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 are some operations and a chunk of memory—compiler, or better compiler writers, do something with them. Consequently, modern compilers need to go the extra mile in order to generate code that is much easier and faster to process by actual CPUs, rather than running code on virtual machines that introduce an additional layer of indirection. 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.[1] For example we do not have to worry about things like register allocations. By using LLVM, however, we also have to pay price in the sense that generating code gets harder…unfortunately.

LLVM and LLVM-IR

LLVM 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 different programming 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 complicated 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. Targetting LLVM-IR will also allow us to benefit from the modular structure of the LLVM compiler and we can let, for example, the compiler generate code for different CPUs, say X86 or ARM. That means we can be agnostic about where our code is actually going to run.[2] We can also be somewhat ignorant about optimising our code and about allocating memory efficiently: the LLVM tools will take care of all this.

[1] `http://llvm.org`

[2] Anybody want to try to run our programs on Android phones?

However, what we have to do in order to make LLVM to play ball 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 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, function calls and so on. Say, we have an expression $((1 + a) + (foo(3 + 2) + (b * 5)))$, then the corresponding SSA format is

```
1 let tmp0 = add 1 a in
2 let tmp1 = add 3 2 in
3 let tmp2 = call foo(tmp1)
4 let tmp3 = mul b 5 in
5 let tmp4 = add tmp2 tmp3 in
6 let tmp5 = add tmp0 tmp4 in
7   return tmp5
```

where every tmp-variable is used only once (we could, for example, not write `tmp1 = add tmp2 tmp3` in Line 5 even if this would not change the overall result).

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 however 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 "local" 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 (and which we lift in the final CW).

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 are 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 of booleans.

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. I assume you know what this means, otherwise just ask.

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 and numbers, but not compound expressions.

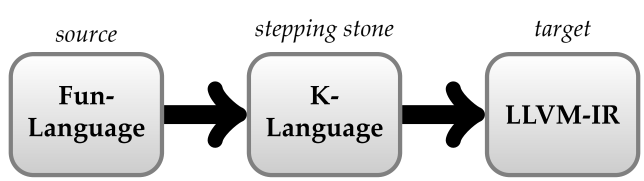
Conveniently, you can use the program `lli`, which comes with LLVM, to interpret programs written in the LLVM-IR. So you can easily check whether the code you produced actually works. To get a running program that does something interesting you need to add some boilerplate about printing out numbers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

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

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

Our Own Intermediate Language

Let's get back to our compiler: 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*.



To see why we need a stepping stone for generating code in SSA-format, considder again arithmetic expressions such as $((1 + a) + (3 + (b * 5)))$. They need to be broken up into smaller "atomic" steps, like so

```
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 the sqr-function in `@main` with the argument `5` (Line 20). The code for the `sqr` function is in Lines 13 – 16. It stores the result of sqr in the variable called `%i` and then prints out the contents of this variable in Line 21. The other code is boilerplate for printing out integers.

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

Here `tmp3` will contain the result of what the whole expression stands for. In each individual step we can only perform an "atomic" or "trival" operation, like addition or multiplication of a number or a variable. We are not allowed to have for example a nested addition or an if-condition on the right-hand side of an assignment. Such constraints are forced upon us because of how the SSA format works in the LLVM-IR. To simplify matters we represent assignments with two kinds of syntactic entities, namely *K-values* and *K-expressions*. K-values are all "things" that can appear on the right-hand side of an equal. Say we have the following definition for K-values:

```
enum KVal {
  case KVar(s: String)
  case KNum(n: Int)
  case KAop(op: String, v1: KVal, v2: KVal)
  case KCall(fname: String, args: List[KVal])
}
```

where a K-value can be a variable, a number or a "trivial" binary operation, such as addition or multiplication. The two arguments of a binary operation need to be K-values as well. Finally, we have function calls, but again each argument of the function call needs to be a K-value. One point we need to be careful, however, is that the arguments of the binary operations and function calls are in fact only variables or numbers. To encode this constraint into the type-system of Scala would make things too complicated—to satisfy this constraint is therefore on us. For our Fun-language, we will later on consider some further K-values.

Our K-expressions will encode the `let` and the `return` from the SSA-format (again for the moment we only consider these two constructors—later on we will have if-branches as well).

```
enum KExp {
  case KLet(x: String, v: KVal, e: KExp)
  case KReturn(v: KVal)
}
```

By having in `KLet` taking first a string (standing for an intermediate variable) and second a value, we can fulfil the SSA constraint in assignments "by construction"—there is no way we could write anything else than a K-value. Note that the third argument of a `KLet` is again a K-expression, meaning either another `KLet` or a `KReturn`. In this way we can construct a sequence of computations and return a final result. Of course we also have to ensure that all intermediary computations are given (fresh) names—we will use an (ugly) counter for this.

To sum up, K-values are the atomic operations that can be on the right-hand side of assignemnts. The K-language is restricted such that it is easy to generate the SSA format for the LLVM-IR. What remains to be done is a translation of our Fun-language into the K-language. The main difficulty is that we need to order the computationa—in teh K-language we only have linear sequence of instructions to need to be followed. Before we explain this, we have a look at some CPS-translation.

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:

```
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. So far we did not look at this generalisation in earnest. 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
```

The point of this section is that you are playing around with these simple definitions and make sure they calculate the expected result. In the next step, you should understand *how* these functions calculate the result with the continuations.

Worked Example

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, we use the CPS-translation mechanism. What continuations essentially solve is the following problem: Given an expressions

$$(1+2) * (3+4) \tag{1}$$

which of the subexpressions should be calculated first? We just arbitrarily going to decide that it will be from left to right. This means we have to tear the expression shown in (1) apart as follows:

$$(1+2) \qquad \text{and} \qquad \Box * (3+4)$$

The first one will give us a result, but the second one is not really an expression that makes sense. It will only make sense as the next step in the computation when we fill-in the result of $1+2$ into the "hole" indicated by $\Box$. Such incomplete expressions can be represented in Scala as functions

