Chapter 7

Denotational Semantics [Lecture 11–12]

Version of March 11, 2024

Summary We give an introduction to denotational semantics, first using a preliminary, set-theoretic treatment of unbounded iteration, and then refining it to a simple domain-theoretic formulation.

In this chapter we will consider a more abstract semantics of IMP, based on mathematical functions rather than on derivation trees. (A similar treatment is possible for FUN, but is not covered by this version of the notes.)

The fundamental principle of denotational semantics is that every well-formed syntactic phrase of a programming language (e.g., an expression, a command, a term, a type, etc.) has an intrinsic *meaning* or *denotation*, which is obtained compositionally from the meanings of its subphrases. For example, the meaning of a command will be a partial function from stores to stores, with the meaning of $(c_0; c_1)$ given as the functional composition of the meaning of c_0 followed by the meaning of c_1.

In other words, we define the meaning of all phrases by induction on their syntax. This works straightforwardly as long as the phrases do not involve loops or recursion, at which point the approach would appear to break down, because it doesn't seem possible to define the meaning of, say, a while-loop without some kind of circularity. To resolve this problem, a branch of mathematics collectively known as *domain theory* can be employed to provide a mathematically rigorous framework for working with general recursive definitions of both terms and types, though we will only consider the former here.

7.1 Denotational semantics of IMP

7.1.1 Semantics of arithmetic expressions

Intuitively, since an IMP arithmetic expression in a given store always has a unique value, we can say that the expression's meaning is a mathematical function from stores to integers. That is, for every a we will define a function $\mathcal{A}[\![a]\!] : \Sigma \to \mathbb{Z}$, by induction on the structure (i.e., the syntax) of a.

We will borrow the λ-notation for anonymous functions from functional languages; that is, instead of saying something like "the function f given by $f(n) = n^2 + 1$", we will just write $\lambda n.\, n^2 + 1$. Also, we will often write function applications without parentheses around the argument, unless needed for syntactic disambiguation, e.g., $f\,4$ rather than $f(4)$. Like in FUN, function application groups tighter than other infix operators, so, e.g., $f\,4 + 1$ means $(f\,4) + 1$, not $f\,(4 + 1)$.

With these conventions, we can now define (recalling that $\Sigma = \mathbf{Loc} \to \mathbb{Z}$):

$$\begin{array}{rcl}
\mathcal{A}[\![a]\!] & : & \Sigma \to \mathbb{Z} \\[1ex]
\mathcal{A}[\![\overline{n}]\!] & = & \lambda\sigma.\, n \\
\mathcal{A}[\![X]\!] & = & \lambda\sigma.\, \sigma(X) \\
\mathcal{A}[\![a_0 + a_1]\!] & = & \lambda\sigma.\, \mathcal{A}[\![a_0]\!]\,\sigma + \mathcal{A}[\![a_1]\!]\,\sigma \\
\mathcal{A}[\![a_0 - a_1]\!] & = & \lambda\sigma.\, \mathcal{A}[\![a_0]\!]\,\sigma - \mathcal{A}[\![a_1]\!]\,\sigma \\
\mathcal{A}[\![a_0 \times a_1]\!] & = & \lambda\sigma.\, \mathcal{A}[\![a_0]\!]\,\sigma \times \mathcal{A}[\![a_1]\!]\,\sigma
\end{array}$$

Note that, by convention, the syntactic phrases are written between "semantic brackets" $[\![...]\!]$, and that on the RHS of each equation, we only apply $\mathcal{A}$ to proper subexpressions of the LHS. The definition reads much like an *interpreter* in a functional language, but it is actually more accurate to think of it as a syntax-directed *compiler*, because we can completely eliminate all recursion on source syntax ahead of the evaluation itself.

