COMP3161/COMP9164 Supplementary Lecture Notes

Imperative Programming

Liam O'Connor

September 22, 2020

The term *imperative* ref [illegible] *commands* that manipulate both the exter[illegible] executed is significant in an imp[illegible]

Typically, states can be defined as a mapping from variable names to their values, however some lower-level languages may require more complex models. For example, to specify an assembly language, a detailed model containing a processor's registers and accessible memory may be required.

Early imperative languages allowed not just global state but also global control flow – a `go to` statement could be used to transfer control to anywhere else in the entire program. Edsger Dijkstra was one of the first computer scientists to advocate the notion of *structured* programming, as it allowed imperati[illegible]ng. This movement was responsible for the int[illegible]ns to imperative langua[illegible]

We will define an im[illegible]ic and dynamic semantics, and specify a Hoare logic to demonstrate the benefits of structured programming for program verification.

1 Syntax

We're going to specify a language **TinyImp**. This language consists of state-changing *statements* and pure, evaluable arithmetic *expressions*, as we have defined before.

Stmt	::=	`skip`	*Do nothing*
	|	x `:=` **Expr**	*Assignment*
	|	`var` $y \cdot$ **Stmt**	*Declaration*
	|	`if` **Expr** `then` **Stmt** `else` **Stmt** `fi`	*Conditional*
	|	`while` **Expr** `do` **Stmt** `od`	*Loop*
	|	**Stmt** `;` **Stmt**	*Sequencing*
Expr	::=	$\langle$*Arithmetic expressions*$\rangle$	

The statement `skip` does nothing, $x := e$ updates the (previously-declared) variable x to have a new value resulting from the evaluation of e, `var` $y \cdot s$ declares a *local* variable that is only in scope within the statement s, `if` statements and `while` loops behave much like what you expect, and $s_1; s_2$ is a compound statement consisting of s_1 followed sequentially by s_2.

For the purposes of this language, we will assume that all variables are of integer types. This means for boolean conditions, we will adopt the C convention that zero means false, and non-zero means true.

Here are some example programs written in **TinyImp**. Firstly, a program that computes the factorial of a fixed constant N:

var i ·
var m ·
$i := 0$;
$m := 1$;
while $i < N$ **do**
  $i := i + 1$;
  $m := m \times i$
od

And the program that computes the Nth fibonacci number:

var m · **var** n · **var** i ·

$m := n$;
$n := m + t$;
$i := i + 1$
od

2 Static Semantics

Seeing as all variables ar ... **TinyImp**, however there is more ... before use. It is also desira ... , that is, assigned to a value, before being re ... mine the result from the semantics.

For this reason, we will define a static semantics judgement ... W consisting of two sets of variables in the context: U, which consists of a ... V, which consists of all declared variables that have been written to pr ... to read. The judgement **ok** denotes that the statement s does not read from any uninitialised variables, or refer to any variable not in scope. The set W denotes all the variables that are guaranteed to be written to when s executes.

Firstly, the statement `skip` does not affect any variables, so is valid under any context:

$$\frac{}{U; V \vdash \texttt{skip}\ \textbf{ok} \rightsquigarrow \emptyset}$$

If we assign to a variable x with the expression e, we want to make sure that e only mentions variables that have been initialised. The variable x must be declared, but may be uninitialised. After executing the assignment, we know that x has now been written to.

$$\frac{x \in U \qquad \mathrm{FV}(e) \subseteq V}{U; V \vdash x\ \texttt{:=}\ e\ \textbf{ok} \rightsquigarrow \{x\}}$$

When we declare a variable y, we introduce it to the set of variables. Once the variable y is no longer in scope, we remove it from the set W as it is no longer relevant information for us to check any further statements.

$$\frac{U \cup \{y\}; V \vdash s\ \textbf{ok} \rightsquigarrow W}{U; V \vdash \texttt{var}\ y \cdot s\ \textbf{ok} \rightsquigarrow W \setminus \{y\}}$$

Conditional statements must once again ensure that the condition only mentions initialised variables. The two branches of the condition must both be valid. After the conditional has executed, the only variables that we can guarantee will be written to are those written to regardless of the branch taken. Therefore, our final write set is the intersection of the write sets of the two branches.

$$\frac{\mathrm{FV}(e) \subseteq V \qquad U; V \vdash s_1 \ \mathbf{ok} \rightsquigarrow W_1 \qquad U; V \vdash s_2 \ \mathbf{ok} \rightsquigarrow W_2}{U; V \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \mathbf{ok} \rightsquigarrow W_1 \cap W_2}$$

While loops, much like `if`, must ensure that the guard only mentions initialised variables. Furthermore, we don't know if the body of the loop will run at all, for example, if the guard is initially false. In that case, this statement will never write to any variables. Therefore, we cannot guarantee that any variables will be initialised by running a while loop.

$$\frac{\mathrm{FV}(e) \subseteq V \qquad U; V \vdash s \ \mathbf{ok} \rightsquigarrow W}{U; V \quad \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \mathbf{ok}}$$

Lastly, for sequential compos

the first statement are initialis
variables. The overall set of vari
two write sets.

$$\frac{U; V \vdash s_1 \ \mathbf{ok} \rightsquigarrow W_1 \qquad U; (V \cup W_1) \vdash s_2 \ \mathbf{ok} \rightsquigarrow W_2}{U; V \vdash s_1; s_2 \ \mathbf{ok} \rightsquigarrow W_1 \cup W_2}$$

These rules correspond to a static *over approximation* of the dynamic property that all variables must be initialised before use. This is *static* because the analysis can be completed without running the program.

