Scribe: Andrew Myers (based on earlier student notes) Lecturer: Andrew Myers

These notes cover the denotational semantics of IMP, partial orders, and the fixed-point theorem.

1 Operational Semantics vs. Denotational Semantics

We have described the behavior of programs in an operational manner by inductively defining transition relations to express evaluation and execution. It is fairly easy to to turn the description of the semantics into a an interpreter for the programming language. However, we still have not arrived at a extensional description of programs. We have explored semantic definitions based on translation to a better-understood language, but this in some sense just pushes the problem somewhere else.

The denotational semantics approach is also based on translation, but the target of the translation is mathematical objects, usually functions, rather than a programming language. A semantic function will map syntax to corresponding *denotations* (meanings), arriving at an extensional meaning for the program. Alternatively, we can think of the semantic function as compiling the program into a mathematical object.

2 Semantic Functions

Denotational semantics operates on expressions to produce mathematical objects that are the meaning of the expression. The mechanism for this is a *semantic function*, which is usually a mathematical function.

Example:

$$[\![(\lambda x.\, x)]\!] = \lambda x \in D\,.\, x$$

Here $[\![\cdot]\!]$ is a semantic function that maps the expression to its *denotation*, in this case a mathematical function from some domain D to the same codomain. compiler. The semantic function takes as input $(\lambda x.\, x)$, which is a piece of abstract syntax, which we know is really an abstract syntax tree.

The lambda expression on the right-hand side is a mathematical function, so it can also be represented by its extension: $\{(a, a) \mid a \in D\}$.

3 Modeling issues

We have already been using lambda terms to describe programs, but some of the expressions we have written down have been problematic from the mathematical perspective. When we write down denotations, we need to make sure that all of the denotations are mathematically well-defined.

For example, consider the identity function above. What is the domain D? We have not identified a set for D that can adequately model the lambda calculus and that can satisfy the equational theory of the lambda calculus. And not just any D will work. For the denotational semantics to be sound, the various reductions cannot change the meaning according to the semantic function. For example, we should have $[\![(\lambda x.\, e)e']\!] = [\![e\{e'/x\}]\!]$, because these terms are beta-equivalent.

But coming up with such a D is not easy. Consider the term $\lambda x.\,(x\ x)$, in which x is applied to itself. If x is drawn from some domain D, then it seems that the domain D must be isomorphic to the function space $D \to D$ in order to be able to apply x to itself. But a simple diagonalization argument will convince us that these domains cannot be isomorphic. Suppose we had a one-to-one mapping f from elements of D to elements of $D \to E$, where E is any set containing at least two elements. But then consider the function $\lambda x \in D.\, \neg(f(x)(x))$, where $\neg$ is some function on $E \to E$ that maps no element of E to itself. This function cannot possibly be the image in f of any element of D, because it gives a different answer than $f(x)$ for every $x \in D$. So by contradiction the isomorphism cannot exist.

We have already been using translation to explain language semantics and getting some insight. And the denotational semantics we will be exploring will be similar. But some of the tricks we were using earlier won't work. Because self-application leads to the above paradox (which is really Russell's paradox), the Y combinator is not acceptable as part of the semantics.

4 Parsing lambda expressions and function domains

We will continue to use some conventions to make our semantics more concise. The expression $\lambda xyz.\, e$ means a function that takes the three variables x, y, and z as input and gives e as output. However, this function can be curried, applying one argument at a time, so it is the same as $\lambda x.\, \lambda y.\, \lambda z.\, e$.

$\lambda-$expressions extend as far to the right as possible. Thus, $\lambda x.\, \lambda y.\, \lambda z.\, x\ \lambda w.\, w = \lambda x.\, (\lambda y.\, (\lambda z.\, (x\ (\lambda w.\, w))))$. It doesn't hurt to throw in extra parentheses to make things clearer, though! Application is left associative. Thus, $xyz = (xy)z$.

We will usually write the type (domain) of the argument of a function unless the name of the argument makes it obvious. For example, consider the following function that adds two integers:

$$PLUS = \lambda x \in \mathbb{Z}\,.\, \lambda y \in \mathbb{Z}\,.\, x + y$$

This function takes $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ (one argument at a time) and yields $x + y \in \mathbb{Z}$. Thus, the domain is Z and the codomain is $Z \to Z$. So we have

$$PLUS : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$$

While application associates to the left, the constructor $(\to)$ associates to the right. For example, here we do not need parentheses:

$$PLUS : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$$

Note that we're giving the domain of a function argument using the symbol $\in$. It's more typical to use a colon (:), but we'll reserve that for when we are writing functions in a typed programming language.

