Lecture Notes:
Semantics of WHILE

17-654/17-754: Analysis of Software Artifacts
Jonathan Aldrich (`jonathan.aldrich@cs.cmu.edu`)

Lecture 2

In order to analyze programs rigorously, we need a clear definition of what a program means. There are many ways of giving such definitions; the most common technique for industrial languages is 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 implementions, but more importantly for our purposes, analyses that give incorrect results.

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.

1 The WHILE Language

In this course, we will study the theory of analyses using a simple programming language called WHILE. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties (see Lecture 3). It is a simple imperative language, with assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

We will use the following metavariables to describe several different categories of syntax. The letter on the left will be used as a variable representing a piece of a program; the word in bold represents the set of all program pieces of that kind; and on the right, we describe the kind of program piece we are describing:

$$
\begin{array}{lcll}
S & \in & \mathbf{Stmt} & \text{statements} \\
a & \in & \mathbf{AExp} & \text{arithmetic expressions} \\
x, y & \in & \mathbf{Var} & \text{program variables} \\
n & \in & \mathbf{Num} & \text{number literals} \\
b & \in & \mathbf{BExp} & \text{boolean expressions}
\end{array}
$$

The syntax of WHILE is shown below. Statements S can be an assignment $x := a$, 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 expression b. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators, abstractly represented by op_a. Boolean expressions 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.

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & skip \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S \\[1ex]
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2 \\[1ex]
op_a & ::= & + \mid - \mid * \mid / \mid \ldots \\[1ex]
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2 \\[1ex]
op_b & ::= & \text{and} \mid \text{or} \mid * \mid / \mid \ldots \\[1ex]
op_r & ::= & < \ \mid\ \leq \ \mid\ = \ \mid\ > \ \mid\ \geq \ \mid \ldots
\end{array}
$$

2 Big-Step Expression Semantics

We will first describe the semantics of arithmetic and boolean expressions using big-step semantics. Big-step semantics use judgments to describe how an expression reduces to a value. In general, our judgments may depend on certain assumptions, such as the values of variables. We will write our assumptions about variable values down in an environment η, which maps each variable to a value. For example, the environment $\eta = [x \mapsto 3, y \mapsto 5]$ states that the value of x is 3 and the value of y is 5. Variables not in the mapping have undefined values.

We will use the judgment form $\eta \vdash a \downarrow v$, read, "In the environment η, expression a reduces to value v." Values in WHILE are integers n and booleans (true and false).

We define valid judgments about expression semantics using a set 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.

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

An inference rule with no premises is called an axiom. Axioms are always true. An example of an axiom is that integers always evaluate to themselves:

$$\frac{}{\eta \vdash n \downarrow \mathbf{n}}$$

In the rule above, we have written the $\mathbf{n}$ on the right hand side of the judgment in bold to denote that the program text n has reduced to a mathematical integer $\mathbf{n}$. This distinction is somewhat pedantic and sometimes we will omit the boldface, but it is useful to remember that program semantics are given in terms of mathematics, whereas mathematical numbers and operations also appear as program text.

On the other hand, if we wish to define the meaning of an arithemetic operator like $+$ in the source text, we need to rely on premises that show how the operands to $+$ reduce to values. Thus we use an induction rule with premises:

$$\frac{\eta \vdash a \downarrow v \quad \eta \vdash a' \downarrow v' \quad v'' = v \boldsymbol{+} v'}{\eta \vdash a + a' \downarrow v''}$$

This rule states that if we want to evaluate an expression $a + a'$, we need to first evaluate a to some value v, then evaluate a' to some value v'. Then, we can use the mathematical operator $+$ on the resulting values to find the final result v''. Note that we are using the mathematic operator **+** (in bold) to define the program operator $+$. Of course the definition of $+$ could in principle be different from **+**–for example, the operator $+$ in C, defined on the **`unsigned int`** type, could be defined as follows:

$$\frac{\eta \vdash_C a \downarrow v \quad \eta \vdash_C a' \downarrow v' \quad v'' = (v\textbf{+}v')\ \textbf{mod}\ 2^{32}}{\eta \vdash_C a + a' \downarrow v''}$$

This definition takes into consideration that **`int`** values are represented with 32 bits, so addition wraps around after it reaches $2^32 - 1$–while of course the mathematical + operator does not wrap around. Here we have used the C subscript on the turnstile $\vdash_C$ to remind ourselves that this is a definition for the C language, not the WHILE language.

