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 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. According to the John Regehr, an expert in the field of compilers, overflow checks can cause 5-10% slowdown, and in some languages even 100% for tight loops.[2]

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

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

I assume you are familiar with such grammars.[3] 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—if there is a clash, then 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 in oure programming language 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:
```

As can be seen 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`.

[2] http://blog.regehr.org/archives/1154

[3] 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!

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.

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 real machine code, which is the code that is 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 ridiculously simple, but it is not far 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 of statements, for example, is rather straightforward, there are still places where we need to make real choices. For example consider the 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 one, and `false` as zero? In this way we can dispense with the additional concept of Booleans.

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 programming language, which specifies exactly how programs are supposed to be run…at least we will specify this for all *good* programs. By good programs I mean where all variables are initialised, for example. Our interpreter will just crash if it cannot find out the value for a variable, in case it is not initialised. Also, we will assume that labels in good programs are unique, otherwise our programs will calculate "garbage".

First we will pre-process our programs. This will simplify the definition of our interpreter later on. We will transform programs into *snippets*. Their purpose is to simplify the definition of what the interpreter should do in case of a jump. A snippet is a label and all code that comes after the label. This essentially means a snippet is a *map* from labels to code. Given that programs are sequences (or lists) of statements, we can easily calculate the snippets by just traversing this sequence and recursively generating the map. Suppose a program is of the general form

$$stmt_1\ stmt_2\ \ldots\ stmt_n$$

The idea is to go through this sequence of statements one by one and check whether they are a label. If yes, we add the label and the remaining statements to our map. If no, we just continue with the next statement. To come up with a recursive definition for generating snippets, let us write [] for the program that does not contain any statement. Consider the following definition:

$$\begin{array}{ll} snippets([]) & \overset{\text{def}}{=} \varnothing \\ snippets(stmt\ \ rest) & \overset{\text{def}}{=} \begin{cases} snippets(rest)[label := rest] & \text{if } stmt = label\text{:} \\ snippets(rest) & \text{otherwise} \end{cases} \end{array}$$

In the first clause we just return the empty map for the program that does not contain any statement. In the second clause, we have to distinguish the case where the first statement is a label or not. As said before, if not, then we just "throw away" the label and recursively calculate the snippets for the rest of the program. If yes, then we do the same, but also update the map so that *label* now points to the rest of the statements. There is one small problem we need to overcome: our two programs have no label as *entry point*—that is where the execution starts. We usually assume that the first statement will be run first. To make this the default, it is convenient if we add to all our programs a default label, say `""` (the empty string). With this we can define our pre-processing of programs as follows

$$\mathit{preproc}(\mathit{prog}) \overset{\text{def}}{=} \mathit{snippets}(\texttt{"":}\ \mathit{prog})$$

Let us see how this pans out in practice. If we pre-process the factorial program shown earlier, we obtain the following map:

$\boxed{\texttt{""}} \mapsto$

```
    a := 1
    n := 5
  top:
    jmp? n = 0 done
    a := a * n
    n := n + -1
    goto top
  done:
```

$\boxed{\texttt{top}} \mapsto$