5 Semantic Functions for IMP

Recall now the three kinds of expressions in IMP:

- arithmetic expressions **Aexp**:
  $a ::= n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$
- boolean expressions **Bexp**:
  $b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 \leq a_1 \mid a_0 = a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$
- commands **Com**:
  $x := a_0 \mid \mathbf{skip} \mid \mathbf{if}\, b_0\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mid \mathbf{while}\, b_0\, \mathbf{do}\, c_0$

What is the intrinsic meaning of these syntactic categories?

Take for example the arithmetic expressions. Our first thought when we see an expression is to evaluate it and thus the meaning of an arithmetic expression is an integer. However, this doesn't explain the meaning of an evaluation depends on the particular store we have: given store σ, each expression a denotes a unique integer, so the meaning of an arithmetic expression is really a function from stores to integers. Hence we can define a function $\mathcal{A}$ which translates the syntax of the arithmetic expressions into their meaning:

$$\mathcal{A}[\![a]\!]\sigma = n \Leftrightarrow \langle a, \sigma\rangle \Downarrow n$$

Similarly, given a particular store σ boolean expressions b denotes a unique truth value $t \in \mathbb{T} = \{\mathit{true}, \mathit{false}\}$. We can define:

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

Since a command c maps one store into another, we define:

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

Therefore, the meaning functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$ have the following types:

- $\mathcal{A} \in \mathbf{Aexp} \to (\Sigma \to \mathbb{N})$
- $\mathcal{B} \in \mathbf{Bexp} \to (\Sigma \to \mathbb{T})$
- $\mathcal{C} \in \mathbf{Com} \to (\Sigma \to \Sigma)$

We say that the arithmetic expression a denotes $\mathcal{A}[\![a]\!]$ and $\mathcal{A}[\![a]\!]$ is a denotation of a. Similarly, $\mathcal{B}[\![b]\!]$ is a denotation of the boolean b and $\mathcal{C}[\![c]\!]$ is a denotation of command c. Each denotation is in fact a function:

- $\mathcal{A}[\![a]\!] : \Sigma \to \mathbb{N}$
- $\mathcal{B}[\![b]\!] : \Sigma \to \mathbb{T}$
- $\mathcal{C}[\![c]\!] : \Sigma \to \Sigma$

This signature for $\mathcal{C}$ won't quite work, however, because of the possibility of non-termination. We'll see how to fix it shortly.

The functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are defined by structural induction.

5.1 Arithmetic Denotations

First, we define the denotation of arithmetic expressions $\mathcal{A} \in \mathbf{Aexp} \to (\Sigma \to \mathbb{N})$ using structural induction:

- $\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.\, n$

This means that the denotation of n is a function which associates the natural number n to any state σ.

Similarly,

- $\mathcal{A}[\![x]\!] = \lambda\sigma \in \Sigma.\, \sigma x$
- $\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$
- $\mathcal{A}[\![a_0 - a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma$
- $\mathcal{A}[\![a_0 \times a_1]\!] = \lambda\sigma \in \Sigma.\, \mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma$

Notice that the signs $+, -, \times$ on the left-hand sides represent syntactic signs in IMP, whereas the signs on the right represent operations on numbers.

We can write the last three definitions as inductive definitions, similar to the inference rules in the operational semantics:

$$\frac{\mathcal{A}[\![a_0]\!] = f_0 \quad \mathcal{A}[\![a_1]\!] = f_1}{\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.\, f_0\sigma + f_1\sigma}$$

Instead of using λ notation, we can present the definition of the semantics as a relation between states and numbers:

- $\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$
- $\mathcal{A}[\![x]\!] = \{(\sigma, \sigma(x) \mid \sigma \in \Sigma\}$
- $\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$
- $\mathcal{A}[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$

5.2 Boolean Denotations

As for the arithmetic expressions, the function $\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to T)$ is defined using induction on the structure of expressions.

We start by applying $\mathcal{B}$ to the booleans with no subexpressions as follows:

- $\mathcal{B}[\![\mathbf{true}]\!] = \lambda\sigma \in \Sigma.\mathit{true}$
- $\mathcal{B}[\![\mathbf{false}]\!] = \lambda\sigma \in \Sigma.\mathit{false}$

After applying both sides of the first function to σ, we get $\mathcal{B}[\![\mathbf{true}]\!]\sigma = \mathit{true}$ by a β reduction. Not only is this notation more compact, it makes the meaning more clear.

