Imperative Programming Languages

Johannes Åman Pohjola
UNSW
Term 3 2022
Imperative Programming

**Definition**

*Imperative programming* is where programs are described as a series of *statements* or commands to manipulate mutable *state* or cause externally observable *effects*.

*States* may take the form of a *mapping* from variable names to their values, or even a model of a CPU state with a memory model (for example, in an *assembly language*).
The Old Days

Early microcomputer languages used a line numbering system with GO TO statements used to arrange control flow.
Factorial Example in BASIC (1964)

10 N = 4
20 I = 0
30 M = 1
40 IF I ≥ N THEN GOTO 100
50 I = I + 1
60 M = M * I
70 GOTO 40
100 PRINT M
110 END
The *structured programming* movement brought in *control structures* to mainstream use, such as conditionals and loops.
Factorial Example in Pascal (1970)

```pascal
program factorial;
var n : integer;
    m : integer;
    i : integer;
begin
    n := 5;
    m := 1;
    i := 0;
    while (i < n) do
    begin
        i := i + 1;
        m := m * i;
    end;
    writeln(m);
end.
```
Syntax

We’re going to specify a language TinyImp, based on structured programming. The syntax consists of statements and expressions.

Grammar

\[
\text{Stmt} ::= \text{skip} \quad \text{Do nothing} \\
| \quad x ::= \text{Expr} \quad \text{Assignment} \\
| \quad \text{var} \ y \cdot \text{Stmt} \quad \text{Declaration} \\
| \quad \text{if} \ \text{Expr} \ \text{then} \ \text{Stmt} \ \text{else} \ \text{Stmt} \ \text{fi} \quad \text{Conditional} \\
| \quad \text{while} \ \text{Expr} \ \text{do} \ \text{Stmt} \ \text{od} \quad \text{Loop} \\
| \quad \text{Stmt} ; \ \text{Stmt} \quad \text{Sequencing} \\
\]

\[
\text{Expr} ::= \langle \text{Arithmetic expressions} \rangle
\]

We already know how to make unambiguous abstract syntax, so we will use concrete syntax in the rules for readability.
Examples

Example (Factorial and Fibonacci)

\begin{align*}
\textbf{var} & \ i \\
\textbf{var} & \ m \\
i & := 0; \\
m & := 1; \\
\textbf{while} & \ i < N \ \textbf{do} \\
& \ i := i + 1; \\
& \ m := m \times i \\
\textbf{od} \\
\textbf{var} & \ m \cdot \textbf{var} \ n \cdot \textbf{var} \ i \\
m & := 1; \ n := 1; \\
i & := 1; \\
\textbf{while} & \ i < N \ \textbf{do} \\
& \ \textbf{var} \ t \cdot \ t := m; \\
& \ m := n; \\
& \ n := m + t; \\
& \ i := i + 1 \\
\textbf{od}
\end{align*}
Static Semantics

Types? We only have one type (`int`), so type checking is a wash.

Scopes? We have to check that variables are declared before use.

Anything Else? We have to check that variables are initialized before they are used!

\[ U; V \vdash \text{ok} \leadsto W \]

- Set of declared free variables
- Set of definitely written to free variables
- Set of initialized free variables
- Indicates that no unsafe reads occur

Note: \( V \subseteq U \)
Static Semantics Rules

\[
\begin{array}{c}
\frac{x \in U \quad \text{FV}(e) \subseteq V}{U; V \vdash \text{skip ok} \rightsquigarrow \emptyset}
\quad \frac{U; V \vdash x := e \text{ ok} \rightsquigarrow \{x\}}{}
\end{array}
\]

\[
\frac{U \cup \{y\}; V \vdash s \text{ ok} \rightsquigarrow W}{U; V \vdash \text{var y \cdot s ok} \rightsquigarrow W \setminus \{y\}}
\]

\[
\frac{\text{FV}(e) \subseteq V \quad U; V \vdash s_1 \text{ ok} \rightsquigarrow W_1 \quad U; V \vdash s_2 \text{ ok} \rightsquigarrow W_2}{U; V \vdash \text{if e then s_1 else s_2 fi ok} \rightsquigarrow W_1 \cap W_2}
\]

\[
\frac{\text{FV}(e) \subseteq V \quad U; V \vdash s \text{ ok} \rightsquigarrow W}{U; V \vdash \text{while e do s od ok} \rightsquigarrow \emptyset}
\]

\[
\frac{U; V \vdash s_1 \text{ ok} \rightsquigarrow W_1 \quad U; (V \cup W_1) \vdash s_2 \text{ ok} \rightsquigarrow W_2}{U; V \vdash s_1; s_2 \text{ ok} \rightsquigarrow W_1 \cup W_2}
\]
Dynamic Semantics

We will use big-step operational semantics. What are the sets of evaluable expressions and values here?