Once we have defined a rule that has premises, we must think about how it can be used. The premises themselves have to be proven with other inference rules in order to ensure that the conclusion is correct. A complete proof of a judgment using multiple inference rules is called a derivation. A derivation can be represented as a tree with the conclusion at the root and axioms at the leaves. For example, an axiom is also a derivation, so it is easy to prove that 5 reduces to **5**:

$$\frac{}{\eta \vdash 5 \downarrow \mathbf{5}}$$

Here we have just applied the axiom for natural numbers, substituting the actual number 5 for the variable n in the axiom. To prove that $1 + 2$ evaluates to **3**, we must use the axiom for numbers twice to prove the two premises of the rule for $+$:

$$\frac{\dfrac{}{\eta \vdash 1 \downarrow \mathbf{1}} \quad \dfrac{}{\eta \vdash 2 \downarrow \mathbf{2}} \quad \mathbf{3} = \mathbf{1+2}}{\eta \vdash 1 + 2 \downarrow \mathbf{3}}$$

The third premise, $\mathbf{3} = \mathbf{1+2}$, can be established using ordinary mathematics; we do not need to explicitly cite an inference rule to justify this judgment.

We can write the addition rule above in a more general way to define the semantics of all of the operators in WHILE in terms of the equivalent mathematical operators. Here we have also simplified things slightly by evaluating the mathematical operator in the conclusion.

$$\frac{\eta \vdash a \downarrow v \quad \eta \vdash a' \downarrow v'}{\eta \vdash a\ op\ a' \downarrow v\ \mathbf{op}\ v'}$$

As stated above, the evaluation of a WHILE expression may depend on the value of variables in the environment η. We use η in the rule for evaluating variables. The notation $\eta(x)$ means looking up the value that x maps to in the environment η:

$$\frac{\eta(x) = v}{\eta \vdash x \downarrow v}$$

We complete our definition of WHILE expression semantics with axioms for true, false, and an evaluation rule for not. As before, items in regular font denote program text, whereas items in bold represent mathematical objects and operators:

$$\frac{}{\eta \vdash true \downarrow \mathbf{true}}$$

$$\frac{}{\eta \vdash false \downarrow \mathbf{false}}$$

$$\frac{\eta \vdash b \downarrow v}{\eta \vdash \text{not}\, b \downarrow \mathbf{not}\, v}$$

As a side note, instead of defining not in terms of the mathematical operator **not**, we could have defined the semantics more directly with a pair of inference rules:

$$\frac{\eta \vdash b \downarrow \mathbf{true}}{\eta \vdash \text{not}\, b \downarrow \mathbf{false}}$$

$$\frac{\eta \vdash b \downarrow \mathbf{false}}{\eta \vdash \text{not}\, b \downarrow \mathbf{true}}$$

3 Example Derivation

Consider the following expression, evaluated in the variable environment $\eta = [x \mapsto 5, y \mapsto 2]$: (false and true) or $(x < ((3 * y) + 1))$. I use parentheses to describe how the expression should parse; the precedence of the operators is standard, but as this is not a class on parsing I will generally leave out the parentheses and assume the right thing will be done. We can produce a derivation that reduces this to a value as follows:

$$\dfrac{\dfrac{\dfrac{}{\eta \vdash \text{false} \downarrow \mathbf{false}} \quad \dfrac{}{\eta \vdash \text{true} \downarrow \mathbf{true}}}{\eta \vdash \text{false and true} \downarrow \mathbf{false}} \quad \dfrac{\dfrac{\eta(x) = \mathbf{5}}{\eta \vdash x \downarrow \mathbf{5}} \quad \dfrac{\dfrac{\dfrac{}{\eta \vdash 3 \downarrow \mathbf{3}} \quad \dfrac{\eta(y) = \mathbf{2}}{\eta \vdash y \downarrow \mathbf{2}}}{\eta \vdash 3 * y \downarrow \mathbf{6}} \quad \dfrac{}{\eta \vdash 1 \downarrow \mathbf{1}}}{\eta \vdash 3 * y + 1 \downarrow \mathbf{7}}}{\eta \vdash x < 3 * y + 1 \downarrow \mathbf{true}}}{\eta \vdash (\text{false and true}) \text{ or } (x < 3 * y + 1) \downarrow \mathbf{true}}$$

4 Small-Step Statement Semantics

To express the semantics of statements, we move to a small-step semantics. In small-step semantics, instead of computing the final result of a computation in one judgment, we take one incremental execution step, typically executing one statement. Thus, small-step semantics are ideal for watching the program execute, step by step. This makes the operation of loops more clear, for example. It also will be useful when we later verify properties of analyses, because analyses are often defined statement-by-statement.