The rest of the rules for boolean denotations are as follows:

- $\mathcal{B}[\![a_0 = a_1]\!]\sigma = \mathbf{if}\, \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\, \mathbf{then}\, \mathit{true}\, \mathbf{else}\, \mathit{false}$
- $\mathcal{B}[\![a_0 \leq a_1]\!]\sigma = \mathbf{if}\, \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\, \mathbf{then}\, \mathit{true}\, \mathbf{else}\, \mathit{false}$
- $\mathcal{B}[\![b_0 \wedge b_1]\!]\sigma = \mathbf{if}\, \mathcal{B}[\![b_0]\!]\sigma \ \wedge\ \mathcal{B}[\![b_1]\!]\sigma\, \mathbf{then}\, \mathit{true}\, \mathbf{else}\, \mathit{false}$
- $\mathcal{B}[\![b_0 \vee b_1]\!]\sigma = \mathbf{if}\, \mathcal{B}[\![b_0]\!]\sigma \ \vee\ \mathcal{B}[\![b_1]\!]\sigma\, \mathbf{then}\, \mathit{true}\, \mathbf{else}\, \mathit{false}$

5.3 Command Denotations

In order to derive the rules for command denotations, we first note that some commands c do not terminate. Formally,

$$(\neg\exists\sigma'.\langle c, \sigma\rangle \Downarrow \sigma')$$

An example of such a command is **while true do skip**. One way to think about commands is as partial functions from states to states ($\Sigma \rightharpoonup \Sigma$). For example, consider the command (**while** $x = 0$ **do skip**). Its denotation is given by $\{(\sigma, \sigma) \mid \sigma(x) = 0\}$, which is not defined for $\sigma(x) \neq 0$. Thus, the corresponding command is not total.

In order to make denotations total, we add a new state, $\perp$, called *bottom*, to represent non-termination. We write $\Sigma_\perp$ to mean the set Σ augmented with the element $\perp$, that is, $\Sigma \cup \{\perp\}$. This is read as the "lift" of Σ. Commands become functions from Σ to $\Sigma_\perp$, and $\mathcal{C} \in \mathbf{Com} \to (\Sigma \to \Sigma_\perp)$.

Using $\perp$ we can now extensionally specify the behavior of commands, including nontermination, in a way that wasn't possible with operational semantics.

The function $\mathcal{C} : \mathbf{Com} \to \mathbf{\Sigma} \to \mathbf{\Sigma}_\perp$ is also defined using induction on the structure of commands as follows:

- $\mathcal{C}[\![\mathbf{skip}]\!]\sigma = \sigma$
- $\mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$
- $\mathcal{C}[\![c_0; c_1]\!]\sigma = \begin{cases} \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma) & (\text{if } \mathcal{C}[\![c_0]\!]\sigma) \neq \perp) \\ \perp & (\text{otherwise}) \end{cases}$
- $\mathcal{C}[\![\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1]\!]\sigma = \mathbf{if}\, \mathcal{B}[\![b]\!]\sigma\, \mathbf{then}\, \mathcal{C}[\![c_0]\!]\sigma\, \mathbf{else}\, \mathcal{C}[\![c_1]\!]\sigma$

Note: It's ok to use $\mathcal{B}$ in the definition of the denotations, since it's not circular.

Let us now try to define the denotation for **while**. In a similar manner with the above definitions, we would like to write:

$$\mathcal{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\sigma = \mathbf{if}\ \neg\mathcal{B}[\![b]\!]\sigma\, \mathbf{then}\ \sigma\, \mathbf{else}\ \mathcal{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!](\mathcal{C}[\![c]\!]\sigma)$$

Unfortunately, this definition is circular: it involves $\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]$ on both sides. The above is actually not a definition, but an equation. We can also write this equation as an equation about sets:

$$\begin{aligned}\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = &\{(\sigma,\sigma) \mid \neg\, \mathcal{B}[\![b]\!]\sigma\} \\ \cup\ &\{(\sigma,\bot) \mid \mathcal{B}[\![b]\!]\sigma \wedge \mathcal{C}[\![c]\!]\sigma = \bot\} \\ \cup\ &\{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma \wedge \mathcal{C}[\![c]\!]\sigma = \sigma'' \neq \bot \wedge \mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma'' = \sigma'\}\end{aligned}$$

We can rewrite this as

$$\begin{aligned}\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = &\{(\sigma,\sigma) \mid \neg\mathcal{B}[\![b]\!]\sigma\} \\ \cup\ &\{(\sigma,\bot) \mid \mathcal{B}[\![b]\!]\sigma \wedge (\sigma,\bot) \in \mathcal{C}[\![c]\!]\} \\ \cup\ &\{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in \mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\}\end{aligned}$$

where $\sigma'' \neq \bot$.