Theorem 7.1 *For any arithmetic expression a, we have $\mathcal{A}[\![a]\!]\,\sigma = n \Leftrightarrow \langle a, \sigma\rangle \downarrow n$.*

Proof. The proof is by induction on the structure of a. We could prove each of the directions of the bi-implication in separate lemmas, but for conciseness we show them together. The cases are:

- Case $a = \overline{n_0}$. For $\Rightarrow$, we have $\mathcal{A}[\![\overline{n_0}]\!]\,\sigma = n_0$, so $n = n_0$; and using EA-NUM we easily construct a derivation $\overline{\langle \overline{n_0}, \sigma\rangle \downarrow n_0}$. For $\Leftarrow$, a derivation of $\langle \overline{n_0}, \sigma\rangle \downarrow n$ must have used EA-NUM, and so $n = n_0 = \mathcal{A}[\![\overline{n_0}]\!]\,\sigma$.

- Case $a = X$. Almost the same as in the previous case. For $\Rightarrow$, we have $n = \mathcal{A}[\![X]\!]\sigma = \sigma(X)$, and we can use EA-LOC to construct the derivation $\overline{\langle X, \sigma\rangle \downarrow n}$. Conversely, a derivation of $\langle X, \sigma\rangle \downarrow n$ must have used EA-LOC, so $n = \sigma(X) = \mathcal{A}[\![X]\!]\,\sigma$.

- Case $a = a_0 + a_1$. For $\Rightarrow$, we have $n = \mathcal{A}[\![a_0 + a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$. By IH($\Rightarrow$) on a_0, we get a derivation $\mathcal{E}_0$ of $\langle a_0, \sigma\rangle \downarrow \mathcal{A}[\![a_0]\!]\sigma$; and likewise, by IH on a_1, an $\mathcal{E}_1$ of $\langle a_1, \sigma\rangle \downarrow \mathcal{A}[\![a_1]\!]\sigma$. Putting these derivations together with EA-PLUS, we get the required derivation of $\langle a_0 + a_1, \sigma\rangle \downarrow n$.

  Conversely, any derivation of $\langle a_0 + a_1, \sigma\rangle \downarrow n$ must have used EA-PLUS, and so has subderivations $\mathcal{E}_0$ of $\langle a_0, \sigma\rangle \downarrow n_0$ and $\mathcal{E}_1$ of $\langle a_1, \sigma\rangle \downarrow n_1$, with $n = n_0 + n_1$. By IH($\Leftarrow$) on a_0 with $\mathcal{E}_0$, we get that $\mathcal{A}[\![a_0]\!]\sigma = n_0$; and analogously, $\mathcal{A}[\![a_1]\!]\sigma = n_1$. And thus $\mathcal{A}[\![a_0 + a_1]\!]\sigma = n_0 + n_1 = n$.

  The cases for subtraction and multiplication are analogous.

■

Note that Theorem 7.1($\Rightarrow$) implicitly says that the operational semantics is total: for any expression a and store σ, there exists an n (namely $n = \mathcal{A}[\![a]\!]\sigma$) such that $\langle a, \sigma\rangle \downarrow n$. Conversely, the $\Leftarrow$ direction of the theorem implies determinism: if $\langle a, \sigma\rangle \downarrow n$ and $\langle a, \sigma\rangle \downarrow n'$, then both n and n' must be equal to $\mathcal{A}[\![a]\!]\,\sigma$, and therefore also to each other.

Also, *semantic equivalence* of two arithmetic expressions is the same as *equality of their denotations*:

$$\begin{aligned} a \sim a' &\iff \forall\sigma.\,\forall n.\,\langle a, \sigma\rangle \downarrow n \Leftrightarrow \langle a', \sigma\rangle \downarrow n \\ &\iff \forall\sigma.\,\forall n.\,\mathcal{A}[\![a]\!]\sigma = n \Leftrightarrow \mathcal{A}[\![a']\!]\sigma = n \\ &\iff \forall\sigma.\,\mathcal{A}[\![a]\!]\sigma = \mathcal{A}[\![a']\!]\sigma \\ &\iff \mathcal{A}[\![a]\!] = \mathcal{A}[\![a']\!] \end{aligned}$$

