Handout 9 (LLVM, SSA and CPS)

Reflecting on our two tiny compilers targetting the JVM, the code generation part was actually not so hard, no? Pretty much just some post-traversal of the abstract syntax tree, yes? One of the reasons for this ease is that the JVM is a stack-based virtual machine and it is therefore not hard to translate deeply-nested arithmetic expressions into a sequence of instructions manipulating the stack. The problem is that "real" CPUs, although supporting stack operations, are not really designed to be *stack machines*. The design of CPUs is more like, here is a chunk of memory—compiler, or better compiler writers, do something with it. Consequently, modern compilers need to go the extra mile in order to generate code that is much easier and faster to process by CPUs. To make this all tractable for this module, we target the LLVM Intermediate Language. In this way we can take advantage of the tools coming with LLVM. For example we do not have to worry about things like register allocations.

LLVM[1] is a beautiful example that projects from Academia can make a difference in the World. LLVM started in 2000 as a project by two researchers at the University of Illinois at Urbana-Champaign. At the time the behemoth of compilers was gcc with its myriad of front-ends for other languages (C++, Fortran, Ada, Go, Objective-C, Pascal etc). The problem was that gcc morphed over time into a monolithic gigantic piece of m…ehm software, which you could not mess about in an afternoon. In contrast, LLVM is designed to be a modular suite of tools with which you can play around easily and try out something new. LLVM became a big player once Apple hired one of the original developers (I cannot remember the reason why Apple did not want to use gcc, but maybe they were also just disgusted by gcc's big monolithic codebase). Anyway, LLVM is now the big player and gcc is more or less legacy. This does not mean that programming languages like C and C++ are dying out any time soon—they are nicely supported by LLVM.

We will target the LLVM Intermediate Language, or LLVM Intermediate Representation (short LLVM-IR). The LLVM-IR looks very similar to the assembly language of Jasmin and Krakatau. It will also allow us to benefit from the modular structure of the LLVM compiler and let for example the compiler generate code for different CPUs, like X86 or ARM. That means we can be agnostic about where our code actually runs. We can also be ignorant about optimising code and allocating memory efficiently.

However, what we have to do for LLVM is to generate code in *Static Single-Assignment* format (short SSA), because that is what the LLVM-IR expects from us. A reason why LLVM uses the SSA format, rather than JVM-like stack instructions, is that stack instructions are difficult to optimise—you cannot just re-arrange instructions without messing about with what is calculated on the stack. Also it is hard to find out if all the calculations on the stack are actually necessary and not by chance dead code. The JVM has for all these obstacles

[1] http://llvm.org

sophisticated machinery to make such "high-level" code still run fast, but let's say that for the sake of argument we do not want to rely on it. We want to generate fast code ourselves. This means we have to work around the intricacies of what instructions CPUs can actually process fast. This is what the SSA format is designed for.

The main idea behind the SSA format is to use very simple variable assignments where every tmp-variable is assigned only once. The assignments also need to be primitive in the sense that they can be just simple operations like addition, multiplication, jumps, comparisons and so on. Say, we have an expression $((1 + a) + (3 + (b * 5)))$, then the corresponding SSA format is

```
1 let tmp0 = add 1 a in
2 let tmp1 = mul b 5 in
3 let tmp2 = add 3 tmp1 in
4 let tmp3 = add tmp0 tmp2 in tmp3
```

where every variable is used only once (we could not write `tmp1 = add 3 tmp1` in Line 3 for example).

There are sophisticated algorithms for imperative languages, like C, that efficiently transform a high-level program into SSA format. But we can ignore them here. We want to compile a functional language and there things get much more interesting than just sophisticated. We will need to have a look at CPS translations, where the CPS stands for Continuation-Passing-Style—basically black programming art or abracadabra programming. So sit tight.

LLVM-IR

Before we start, let's first have a look at the *LLVM Intermediate Representation* in more detail. The LLVM-IR is in between the frontends and backends of the LLVM framework. It allows compilation of multiple source languages to multiple targets. It is also the place where most of the target independent optimisations are performed.

What is good about our toy Fun language is that it basically only contains expressions (be they arithmetic expressions, boolean expressions or if-expressions). The exception are function definitions. Luckily, for them we can use the mechanism of defining functions in the LLVM-IR (this is similar to using JVM methods for functions in our earlier compiler). For example the simple Fun program

```
def sqr(x) = x * x
```

can be compiled to the following LLVM-IR function:

```
define i32 @sqr(i32 %x) {
   %tmp = mul i32 %x, %x
   ret i32 %tmp
}
```

First notice that all variable names, in this case `x` and `tmp`, are prefixed with `%` in the LLVM-IR. Temporary variables can be named with an identifier, such as

`tmp`, or numbers. In contrast, function names, since they are "global", need to be prefixed with an @-symbol. Also, the LLVM-IR is a fully typed language. The `i32` type stands for 32-bit integers. There are also types for 64-bit integers (`i64`), chars (`i8`), floats, arrays and even pointer types. In the code above, `sqr` takes an argument of type `i32` and produces a result of type `i32` (the result type is in front of the function name, like in C). Each arithmetic operation, for example addition and multiplication, are also prefixed with the type they operate on. Obviously these types need to match up… but since we have in our programs only integers, for the moment `i32` everywhere will do. We do not have to generate any other types, but obviously this is a limitation in our Fun language.