We will use the judgment form $(\eta, S) \mapsto (\eta', S')$, read, "In the environment η, statement S takes a single step, resulting in a new environment η' and a new statement S'." For example, consider the rule for evaluating an assignment statement:

$$\frac{\eta \vdash a \downarrow v}{(\eta,\ x{:=}a) \ \ \mapsto \ \ (\eta[x{\mapsto}v],\ \text{skip})}$$

This rule uses as its premise a big-step judgment evaluating the right-hand-side expression a to a value v. It then produces a new environment which is the same as the old environment η except that the mapping for x is updated to refer to v. The notation $\eta[x{\mapsto}v]$ means exactly this. The assignment statement has executed, so we produce the statement skip indicating that this statement cannot execute any further. In fact, in our small-step semantics, all programs that terminate eventually reduce to a single skip statement.

Of course, realistic programs are made up of more than one statement. For a sequence of two statements, we simply reduce the first and leave the second alone. Once the first statement reduces all the way down to skip, we remove the skip statement and proceed to evaluate the second statement:

$$\frac{(\eta,\ S_1) \ \ \mapsto \ \ (\eta',\ S_1')}{(\eta,\ S_1; S_2) \ \ \mapsto \ \ (\eta',\ S_1'; S_2)}$$

$$\frac{}{(\eta,\ \text{skip}; S_2) \ \ \mapsto \ \ (\eta,\ S_2)}$$

For if statements, we evaluate the boolean condition using big-step semantics. If the result is `true`, we replace the if statement with the statement in the then clause. This has the effect of selecting the then branch of the if (and ignoring the else branch). Of course, we need another rule stating that if the result of the condition is **false**, we will replace the if statement with the statement in the else clause.

$$\frac{\eta \vdash b \downarrow \mathbf{true}}{(\eta,\ \text{if } b \text{ then } S_1 \text{ else } S_2) \quad \mapsto \quad (\eta,\ S_1)}$$

$$\frac{\eta \vdash b \downarrow \mathbf{false}}{(\eta,\ \text{if } b \text{ then } S_1 \text{ else } S_2) \quad \mapsto \quad (\eta,\ S_2)}$$

While loops work much like if statements. If the loop condition evaluates to true, we replace the while loop with the loop body. However, because the loop must evaluate again if the condition is still true after execution of the body, we copy the entire while loop after the loop body statement. Thus, the rewriting rules produce a copy of the loop body for each iteration of the loop until the loop guard evaluates to false, at which point the loop is replaced with skip.

$$\frac{\eta \vdash b \downarrow \mathbf{true}}{(\eta,\ \text{while } b \text{ do } S) \quad \mapsto \quad (\eta,\ S; \text{while } b \text{ do } S)}$$

$$\frac{\eta \vdash b \downarrow \mathbf{false}}{(\eta,\ \text{while } b \text{ do } S) \quad \mapsto \quad (\eta,\ skip)}$$

5 Example Executions

Consider the following program $r := 1;\ i := 0;$ while $i < m$ do $r := r * n; i := i + 1$ executing in the environment $\eta = [n{\mapsto}3, m{\mapsto}2]$. We can determine the first execution step with the following derivation:

$$\frac{\dfrac{[n{\mapsto}3, m{\mapsto}2] \vdash 1 \downarrow 1}{([n{\mapsto}3, m{\mapsto}2], r := 1) \mapsto ([n{\mapsto}3, m{\mapsto}2, r{\mapsto}1], \text{skip})}}{([n{\mapsto}3, m{\mapsto}2], r := 1; i := 0; \ldots) \mapsto ([n{\mapsto}3, m{\mapsto}2, r{\mapsto}1], \text{skip}; i := 0; \ldots)}$$