3 Dynamic Se

We will use *big-step* operational semantics, describing evaluation from a *pair* containing a statement to execute and a program *state* σ. A *state* is a s to their values. We will use the following notation to describe state operatio

- To *read* a variable x from the state σ, we write
- To *update* an existing variable x to have value v inside the state σ, we write $(\sigma : x \mapsto v)$.
- To *extend* a state σ with a new, previously undeclared variable x, we write $\sigma \cdot x$. In such a state, x has undefined value.

We will assume we have defined a relation $\sigma \vdash e \Downarrow v$ for arithmetic expressions, which evaluates arithmetic expression e with the values for variables provided by σ. The evaluation rules resemble those we have done previously.

The rule for `skip` simply leaves the state unchanged.

$$\frac{}{(\sigma, \texttt{skip}) \Downarrow \sigma}$$

The rule for sequential composition $s_1; s_2$ threads the the state through the execution of the two statements in order. This forces us to evaluate s_1 before s_2, as the input state of s_2 is the output state of s_1:

$$\frac{(\sigma_1, s_1) \Downarrow \sigma_2 \qquad (\sigma_2, s_2) \Downarrow \sigma_3}{(\sigma_1, s_1; s_2) \Downarrow \sigma_3}$$

The rule for assignment updates the state to reflect the new value for the variable, after evaluating the expression:

$$\frac{\sigma \vdash e \Downarrow v}{(\sigma, x := e) \Downarrow (\sigma : x \mapsto v)}$$

The rule for variable declarations introduces a new variable into the state when evaluating the statement for which it is in scope, then removes it again before returning the result state:

$$\frac{(\sigma_1 \cdot x, s) \Downarrow (\sigma_2 \cdot x)}{(\sigma_1, \texttt{var } x \cdot s) \Downarrow \sigma_2}$$

Conditionals are defined with two rules. If the condition expression evaluates to a non-zero value, then the `then` case is executed:

$$\frac{\sigma_1 \vdash e \Downarrow v \qquad v \neq 0 \qquad (\sigma_1, s_1) \Downarrow \sigma_2}{(\sigma_1, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}) \Downarrow \sigma_2}$$

Otherwise, the `else` case is executed:

For `while` loops, the situation is similar. If the guard is false, we do not execute anything:

$$\frac{\sigma_1 \vdash e \Downarrow 0}{(\sigma_1, \texttt{while } e \texttt{ do } s \texttt{ od}) \Downarrow \sigma_1}$$

Otherwise, we execute the loop body and then execute the whole loop again with the resulting state:

$$\sigma_1 \quad e \quad v \quad v \neq 0$$

4 Hoare Logic

Because our language has been defined with compositional con ctured programming, we can write a compositional *proof cal* roperties of our programs. This is a common type of *axiomatic semantics*, an alternative to the operational semantics we have defined earlier.

Hoare Logic involves proving judgements of the following format:

$$\{\varphi\}\ s\ \{\psi\}$$

Here φ and ψ are logical *assertions*, propositions that may mention variables from the state. The above judgement, called a *Hoare Triple*, states that if the program s starts in any state σ that satisfies the *precondition* φ and $(\sigma, s) \Downarrow \sigma'$, then σ' will satisfy the *postcondition* ψ.

To prove a hoare triple like:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\textbf{var } i \cdot \textbf{var } m \cdot \\
i := 0; \\
m := 1; \\
\textbf{while } i < N \textbf{ do} \\
\quad i := i + 1; \\
\quad m := m \times i \\
\textbf{od} \\
\{m = N!\}
\end{array}
$$

It is undesirable to look at every possible derivation of the operational semantics given a starting state that satisfies the precondition, in order to prove that the postcondition is satisfied. Instead we shall define a set of rules to prove Hoare triples directly, without appealing to the operational semantics.

The rule for `skip` simply states that any condition about the state that was true before `skip` was executed is still true afterwards, as this statement does not change the state:

$$\frac{}{\{\varphi\}\ \texttt{skip}\ \{\varphi\}}$$

The rule for sequential composition states that in order for $s_1; s_2$ to move from a precondition of φ to a postcondition of ψ, then s_1 must, if starting from a state satisfying φ, evaluate to a state satisfying some intermediate assertion α, and s_2, starting from α, must evaluate to a state satisfying ψ.

$$\frac{\{\varphi\}\ s_1\ \{\alpha\} \qquad \{\alpha\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1; s_2\ \{\psi\}}$$

For a conditional to satisfy the postcondition ψ under precondition φ, both branches must satisfy ψ under the precondition that

respectively:

$$\frac{}{\{\varphi\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}}$$

Seeing as while loops may execute zero times, the precondition φ must remain true after the while loop has finished. In addition, after the loop has finished, we know that the guard must be false. Furthermore, because the loop body may execute any number of times, the loop body must maintain the assertion φ to be true after each iteration. This is called a *loop invariant*

$$\frac{\varphi \quad e \quad s \quad \varphi}{}$$

For an assignment s ition must effectively state that φ hold leted, x will indeed be replaced with e and therefore φ will hold. Tr re not convinced:

$$\frac{}{\{\varphi[x := e]\}\ x :}$$

There is one more rule, called the *rule of consequence*, that we need to insert ordinary logical reasoning into our Hoare logic proofs, allowing us to change the pre- and post-conditions we have to prove by way of logical implications:

$$\frac{\varphi \Rightarrow \alpha \qquad \{\alpha\}\ s\ \{\beta\} \qquad \beta \Rightarrow \psi}{\{\varphi\}\ s\ \{\psi\}}$$

This is the only rule that is not directed entirely by syntax. This means a Hoare logic proof need not look like a derivation tree. Instead we can sprinkle assertions through our program, and specially note uses of the consequence rule.

Note: An example verification of factorial using Hoare logic is provided in the Week 5 Friday slides.