**Evaluable Expressions**: A pair containing a statement to execute and a state $\sigma$.

**Values**: The final state that results from executing the statement.

**States**: mutable mappings from states to values.
States

A *state* is a mutable mapping from variables to their values. We use the following notation:

- To **read** a variable \( x \) from the state \( \sigma \), we write \( \sigma(x) \).
- To **update** an existing variable \( x \) to have value \( v \) inside the state \( \sigma \), we write \( (\sigma : x \mapsto v) \).
- To **extend** a state \( \sigma \) with a new, previously undeclared variable \( x \), we write \( \sigma \cdot x \). In such a state, \( (\sigma \cdot x)(x) \) is undefined.
- To **remove** a variable \( x \) from the set of declared variables, we write \( (\sigma|_x) \).
- To exit a local scope for \( x \), returning to the previous scope \( \sigma' \):

\[
\sigma|_{\sigma'}^x = \begin{cases} 
\sigma|_x & \text{if } x \text{ is undeclared in } \sigma' \\
(\sigma|_x) \cdot x & \text{if } x \text{ is declared but undefined in } \sigma' \\
(\sigma : x \mapsto \sigma'(x)) & \text{if } \sigma'(x) \text{ is defined}
\end{cases}
\]
Evaluation Rules

We will assume we have defined a relation \( \sigma \vdash e \Downarrow v \) for arithmetic expressions, much like in the previous lecture.

\[
\begin{align*}
\text{(\(\sigma, \text{skip}\)) \Downarrow \sigma} & \quad \text{(\(\sigma_1, s_1\)) \Downarrow \sigma_2} & \quad \text{(\(\sigma_2, s_2\)) \Downarrow \sigma_3} \\
\sigma \vdash e \Downarrow v & \quad \text{(\(\sigma_1, s_1; s_2\)) \Downarrow \sigma_3} \\
\text{(\(\sigma, x := e\)) \Downarrow (\(\sigma : x \mapsto v\))} & \quad (\(\sigma_1 \cdot x, s\)) \Downarrow \sigma_2 \\
\sigma_1 \vdash e \Downarrow v & \quad v \neq 0 & \quad (\sigma_1, s_1) \Downarrow \sigma_2 \\
\text{(\(\sigma_1, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}\)) \Downarrow \sigma_2} & \quad \sigma_1 \vdash e \Downarrow 0 & \quad (\sigma_1, s_2) \Downarrow \sigma_2 \\
\text{(\(\sigma_1, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}\)) \Downarrow \sigma_2} & \quad \sigma_1 \vdash e \Downarrow v & \quad v \neq 0 \\
\text{(\(\sigma_1, \text{while } e \text{ do } s \text{ od}\)) \Downarrow \sigma_1} & \quad (\sigma_1, s) \Downarrow \sigma_2 & \quad (\sigma_2, \text{while } e \text{ do } s \text{ od}) \Downarrow \sigma_3 \\
\text{(\(\sigma_1, \text{while } e \text{ do } s \text{ od}\)) \Downarrow \sigma_3} & \quad (\sigma_1, \text{while } e \text{ do } s \text{ od}) \Downarrow \sigma_3
\end{align*}
\]
Alternative declaration semantics

What should happen when an uninitialised variable is used?

\[(\sigma \cdot y, \text{var } x \cdot y := x + 1) \Downarrow ??\]

\[\begin{align*}
?? \\
(\sigma \cdot y \cdot x, y := x + 1) \Downarrow ?? \\
(\sigma \cdot y, \text{var } x \cdot y := x + 1) \Downarrow ??
\end{align*}\]

We can't apply the assignment rule here, because in the state \(\sigma \cdot y \cdot x\), \(\sigma(x)\) is undefined.
**Alternative declaration semantics**

**Crash and burn:** \((\sigma \cdot y, \text{var } x \cdot y := x + 1) \not\Downarrow\)

\[
\frac{(\sigma_1 \cdot x, s) \Downarrow \sigma_2}{(\sigma_1, \text{var } x \cdot s) \Downarrow \sigma_2|_{x}^{\sigma_1}}
\]

**Default value:** \((\sigma \cdot y, \text{var } x \cdot y := x + 1) \Downarrow (\sigma \cdot y) : y \mapsto 1\)

\[
\frac{((\sigma_1 \cdot x) : x \mapsto 0, s) \Downarrow \sigma_2}{(\sigma_1, \text{var } x \cdot s) \Downarrow \sigma_2|_{x}^{\sigma_1}}
\]

