# Handout 9 (Static Analysis)

If we want to improve the safety and security of our programs, we need a more principled approach to programming. Testing is good, but as Dijkstra famously wrote:

> *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."*

While such a more principled approach has been the subject of intense study for a long, long time, only in the past few years some impressive results have been achieved. One is the complete formalisation and (mathematical) verification of a microkernel operating system called seL4.

`http://sel4.systems`

In 2011 this work was included in the MIT Technology Review in the annual list of the world's ten most important emerging technologies.[1] While this work is impressive, its technical details are too enormous for an explanation here. Therefore let us look at something much simpler, namely finding out properties about programs using *static analysis*.

Static analysis is a technique that checks properties of a program without actually running the program. This should raise alarm bells with you—because almost all interesting properties about programs are equivalent to the halting problem, which we know is undecidable. For example estimating the memory consumption of programs is in general undecidable, just like the halting problem. Static analysis circumvents this undecidability-problem by essentially allowing answers *yes* and *no*, but also *don't know*. With this "trick" even the halting problem becomes decidable…for example we could always say *don't know*. Of course this would be silly. The point is that we should be striving for a method that answers as often as possible either *yes* or *no*—just in cases when it is too difficult we fall back on the *don't-know*-answer. This might sound all like abstract nonsense. Therefore let us look at a concrete example.

### A Simple, Idealised Programming Language

Our starting point is a small, idealised programming language. It is idealised because we cut several corners in comparison with real programming languages. The language we will study contains, amongst other things, variables holding integers. Using static analysis, we want to find out what the sign of these integers (positive or negative) will be when the program runs. This sign-analysis seems like a very simple problem. But it will turn out even such simple problems, if approached naively, are in general undecidable, just like Turing's halting problem. I let you think why?

[1] `http://www2.technologyreview.com/tr10/?year=2011`

Is sign-analysis of variables an interesting problem? Well, yes—if a compiler can find out that for example a variable will never be negative and this variable is used as an index for an array, then the compiler does not need to generate code for an underflow-test. Remember some languages are immune to buffer-overflow attacks, but they need to add underflow and overflow checks everywhere. If the compiler can omit the underflow test, for example, then this can potentially drastically speed up the generated code.

What do programs in our programming language look like? The following grammar gives a first specification:

$$
\begin{array}{rcl}
\langle \mathit{Stmt} \rangle & ::= & \langle \mathit{label} \rangle \texttt{ :} \\
& | & \langle \mathit{var} \rangle \texttt{ := } \langle \mathit{Exp} \rangle \\
& | & \texttt{jmp? } \langle \mathit{Exp} \rangle \ \langle \mathit{label} \rangle \\
& | & \texttt{goto } \langle \mathit{label} \rangle \\
\langle \mathit{Prog} \rangle & ::= & \langle \mathit{Stmt} \rangle \ \ldots \ \langle \mathit{Stmt} \rangle
\end{array}
\qquad
\begin{array}{rcl}
\langle \mathit{Exp} \rangle & ::= & \langle \mathit{Exp} \rangle \texttt{ + } \langle \mathit{Exp} \rangle \\
& | & \langle \mathit{Exp} \rangle \texttt{ * } \langle \mathit{Exp} \rangle \\
& | & \langle \mathit{Exp} \rangle \texttt{ = } \langle \mathit{Exp} \rangle \\
& | & \langle \mathit{num} \rangle \\
& | & \langle \mathit{var} \rangle
\end{array}
$$

I assume you are familiar with such grammars.[2] There are three main syntactic categories: *statments* and *expressions* as well as *programs*, which are sequences of statements. Statements are either labels, variable assignments, conditional jumps (`jmp?`) and unconditional jumps (`goto`). Labels are just strings, which can be used as the target of a jump. We assume that in every program the labels are unique—otherwise if there is a clash we do not know where to jump to. The conditional jumps and variable assignments involve (arithmetic) expressions. Expressions are either numbers, variables or compound expressions built up from `+`, `*` and `=` (for simplicity reasons we do not consider any other operations). We assume we have negative and positive numbers, …`-2`, `-1`, `0`, `1`, `2`… An example program that calculates the factorial of 5 is as follows:

```
1        a := 1
2        n := 5
3  top:
4        jmp? n = 0 done
5        a := a * n
6        n := n + -1
7        goto top
8  done:
```

Each line of the program contains a statement. In the first two lines we assign values to the variables `a` and `n`. In line 4 we test whether `n` is zero, in which case we jump to the end of the program marked with the label `done`. If `n` is not zero, we multiply the content of `a` by `n`, decrease `n` by one and jump back to the beginning of the loop, marked with the label `top`. Another program in our language is shown in Figure 1. I let you think what it calculates.

[2] http://en.wikipedia.org/wiki/Backus-Naur_Form

```
      n := 6
      m1 := 0
      m2 := 1
loop:
      jmp? n = 0 done
      tmp := m2
      m2 := m1 + m2
      m1 := tmp
      n := n + -1
      goto top
done:
```

Figure 1: A mystery program in our idealised programming language. Try to find out what it calculates!

Even if our language is rather small, it is still Turing complete—meaning quite powerful. However, discussing this fact in more detail would lead us too far astray. Clearly, our programming is rather low-level and not very comfortable for writing programs. It is inspired by machine code, which is the code that is actually executed by a CPU. So a more interesting question is what is missing in comparison with real machine code? Well, not much…in principle. Real machine code, of course, contains many more arithmetic instructions (not just addition and multiplication) and many more conditional jumps. We could add these to our language if we wanted, but complexity is really beside the point here. Furthermore, real machine code has many instructions for manipulating memory. We do not have this at all. This is actually a more serious simplification because we assume numbers to be arbitrary small or large, which is not the case with real machine code. In real code basic number formats have a range and might over-flow or under-flow from this range. Also the number of variables in our programs is potentially unlimited, while memory in an actual computer, of course, is always limited somehow on any actual. To sum up, our language might look very simple, but it is not completely removed from practically relevant issues.

**An Interpreter**

Designing a language is like playing god: you can say what names for variables you allow; what programs should look like; most importantly you can decide what each part of the program should mean and do. While our language is rather simple and the meaning is rather straightforward, there are still places where we need to make a real choice. For example with conditional jumps, say the one in the factorial program:

```
jmp? n = 0 done
```

How should they work? We could introduce Booleans (`true` and `false`) and then jump only when the condition is `true`. However, since we have numbers in our language anyway, why not just encoding *true* as zero, and `false` as anything else? In this way we can dispense with the additional concept of Booleans, but also we could replace the jump above by

```
jmp? n done
```

which behaves exactly the same. But what does it mean that two jumps behave the same?

I hope the above discussion makes it already clear we need to be a bit more careful with our programs. Below we shall describe an interpreter for our programs, which specifies exactly how programs are supposed to be run…at least we will specify this for all *good* programs. By good programs we mean where for example all variables are initialised. Our interpreter will just crash if it cannot find out the value for a variable, because it is not initialised.

First we will pre-process our programs. This will simplify our definition of our interpreter later on. We will transform programs into *snippets*.