```
r => r * (3 + 4)
```

where `r` is any result that has been computed earlier. By the way in lambda-calculus notation we would write $\lambda r. r * (3+4)$ for the same function. To sum up, we 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

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

Lets look at concrete code. To simplify matters first, suppose our source language consists just of three kinds of expressions

```
enum Expr {
    case Num(n: Int)
    case Bop(op: String, e1: Expr, e2: Expr)
    case Call(fname: String, args: List[Expr])
}
```

The code of the CPS-translation is then of the form

```
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, we can just pass them to the continuations because numbers need not be further teared apart as they are already atomic. Passing the number to the continuation means we apply the continuation like

```
  case Num(i) => k(KNum(i))
```

This would just fill in the □ in a `KLet`-expression. There is no need to create a temporary variable for simple numbers. More interesting is the case for arithmetic operations.

```
case Aop(op, e1, e2) => {
  val z = Fresh("tmp")
  CPS(e1)(r1 =>
    CPS(e2)(r2 => KLet(z, KAop(op, r1, r2), k(KVar(z)))))
}
```

What we essentially have to do in this case is the following: compile the subexpressions `e1` and `e2`. They will produce some result that is stored in two temporary variables (assuming they are more complicated than just numbers). We need to use these two temporary variables and feed them into a new assignment of the form

```
let z = op □r1 □r2 in
...
```

Note that this assignment has two "holes", one for the left subexpression and one for the right subexpression. We can fill both holes by calling the CPS function twice—this is a bit of the black art in CPS. We can use the second call of CPS as the continuation of the first call: The reason is that

```
r1 => CPS(e2)(r2 => KLet(z, KAop(op, r1, r2), k(KVar(z))))
```

is a function from `KVal` to `KExp`. Once we created the assignment with the fresh temporary variable `z`, we need to "communicate" that the result of the computation of the arithmetic expression is stored in `z` to the computations that follow. In this way we apply the continuation `k` with this new variable (essentially we are plugging in a hole further down the line).

The last case we need to consider in our small expression language are function calls.

```
case Call(fname, args) => {
  def aux(args: List[Expr], vs: List[KVal]): KExp = args match {
       case Nil => {
         val z = Fresh("tmp")
         KLet(z, KCall(fname, vs), k(KVar(z)))
       }
       case a::as => CPS(a)(r => aux(as, vs ::: List(r)))
  }
  aux(args, Nil)
}
```

For them we introduce an auxiliary function that compiles first all function arguments—remember in our source language we can have calls such as $foo(3+4, g(3))$ where we first have to create temporary variables (and computations) for each argument. Therefore `aux` analyses the arguments from left to right. In case there is an argument `a` on the front of the list (the case `a::as`), we call CPS recursively for the corresponding subexpression. The temporary variable containing the result for this argument we add to the end of the K-values we have already analysed before. Again very conveniently we can use the recursive call to `aux` as the continuation for the computations that follow. If we reach the end of the argument list (the `Nil`-case), then we collect all the K-values `v1` to `vn` and call the function in the K-language like so

```
let z = call foo(v1,...,vn) in
...
```

Again we need to communicate the result of the function call, namely the fresh temporary variable `z`, to the rest of the computation. Therefore we apply the continuation `k` with this variable.

The last question we need to answer in the code (see Figure 2) is how we start the CPS-translation function, or more precisely with which continuation we should start CPS? It turns out that this initial continuation will be the last one that is called. What should be the last step in the computation? Yes, we need to return the temporary variable where the last result is stored (if it is a simple number, then we can just return this number). Therefore we cal CPS

```scala
// Source language: arithmetic expressions with function calls
enum Expr {
    case Num(n: Int)
    case Aop(op: String, e1: Expr, e2: Expr)
    case Call(fname: String, args: List[Expr])
}
import Expr._

// Target language
// "trivial" KValues
enum KVal {
    case KVar(s: String)
    case KNum(n: Int)
    case KAop(op: String, v1: KVal, v2: KVal)
    case KCall(fname: String, args: List[KVal])
}
import KVal._

// KExpressions
enum KExp {
    case KReturn(v: KVal)
    case KLet(x: String, v: KVal, e: KExp)
}
import KExp._

 ...

def CPS(e: Expr)(k: KVal => KExp): KExp = e match {
    case Num(i) => k(KNum(i))
    case Aop(op, l, r) => {
        val z = Fresh("z")
        CPS(l)(l =>
          CPS(r)(r => KLet(z, KAop(op, l, r), k(KVar(z)))))
    }
    case Call(fname, args) => {
        def aux(args: List[Expr], vs: List[KVal]) : KExp = args match {
            case Nil => {
                val z = Fresh("tmp")
                KLet(z, KCall(fname, vs), k(KVar(z)))
            }
            case a::as => CPS(a)(r => aux(as, vs ::: List(r)))
        }
        aux(args, Nil)
    }
}

def CPSi(e: Expr) : KExp = CPS(e)(KReturn(_))
```

Figure 2: CPS-translation for a simple expression language.

with the code

```
def CPSi(e: Expr) : KExp = CPS(e)(KReturn(_))
```

where we give the function `KReturn(_)`.

```scala
// 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 3: Abstract syntax trees for the Fun-language.