We must come up with an alternative definition for **while**. In order to do this, we will construct a function F such that the denotation of **while** is a fixed point of F. We can define F equivalently using either sets or lambda notation (here d is a command denotation):

$$\begin{aligned}F = \lambda d \in \Sigma \to \Sigma_\bot\,.\, &\{(\sigma,\sigma) \mid \neg\mathcal{B}[\![b]\!]\sigma\} \\ &\cup\ \{(\sigma,\bot) \mid \mathcal{B}[\![b]\!]\sigma \wedge (\sigma,\bot) \in \mathcal{C}[\![c]\!]\} \\ &\cup\ \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in d\} \\ F = \lambda d \in \Sigma \to \Sigma_\bot\,.\, &\lambda\sigma \in \Sigma\,.\, \mathit{if}\ \neg\mathcal{B}[\![b]\!]\ \mathit{then}\ \sigma\ \mathit{else}\ d^*(\mathcal{C}[\![c]\!]\sigma).\end{aligned}$$

where given $f : D \to E_\bot$, we define $f^* : D_\bot \to E_\bot = \lambda x \in D_\bot\,.\, \mathit{if}\ x = \bot\ \ \mathit{then}\ \ \bot\ \ \mathit{else}\ f(x)$ Then, the denotation of *while* is a fixed point of F, i.e., $\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = F(\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!])$.

6 Finding fixed points

Suppose we had an operator *fix* that could find fixed points. What we would like to do is define

$$\begin{aligned}\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] &= \mathit{fix}(F) \\ &= \mathit{fix}(\lambda d \in \Sigma \to \Sigma_\bot\,.\, \lambda\sigma \in \Sigma.\, \mathit{if}\ \neg\mathcal{B}[\![b]\!]\sigma\ \mathit{then}\ \sigma\ \mathit{else}\ d^*(\mathcal{C}[\![c]\!]\sigma))\end{aligned}$$

But which fixed point of F do we want? We would like to take the "least" fixed point, in the sense that we want $\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]$ to give a non-$\bot$ result only when required by the intended semantics. (For example, we want $\mathcal{C}[\![\textbf{while true do skip}]\!]\sigma = \bot$ for all σ.)

First, let's get an idea what this least fixed point might look like. Iterating F applied to some "minimal" function $d_\bot = \lambda\sigma.\ \bot$ yields a sequence of successively better approximations to $\mathcal{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]$:

$$\begin{aligned}d_0 &= d_\bot \\ d_1 &= F(d_\bot) \\ &= \lambda\sigma.\, \mathit{if}\ \neg\mathcal{B}[\![b]\!]\ \mathit{then}\ \sigma\ \mathit{else}\ \bot \\ d_2 &= F(F(d_\bot)) \\ &= \lambda\sigma.\, \mathit{if}\ \neg\mathcal{B}[\![b]\!]\sigma\ \mathit{then}\ \sigma\ \mathit{else} \\ &\qquad \mathit{if}\ \neg\mathcal{B}[\![b]\!]\mathcal{C}[\![c]\!]\sigma\ \mathit{then}\ \mathcal{C}[\![c]\!]\sigma\ \mathit{else}\ \bot \\ d_3 &= F(F(F(d_\bot))) \\ &= \lambda\sigma.\, \mathit{if}\ \neg\mathcal{B}[\![b]\!]\sigma\ \mathit{then}\ \sigma\ \mathit{else} \\ &\qquad \mathit{if}\ \neg\mathcal{B}[\![b]\!]\mathcal{C}[\![c]\!]\sigma\ \mathit{then}\ \mathcal{C}[\![c]\!]\sigma\ \mathit{else} \\ &\qquad\ \ \mathit{if}\ \neg\mathcal{B}[\![b]\!]\mathcal{C}[\![c]\!]\mathcal{C}[\![c]\!]\sigma\ \mathit{then}\ \mathcal{C}[\![c]\!]\mathcal{C}[\![c]\!]\sigma\ \mathit{else}\ \bot \\ &\vdots \\ d_n &= F^n(d_\bot) \\ &\vdots\end{aligned}$$

Here, d_0 doesn't really approximate **while** at all, but d_1 correctly describes a loop whose guard is evaluated once (to false), d_2 correctly describes a loop whose guard is evaluated once or twice, and in general d_n describes a loop whose guard is evaluated up to n times.

