COMP3161/COMP9164 Supplementary Lecture Notes

Imperative Programming

Liam O'Connor Johannes Åman Pohjola

October 2, 2022

The term *imperative* refers to programming languages that are defined as a series of *commands* that manipulate both the external world and internal state. The order in which commands are executed is significant in an imperative language, as the state changes over time.

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 imperative control flow to be made amenable to *local* reasoning. This movement was responsible for the introduction of things like loops, conditionals, and subroutines or functions to imperative languages.

We will define an imperative language based on structured programming, describe its static 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	::=	⟨*Arithmetic expressions*⟩	

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**. First, 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 for some constant N:[1]

```
var $m$ · var $n$ · var $i$ ·
$m := 1; n := 1$;
$i := 1$;
while $i < N$ do
   var $t \cdot t := m$;
   $m := n$;
   $n := m + t$;
   $i := i + 1$
od
```

2 Static Semantics

Seeing as all variables are of integer type, type checking is not really an issue for **TinyImp**. We want our programs to be well-scoped—that is, all variables should be declared before use. But there is more to check. In particular, are all variables *initialised*? That is, are they assigned to a value before being read from? Otherwise, their values may be undefined, and we cannot determine the result from the semantics.

For this reason, we will define a static semantics judgement $U; V \vdash s\ \mathbf{ok} \rightsquigarrow W$ consisting of two sets of variables in the context: U, which consists of all declared variables in scope; and V, which consists of all initialised variables in scope. 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:

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

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.

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

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.

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

[1]The notational convention is that lower-case variables are *program variables*, i.e., they are part of the syntax of the **TinyImp** program. Upper-case variables are *meta-variables*. We can think of expressions involving meta-variables as describing as a family of concrete programs: one where the programmer wrote 0 in place of N, another where the programmer wrote 1, and so on.

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—the guard might be false initially. 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.

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

Lastly, for sequential composition, we first check the first statement. Any variables written to by the first statement are considered initialised in the second statement, so we add them to the set of initialised variables. The overall set of variables written to by the sequential composition is the union of the 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. We call it static because the analysis can be completed without running the program. It's an over-approximation because there exists programs where all variables are initialised, but that are not **ok**.

3 Dynamic Semantics

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 mutable mapping from variables to their values. It allows us to keep track of which variables are declared, whether a declared variable is assigned to a value, and if so, which value.[2] We will use the following notation to describe state operations

- To *read* a variable x from the state σ, we write $\sigma(x)$.
- To *update* a state σ to give a previously declared variable x the v we write $(\sigma : x \mapsto v)$.
- To *extend* a state σ with a new, previously undeclared variable x, we write $\sigma \cdot x$. This does not assign a value to x, so $(\sigma \cdot x)(x)$ is undefined.
- To *remove* a variable x from the set of declared variables, we write $(\sigma|_x)$.
- The final operation caters to highly specific needs. $(\sigma|_x^{\sigma'})$ updates the state σ by first removing x from the set of declared variables, then replacing it with whatever x is in the state σ'. More formally:

  $$\sigma|_x^{\sigma'} = \begin{cases} \sigma|_x & \text{if } x \text{ is undeclared in } \sigma' \\ (\sigma|_x) \cdot x & \text{if } x \text{ is declared in } \sigma'\text{, but } \sigma'(x) \text{ is undefined.} \\ (\sigma : x \mapsto \sigma'(x)) & \text{if } \sigma'(x) \text{ is defined} \end{cases}$$

  Intuitively, this represents exiting the scope of x. When we do so, any previous declaration of x that may have been locally shadowed comes back into scope. Think of σ as the state when we exit the scope of x, and σ' as the state right before we entered the scope of x.

[2]This can be formalised with partial functions.

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.

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

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

Sequencing 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:

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

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

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

If x is undeclared in σ, the assignment rule is inapplicable, because $\sigma : x \mapsto v$ is not defined. If e contains any variables that are uninitialised in σ, the assignment rule is also inapplicable—there will be no v such that $\sigma e \vdash v$.

Declaration 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:

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

Conditionals Conditionals are defined with two rules. If the condition 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:

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

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

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

Here, we're trying to use x immediately after declaring it, without initialising it. Otherwise, we execute the loop body and then execute the whole loop again in the resulting state:

$$\frac{\begin{array}{c}\sigma_1 \vdash e \Downarrow v \quad v \neq 0 \\ (\sigma_1, s) \Downarrow \sigma_2 \quad (\sigma_2, \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) \Downarrow \sigma_3\end{array}}{(\sigma_1, \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) \Downarrow \sigma_3}$$

4 Variable declaration semantics

There are (at least) three approaches to dealing with uninitialised variables:

1. Crash and burn
2. Default values
3. Read junk data

In the following, we discuss how each of these can be captured in the big-step semantics. These should be indistinguishable for programs that are **ok**.

4.1 Crash-and-burn semantics

In the semantics given in Section 3, a statement that attempts to use an uninitialised value does not evaluate to anything. Consider, for example, the following program fragment, where we assume y is a variable that has already been declared:

$$\mathbf{var}\ x \cdot y := x + 1$$

If we start from a state σ where y is defined, we cannot derive any judgement that tells us what the final state should be, because we'll be unable to close the following derivation

$$\dfrac{\dfrac{???}{(\sigma \cdot x, y := x + 1) \Downarrow ??}}{(\sigma, \mathbf{var}\ x \cdot y := x + 1) \Downarrow ??}$$

since the assignment rule is inapplicable. In implementation terms, this means the program will not terminate successfully. This is analogous to how uninitialised variables in Python behave:[3]

```
>>> x+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

4.2 Default value semantics

In many programming languages, variables that are declared are automatically initialised to a default value. For example, the following Java program will always print `1` because Java initialises all fields to a default value—in the case of `int`, that's `0`.

