LLVM: Generate LLVM Intermediate Representation

Compiler Construction '13 Final Report

Andy Roulin Sacha Bron

EPFL

andy.roulin@epfl.ch sacha.bron@epfl.ch

1. Introduction

The goal of our Compiler Construction course was to build a full compiler for a toy object-oriented language named Tool targeting the Java Virtual Machine (JVM). We have proceeded by first implementing the front-end part which consists of a lexical analyzer, a parser, a name analyzer and a type checker. The back-end part was then written in order to generate Java Bytecode for the JVM from the Abstract Syntax Tree obtained with the front-end.

Our project aims to replace the back-end targeting the JVM with a new one targeting the LLVM platform. LLVM is a fast-developing compiler platform consisting of a architecture-independent intermediate representation (LLVM-IR), optimizers for programs written in LLVM-IR and a set of front-end and back-end compilers working with this intermediate representation. Front-end compilers translate a language to LLVM-IR while back-end go from LLVM-IR to native code (or other languages).

2. Examples

As an introductory example, a detailed compiled version of a factorial program written in Tool is given. As LLVM-IR, like other assembly languages, results in long code, this example is simple and not complete. Further difficulties like object-orientation or static single assignment will be detailed later in the text.

One Tool code for factorial function:

```
def computeFactorial(num : Int) : Int = {
    var num_aux : Int;
    if (num < 0 || num == 1)
        num_aux = 1;
    else
        num_aux = num *
            (this.computeFactorial(num - 1));
    return num_aux; }
```

and its compiled version (lines beginning with ";" are comments):

```
define i32 @Fact$computeFactorial(%struct.Fact*
        %this, i32 %\_num) nounwind ssp

    ; Store the argument in a pointer
    ; %num is of type i32*
    %num = alloca i32, align 4
    store i32 %\_num, i32* %num, align 4

    ; var num_aux: Int
    %num\_aux = alloca i32, align 4

    ; %1 = num
    %1 = load i32* %num, align 4
    ; %3 = 0
    %2 = alloca i32, align 4
    store i32 0, i32* %2, align 4
    %3 = load i32* %2, align 4
    ; %4 = num < 0
    %4 = icmp slt i32 %1, %3
    br label %5

; <label>: %5
    ; Short-circuiting OR if num < 0
    br i1 %4, label %12, label %6

; <label>: %6
    ; %7 = num
    %7 = load i32* %num, align 4
    ; %9 = 1
    %8 = alloca i32, align 4
    store i32 1, i32* %8, align 4
    %9 = load i32* %8, align 4
    ; %10 = (num == 1)
```

```
    %10 = icmp eq  i32 %7, %9
    br label %11

; <label>: %11
    br label %12

; <label>: %12
    ; solve short-circuiting with PHI function
    ; that decides which value to give to %13
    ; based on the branch we are coming from
    %13 = phi i1 [ true, %5 ], [ %10, %11 ]

    ; if branching
    br i1 %13, label %14, label %17

; <label>: %14
    ; %16 = 1
    %15 = alloca i32, align 4
    store i32 1, i32* %15, align 4
    %16 = load i32* %15, align 4
    ; num_aux = 1
    store i32 %16, i32* %num\_aux, align 4
    br label %31

; <label>: %17
    ; %18 = num
    %18 = load i32* %num, align 4
    ; %20 = this
    %19 = alloca %struct.Fact*, align 8
    store %struct.Fact* %this,%struct.Fact**
        %19, align 8
    %20 = load %struct.Fact** %19, align 8
    ; %21 = this
    %21 = load i32* %num, align 4
    ; %23 = 1
    %22 = alloca i32, align 4
    store i32 1, i32* %22, align 4
    %23 = load i32* %22, align 4
    ; %24 = num - 1
    %24 = sub nsw i32 %21, %23
    ; %26 = this.vtable
    %25 = getelementptr inbounds
        %struct.Fact* %20, i32 0, i32 0
    %26 = load %struct.Fact$vtable** %25,
        align 8
    ; %28 = this.vtable.computeFactorial
    %27 = getelementptr inbounds
        %struct.Fact$vtable* %26,i32 0,i32 0
    %28 = load i32 (%struct.Fact*, i32)**
        %27, align 8
    ; %29 = this.computeFactorial(num-1)
    %29 = call i32 %28(%struct.Fact*
        %20, i32 %24)
    ; %30 = num_aux
    ;   * this.computeFactorial(num-1)
    %30 = mul nsw i32 %18, %29
    ; update num_aux
    store i32 %30, i32* %num\_aux, align 4
    br label %31

; <label>: %31
    ; return num_aux
    %32 = load i32* %num\_aux, align 4
    ret i32 %32
```