The "limit" of this sequence will be the denotation of **while** b **do** c. To take this limit, and to show that it is the *least* fixed point, we will need more mathematical machinery, starting with the the theory of *partial orders*.

7 Partial Orders

A *partial order* (also known as a *partially ordered set* or *poset*) is a pair $(S, \sqsubseteq)$, where

- S is a set of elements.
- $\sqsubseteq$ is a relation on S which is:
  - *i.* reflexive: $x \sqsubseteq x$
  - *ii.* transitive: $(x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$
  - *iii.* antisymmetric: $(x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$

Examples:

- $(\mathbb{Z}, \leq)$, where $\mathbb{Z}$ is the integers and $\leq$ is the usual ordering.
- $(\mathbb{Z}, =)$ (Note that unequal elements are incomparable in this order. Partial orders ordered by the identity relation, $=$, are called *discrete*.)
- $(2^S, \subseteq)$ (Here, 2^S denotes the powerset of S, the set of all subsets of S, often written $\mathcal{P}(S)$, and in Winskel, $\mathcal{P}ow(S)$.)
- $(2^S, \supseteq)$
- $(S, \sqsupseteq)$, if we are given that $(S, \sqsubseteq)$ is a partial order.
- $(\omega, |)$, where $\omega = \{0, 1, 2, \ldots\}$ and $a|b \Leftrightarrow (a \text{ divides } b) \Leftrightarrow (b = ka \text{ for some } k \in \omega)$. Note that for any $n \in \omega$, we have $n|0$; we call 0 an upper bound for ω (but only in this ordering, of course!).

Non-examples:

- $(\mathbb{Z}, <)$ is not a partial order, because $<$ is not reflexive.
- $(\mathbb{Z}, \sqsubseteq)$, where $m \sqsubseteq n \Leftrightarrow |m| \leq |n|$, is not a partial order because $\sqsubseteq$ is not anti-symmetric: $-1 \sqsubseteq 1$ and $1 \sqsubseteq -1$, but $-1 \neq 1$.

The "partial" in partial order comes from the fact that our definition does not require these orders to be total; e.g., in the partial order $(2^{\{a,b\}}, \subseteq)$, the elements $\{a\}$ and $\{b\}$ are incomparable: neither $\{a\} \subseteq \{b\}$ nor $\{b\} \subseteq \{a\}$ hold.

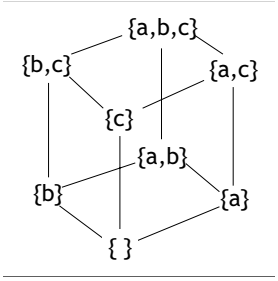
Hasse diagrams Partial orders can be described pictorially using *Hasse diagrams*[1]. In a Hasse diagram, each element of the partial order is displayed as a (possibly labeled) point, and lines are drawn between these points, according to these rules:

1. If x and y are elements of the partial order, and $x \sqsubseteq y$, then the point corresponding to x is drawn lower in the diagram than the point corresponding to y.

[1]Named after Helmut Hasse, 1898-1979. Hasse published fundamental results in algebraic number theory, including the Hasse (or "local-global") principle. He succeeded Hilbert and Weyl as the chair of the Mathematical Institute at Göttingen.

2. A line is drawn between the points representing two elements x and y *iff* $x \sqsubseteq y$ and $\neg\exists z$ in the partial order, distinct from x and y, such that $x \sqsubseteq z$ and $z \sqsubseteq y$ (i.e., the ordering relation between x and y is not due to transitivity).

An example of a Hasse diagram for the partial order on the set $2^{\{a,b,c\}}$ using $\subseteq$ as the binary relation is:



Least upper bounds Given a partial order $(S, \sqsubseteq)$, and a subset $B \subseteq S$, y is an *upper bound* of B *iff* $\forall x \in B.x \sqsubseteq y$. In addition, y is a *least upper bound iff* y is an upper bound and $y \sqsubseteq z$ for all upper bounds z of B. We may abbreviate "least upper bound" as LUB or lub. We shall notate the LUB of a subset B as $\bigsqcup B$. We may also make this an infix operator, as in $\bigsqcup\{x_1, \ldots, x_m\} = x_1 \sqcup \ldots \sqcup x_m$. This is also known as the *join* of elements $x_1, \ldots, x_m$.

