Transformational Imperative Programming

In this part of the course we will:

- Look at requirements specification for simple programming problems.
- Look at programs as systems
- Look at ways of putting systems together (composing systems)
- Use the above to develop techniques for designing programs that meet their specification.

We look at transformational programming rather than interactive programming. That, is we will assume no output is required by the environment until after the whole computation is complete.

Imperative programming means programming with commands.

States and Behaviours

States of the computer are modeled as mappings from (unprimed) variable names to values.

- For example, if we have variable names x and y of type **int**, then example states include

$$\begin{aligned}\Sigma &= \{\text{“}x\text{”} \mapsto \mathbb{Z}, \text{“}y\text{”} \mapsto \mathbb{Z}\} : \\ i &= \{\text{“}x\text{”} \mapsto 10, \text{“}y\text{”} \mapsto 5\} : \Sigma \\ o &= \{\text{“}x\text{”} \mapsto 6, \text{“}y\text{”} \mapsto 5\} : \Sigma\end{aligned}$$

  For this section of the course we will ignore internal states, so:

- A **behaviour** consists of two states, an initial state and a final state. E.g.

$$i \dagger o = \{\text{“}x\text{”} \mapsto 10, \text{“}y\text{”} \mapsto 5, \text{“}x'\text{”} \mapsto 6, \text{“}y'\text{”} \mapsto 5\}$$

  is a behaviour.

Both input and output states belong to the same signature Σ; so behaviours belong to $\Sigma \dagger \Sigma$.

For the rest of the notes on programming the signature for specifications will be $\Sigma \dagger \Sigma$ for some Σ.

Some Example Specifications.

- Consider the problem of computing the minimum of two natural numbers

$$\Sigma = \{\text{“}x\text{”} \mapsto \mathbb{N}, \text{“}y\text{”} \mapsto \mathbb{N}\}$$
$$\langle x' = \min(x, y)\rangle$$

- Consider the problem of computing the greatest common denominator of two natural numbers

$$\Sigma = \{\text{“}x\text{”} \mapsto \mathbb{N}, \text{“}y\text{”} \mapsto \mathbb{N}\}$$
$$\langle x' = \gcd(x, y)\rangle$$

- Consider the problem of searching for an element x in an array a of N integers

$$\Sigma = \left\{\text{“}b\text{”} \mapsto \mathbb{B}, \text{“}a\text{”} \mapsto \left(\{0, ..N\} \xrightarrow{\text{tot}} \mathbb{Z}\right), \text{“}x\text{”} \mapsto \mathbb{Z}\right\}$$

and $\{0, ..N\} = \{i \in \mathbb{N} \mid 0 \leq i < N\}$.

(We model the value of array a with a total function from $\{0, ..N\}$ to $\mathbb{Z}$.)

A specification is

$$\langle b' = (\exists i \in \{0, ..N\} \cdot a(i) = x)\rangle$$

A better specification would be

$$\langle b' = (\exists i \in \{0, ..N\} \cdot a(i) = x) \wedge x' = x \wedge a' = a\rangle$$

if we require that program variables x and a not change.

- Here is another array search problem. This time the goal is to report the location of an item that may be assumed to be in the array

$$\Sigma = \left\{ \text{“}k\text{”} \mapsto \mathbb{N}, \text{“}a\text{”} \mapsto \{0, ..N\} \xrightarrow{\text{tot}} \mathbb{Z}, \text{“}x\text{”} \mapsto \mathbb{Z} \right\}$$

Note that a specification

$$\langle a(k') = x \rangle$$

is unimplementable. (Remember the commandment.)

We can state assumptions about the input as the antecedent of an implication. A reasonable (implementable) specification would be

$$\langle (\exists i \in \{0, ..N\} \cdot a(i) = x) \Rightarrow a(k') = x \rangle$$

This illustrates an important pattern. Specifications are often of the form

$$\langle P \Rightarrow Q \rangle$$

where P represents an assumption about the input. Then P is called the **precondition** and Q is called the **postcondition**. For inputs, for which P is false, the expression simplifies as follows

$$\begin{aligned} & P \Rightarrow Q \\ = & \\ & \mathfrak{false} \Rightarrow Q \\ = & \\ & \mathfrak{true} \end{aligned}$$

For such inputs, the specification imposes no restrictions on the output.