Note that the above derivation tree assumes that the structure of the program is parsed so that the top level is a statement sequence $S_1; S_2$ with S_1 being $r := 1$ and S_2 being the rest of the program. Another possible choice would have been S_1 as the statement sequence $r := 1; i := 0$; because we have not specified the associativity of the semicolon statement sequencing operator, either parsing choice is possible. Fortunately, although the derivations differ, the program ends up executing in exactly the same way; check this yourself if you're unsure. While we're on the topic of parsing, the intent of the program above is that both $r := r * n$ and $i := i + 1$ are within the while loop, not just the first one (if we were writing a parser we would close with "od" to remove the ambiguity).

In the next step, we skip the skip statement. The derivation is simple because we are using an axiom with no premises:

$$\frac{}{([n\mapsto 3, m\mapsto 2, r\mapsto 1], \mathsf{skip}; i := 0; \ldots) \mapsto ([n\mapsto 3, m\mapsto 2, r\mapsto 1], i := 0; \ldots)}$$

Next, we evaluate the next assignment:

$$\frac{\dfrac{[n\mapsto 3, m\mapsto 2] \vdash 0 \downarrow 0}{([n\mapsto 3, m\mapsto 2, r\mapsto 1], i := 0) \mapsto ([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], \mathsf{skip})}}{([n\mapsto 3, m\mapsto 2, r\mapsto 1], i := 0; \mathsf{while}\ \ldots) \mapsto ([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], \mathsf{skip}; \mathsf{while}\ \ldots)}$$

Next we skip the skip statement:

$$\frac{}{([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], \mathsf{skip}; \mathsf{while}\ \ldots) \mapsto ([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], \mathsf{while}\ \ldots)}$$

Next we evaluate the while loop. Since the condition is true (evaluated using big-step semantics) we copy the loop body out in front of the while loop:

$$\frac{\dfrac{\dfrac{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0](i) = 0}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash i \downarrow 0} \quad \dfrac{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0](i) = 2}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash m \downarrow 2}}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash i < m \downarrow \mathbf{true}}}{\begin{array}{c}([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], \mathsf{while}\ i < m\ \mathsf{do}\ r := r * n; i := i + 1) \mapsto \\ ([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], r := r * n; i := i + 1; \mathsf{while}\ i < m\ \mathsf{do}\ r := r * n; i := i + 1)\end{array}}$$

Next we assign to r:

$$\frac{\dfrac{\dfrac{\dfrac{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0](r) = 1}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash r \downarrow 1} \quad \dfrac{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0](n) = 3}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash n \downarrow 3}}{[n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0] \vdash r * n \downarrow 3}}{([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], r := r * n) \mapsto ([n\mapsto 3, m\mapsto 2, r\mapsto 3, i\mapsto 0], \mathsf{skip})}}{\begin{array}{c}([n\mapsto 3, m\mapsto 2, r\mapsto 1, i\mapsto 0], r := r * n; i := i + 1; \mathsf{while}\ i < m\ \mathsf{do}\ r := r * n; i := i + 1) \mapsto \\ ([n\mapsto 3, m\mapsto 2, r\mapsto 3, i\mapsto 0], \mathsf{skip}; i := i + 1; \mathsf{while}\ i < m\ \mathsf{do}\ r := r * n; i := i + 1)\end{array}}$$

Assuming the examples above are sufficient to illustrate derivations, we can proceed just by showing the remaining execution steps:

$$
\begin{array}{ll}
 & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}3, i{\mapsto}0], \text{skip}; i := i+1; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}3, i{\mapsto}0], i := i+1; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}3, i{\mapsto}1], \text{skip}; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}3, i{\mapsto}1], \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}3, i{\mapsto}1], r := r*n; i := i+1; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}9, i{\mapsto}1], \text{skip}; i := i+1; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}9, i{\mapsto}1], i := i+1; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}9, i{\mapsto}2], \text{skip}; \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}9, i{\mapsto}2], \text{while } i < m \text{ do } r := r*n; i := i+1 \\
\mapsto & [n{\mapsto}3, m{\mapsto}2, r{\mapsto}9, i{\mapsto}2], \text{skip}
\end{array}
$$

We can see that this program computes the mth power of n, leaving the result in r. For the input of $n = 3, m = 2$ we get $r = 9$ as expected.