There are a few interesting instructions in the LLVM-IR which are quite different than what we have seen in the JVM. Can you remember the kerfuffle we had to go through with boolean expressions and negating the condition? In the LLVM-IR, branching if-conditions is implemented differently: there is a separate `br`-instruction as follows:

```
br i1 %var, label %if_br, label %else_br
```

The type `i1` stands for booleans. If the variable is true, then this instruction jumps to the if-branch, which needs an explicit label; otherwise it jumps to the else-branch, again with its own label. This allows us to keep the meaning of the boolean expression "as is" when compiling if's—thanks god no more negating the boolean. A value of type boolean is generated in the LLVM-IR by the `icmp`-instruction. This instruction is for integers (hence the `i`) and takes the comparison operation as argument. For example

```
icmp eq  i32 %x, %y      ; for equal
icmp sle i32 %x, %y      ; signed less or equal
icmp slt i32 %x, %y      ; signed less than
icmp ult i32 %x, %y      ; unsigned less than
```

Note that in some operations the LLVM-IR distinguishes between signed and unsigned representations of integers.

It is also easy to call another function in LLVM-IR: as can be seen from Figure 1 we can just call a function with the instruction `call` and can also assign the result to a variable. The syntax is as follows

```
%var = call i32 @foo(...args...)
```

where the arguments can only be simple variables, not compound expressions.

Conveniently, you can use the program `lli`, which comes with LLVM, to interpret programs written in the LLVM-IR. So you can easily check whether the code you produced actually works. To get a running program that does something interesting you need to add some boilerplate about printing out numbers and a main-function that is the entry point for the program (see Figure 1 for a complete listing). Again this is very similar to the boilerplate we needed to add in our JVM compiler.

```
1  @.str = private constant [4 x i8] c"%d\0A\00"
2
3  declare i32 @printf(i8*, ...)
4
5  ; prints out an integer
6  define i32 @printInt(i32 %x) {
7     %t0 = getelementptr [4 x i8], [4 x i8]* @.str, i32 0, i32 0
8     call i32 (i8*, ...) @printf(i8* %t0, i32 %x)
9     ret i32 %x
10 }
11
12 ; square function
13 define i32 @sqr(i32 %x) {
14    %tmp = mul i32 %x, %x
15    ret i32 %tmp
16 }
17
18 ; main
19 define i32 @main() {
20   %1 = call i32 @sqr(i32 5)
21   %2 = call i32 @printInt(i32 %1)
22   ret i32 %1
23 }
```

Figure 1: An LLVM-IR program for calculating the square function. It calls this function in `@main` with the argument 5. The code for the `sqr` function is in Lines 13 – 16. The main function calls `sqr` and then prints out the result. The other code is boilerplate for printing out integers.

You can generate a binary for the program in Figure 1 by using the `llc`-compiler and then `gcc`, whereby `llc` generates an object file and `gcc` (that is clang) generates the executable binary:

```
llc -filetype=obj sqr.ll
gcc sqr.o -o a.out
./a.out
> 25
```

Our Own Intermediate Language

Remember compilers have to solve the problem of bridging the gap between "high-level" programs and "low-level" hardware. If the gap is too wide for one step, then a good strategy is to lay a stepping stone somewhere in between. The LLVM-IR itself is such a stepping stone to make the task of generating and optimising code easier. Like a real compiler we will use our own stepping stone which I call the *K-language*. For what follows recall the various kinds of expres-

sions in the Fun language. For convenience the Scala code of the corresponding abstract syntax trees is shown on top of Figure 2. Below is the code for the abstract syntax trees in the K-language. In K, here are two kinds of syntactic entities, namely *K-values* and *K-expressions*. The central constructor of the K-language is `KLet`. For this recall in SSA that arithmetic expressions such as $((1 + a) + (3 + (b * 5)))$ need to be broken up into smaller "atomic" steps, like so

```
let tmp0 = add 1 a in
let tmp1 = mul b 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  tmp3
```

Here `tmp3` will contain the result of what the whole expression stands for. In each individual step we can only perform an "atomic" operation, like addition or multiplication of a number and a variable. We are not allowed to have for example an if-condition on the right-hand side of an equals. Such constraints are enforced upon us because of how the SSA format works in the LLVM-IR. By having in `KLet` taking first a string (standing for an intermediate result) and second a value, we can fulfil this constraint "by construction"—there is no way we could write anything else than a value.