Chains A *chain* is a pairwise comparable sequence of elements from a partial order (i.e., elements $x_0, x_1, x_2 \ldots$ such that $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$). For any finite chain, its LUB is its last element (e.g., $\bigsqcup\{x_0, x_1, \ldots, x_n\} = x_n$). Infinite chains (Winskel: ω-chains) may also have LUBs.

Complete partial orders A *complete partial order* (CPO) is a partial order in which every chain has a LUB. Note that the requirement for *every* chain is trivial for finite chains (and thus finite partial orders) – it is the infinite chains that can cause trouble.

Some examples of CPOs:

- $(2^S, \subseteq)$ Here S itself is the LUB for the chain of all elements.
- $(\omega \cup \{\infty\}, \leq)$ Here ∞ is the LUB for any infinite chain: $\forall w \in \omega.w \leq \infty$.
- $([0,1], \leq)$ where $[0,1]$ is the closed continuum, and 1 is a LUB for infinite chains. Note that making the continuum open at the top – $[0,1)$ – would cause this to no longer be a CPO, since there would be no LUB for infinite chains such as $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$
- $(S, =)$ This is a discrete CPO, just as it is a discrete partial order. The only infinite chains are of the sort $x_i \sqsubseteq x_i \sqsubseteq x_i \ldots$, of which x_i is itself a LUB.

Even if $(S, \sqsubseteq)$ is a CPO, $(S, \sqsupseteq)$ is not necessarily a CPO. Consider $((0,1], \leq)$, which is a CPO. Reversing its binary relation yields $((0,1], \geq)$ which is not a CPO, just as $([0,1), \leq)$ above was not.

CPOs can also have a least element, written $\bot$, such that $\forall x.\bot \sqsubseteq x$. We call a CPO with such an element a *pointed CPO*. Winskel instead uses *CPO with bottom*.

8 Least fixed points of functions

Recall that at the end of the last lecture we were attempting to define the least fixed point operator *fix* over the domain $(\Sigma \to \Sigma_\bot)$ so that we could determine calculate fixed points of $F : (\Sigma \to \Sigma_\bot) \to (\Sigma \to \Sigma_\bot)$. It was unclear, however, what the "least" fixed point of this domain would be—how is one function from states to states "less" than another? We've now developed the theory to answer that question.

We define the ordering of states by *information content*: $\sigma \sqsubseteq \sigma'$ *iff* σ gives less (or at most as much) information than σ'. Non-termination is defined to provide less information than any other state: $\forall \sigma \in \Sigma.\bot \sqsubseteq \sigma$. In addition, we have that $\sigma \sqsubseteq \sigma$. No other pairs of states are defined to be comparable. The lifted set of possible states $\Sigma_\bot$ can now be characterized as a flat CPO (a lifted discrete CPO):

- Its elements are elements of $\Sigma \cup \{\bot\}$.
- The ordering relation $\sqsubseteq$ satisfies the reflexive, transitive, and anti-symmetric properties.
- There are three types of infinite chains, each with a LUB:
  1. $\bot \sqsubseteq \bot \sqsubseteq \ldots$, LUB $= \bot$
  2. $\sigma \sqsubseteq \sigma \sqsubseteq \ldots$, LUB $= \sigma$
  3. $\bot \sqsubseteq \bot \sqsubseteq \ldots \sqsubseteq \sigma \sqsubseteq \sigma \sqsubseteq \ldots$, LUB $= \sigma$

9 Functions

We are now ready to define an ordering relation on functions. Functions will be ordered using a *pointwise ordering* on their results. Given a CPO E, a domain D, $f \in D \to E$, and $g \in D \to E$:

$$f \sqsubseteq_{D \to E} g \stackrel{def}{\iff} \forall x \in D. f(x) \subseteq_E g(x)$$

Note that we are defining a new partial order over $D \to E$, and that this CPO is pointed if E is pointed, since $\bot_{D \to E} = \lambda x \in D. \bot_E$.

As an example, consider two functions $\mathbb{Z} \to \mathbb{Z}_\bot$:

$$\begin{aligned} f &= \lambda x \in \mathbb{Z}. \textbf{if}\, x = 0\, \textbf{then}\, \bot\, \textbf{else}\, x \\ g &= \lambda x \in \mathbb{Z}. x \end{aligned}$$

We conclude $f \sqsubseteq g$ because $f(x) \sqsubseteq g(x)$ for all x; in particular, $f(0) = \bot \sqsubseteq 1 = g(0)$.

