Lecture Notes: The WHILE and WHILE3ADDR Language

17-355/17-665: Program Analysis *(Spring 2017)*
Jonathan Aldrich
aldrich@cs.cmu.edu

1 The WHILE Language

In this course, we will study the theory of analyses using a simple programming language called WHILE, along with various extensions. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties (to be discussed in a later lecture). 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
e	arithmetic expressions
x, y	program variables
n	number literals
P	boolean predicates

The syntax of WHILE is shown below. Statements S can be an assignment $x := e$, a skip statement, which does nothing (similar to a lone semicolon or open/close bracket in C or Java), and `if` and `while` statements, whose condition is a boolean predicate P. Arithmetic expressions e include variables x, numbers n, and one of several arithmetic operators, abstractly represented by 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.

$$\begin{array}{rcl}
S & ::= & x := e \\
& | & \text{skip} \\
& | & S_1;\ S_2 \\
& | & \text{if } P \text{ then } S_1 \text{ else } S_2 \\
& | & \text{while } P \text{ do } S
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & x \\
& | & n \\
& | & e_1\ op_a\ e_2 \\
& & \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}$$

$$
\begin{array}{lll}
P & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } P \\
 & | & P_1 \; op_b \; P_2 \\
 & | & e_1 \; op_r \; e_2 \\
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \;\mid\; \leq \;\mid\; = \;\mid\; > \;\mid\; \geq
\end{array}
$$

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 how to analyze each form separately. Furthermore, if we want to precisely understand what programs mean—e.g. so that we can reason about the correctness of our analysis definitions—we will also have to define the semantics of each form separately. 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:

$$
\begin{array}{l}
t_1 = x * y \\
w = t_1 + z
\end{array}
$$

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 both compilers and analysis tools.

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

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

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:[1]

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

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 examples above should suffice as intuition. We will focus instead on semantics and on formalizing program analyses.

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:

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

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 Operational Semantics

To reason about the correctness of an analysis, we need a clear definition of what a program *means*. There are many ways of giving such definitions; the most common technique in industry is to define a language using an English document, such as the Java Language Specification. However, natural language specifications, while accessible to all programmers, are often imprecise. This imprecision can lead to many problems, such as incorrect or incompatible compiler implementations, but more importantly for our purposes, analyses that give incorrect results.

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

A better alternative, from the point of view of reasoning precisely about programs, is a formal definition of program semantics. In this class we will deal with *operational semantics*, so named because they show how programs operate. In particular, we will use a form of operational semantics known as an *abstract machine*, in which the semantics mimics, at a high level, the operation of the computer that is executing the program, including a program counter, values for program variables, and (eventually) a representation of the heap. Such a semantics also reflects the way that techniques such as dataflow analysis or Hoare Logic reason about the program, so it is convenient for our purposes.

We now define an abstract machine that evaluates programs in WHILE3ADDR. A configuration c of the abstract machine includes the stored program P (which we will generally treat implicitly), along with an environment E that defines a mapping from variables to values (which for now are just numbers) and the current program counter n representing the next instruction to be executed:

$$\begin{array}{rcl} E & \in & \mathit{Var} \to \mathbb{Z} \\ c & \in & E \times \mathbb{N} \end{array}$$

The abstract machine executes one step at a time, executing the instruction that the program counter points to, and updating the program counter and environment according to the semantics of that instruction. We will represent execution of the abstract machine with a mathematical judgment of the form $P \vdash E, n \leadsto E', n'$ The judgment reads as follows: "When executing the program P, executing instruction n in the environment E steps to a new environment E' and program counter n'."

We can now define how the abstract machine executes with a series of inference rules. As shown below, an inference rule is made up of a set of judgments above the line, known as premises, and a judgment below the line, known as the conclusion. The meaning of an inference rule is that the conclusion holds if all of the premises hold.

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

We now consider a simple rule defining the semantics of the abstract machine for WHILE3ADDR in the case of the constant assignment instruction:

$$\frac{P[n] = x := m}{P \vdash E, n \leadsto E[x \mapsto m], n+1} \ \textit{step-const}$$

This rule states that in the case where the nth instruction of the program P (which we look up using $P[n]$) is a constant assignment $x := m$, the abstract machine takes a step to a state in which the environment E is updated to map x to the constant m, written as $E[x \mapsto m]$, and the program counter now points to the instruction at the following address $n + 1$.

We similarly define the remaining rules:

$$\frac{P[n] = x := y}{P \vdash E, n \leadsto E[x \mapsto E[y]], n+1} \textit{ step-copy}$$

$$\frac{P[n] = x := y \; op \; z \quad E[y] \; \mathbf{op} \; E[z] = m}{P \vdash E, n \leadsto E[x \mapsto m], n+1} \textit{ step-arith}$$

$$\frac{P[n] = \text{goto } m}{P \vdash E, n \leadsto E, m} \textit{ step-goto}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = true}{P \vdash E, n \leadsto E, m} \textit{ step-iftrue}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = false}{P \vdash E, n \leadsto E, n+1} \textit{ step-iffalse}$$

Now that we have specified how each statement affects the WHILE3ADDR abstract machine, we can formalize program execution as a trace:

Definition (Program Trace). A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, ...\}$ of program configurations, where $c_0 = E_0, 1$ is called the initial configuration, and for every $i \geq 0$ we have $P \vdash c_i \leadsto c_{i+1}$.

Acknowledgements

I thank Claire Le Goues for greatly appreciated refinements to these notes.