Programming Constructs

Syntax (Form)

We will consider commands of the following forms

$\mathbf{skip}_{\Sigma\dagger\Sigma}$	Skip
$\mathcal{V} :=_{\Sigma\dagger\Sigma} \mathcal{E}$	Assignment
$\mathcal{V}_0, \mathcal{V}_1, ..., \mathcal{V}_k :=_{\Sigma\dagger\Sigma} \mathcal{E}_0, \mathcal{E}_1, ..., \mathcal{E}_k$	Parallel asignment
$f_{\Sigma\dagger\Sigma}; g_{\Sigma\dagger\Sigma}$	Sequential composition
$\mathbf{if}\ \mathcal{B}\ \mathbf{then}\ f_{\Sigma\dagger\Sigma}\ \mathbf{else}\ g_{\Sigma\dagger\Sigma}$	Alternation
$\mathbf{while}\ \mathcal{B}\ \mathbf{do}\ f_{\Sigma\dagger\Sigma}$	Iteration
$(f_{\Sigma\dagger\Sigma})$	Grouping

where

- $\mathcal{V}$ and each $\mathcal{V}_i$ is a variable name mapped in Σ
- $\mathcal{E}$ is an expression of type $\Sigma(\mathcal{V})$
- each $\mathcal{E}_i$ is an expression of type $\Sigma(\mathcal{V}_i)$
- $\mathcal{B}$ is an expression of type $\mathbb{B}$.
- $f_{\Sigma\dagger\Sigma}$ and $g_{\Sigma\dagger\Sigma}$ are specifications

Note that expressions $\mathcal{E}$ and $\mathcal{B}$ should use only unprimed variables

Precedence: “()”, then “if-then-else” and “while-do”, then “;”

Normally the signature is clear from context, so we just write $\mathbf{skip}$ or $:=$.

Comparison to C and Java

The syntax above is based on the Algol and Pascal languages.

The corresponding syntax in C/C++ and Java, *which will **not** be used in this course*, is

The Course	**C/C++/Java**
$\mathbf{skip}$	`;` or `{}`
$\mathcal{V} := \mathcal{E}$	$\mathcal{V} = \mathcal{E}$ `;`
$f ; g$	$f\ g$
$\mathbf{if}\ \mathcal{B}\ \mathbf{then}\ f\ \mathbf{else}\ g$	`if(` $\mathcal{B}$ `)` f `else` g
$\mathbf{while}\ \mathcal{B}\ \mathbf{do}\ f$	`while(` $\mathcal{B}$ `)` f
(f)	`{` f `}`

Semantics (Meaning)

Each command describes a set of acceptable behaviours.

Thus each command *is* a specification.

Furthermore, we will define composition operators so that they operate on arbitrary specifications, not just on commands.

Skip

skip is the do-nothing transformation. Its output must be the same as its input

$$\mathbf{skip}(i \dagger o) = (i = o)$$

Suppose that Σ maps names "x", "y", and "z". Then

$$\mathbf{skip} = \langle x' = x \wedge y' = y \wedge z' = z \rangle$$

Assignment ('becomes')

Suppose that Σ maps names "x", "y", and "z". Then

$$x := \mathcal{E}$$

is defined to be

$$\langle x' = \mathcal{E} \wedge y' = y \wedge z' = z \rangle$$

Example

- Consider $\Sigma = \{\text{"}x\text{"} \mapsto \mathbb{N}, \text{"}y\text{"} \mapsto \mathbb{N}\}$ then

$$(x := x - y) = \langle x' = x - y \wedge y' = y \rangle$$

Parallel assignment

Suppose that Σ maps names "x", "y", and "z" Then $(x, y := \mathcal{E}, \mathcal{F})$ is defined to be

$$\langle x' = \mathcal{E} \wedge y' = \mathcal{F} \wedge z' = z \rangle$$

Example

- Consider $\Sigma = \{\text{"}x\text{"} \mapsto \mathbb{N}, \text{"}y\text{"} \mapsto \mathbb{N}\}$ then $x, y := y, x$ is

$$\langle x' = y \wedge y' = x \rangle$$

Sequential composition

The sequential composition $f ; g$ performs the transformation f and then the transformation g on the result.

We call the output of f (which is the input to g) m.

If the input is i and the output is o, then we want

$$f(i \dagger m) \quad \text{and} \quad g(m \dagger o)$$

Since we are ignoring intermediate states, we don't care what the value of m is, only that it *exists*.