```
    jmp? n = 0 done
    a := a * n
    n := n + -1
    goto top
  done:
```

$\boxed{\texttt{done}} \mapsto []$

I highlighted the *keys* in this map. Since there are three labels in the factorial program (remember we added `""`), there are three keys. When running the factorial program and encountering a jump, then we only have to consult this snippets-map in order to find out what the next instruction should be.

We should now be in the position to define how a program should be run. In the context of interpreters, this "running" of programs is often called *evaluation*. Let us start with the definition of how expressions are evaluated. A first attempt might be the following recursive function:

$$\begin{aligned}
\mathit{eval\_exp}(\texttt{n}) &\overset{\text{def}}{=} n \qquad \text{if } \texttt{n} \text{ is a number like } \ldots\texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}\ldots \\
\mathit{eval\_exp}(\texttt{e}_1\, \texttt{+}\, \texttt{e}_2) &\overset{\text{def}}{=} \mathit{eval\_exp}(\texttt{e}_1) + \mathit{eval\_exp}(\texttt{e}_2) \\
\mathit{eval\_exp}(\texttt{e}_1\, \texttt{*}\, \texttt{e}_2) &\overset{\text{def}}{=} \mathit{eval\_exp}(\texttt{e}_1) * \mathit{eval\_exp}(\texttt{e}_2) \\
\mathit{eval\_exp}(\texttt{e}_1\, \texttt{=}\, \texttt{e}_2) &\overset{\text{def}}{=} \begin{cases} 1 & \text{if } \mathit{eval\_exp}(\texttt{e}_1) = \mathit{eval\_exp}(\texttt{e}_2) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

While this should look all relatively straightforward, still be very careful. There is a subtlety which can be easily overlooked: The function *eval_exp* takes an expression of our programming language as input and returns a number as output. Therefore whenever we have a number in our program, we just return this number—this is defined in the first clause above. Whenever we encounter an addition, well then we first evaluate the left-hand side, $\texttt{e}_1$ of the addition (this will give a number), then evaluate the right-hand side $\texttt{e}_2$ (this gives another number), and finally add both numbers together. Here is the subtlety: on the left-hand side of the $\overset{\text{def}}{=}$ we have a `+` (in the teletype font) which is the symbol for addition in our programming language. On the right-hand side we have $+$ which stands for the arithmetic operation from "mathematics" of adding two numbers. These are rather different concepts—one is a symbol (which we made up), and the other a mathematical operation. When we will have a look at an actual implementation of our interpreter, the mathematical operation will be the function for addition from the programming language in which we implement our interpreter. While the `+` is just a symbol that is used in our programming language. Clearly we have to use a symbol that is a good mnemonic for addition otherwise we will confuse the programmers working with our language. Therefore we use `+`. A similar choice is made for times in the third clause and equality in the fourth clause. Remember I wrote at the beginning of this section about being god when designing a programming language. You can see this here: we need to give meaning to symbols.

At the moment however, we are a poor fallible god. Look again at the grammar of our programming language and our definition. Clearly, an expression can contain variables. So far we we ignored them. What should our interpreter do with variables? They might change during the evaluation of a program. For example the variable `n` in the factorial program counts down from 5 up to 0. How can we improve our definition above to give also an answer whenever our interpreter encounters a variable in an expression? The solution is to add an *environment*, *env* as an additional input argument to our *eval_exp* function.

$$
\begin{array}{lcl}
\mathit{eval\_exp}(\texttt{n}, \mathit{env}) & \stackrel{\text{def}}{=} & n \qquad \text{if } \texttt{n} \text{ is a number like } \ldots\texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}\ldots \\
\mathit{eval\_exp}(\texttt{e}_1 \texttt{ + } \texttt{e}_2, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{eval\_exp}(\texttt{e}_1, \mathit{env}) + \mathit{eval\_exp}(\texttt{e}_2, \mathit{env}) \\
\mathit{eval\_exp}(\texttt{e}_1 \texttt{ * } \texttt{e}_2, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{eval\_exp}(\texttt{e}_1, \mathit{env}) * \mathit{eval\_exp}(\texttt{e}_2, \mathit{env}) \\
\mathit{eval\_exp}(\texttt{e}_1 \texttt{ = } \texttt{e}_2, \mathit{env}) & \stackrel{\text{def}}{=} & \begin{cases} 1 & \text{if } \mathit{eval\_exp}(\texttt{e}_1, \mathit{env}) = \mathit{eval\_exp}(\texttt{e}_2, \mathit{env}) \\ 0 & \text{otherwise} \end{cases} \\
\mathit{eval\_exp}(\texttt{x}, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{env}(x)
\end{array}
$$

This environment *env* also acts like a map: it associates variable names with the current value. In the clause for variables, we therefore consult this environment and return whatever value is currently stored for this variable. This is written $\mathit{env}(x)$ indicating that the environment acts like a function. If we call the function with x we obtain the corresponding number. What happens if an environment does not contain any value for, say, the variable x. Well, then our interpreter just crashes, or will raise an exception. In this case we had a "bad" program that tried to use a variable before it was initialised. With the second version of *eval_exp* we completed our definition for evaluating expressions.

Next comes the evaluation function for statements. We define this function in such a way that we evaluate a whole sequence of statements. Assume a program p and its preprocessed snippets sn. Then we define:

$$
\begin{array}{lcl}
\mathit{eval\_stmts}([], \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{env} \\
\mathit{eval\_stmts}(\texttt{label:}\ \mathit{rest}, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{eval\_stmts}(\mathit{rest}, \mathit{env}) \\
\mathit{eval\_stmts}(\texttt{x := e}\ \mathit{rest}, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{eval\_stmts}(\mathit{rest}, \mathit{env}[x := \mathit{eval\_exp}(\texttt{e}, \mathit{env})]) \\
\mathit{eval\_stmts}(\texttt{jmp? e lbl}\ \mathit{rest}, \mathit{env}) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{eval\_stmts}(\mathit{sn}(\texttt{lbl}), \mathit{env}) & \\ & \text{if } \mathit{eval\_exp}(\texttt{e}, \mathit{env}) = 1 \\ \mathit{eval\_stmts}(\mathit{rest}, \mathit{env}) & \text{otherwise} \end{cases} \\
\mathit{eval\_stmts}(\texttt{goto lbl}\ \mathit{rest}, \mathit{env}) & \stackrel{\text{def}}{=} & \mathit{eval\_stmts}(\mathit{sn}(\texttt{lbl}), \mathit{env})
\end{array}
$$