**Junk data:** \((\sigma \cdot y, \text{var } x \cdot y := x + 1) \Downarrow (\sigma \cdot y) : y \mapsto 3 \text{ (or 4, or whatever we want...)}\)

\[
\frac{((\sigma_1 \cdot x) : x \mapsto n, s) \Downarrow \sigma_2}{(\sigma_1, \text{var } x \cdot s) \Downarrow \sigma_2|_{x}^{\sigma_1}}
\]
For a taste of axiomatic semantics, let’s define a Hoare Logic for TinyImp (without var). We write a Hoare triple judgement as:

\[
\{ \varphi \} \; s \; \{ \psi \}
\]

Where \( \varphi \) and \( \psi \) are logical formulae about states, called assertions, and \( s \) is a statement. This triple states that if the statement \( s \) successfully evaluates from a starting state satisfying the precondition \( \varphi \), then the final state will satisfy the postcondition \( \psi \):

\[
\varphi(\sigma) \land (\sigma, s) \downarrow \sigma' \Rightarrow \psi(\sigma')
\]
Proving Hoare Triples

To prove a Hoare triple like:

\[
\{ \text{True} \}
\]
\[
i := 0; \\
m := 1; \\
\textbf{while } i \neq N \textbf{ do} \\
\quad i := i + 1; \\
\quad m := m \times i \\
\textbf{od} \\
\{ m = N! \}
\]

We could prove this using the operational semantics. This is cumbersome, and requires an induction to deal with the \textbf{while} loop. Instead, we’ll define a set of rules to prove Hoare triples directly (called a proof calculus).
Hoare Rules

\[
\begin{align*}
(\sigma, \text{skip}) & \Downarrow \sigma \\
(\sigma_1, s_1) & \Downarrow \sigma_2 \quad (\sigma_2, s_2) \Downarrow \sigma_3 \\
(\sigma_1, s_1; s_2) & \Downarrow \sigma_3 \\
\sigma \vdash e & \Downarrow v \\
(\sigma, x := e) & \Downarrow (\sigma : x \mapsto v) \\
{\{\varphi \land e\}} s_1 & {\{\psi\}} \quad {\{\varphi \land \neg e\}} s_2 & {\{\psi\}} \\
{\{\varphi\}} \text{if} \ e \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{fi} & {\{\psi\}} \\
{\{\varphi\}} \text{while} \ e \ \text{do} \ s \ \text{od} & {\{\varphi \land \neg e\}}
\end{align*}
\]

Continuing on, we can get rules for if, and while with a loop invariant:
There is one more rule, called the *rule of consequence*, that we need to insert ordinary logical reasoning into our Hoare logic proofs:

$$\varphi \Rightarrow \alpha \quad \{\alpha\} \ s \ \{\beta\} \quad \beta \Rightarrow \psi$$

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

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.
Factorial Example

Let’s verify the Factorial program using our Hoare rules:

\[
\{\text{True}\} \\
\{1 = 0!\} \ i := 0; \ \{1 = i!\} \\
\{1 = i!\} \ m := 1; \ \{m = i!\} \\
\{m = i!\} \\
\text{while} \ i \neq N \ \text{do} \ \{m = i! \land i \neq N\} \\
\quad \{m \times (i + 1) = (i + 1)!\} \\
\quad i := i + 1; \\
\quad \{m \times i = i!\} \\
\quad m := m \times i \\
\quad \{m = i!\} \\
\text{od} \ \{m = i! \land i = N\} \\
\{m = N!\} \\
\]

\[
\{\varphi \land e\} \ s_1 \ \{\psi\} \quad \{\varphi \land \neg e\} \ s_2 \ \{\psi\} \\
\{\varphi\} \ \text{if} \ e \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{fi} \ \{\psi\} \\
\{\varphi[x := e]\} \ x := e \ \{\varphi\} \\
\{\varphi \land e\} \ s \ \{\varphi\} \\
\{\varphi\} \ \text{while} \ e \ \text{do} \ s \ \text{od} \ \{\varphi \land \neg e\} \\
\{\varphi\} \ s_1 \ \{\alpha\} \ \{\alpha\} \ s_2 \ \{\psi\} \\
\{\varphi\} \ s_1; \ s_2 \ \{\psi\} \\
\varphi \Rightarrow \alpha \quad \{\alpha\} \ s \ \{\beta\} \quad \beta \Rightarrow \psi \\
\{\varphi\} \ s \ \{\psi\} \\
\]

note: \((i + 1)! = i! \times (i + 1)\)