocks of instructions, delimited by the labels. The `br` instructions are used to transfer control flow to a different block within the same function.

The first `br` instruction is a *conditional branch*. It takes a single `i1` value (the boolean condition) and two labels. If the `i1` value is true, control flows to the first label; otherwise, control flows to the second label. The `i1` value (temporary `%3`) is computed by the `icmp` instruction, which is the integer comparison instruction. Therefore, if `n` is not equal to zero (`icmp ne i32 %2, 0` where `%2 = n`) the program branches to label `L1then`, otherwise it branches to label `L1else`.

The other two `br` instructions are *unconditional branches*, taking a single label as target. Such branches are always taken, that is, control flow is unconditionally transferred to the target label.

Finally, it should be noted that the `factorial` function returns the value stored at `i32* %1` which is a pointer to an integer stored in the stack frame. That pointer is allocated through the first `alloca` instruction.

The recursive code generation algorithm

The input program is fully represented by an AST annotated with the relevant attributes. Code generation is expressed through a recursive traversal of the AST. Function `codegen_program` (in file `codegen.c`) generates code for the root `Program` node by generating code for each child `Function` node and then emitting code for the `main` entry point:

```
void codegen_program(struct node *program) {
    struct node_list *function = program->children;
    while((function = function->next) != NULL)
        codegen_function(function->node);

    struct symbol_list *entry = search_symbol(symbol_table, "main");
    if(entry != NULL && entry->node->category == Function)
        printf("define i32 @main() {\n"
               "  %%1 = call i32 @_main(i32 0)\n"
               "  ret i32 %%1\n"
               "}\n");
}
```

The `while` loop iterates through the children of the `Program` node, which are the `Function` nodes, calling `codegen_function` to generate code for each of them.

```
void codegen_function(struct node *function) {
    temporary = 1;
    printf("define i32 @_%s(", getchild(function, 0)->token);
    codegen_parameters(getchild(function, 1));
    printf(") {\n");
    codegen_expression(getchild(function, 2));
    printf("  ret i32 %%%d\n", temporary-1);
    printf("}\n\n");
}
```

The code above initialises the counter of temporary registers (`%1`, `%2`, etc.). Then, it emits code to `define` the function, calls `codegen_parameters` to generate the list of parameters and calls `codegen_expression` to evaluate the expression. Small fragments of code are also emitted (the `ret` instruction, brackets, etc.).

```
void codegen_parameters(struct node *parameters) {
    struct node *parameter;
    int curr = 0;
    while((parameter = getchild(parameters, curr++)) != NULL) {
        if(curr > 1)
            printf(", ");
        printf("i32 %%%s", getchild(parameter, 1)->token);
    }
}
```

Function `codegen_parameters` (above) is quite simple, as it only emits a sequence of identifier names separated by commas. Function `codegen_expression` (below) generates code to evaluate expressions:

```
int codegen_expression(struct node *expression) {
    int tmp = -1;
    switch(expression->category) {
        case Natural:
            tmp = codegen_natural(expression);
            break;
        default:
            break;
    }
    return tmp;
}
```

Function `codegen_expression` is incomplete, because it only supports expressions of category `Natural`, which is the simplest case. The exercises challenge you to continue the code generation for the other cases (additions, calls, etc.).

Generating code to evaluate an expression of category `Natural` consists of loading the value of the natural number (given by the lexical token) into a temporary register. That code is generated by the function `codegen_natural` below:

```
int codegen_natural(struct node *natural) {
    printf("  %%%d = add i32 %s, 0\n", temporary, natural->token);
    return temporary++;
}
```

Notice that the function `codegen_natural` emits the code to load the value of the natural number into *the next* temporary register (using the `temporary` counter). Then, the number of the temporary register is returned and incremented.

Exercises

In this tutorial we only consider the `integer` type and completely forget about `double` values (for now). Hence, all instructions operate on `i32` values exclusively.

1. Take the above example in file `factorial.ll` as a starting point. Manually compose the LLVM IR for the `main` function to read an integer value, calculate its factorial, and write the result. The read/write functions are available in `io.c` and the commands that follow should correctly output the result. Notice that we must call `read(0)` to read a new value.

   ```
   $ llc factorial.ll -o factorial.s
   $ clang factorial.s io.c -o factorial
   $ ./factorial
   12                  [input]
   479001600           [output]
   ```

2. The code generator already compiles functions like `func(integer i) = 10` where the expression consists of a single `Natural` value. Modify the code generator to support the `Identifier` case of expressions. It should then be able to compile the program `identity(integer n) = n` by generating code that reads a variable into a temporary register.

3. Implement code generation for the multiplication operation. The code generation process for this operation can be outlined with the following pseudocode:

   - `t1 = codegen_expression(left child)`
   - `t2 = codegen_expression(right child)`
   - `new_temporary =` result of multiplying `t1 * t2`
   - return `new_temporary` and post-increment the temporary counter

4. Implement code generation for the other three arithmetic operations (addition, subtraction and division). At this point it should be able to compile `mod(integer a, integer b) = a-a/b*b` and similar functions.

5. Implement code generation for function calls, ensuring the generation of calls with appropriate arguments. Please be aware of the convention to prefix function names with an underscore.

6. Implement code generation for if-then-else expressions, with the same exact semantics as the the ternary operator `?:` existing in C and Java.

Finally, test your solution with the following Petit program:

```
factorial(integer n) = if n then n * factorial(n-1) else 1
main(integer i) = write(factorial(read(0)))
```

By following the link in the references below, you can find other Petit programs such as `primes.pt` to test your solution with more advanced programs.

Author

Raul Barbosa (University of Coimbra)

References

M. Rodler, M. Egevig (2023). Mapping High Level Constructs to LLVM IR. https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io

Aho, A. V. (2006). Compilers: Principles, techniques and tools, 2nd edition. Pearson Education.

Barbosa, R. (2023). Petit programming language and compiler. https://github.com/rbbarbosa/Petit

LLVM Project (2023). LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html