```
public class test {
    static int x;

    public static void main(String[] args) {
        System.out.println(x+1);
    }
}
```

To adapt our operational semantics above to default-value semantics, we could replace the declaration rule with the following:

$$\dfrac{((\sigma_1 \cdot x) : x \mapsto 0, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var}\ x \cdot s) \Downarrow (\sigma_2|_x^{\sigma_1})}$$

[3]Not a precise analogy, since Python doesn't do explicit variable declarations.

Here, initialisation is modelled by immediately updating all variables to 0 as soon as they are declared. Now, our crash-and-burn example can be evaluated as follows:

$$\dfrac{\dfrac{(\sigma \cdot x) : x \mapsto 0 \ \vdash \ x + 1 \Downarrow 1}{((\sigma \cdot x) : x \mapsto 0, y := x + 1) \ \Downarrow \ (\sigma \cdot x) : x \mapsto 0 : y \mapsto 1}}{(\sigma, \textbf{var}\ x \cdot y := x + 1) \ \Downarrow \ \sigma : y \mapsto 1}$$

Note that the last step in the derivation relies on the fact that the expression

$$((\sigma \cdot x) : x \mapsto 0 : y \mapsto 1)|_{x}^{\sigma}$$

simplifies to $\sigma : y \mapsto 1$. You may want to convince yourself that this makes sense.

4.3 Junk data semantics

Most implementations of C will "initialise" variables by letting them inherit whatever junk data happened to live in the variable's memory location beforehand.[4] This can lead to unpredictable runtime behaviour. One way of capturing this unpredictability is with the following rule:

$$\frac{((\sigma_1 \cdot x) : x \mapsto n, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var}\ x \cdot s) \Downarrow (\sigma_2|_{x}^{\sigma_1})}$$

This is almost identical to the default-value rule, except we've replaced the 0 with a free variable n. By choosing to instantiate n to 0, we can derive the judgement

$$(\sigma, \textbf{var}\ x \cdot y := x + 1) \ \Downarrow \ \sigma : y \mapsto 1$$

using the exact same proof tree as for the default-value example. But we can also prove the following:

$$\dfrac{\dfrac{(\sigma \cdot x) : x \mapsto 5 \ \vdash \ x + 1 \Downarrow 6}{((\sigma \cdot x) : x \mapsto 5, y := x + 1) \ \Downarrow \ (\sigma \cdot x) : x \mapsto 5 : y \mapsto 6}}{(\sigma, \textbf{var}\ x \cdot y := x + 1) \ \Downarrow \ \sigma : y \mapsto 6}$$

The resulting semantics is *non-deterministic*: the final state (if one exists) is not unique. In real-world terms, this means the program may be observed to behave differently depending on lowerer-level details, e.g., in the compiler implementation, the hardware, or the runtime environment.

5 Hoare Logic

Because our language has been defined with compositional control structures inspired by structured programming, we can write a compositional *proof calculus* for proving properties about our programs. This is a common type of *axiomatic semantics*, an alternative to the operational semantics we have defined earlier.

While scopes were important in previous sections, Hoare logic is usually presented in a setting where all variables are global. To avoid bogging down the presentation, we'd like to do the same. Therefore, from now on, we will start pretending that the **var** construct does not exist, and that all variables are declared.

Hoare Logic involves proving judgements of the following format:

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

[4]The situation in C is actually a bit more nuanced than this. Initialisation behaviour depends on the storage class, and uninitialised variables may contain bit patterns that do not represent values of the intended type. This may lead to crashing and burning.

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.

Here's an example of a Hoare triple:

```
{True}
$i := 0$;
$m := 1$;
while $i < N$ do
  $i := i + 1$;
  $m := m \times i$
od
{$m = N!$}
```

To prove a Hoare triple, it is undesirable to use the operational semantics directly. We could, but it gets messy, and requires setting up an induction every time you encounter a **while** loop. 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 φ holds and that the condition either holds or does not hold, respectively:

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

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*.

$$\frac{\{\varphi \land e\}\ s\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \land \neg e\}}$$

For an assignment statement $x := e$ to satisfy a postcondition φ, the precondition must effectively state that φ holds if x is replaced with e. Therefore, once the assignment has completed, x will indeed be replaced with e and therefore φ will hold. Try this on a few simple examples if you are not convinced:

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

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 slides.

Appendix A: Semantics of arithmetic expressions

In the above, we left the treatment of arithmetic expressions to the reader's imagination. This was to above bogging down the presentation, and because there's no need at this point to fix the set of allowed arithmetic expressions. However, you may have noticed that none of the previously introduced operational semantics for arithmetic expressions fit what we want precisely, because they were only defined on closed expressions.

Here's an example of how to set up the big-step operational semantics so that it takes states into account:

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2}{\sigma \vdash e_1 + e_2 \;\Downarrow\; v_1 + v_2} \qquad \frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2}{\sigma \vdash e_1 * e_2 \;\Downarrow\; v_1 * v_2} \qquad \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v} \qquad \frac{}{\sigma \vdash n \Downarrow n}$$

n and x in the above stand for numerical constants and variables, respectively. Note that this is very close to the denotational semantics.