7.1.2 Semantics of boolean expressions

The semantics of boolean expressions is very similar to the one for arithmetic expressions. Recall that $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$. We first introduce two auxiliary functions:

$$\begin{aligned} eq, leq &: \mathbb{Z} \times \mathbb{Z} \to \mathbf{T} \\ eq\,(n_0, n_1) &= \begin{cases} \mathbf{true} & \text{if } n_0 = n_1 \\ \mathbf{false} & \text{otherwise} \end{cases} \\ leq\,(n_0, n_1) &= \begin{cases} \mathbf{true} & \text{if } n_0 \leq n_1 \\ \mathbf{false} & \text{otherwise} \end{cases} \end{aligned}$$

Also, for any set A, we define a function

$$\begin{aligned} cond &: \mathbf{T} \times A \times A \to A \\ cond\,(\mathbf{true}, a_0, a_1) &= a_0 \\ cond\,(\mathbf{false}, a_0, a_1) &= a_1 \end{aligned}$$

We can then express the semantics of boolean expressions very concisely:

$$\begin{aligned} \mathcal{B}[\![b]\!] &: \Sigma \to \mathbf{T} \\ \mathcal{B}[\![t]\!] &= \lambda\sigma.\,t \\ \mathcal{B}[\![a_0 = a_1]\!] &= \lambda\sigma.\,eq\,(\mathcal{A}[\![a_0]\!]\sigma, \mathcal{A}[\![a_1]\!]\sigma) \\ \mathcal{B}[\![a_0 \leq a_1]\!] &= \lambda\sigma.\,leq\,(\mathcal{A}[\![a_0]\!]\sigma, \mathcal{A}[\![a_1]\!]\sigma) \\ \mathcal{B}[\![\neg b_0]\!] &= \lambda\sigma.\,cond\,(\mathcal{B}[\![b_0]\!]\sigma, \mathbf{false}, \mathbf{true}) \\ \mathcal{B}[\![b_0 \wedge b_1]\!] &= \lambda\sigma.\,cond\,(\mathcal{B}[\![b_0]\!]\sigma, \mathcal{B}[\![b_1]\!]\sigma, \mathbf{false}) \end{aligned}$$

As expected, we also have:

Theorem 7.2 *For any boolean expression b, $\mathcal{B}[\![b]\!]\sigma = t \Leftrightarrow \langle b, \sigma\rangle \downarrow t$.*

Proof. By structural induction on b; left as an exercise. ■

When reasoning about denotations involving the auxiliary functions, we can just unfold their definitions, and – where relevant – consider cases for whether some boolean value is **true** or **false**. For example, it is immediate to verify the following equations:

$$\begin{array}{rcl} \mathit{eq}\,(n_0, n_1) & = & \mathit{eq}\,(n_0 + n, n_1 + n) \\ f\,(\mathit{cond}\,(t, a_0, a_1)) & = & \mathit{cond}\,(t, f\,a_0, f\,a_1) \\ \mathit{cond}\,(\mathit{eq}\,(n_0, n_1), f\,n_0, a) & = & \mathit{cond}\,(\mathit{eq}\,(n_0, n_1), f\,n_1, a) \end{array}$$

7.1.3 Interlude: the significance of compositionality

More subtly than a mere agreement between the two semantics, the fact that boolean expressions can be given a proper denotational semantics also immediately tells us that they can depend only on the formal *meanings* of any arithmetic expressions they may contain, but not on any *syntactic* properties of such arithmetic expressions, such as their sizes/depths, or what constructs they contain.

This might seem like an obvious property, but it can easily be broken by a careless or ill-advised language design. For example, suppose we wish to extend the syntax of boolean expressions to add a new form:

$$b ::= \cdots \mid \mathbf{iscst}(a)\,,$$

with evaluation rules:

$$\frac{}{\langle \mathbf{iscst}(a), \sigma\rangle \downarrow \mathbf{true}}(\mathit{locs}(a) = \emptyset) \qquad \frac{}{\langle \mathbf{iscst}(a), \sigma\rangle \downarrow \mathbf{false}}(\mathit{locs}(a) \neq \emptyset)$$

