1 Recap

The domain of computation that we (*Winskel*) have been using (for **IMP** terms or **Aexp**s) is the number theory domain $\mathcal{N}$. In general, we may define terms over any domain of computation $\mathcal{D}$ given by:

$$\mathcal{D} = (D, f_1^{\mathcal{D}}, \ldots, f_l^{\mathcal{D}}, a_1^{\mathcal{D}}, \ldots, a_m^{\mathcal{D}}, R_1^{\mathcal{D}}, \ldots, R_n^{\mathcal{D}})$$

where:

D is any set,
$f_i^{\mathcal{D}}$s are some operators or functions (unary, binary etc.) on elements of D,
$a_i^{\mathcal{D}}$s are constants or nullary functions and
$R_i^{\mathcal{D}}$s are relations defined on D.

In particular, the number theory domain $\mathcal{N}$ used in *Winskel* for defining **IMP** terms is given by:

$$\mathcal{N} = (\omega, +, \cdot, 0, 1, =, \leq)$$

where:

ω is the set of whole numbers
$+, \cdot$ are binary functions
$0, 1$ are constants or nullary functions and
$=, \leq$ are binary relations defined on ω

Recall that our states $\sigma \in \Sigma$ were mappings $\sigma : Var \to D$. We can generalize this notion of state to be a mapping $\sigma : Term \to D$ where $Term$ is any well formed expression over the "vocabulary" of the structure $\mathcal{D}$. *eg.* Over $\mathcal{N}$, $(x + y) \cdot 5 + z$ is a term, or over $\mathcal{D}$, we might have $f(g(x), g(a))$, where f is a binary function, g is a unary function, x is a variable and a is some constant. This extension of $\sigma : Var \to D$ to $\sigma : Term \to D$ can be done *homomorphically*, *viz.*

$$\sigma(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} f^{\mathcal{D}}(\sigma(t_1), \ldots, \sigma(t_n))$$

where $f^{\mathcal{D}}$ denotes the *meaning* of the n-ary function f.

Recall that the **IMP** programming language contained the following commands:

skip	(Does nothing)
p ; q	(Sequential composition)
if b **then** p **else** q	(Conditional)
while b **do** p	(While loop)
X := t	(Assignment)

As noted before, *Winskel* restricts the t in the assignment statement to be a term over the number theory domain $\mathcal{N}$, but in general t could be a term over any well formed domain $\mathcal{D}$.

Recap of Structural Operational Semantics (SOS)

Configuration: $\langle p, \sigma \rangle \Downarrow \sigma'$, where p is a *program*, σ is the input state, and σ' is the output state. The semantics is specified using inference rules which help us trace the execution of the program step by step. Example of a rule:

$$\frac{\langle p, \sigma \rangle \Downarrow \sigma', \langle q, \sigma' \rangle \Downarrow \sigma''}{\langle p; q, \sigma \rangle \Downarrow \sigma'}$$

Booleans (and arithmetic expressions) are evaluated slightly differently, in that $\langle b, \sigma\rangle$ maps to a truth value (and $\langle a, \sigma\rangle$ to an element of ω or D). Thus, $\langle b, \sigma\rangle \Downarrow \{0, 1\}$ and an example of a rule involving a boolean could be:

$$\frac{\langle b, \sigma\rangle \Downarrow 1, \langle c, \sigma\rangle \Downarrow 1}{\langle b \wedge c, \sigma\rangle \Downarrow 1}$$

Recap of Denotational Semantics (DS)

Here programs are defined as partial functions. $\mathcal{C}[\![p]\!]$ is a partial function from Σ to Σ. This could be made a total function by having $\mathcal{C}[\![p]\!] : \Sigma_\perp \to \Sigma_\perp$. For booleans, we have: $\mathcal{B}[\![b]\!] : \Sigma \to \{0, 1\}$, and for terms, we have: $\mathcal{A}[\![a]\!] : \Sigma \to D$

Relationship between SOS and DS

$$\mathcal{C}[\![p]\!]\sigma = \sigma' \Leftrightarrow \langle p, \sigma\rangle \Downarrow \sigma'$$

$$\mathcal{B}[\![b]\!]\sigma = 1 \Leftrightarrow \langle b, \sigma\rangle \Downarrow 1$$

We shall now look at two other types of semantics, *Binary Relational Semantics* and *Axiomatic Semantics*.

2 Relational Semantics

- Programs are interpreted as a set of pairs (viz. as a binary relation) of input/output states.
- Booleans (*ie.* tests) are also interpreted as sets of pairs of input states. (Actually a subset of the identity relation).