If E is a CPO, then the function space $D \to E$ is also a CPO. We show that given a chain of functions $f_1 \sqsubseteq f_2 \sqsubseteq f_3 \ldots$, the function $\lambda d \in D. \bigsqcup_{n \in \omega} f_n(d)$ is a least upper bound for this chain. Consider any function g that is an upper bound for all the f_n. In that case, we have:

$$\begin{aligned} & \forall n \in \omega. \forall d \in D. f_n(d) \sqsubseteq g(d) \\ \iff \quad & \forall d \in D. \forall n \in \omega. f_n(d) \sqsubseteq g(d) \end{aligned}$$

Because the f_n form a chain, so do the $f_n(d)$, and because E is a CPO, it has a least upper bound that is necessarily less than the upper bound $g(d)$:

$$\begin{aligned} \Rightarrow \quad & \forall d \in D. (\bigsqcup_{n \in \omega} f_n(d)) \sqsubseteq g(d) \\ \iff \quad & \forall d \in D (\bigsqcup_{n \in \omega} f_n)(d) \sqsubseteq g(d) \\ \iff \quad & \bigsqcup_{n \in \omega} f_n \sqsubseteq g \end{aligned}$$

Therefore, $D \Rightarrow E$ is a CPO under the pointwise ordering.

10 Back to **while**

It's now time to unify our dual understanding of the denotation of **while** as both a limit and a fixed point.

We previously defined the denotation of **while** as both:

$$\begin{aligned}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \mathit{fix}(F) \\ &= \text{limit of } F^n(\bot)\end{aligned}$$

However, we did not know how to define the *fix* operator over the range of F, nor did we have a definition for the least fixed point of F to take as its limit. CPOs have given us the machinery to handle these definitions now.

We assert that:

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \bigsqcup_{n\in\omega} F^n(\bot)$$

As an example to give us confidence that this is the correct definition, we see that:

$$\begin{aligned}\mathcal{C}[\![\textbf{while true do skip}]\!] &= \bigsqcup_{n\in\omega} F^n(\bot) \\ &= \bot_{\Sigma\to\Sigma_\bot} \\ &= \lambda\sigma \in \Sigma.\bot\end{aligned}$$

As we begin to construct a proof that this denotation is correct, we want to show that this limit, or LUB, is a least fixed point of F. That is, we want to show that

$$\bigsqcup_{n\in\omega} F^n(\bot)$$

is the least solution to

$$x = F(x)$$

This will not be true for arbitrary F! We need F to be both monotonic and continuous.

Consider a non-monotonic F:

$$\begin{aligned}F(x) = \;& \textbf{if } x = \bot \textbf{ then } 1 \\ & \textbf{else if } x = 1 \textbf{ then } \bot \\ & \textbf{else if } x = 0 \textbf{ then } 0\end{aligned}$$

Although 0 is clearly a fixed point of this F, $F^n(\bot)$ is not a chain (the elements cycle between $\bot$ and 1), and so we cannot take the LUB of it. Monotonicity would avoid this problem.

Monotonicity guarantees that the elements $F^n(\bot)$ are a chain and hence that we can find a LUB. But it doesn't mean we have a fixed point. Consider a monotonic but non-continuous F defined over the pointed CPO $(\mathbb{R} \cup \{-\infty, \infty\}, \leq)$:

$$F(x) = \textbf{if } x < 0 \textbf{ then } \tan^{-1}(x) \textbf{ else } 1$$

The least fixed point of this F is 1. However,

$$\begin{aligned}F^1(\bot) &= \tan^{-1}(-\infty) = -\frac{\pi}{2} \\ F^2(\bot) &= \tan^{-1}(-\frac{\pi}{2}) = -1 \\ F^2(\bot) &= \tan^{-1}(-1) \approx -0.78\end{aligned}$$

For $x < 0$, $F(x) > x$ and $F(x) < 0$: $F^n(\bot)$ is a chain that approaches 0 arbitrarily closely: its LUB is 0. But $F(0) = 1$, so the LUB is not a fixed point! The least fixed point of this monotonic function is actually $1 = F(1)$. The problem with this function F is that it is not continuous at 0. In general, we will look for some form of *continuity* in F for *fix* to guarantee that the LUB formula gives us a (least) fixed point.