To sum up, K-values are the atomic operations that can be on the right-hand side of equal-signs. The K-language is restricted such that it is easy to generate the SSA format for the LLVM-IR.

CPS-Translations

CPS stands for Continuation-Passing-Style. It is a kind of programming technique often used in advanced functional programming. Before we delve into the CPS-translation for our Fun language, let us look at CPS-versions of some well-known functions. Consider

```
def fact(n: Int) : Int =
  if (n == 0) 1 else n * fact(n - 1)
```

This is clearly the usual factorial function. But now consider the following version of the factorial function:

```
def factC(n: Int, ret: Int => Int) : Int =
  if (n == 0) ret(1)
  else factC(n - 1, x => ret(n * x))

factC(3, identity)
```

This function is called with the number, in this case 3, and the identity-function (which returns just its input). The recursive calls are:

```
// Fun language (expressions)
abstract class Exp
abstract class BExp

case class Call(name: String, args: List[Exp]) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
case class Var(s: String) extends Exp
case class Num(i: Int) extends Exp
case class Aop(o: String, a1: Exp, a2: Exp) extends Exp
case class Sequence(e1: Exp, e2: Exp) extends Exp
case class Bop(o: String, a1: Exp, a2: Exp) extends BExp


// K-language (K-expressions, K-values)
abstract class KExp
abstract class KVal

case class KVar(s: String) extends KVal
case class KNum(i: Int) extends KVal
case class Kop(o: String, v1: KVal, v2: KVal) extends KVal
case class KCall(o: String, vrs: List[KVal]) extends KVal
case class KWrite(v: KVal) extends KVal

case class KIf(x1: String, e1: KExp, e2: KExp) extends KExp
case class KLet(x: String, v: KVal, e: KExp) extends KExp
case class KReturn(v: KVal) extends KExp
```

Figure 2: Abstract syntax trees for the Fun language.

```
factC(2, x => identity(3 * x))
factC(1, x => identity(3 * (2 * x)))
factC(0, x => identity(3 * (2 * (1 * x))))
```

Having reached 0, we get out of the recursion and apply 1 to the continuation (see if-branch above). This gives

```
identity(3 * (2 * (1 * 1)))
= 3 * (2 * (1 * 1))
= 6
```

which is the expected result. If this looks somewhat familiar to you, than this is because functions with continuations can be seen as a kind of generalisation of tail-recursive functions. Anyway notice how the continuations is "stacked up" during the recursion and then "unrolled" when we apply 1 to the continuation. Interestingly, we can do something similar to the Fibonacci function where in the traditional version we have two recursive calls. Consider the following function

```
def fibC(n: Int, ret: Int => Int) : Int =
  if (n == 0 || n == 1) ret(1)
  else fibC(n - 1,
              r1 => fibC(n - 2,
                r2 => ret(r1 + r2)))
```

Here the continuation is a nested function essentially wrapping up the second recursive call. Let us check how the recursion unfolds when called with 3 and the identity function:

```
fibC(3, id)
fibC(2, r1 => fibC(1, r2 => id(r1 + r2)))
fibC(1, r1 =>
   fibC(0, r2 => fibC(1, r2a => id((r1 + r2) + r2a))))
fibC(0, r2 => fibC(1, r2a => id((1 + r2) + r2a)))
fibC(1, r2a => id((1 + 1) + r2a))
id((1 + 1) + 1)
(1 + 1) + 1
3
```

Let us now come back to the CPS-translations for the Fun language. The main difficulty of generating instructions in SSA format is that large compound expressions need to be broken up into smaller pieces and intermediate results need to be chained into later instructions. To do this conveniently, CPS-translations have been developed. They use functions ("continuations") to represent what is coming next in a sequence of instructions. In our case, continuations are functions of type `KVal` to `KExp`. They can be seen as a sequence of `KLets` where there is a "hole" that needs to be filled. Consider for example

```
let tmp0 = add 1 a in
let tmp1 = mul □ 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  tmp3
```

where in the second line is a □ which still expects a `KVal` to be filled in before it becomes a "proper" `KExp`. When we apply an argument to the continuation (remember they are functions) we essentially fill something into the corresponding hole. The code of the CPS-translation is

```
def CPS(e: Exp)(k: KVal => KExp) : KExp =
  e match { ... }
```

where `k` is the continuation and `e` is the expression to be compiled. In case we have numbers or variables, we can just apply the continuation like

```
k(KNum(n))     k(KVar(x))
```

This would just fill in the □ in a `KLet`-expression. More interesting is the case for an arithmetic expression.

```
case Aop(o, e1, e2) => {
    val z = Fresh("tmp")
    CPS(e1)(y1 =>
      CPS(e2)(y2 => KLet(z, Kop(o, y1, y2), k(KVar(z)))))
}
```

For more such rules, have a look to the code of the fun-llvm compiler.