Thus we define

$$(f ; g)(i \dagger o) = (\exists m : \Sigma \cdot f\,(i \dagger m) \wedge g\,(m \dagger o))$$

Suppose that Σ maps names "x", "y", and "z" Then

$$\langle \mathcal{A} \rangle \,;\, \langle \mathcal{B} \rangle$$

is

$$\langle \exists \dot{x}, \dot{y}, \dot{z} \cdot \mathcal{A}[x', y', z' : \dot{x}, \dot{y}, \dot{z}] \wedge \mathcal{B}[x, y, z : \dot{x}, \dot{y}, \dot{z}] \rangle$$

Example

- Suppose that Σ maps names "x", "y", and "z".

$$
\begin{aligned}
& x := x - 1; y := 2 \times x \\
= \quad & \text{“definition of assignment”} \\
& \langle x' = x - 1 \wedge y' = y \wedge z' = z \rangle \, ; \\
& \langle x' = x \wedge y' = 2 \times x \wedge z' = z \rangle \\
= \quad & \text{“definition of sequential composition”} \\
& \left\langle \exists \dot{x}, \dot{y}, \dot{z} \cdot \left(\begin{array}{r} (x' = x - 1 \wedge y' = y \wedge z' = z) \\ [x', y', z' : \dot{x}, \dot{y}, \dot{z}] \\ \wedge \ (x' = x \wedge y' = 2 \times x \wedge z' = z) \\ [x, y, z : \dot{x}, \dot{y}, \dot{z}] \end{array} \right) \right\rangle \\
= \quad & \text{“making the substitutions”} \\
& \left\langle \exists \dot{x}, \dot{y}, \dot{z} \cdot \left(\begin{array}{c} (\dot{x} = x - 1 \wedge \dot{y} = y \wedge \dot{z} = z) \\ \wedge \ (x' = \dot{x} \wedge y' = 2 \times \dot{x} \wedge z' = \dot{z}) \end{array} \right) \right\rangle \\
= \quad & \text{“one-point law”} \\
& \langle x' = x - 1 \wedge y' = 2 \times (x - 1) \wedge z' = z \rangle
\end{aligned}
$$

Alternation

First we define how the propositional operators to act on boolean functions f and g

$$\begin{aligned}(\neg f)\,(b) &= \neg f(b)\\ (f \wedge g)\,(b) &= f(b) \wedge g(b)\\ (f \vee g)\,(b) &= f(b) \vee g(b)\\ (f \Rightarrow g)\,(b) &= f(b) \Rightarrow g(b)\end{aligned}$$

And therefore we have the following distribution laws.

$$\begin{aligned}\neg \langle \mathcal{A} \rangle &= \langle \neg \mathcal{A} \rangle\\ \langle \mathcal{A} \rangle \wedge \langle \mathcal{B} \rangle &= \langle \mathcal{A} \wedge \mathcal{B} \rangle\\ \langle \mathcal{A} \rangle \vee \langle \mathcal{B} \rangle &= \langle \mathcal{A} \vee \mathcal{B} \rangle\\ \langle \mathcal{A} \rangle \Rightarrow \langle \mathcal{B} \rangle &= \langle \mathcal{A} \Rightarrow \mathcal{B} \rangle\end{aligned}$$

Now we can define

$$\textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else } g$$

to be

$$(\langle \mathcal{A} \rangle \wedge f) \vee (\neg \langle \mathcal{A} \rangle \wedge g)$$

Exercise. Show that

$$(\textbf{if } \mathcal{A} \textbf{ then } f \textbf{ else } g) = (\langle \mathcal{A} \rangle \Rightarrow f) \wedge (\neg \langle \mathcal{A} \rangle \Rightarrow g)$$

Example

- Suppose that Σ maps names "x", "y", and "z"

$$
\begin{array}{ll}
 & \mathbf{if}\ y < x\ \mathbf{then}\ x := y\ \mathbf{else}\ y := x \\
= & \text{“Definition assignment”} \\
 & \mathbf{if}\ y < x \\
 & \mathbf{then}\ \langle x' = y' = y \wedge z' = z \rangle \\
 & \mathbf{else}\ \langle x' = y' = x \wedge z' = z \rangle \\
= & \text{“Definition of alternation”} \\
 & \quad (\langle y < x \rangle \wedge \langle x' = y' = y \wedge z' = z \rangle) \\
 & \vee\ (\langle y \geq x \rangle \wedge \langle x' = y' = x \wedge z' = z \rangle) \\
= & \text{“Distributing angle brackets”} \\
 & \left\langle \begin{array}{r} \quad (y < x \wedge x' = y' = y \wedge z' = z) \\ \vee\ (y \geq x \wedge x' = y' = x \wedge z' = z) \end{array} \right\rangle \\
= & \text{“Distributivity”} \\
 & \left\langle \left(\begin{array}{r} \quad (y < x \wedge x' = y' = y) \\ \vee\ (y \geq x \wedge x' = y' = x) \end{array} \right) \wedge z' = z \right\rangle \\
= & \text{“Definition of } \min \text{”} \\
 & \langle x' = y' = \min(x, y) \wedge z' = z \rangle
\end{array}
$$

Iteration

Iteration (**while**-loop) is a bit more complicated. We will return to it later, once the non-iterative constructs are better understood.