11 Monotonicity and Continuity

Definition: Let $(D, \sqsubseteq)$ be a cpo, $F : D \to D$ a function. F is *monotonic* if

$$\forall x, y \in D \quad x \sqsubseteq y \to F(x) \sqsubseteq F(y).$$

Claim: If $(D, \sqsubseteq, \bot)$ is a pointed cpo and $F : D \to D$ is monotonic then the elements $F^n(\bot)$ form an increasing chain in D:

$$\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq \dots$$

Proof: Since $\bot$ is the least element of D, we have

$$\bot \sqsubseteq F(\bot).$$

Monotonicity of F gives

$$\forall n \in \omega \quad F^n(\bot) \sqsubseteq F^{n+1}(\bot) \Rightarrow F^{n+1}(\bot) \sqsubseteq F^{n+2}(\bot).$$

The result follows by induction.

Notice that if $F : D \to D$ is monotonic and $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ is a chain in D, then $F(x_0) \sqsubseteq F(x_1) \sqsubseteq F(x_2) \sqsubseteq \dots$ is also a chain in D. This permits the following definition.

Definition: Let $(D, \sqsubseteq)$ be a cpo, $F : D \to D$ a monotonic function. F is *continuous* if for every chain

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$$

in D, F preserves the LUB operator:

$$\bigsqcup_{n \in \omega} F(x_n) \quad = \quad F(\bigsqcup_{n \in \omega} x_n).$$

12 The Fixed Point Theorem

We will now show that the properties of monotonicity and continuity allow us to compute the least fixed point as desired.

Claim: Let $(D, \sqsubseteq)$ be a cpo, and let $F : D \to D$ be a monotonic continuous function. Then $\bigsqcup_{n \in \omega} F^n(\bot)$ is a fixed point of F.

Proof: By continuity of F,

$$F(\bigsqcup_{n \in \omega} F^n(\bot)) = \bigsqcup_{n \in \omega} F(F^n(\bot))$$

Applying F,

$$= \bigsqcup_{n \in \omega} F^{n+1}(\bot)$$

Reindexing,

$$= \bigsqcup_{n=1,2,\dots} F^n(\bot)$$

By definition of $\perp$,

$$= \perp \sqcup \bigsqcup_{n=1,2,\ldots} F^n(\perp)$$

And, finally, absorbing the join with $\perp$ into the big join,

$$= \bigsqcup_{n\in\omega} F^n(\perp)$$

We now know that monotonicity and continuity guarantee that $\bigsqcup_{n\in\omega} F^n(\perp)$ is a fixed point of F. We also want $\bigsqcup_{n\in\omega} F^n(\perp)$ to be the *least* fixed point of F. To show this, we must prove that $y = F(y) \Rightarrow \bigsqcup_{n\in\omega} F^n(\perp) \sqsubseteq y$. We can actually prove something even stronger.

Definition: Let $(D, \sqsubseteq)$ be a cpo, $F : D \to D$ a function. $x \in D$ is a *prefixed* point of F if $F(x) \sqsubseteq x$.

Notice that every fixed point of F is also a prefixed point. As a consequence, if a fixed point of F is the least prefixed point of F, it is also the least fixed point of F.

Claim: Let $(D, \sqsubseteq, \perp)$ be a pointed cpo. For any monotonic continuous function, $F : D \to D$, $\bigsqcup_{n\in\omega} F^n$ is the least prefixed point of F.

Proof: Suppose y is a prefixed point of F. By definition of $\perp$,

$$\perp \sqsubseteq y$$

Taking F of both sides,

$$F(\perp) \sqsubseteq F(y) \sqsubseteq y$$

Inductively, for all $n \geq 0$,

$$F^n(\perp) \sqsubseteq y$$

Because y is an upper bound for all the $F^n(\perp)$, it must be at least as large as their least upper bound:

$$\bigsqcup_{n\in\omega} F^n(\perp) \sqsubseteq y$$

We have now proven:

The Fixed Point Theorem: Let $(D, \sqsubseteq, \perp)$ be a pointed cpo. For any monotonic continuous function, $F : D \to D$, $\bigsqcup_{n\in\omega} F^n$ is the least fixed point of F.

13 An instance of the FPT

We have actually encountered an instance of the fixed-point theorem before. Recall lecture 6, when we defined the set of all elements derivable in some rule system to be the least fixed point of the rule operator, R. Our proof in that case was an instantiation of the fixed point theorem on the CPO consisting of all subsets of a set, ordered by set inclusion:

$$R = F$$

$$\emptyset = \perp$$

$$\bigcup = \bigsqcup$$

$$\subseteq = \sqsubseteq$$

The tricky part of the earlier proof corresponded to showing that R is a continuous operator, which was true because we only allow inference rules with a finite number of premises.