Thus, $\mathcal{R}[\![p]\!] \subseteq \Sigma \times \Sigma$ is a binary relation on the set of states Σ. Relational semantics can be related to denotational semantics and structural semantics as follows:

$$\begin{aligned} \mathcal{R}[\![p]\!] &= \{(\sigma, \tau) \mid \tau = \mathcal{C}[\![p]\!]\sigma\} \quad (DS) \\ &= \{(\sigma, \tau) \mid \langle p, \sigma\rangle \Downarrow \tau\} \quad (SOS) \end{aligned}$$

We can also define booleans as binary relations ($\mathcal{R}[\![b]\!] \subseteq \Sigma \times \Sigma$.), but as subsets of the identity relation, since booleans do not change state in **IMP**.

$$\begin{aligned} \mathcal{R}[\![b]\!] &= \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = 1\} \quad (DS) \\ &= \{(\sigma, \sigma) \mid \langle b, \sigma\rangle \Downarrow 1\} \quad (SOS) \\ &\subseteq \{(\sigma, \sigma) \mid \sigma \in \Sigma\} = \textit{identity relation on } \Sigma \end{aligned}$$

In particular, notice that: $\mathcal{R}[\![\mathbf{skip}]\!] = \mathcal{R}[\![1]\!] = \mathcal{R}[\![\mathbf{true}]\!] =$ identity relation $= \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$.

3 Axiomatic Semantics (Hoare '69)

The name *Axiomatic Semantics* is a slight misnomer, since it doesn't define the meaning of a program, but instead it is mainly used to prove properties of programs, for verification etc.

The intuition here is that we want to define a wider class of assertions, to have some extra notations in addition to the existing boolean b, which Winskel denotes as *Bexp* (what we call *tests* in this lecture),which are Boolean combinations of atomic formulae in the language of $\mathcal{N}$. For example:

$$((x - y + 3) - z \leq x \cdot 4) \wedge y \neq 0)$$

But we want the assertions to be stronger, so we will extend the assertions to include quantifiers, and get something like the first order formulae in language of number theory. So we add:

- New variables: $i, j, k \ldots$ that range over $\mathcal{N}$ and we call them *IntVar* (in addition to $Var = x, y, z, \ldots$)
- Quantifiers: *eg.* $\forall i, \exists j$

With this extended language, we will be able to express more complicated things like:
$j \mid i =$ j divides i $\stackrel{\text{def}}{=} \exists k.(i = jk)$
y is a prime $\stackrel{\text{def}}{=} (\forall \omega(\omega \mid y \Rightarrow (\omega = 1 \vee \omega = y))) \wedge y \neq 1$
i is a power of 2 $\stackrel{\text{def}}{=} \forall j\ j$ is a prime $\wedge\ j \mid i \Rightarrow j = 2$

To move on, make sure we know the concept of:

- Scope of quantifier (just like the concept of scope of a variable in a λ-expression)
- Free & bound variables:
  eg. $\exists j.\ (i = jk)$: Here i, k are free variables, j is a bound variable
- Safe substitution:
  eg. In $(\exists i\ \forall j\ .\ ij = x)[\ (i + j)/x\ ]$, i, j in the assertion are captured, so just as we do in lambda calculus, we have to rename i, j (α renaming) before substitution.

Winskel extends denotational semantics to give a new way of interpreting a term:

- σ : $\quad Var \to D$ (called *State* by Winskell)
- I : $\quad IntVar \to D$ (called *Interpretation* by Winskell)
- $\mathcal{A}v[\![t]\!]$: $\quad \Sigma \to (I \to D)$
- $\mathcal{B}v[\![b]\!]$: $\quad \Sigma \to (I \to \{0, 1\})$
- For programs, instead of using $\mathcal{C}v$, Winskell changes the notation to $\sigma \stackrel{I}{\models} A$, which means state σ satisfies assertion A under interpretation I, defined inductively as:
  $\sigma \stackrel{I}{\models} \exists i.\ A \stackrel{def}{\Leftrightarrow}$ there exsits $n \in D$ such that $\sigma \stackrel{I[n/i]}{\models} A$ (if we set the value of i to be n, A is satisfied.)
  $I[n/i](j) = \begin{cases} I(j) & i \neq j \\ n & i = j \end{cases}$
  $\mathcal{B}v[\![A]\!]\sigma I = 1 \stackrel{def}{\Leftrightarrow} \sigma \stackrel{I}{\models} A$
  $\mathcal{B}v[\![\exists i.\ A]\!]\sigma I = 1 \stackrel{def}{\Leftrightarrow}$ there exsits $n \in D$ such that $\mathcal{B}v[\![A]\!]\sigma(I[n/i]) = 1$

4 Partial Correctness Assertions (PCA)

Partial correctness assertions have the form: $\{A\}\ p\ \{B\}$
where:

$p =$ program
$A =$ precondition
$B =$ postcondition

A and B are assertions from the extended assertion language having quantifiers.

The above PCA can be interpreted as follows: If program p is started in any state satisfying assertion A, then, if and when p halts, the halting state satisfies assertion B.
Note: This does not imply that p halts. (There exists a counterpart called *Total Correctness Assertion* which implies that p must halt)