Lecture Notes: The WHILE Language and Program Semantics

17-355/17-665/17-819O: Program Analysis (Spring 2019)
Jonathan Aldrich
aldrich@cs.cmu.edu
course material developed with Claire Le Goues

1 The WHILE Language

We will study the theory of analyses using a simple programming language called WHILE, with various extensions. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties. It is a simple imperative language, with (to start!) assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

We use the following metavariables to describe different categories of syntax. The letter on the left will be used as a variable representing a piece of a program. On the right, we describe the kind of program piece that variable represents:

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
P	boolean predicates (BExp)

The syntax of WHILE is shown below. Statements S can be an assignment $x := a$; a skip statement, which does nothing;[1] and `if` and `while` statements, with boolean predicates P as conditions. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators (op_a). Predicates are represented by Boolean expressions that include true, false, the negation of another Boolean expression, Boolean operators op_b applied to other Boolean expressions, and relational operators op_r applied to arithmetic expressions.

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl@{\qquad}lcl}
S & ::= & x := a & P & ::= & \text{true} & a & ::= & x & op_b & ::= & \text{and} \mid \text{or} \\
 & \mid & \text{skip} & & \mid & \text{false} & & \mid & n & op_r & ::= & < \mid \leq \mid = \\
 & \mid & S_1;\ S_2 & & \mid & \text{not } P & & \mid & a_1\ op_a\ a_2 & & \mid & > \mid \geq \\
 & \mid & \text{if } P \text{ then } S_1 \text{ else } S_2 & & \mid & P_1\ op_b\ P_2 & & & & op_a & ::= & + \mid - \mid * \mid / \\
 & \mid & \text{while } P \text{ do } S & & \mid & a_1\ op_r\ a_2 & & & & & &
\end{array}$$

[1]Similar to a lone semicolon or open/close bracket in C or Java

2 WHILE3ADDR: A Representation for Analysis

For analysis, the source-like definition of WHILE can sometimes prove inconvenient. For example, WHILE has three separate syntactic forms—statements, arithmetic expressions, and boolean predicates—and we would have to define the semantics and analysis of each separately to reason about it. A simpler and more regular representation of programs will help simplify certain of our formalisms.

As a starting point, we will eliminate recursive arithmetic and boolean expressions and replace them with simple atomic statement forms, which are called *instructions*, after the assembly language instructions that they resemble. For example, an assignment statement of the form $w = x * y + z$ will be rewritten as a multiply instruction followed by an add instruction. The multiply assigns to a temporary variable t_1, which is then used in the subsequent add:

$$\begin{aligned} t_1 &= x * y \\ w &= t_1 + z \end{aligned}$$

As the translation from expressions to instructions suggests, program analysis is typically studied using a representation of programs that is not only simpler, but also lower-level than the source (WHILE, in this instance) language. Many Java analyses are actually conducted on byte code, for example. Typically, high-level languages come with features that are numerous and complex, but can be reduced into a smaller set of simpler primitives. Working at the lower level of abstraction thus also supports simplicity in the compiler.

Control flow constructs such as `if` and `while` are similarly translated into simpler jump and conditional branch constructs that jump to a particular (numbered) instruction. For example, a statement of the form `if` P `then` S_1 `else` S_2 would be translated into:

$$\begin{array}{rl} 1: & \text{if } P \text{ then goto } 4 \\ 2: & S_2 \\ 3: & \text{goto } 5 \\ 4: & S_1 \end{array}$$

Exercise 1. How would you translate a WHILE statement of the form `while` P `do` S?

This form of code is often called 3-address code, because every instruction has at most two source operands and one result operand. We now define the syntax for 3-address code produced from the WHILE language, which we will call WHILE3ADDR. This language consists of a set of simple instructions that load a constant into a variable, copy from one variable to another, compute the value of a variable from two others, or jump (possibly conditionally) to a new address n. A program P is just a map from addresses to instructions:[2]

$$\begin{array}{rcl@{\qquad}rcl} I & ::= & x := n & op & ::= & + \mid - \mid * \mid / \\ & \mid & x := y & op_r & ::= & < \mid = \\ & \mid & x := y \; op \; z & P & \in & \mathbb{N} \to I \\ & \mid & \text{goto } n & & & \\ & \mid & \text{if } x \; op_r \; 0 \text{ goto } n & & & \end{array}$$

[2]The idea of the mapping between numbers and instructions maps conceptually to Nielsens' use of *labels* in the WHILE language specification in the textbook. This concept is akin to mapping line numbers to code.

Formally defining a translation from a source language such as WHILE to a lower-level intermediate language such as WHILE3ADDR is possible, but more appropriate for the scope of a compilers course. For our purposes, the above should suffice as intuition. We will formally define the semantics of WHILE3ADDR in subsequent lectures.

3 Extensions

The languages described above are sufficient to introduce the fundamental concepts of program analysis in this course. However, we will eventually examine various extensions to WHILE and WHILE3ADDR, so that we can understand how more complicated constructs in real languages can be analyzed. Some of these extensions to WHILE3ADDR will include:

$$\begin{array}{lcll}
I & ::= & \ldots & \\
 & \mid & x := f(y) & \textit{function call} \\
 & \mid & \text{return } x & \textit{return} \\
 & \mid & x := y.m(z) & \textit{method call} \\
 & \mid & x := \&p & \textit{address-of operator} \\
 & \mid & x := *p & \textit{pointer dereference} \\
 & \mid & *p := x & \textit{pointer assignment} \\
 & \mid & x := y.f & \textit{field read} \\
 & \mid & x.f := y & \textit{field assignment}
\end{array}$$

We will not give semantics to these extensions now, but it is useful to be aware of them as you will see intermediate code like this in practical analysis frameworks.

4 Control flow graphs

Program analysis tools typically work on a representation of code as a *control-flow graph* (CFG), which is a graph-based representation of the flow of control through the program. It connects simple instructions in a way that statically captures all possible execution paths through the program and defines the execution order of instructions in the program. When control could flow in more than one direction, depending on program values, the graph branches. An example is the representation of an `if` or `while` statement. At the end of the instructions in each branch of an `if` statement, the branches merge together to point to the single instruction that comes afterward. Historically, this arises from the use of program analysis to optimize programs.

More precisely, a control flow graph consists of a set of nodes and edges. The nodes $\mathcal{N}$ correspond to *basic blocks*: Sequences of program instructions with no jumps in or out (no gotos, no labeled targets). The edges $\mathcal{E}$ represent the flow of control between basic blocks. We use *Pred(n)* to denote the set of all predecessors of the node *n*, and *Succ(n)* the set of all successors. A CFG has a start node, and a set of final nodes, corresponding to return or other termination of a function. Finally, for the purposes of dataflow analysis, we say that a *program point* exists before and after each node. Note that there exists considerable flexibility in these definitions, and the precision of the representation can vary based on the desired precision of the resulting analysis as well as the peculiarities of the language. In this course we will in fact often ignore the concept of a basic block and just treat instructions as the nodes in a graph; this view is semantically equivalent and simpler, but less efficient in practice. Further defining and learning how to construct CFGs is a subject best left to a compilers course; this discussion should suffice for our purposes.