3. Implementation

3.1 Sketch of the implementation

The code generation begins with the generation of some headers for the LLVM-IR files. These are the Module Identifier, the target datalayout which specifies the size in bits of common types (for example i32, integer 32 bits, takes 32 bits) and the target triple which is the target architecture, vendor, operating system and, eventually, environment (in order to generate native code later). Headers also include structures declarations for the List type (see memory management part) and the IntArray type (see Int Arrays part).

Then we generate the class headers, i.e. the structures and vtable global variables for all classes (See object orientation part). Following all structures declarations comes the definition of all functions from all classes. Then comes the main function (no structures needs to be generated for the main object).

Finally declarations of C standard library functions(malloc, printf, strlen, ...) and some helper functions written directly in LLVM-IR (concatString, concatStringInt, List$add, ...) are included at the end of a LLVM-IR file.

The compilation of a function follows the pattern seen in the course with two scala functions ”compileExpression” and ”compileStatement”.

Note that we generate LLVM-IR by hand, i.e. we have not used the C/C++ API (difficult to interface with scala) nor have we used a Java/Scala library (available ones use LLVM-IR outdated versions)

3.2 Static Single Assignment Form

As an intermediate representation (IR), LLVM-IR has an infinite number of registers (named %<regName>) but a register can only be assigned once. This last property is what is called the Static Single Assignment Form (SSA).

Therefore, a code like

```
%x = 1;
%x = 2;
%y = x;
```

should be for example translated as

```
%x = 1;
%x2 = 2;
%y = x2;
```

The first consequence of this form is to force us to have a convention when compiling expressions of unknown size that will therefore use an unknown number of registers: the value of the expression should be stored in the last register used. For example, an assignment in Tool:

```
a = expr
```

Let [...] denote compilation operator:

```
[ a = expr ] =
    // This should as a last instruction put
    // expr value in a fresh register
    [ expr ]
     // So that this instruction know where
     // the value is (last register used)
    store typeOfExpr %freshRegister, typeOfA* %a
```

A second consequence is about the branching structures which implies a PHI instruction. Let's consider a non-SSA code:

```
if(z) { x = 1; }
else { x = 2; }
y = x;
```

Its SSA version:

```
if(%z) { %x1 = 1; }
else { %x2 = 2; }
%y = %x???; // Which register to use ?
```

This problem can be handled in two ways. The first one is to use pointers and store values in memory, as memory is not in SSA in LLVM:

```
if(%z) { store i32 1, i32* %x } // %x is now a pointer
else { store i32 2, i32* %x }
%y = load i32* %x; // load from memory
```

The other way is to use the PHI instruction which is a native LLVM instruction. It decides which register to take based on the branch we are coming from:

```
if(%z) { %x1 = 1; }
else { %x2 = 2; }
%y = phi [ %x1, %ifTrueLabel ], [ %x2, %ifFalseLabel];
```

3.3 Object-Orientation

3.3.1 Object Layout

A class C with fields $f1, ..., f_n$ and methods m_1, m_m will be compiled to two structures: The first one is $C\$vtable$ containing m function pointers with same signatures as methods $m_1, ..., m_m$. A global variable $C\$gvtable$ of type $C\$vtable$ is also generated. His elements are set to $m_1, ..., m_m$. The second structure is the structure C which will have as a first element a $C\$vtable$ structure and then all $f_1, ..., f_n$ fields.

Then the methods code are compiled into LLVM methods named $C\$m_1, ..., C\$m_m$ and a new function $new\$C$ is also generated. Its goal is to malloc a new object o of type C and set $o.vtable$ to $C\$gvtable$ (Dynamic dispatch).

Now to call m_1 on object o, we have simply to call $o.vtable.m_1()$.