6 Proofs Using WHILE Semantics

We would like to be able to prove that the program above in fact computes the mth power of n. We assume that $m > 0$ and $n \geq 0$. Assuming that $\mapsto^*$ is the reflexive, transitive closure of $\mapsto$, we would like to show that:

$$
\begin{array}{l}
([n{\mapsto}N, m{\mapsto}M], r := 1;\ i := 0;\ \text{while } i < m \text{ do } r := r*n; i := i+1) \\
\mapsto ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^M, i{\mapsto}M], \text{skip})
\end{array}
$$

A full proof would show the complete derivation for every step applied in the static semantics. In this class, we will be satisfied to prove programs correct by showing only each small-step execution. Thus, we will first observe that the semantics means the program will take the following initial steps:

$$
\begin{array}{ll}
 & ([n{\mapsto}N, m{\mapsto}M], r := 1;\ i := 0;\ \text{while } i < m \text{ do } r := r*n; i := i+1) \\
\mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}1], \text{skip};\ i := 0;\ \text{while } i < m \text{ do } r := r*n; i := i+1) \\
\mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}1], i := 0;\ \text{while } i < m \text{ do } r := r*n; i := i+1) \\
\mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}1, i{\mapsto}0], \text{skip};\ \text{while } i < m \text{ do } r := r*n; i := i+1) \\
\mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}1, i{\mapsto}0], \text{while } i < m \text{ do } r := r*n; i := i+1)
\end{array}
$$

Now, we know that $M \geq 0$ by our assumptions from the theorem to be proved, but we don't know what step comes next. If $M = 0$ then the while loop will reduce to a skip, but if $M > 0$ we will execute the while loop one or more times. In order to verify the desired property, we need to prove the following lemma. We assume $N > 0$, $M \geq 0$, and $0 \leq i \leq M$.

$$\begin{array}{l}([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I, i{\mapsto}I], \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^M, i{\mapsto}M], \text{skip})\end{array}$$

Notice that this lemma generalizes the fact we need to prove, because it applies to any value of I, and allows r to map to N^I, not just to 0. Of course, in the case where $I = 0$, $N^I = 1$ and so the lemma will allow us to directly conclude the needed result for the theorem above. This generalization, it turns out, corresponds directly to the loop invariant we will use to prove this same program correct in the next lecture using Hoare Logic.

We prove the lemma by induction on $M - I$. Note that this is backwards from induction on I, in that we are starting with $I = M$ and counting down towards zero. This is because the final result ($r = N^M$) is fixed and we can vary the precondition by changing the initial value of I. Thus we are working backwards–first proving that the last evaluation of the loop is correct, assuming the loop invariant for previous iterations–then working backwards one loop iteration of the time using induction.

We prove the base case first, where $M - I = 0$. But this case is easy, because then $I = M$ and so we reduce directly to skip:

$$\begin{array}{l}([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^M, i{\mapsto}M], \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^M, i{\mapsto}M], \text{skip})\end{array}$$

We next prove the inductive case. We will assume the lemma holds for $M - I - 1$, that is, substituting $I + 1$ into the lemma:

$$\begin{array}{l}([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^{I+1}, i{\mapsto}I + 1], \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^M, i{\mapsto}M], \text{skip})\end{array}$$

Now, we know that $I < M$, because otherwise we would be in the base case. Therefore we can conclude that:

$$\begin{array}{ll} & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I, i{\mapsto}I], \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I, i{\mapsto}I], r := r * n; i := i + 1;\ \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I * N, i{\mapsto}I], \text{skip}; i := i + 1;\ \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I * N, i{\mapsto}I], i := i + 1;\ \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I * N, i{\mapsto}I + 1], \text{skip};\ \text{while } i < m \text{ do } r := r * n; i := i + 1) \\ \mapsto & ([n{\mapsto}N, m{\mapsto}M, r{\mapsto}N^I * N, i{\mapsto}I + 1], \text{while } i < m \text{ do } r := r * n; i := i + 1)\end{array}$$

Now, $N^I * N = N^{I+1}$. Therefore, we can directly apply the inductive hypothesis given above to prove the inductive case of the lemma.

Since the theorem follows directly from the first few reductions and the lemma, we have proved the theorem as well.