Handout 9 (LLVM, SSA and CPS)

Reflecting on our tiny compiler targetting the JVM, code generation was actually not so hard, no? One of the main reason for this ease is that the JVM is a stack-based virtual machine and it is, for example, not hard to translate arithmetic expressions into instructions manipulating the stack. The problem is that "real" CPUs, although supporting stack operations, are not really *stack machines*. They are just not optimised for this way of calculating things. The design of CPUs is more like, here is a piece of memory—compiler, or compiler writer, do something with it. Otherwise get lost. So in the name of raw speed, modern compilers go the extra mile and generate code that is much easier and faster to process by CPUs.

Another reason why it makes sense to go the extra mile is that stack instructions are very 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 this sophisticated machinery to make such "high-level" code still run fast, but let's say that for the sake of argument we want to not 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. To make this all tractable, we target the LLVM Intermediate Language. In this way we can take advantage of the tools coming with LLVM and for example do not have to worry about things like that CPUs have only a limited amount of registers.

LLVM[1] is a beautiful example that projects from Academia can make a difference in the world. LLVM was started in 2000 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 (e.g. gfortran, 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 was a modular suite of tools with which you could 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 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.

Targetting the LLVM-IR also means we can profit from the very modular structure of the LLVM compiler and let for example the compiler generate code for X86, or ARM etc. We can be agnostic about where our code actually runs. However, what we have to do is to generate code in *Static Single-Assignment* format (short SSA), because that is what the LLVM-IR expects from us and which

[1]http://llvm.org

is the intermediate format that LLVM can use to do cool things (like targetting strange architectures) and allocating memory efficiently.

The idea behind SSA is to use very simple variable assignments where every variable is assigned only once. The assignments also need to be extremely primitive in the sense that they can be just simple operations like addition, multiplication, jumps and so on. A typical program in SSA is

```
    x := 1
    y := 2
    z := x + y
```

where every variable is used only once. You cannot for example have

```
    x := 1
    y := 2
    x := x + y
```

because in this snippet `x` is assigned twice. There are sophisticated algorithm for imperative languages, like C, for efficiently transforming a program into SSA format, but we do not have to be concerned about those. 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, which stands for Continuation-Passing-Style—basically black art. So sit tight.

CPS-Translations

What is good about our simple fun language is that it basically only contains expressions (be they arithmetic expressions or boolean expressions). The only exceptions are function definitions, for which we can easily use the mechanism of defining functions in LLVM-IR. For example the simple fun program

```
def dble(x) = x * x
```

can be compiled into the following LLVM-IR function:

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