3.3.2 Inheritance

A class D which extends from C with new fields $g_1, ..., g_k$ and new methods $n_1, ..., n_l$ also generates the same types of structures but adds elements from C. $D\$vtable$ first has the same function pointers as $C\$vtable$ and then adds D new methods at the end.

Therefore $D\$vtable$ will contains first function pointers with signatures of $m_1, ..., m_m$ and then of $n_1, ..., n_l$. This way a $D\$vtable$ can be cast to a $C\$vtable$ because the of the order of elements in the structure. For example a code like $Cc = newD(); c.m1()$ will make such a cast.

The global variable $D\$gvtable$ will be of type $D\$vtable$ and will have its fields set to the function of C $m1, ..., m_m$ for the function pointers inherited by C and the functions of D $n_1, ..., n_l$ for the new methods (no polymorphism/method overrriding for now).

The structure D has as a first element a $D\$vtable$ and then in order the fields from C $f_1, ..., f_n$ and the

new fields $g_1, ..., g_k$. Again the order of elements will allow a cast to C structure in a code like C c = new $D()$.

Finally $D\$n_1, ..., D\$n_l$ codes are compiled and the $new\$D$ function must malloc a new object of type D and set its vtable to $D\$gvtable$.

3.3.3 Polymorphism/Method overriding

If the class D overrides methods m_i from C, then $D\$m_i$ will be generated and the only thing we have to do is to change in $D\$gvtable$ the function pointer which previously pointed to $C\$m_i$ and make it points to $D\$m_i$.

3.4 Println and String/Integers concatenation

In order to print strings, we use the printf function from the C standard library. Concatenate strings and/or integers was a bit more involved since no standard functions implements this functionality. We simply implemented these functions in C, compiled them to LLVM-IR using the clang compiler and packaged these with every compiled Tool code.

3.5 Int Arrays

Integers arrays of runtime-known length are not supported natively in LLVM-IR. As in C, we have to use a integer pointer and malloc the array of integers. Another problem was that, again as in C, you must always pass the array along with its size, otherwise one does not know where the array ends (this problem does not arise with strings as the null byte ends a string). To solve this little inconvenience, we designed a structure "IntArray" that packs together the integer pointer (the array) and its size (an integer). We also implemented some methods for this structure as for example IntArray$Length or IntArray$AssignAtIndex. In fact we add an new class to every LLVM-IR we compile.

3.6 Memory Management

For allocation of objects on the heap, we use malloc from the C standard library. To avoid memory leaks at the end of the program (for example reported by valgrind), we have put in place a simple system for freeing memory allocated. Each time we malloc an object, we put a pointer to this object in a global linked-list. At the end of the program (last line before return of the function main), we traverse the list and free all pointers.

3.7 Name collisions

By adding new structures and functions that the original Tool programmer ignore (List for memory management, IntArray or simply vtable structures) we face name collisions i.e. the Tool programmer could have given a name such as IntArray to an object and the latter will collide with our hidden class. To solve this problem, we used characters in such class names and functions definition that are forbidden in Tool identifiers (for example dot or dollar signs). C Standard library functions such as strlen were not changed but they cannot cause a collision because any function named strlen of a Tool program will have the form $ClassName\$strlen$ and a class named strlen will lead to a structure named $struct.className$ where the dot is forbidden in Tool.

4. Possible Extensions

One way to enhance our back-end to LLVM-IR would be to make better use of the PHI function. We did not used them much, prefering to use pointers and access memory which is not SSA. This is not considered as a good practice in LLVM-IR.

Another interesting and useful extension would be to implement a garbage collector (mark and sweep or reference counting) to properly do automatic memory management.

A great advantage of generating LLVM-IR is to now dispose of a set of compiler from LLVM-IR to other languages espcially native code. Using "llc", the official LLVM-IR compiler or simply "clang" the C compiler from LLVM (which is able to compile LLVM-IR), we can obtain a machine-code binary for our operating system. Another great back-end compiler is "emscripten" which translate LLVM-IR to javascript and let us have Tool code running in our browsers! These two examples (binaries and javascript) are included in the examples directory of this report.

References

LLVM Documentation: http://llvm.org/docs/ (last access: January 10, 2014)