(where $\mathit{locs}(a)$ is the set of locations occurring in a, as defined by structural induction in Section 2.2.4). Informally, $\mathbf{iscst}(a)$ checks that the value of a is completely independent of the store, and could hence be determined once and for all (perhaps at compilation time), instead of repeatedly. While such a construct does not break totality or determinism of boolean expressions, it cannot be given a denotational semantics building upon the one for arithmetic expressions. This is because, in the the latter, we have, e.g., $x - x \sim \overline{0}$ (i.e., both expressions denote the constant-zero function from stores to integers), but $\mathbf{iscst}(x - x)$ evaluates to **false** in any store, while $\mathbf{iscst}(\overline{0})$ evaluates to **true**. That is, knowing that two arithmetic expressions are equivalent does *not* guarantee that they can always be substituted for each other without changing the meaning of any larger expression or program.

If we insist on the above operational semantics of $\mathbf{iscst}()$, we can still give it an equivalent denotational semantics, but we will then need to *refine* the semantics of arithmetic expressions. For example, we can make their meanings consist of not only a function from stores to numbers, but also include a boolean flag indicating whether the expression was free of store references:

$$\begin{array}{rcl}
\mathcal{A}'[\![a]\!] & : & (\Sigma \to \mathbb{Z}) \times \mathbf{T} \\[1ex]
\mathcal{A}'[\![\overline{n}]\!] & = & (\lambda\sigma.\, n, \mathbf{true}) \\
\mathcal{A}'[\![X]\!] & = & (\lambda\sigma.\, \sigma(X), \mathbf{false}) \\
\mathcal{A}'[\![a_0 + a_1]\!] & = & (\lambda\sigma.\, f_0\,\sigma + f_1\,\sigma, \mathit{cond}\,(t_0, t_1, \mathbf{false})), \quad \text{where} \\
& & \quad (f_0, t_0) = \mathcal{A}'[\![a_0]\!] \text{ and } (f_1, t_1) = \mathcal{A}'[\![a_1]\!] \\
& \vdots &
\end{array}$$

Then, in the denotational semantics of boolean expressions, we could simply take

$$\begin{array}{rcl}
\mathcal{B}[\![b]\!] & : & \Sigma \to \mathbf{T} \\
& \vdots & \\
\mathcal{B}[\![a_0 = a_1]\!] & = & \lambda\sigma.\, \mathit{eq}\,(f_0\,\sigma, f_1\,\sigma), \quad \text{where} \\
& & \quad (f_0, t_0) = \mathcal{A}'[\![a_0]\!] \text{ and } (f_1, t_1) = \mathcal{A}'[\![a_1]\!] \\
& \vdots & \\
\mathcal{B}[\![\mathbf{iscst}(a)]\!] & = & \lambda\sigma.\, t, \text{ where } (f, t) = \mathcal{A}'[\![a]\!]
\end{array}$$

This refined semantics of arithmetic expressions induces a *finer* notion of semantic equivalence, which requires not only that the two expressions evaluate to the same number in any store, but also that either neither of them contains any locations, or that both do. For example, we would still have $a_0 + a_1 \sim a_1 + a_0$, for all a_0 and a_1, but no longer, e.g., $a \times \overline{0} \sim \overline{0}$. Depending on our purpose with the semantics, this may or may not be an acceptable trade-off.

A quite different approach would be to keep the original semantics of arithmetic expressions (and the natural notion of equivalence it induces), and instead redefine $\mathbf{iscst}(a)$ to be a *semantic*, rather than *syntactic*, property. That is, we could simply require that a be a constant-valued expression, regardless of its form. This is easy enough to specify in a denotational semantics:

$$\mathcal{B}[\![\mathbf{iscst}(a)]\!] \;=\; \lambda\sigma.\begin{cases} \mathbf{true} & \text{if } \exists n.\, \forall\sigma'.\, \mathcal{A}[\![a]\!]\sigma' = n \\ \mathbf{false} & \text{otherwise} \end{cases}$$

However, it is not obvious whether such a specification can actually be implemented, especially while keeping the semantics of boolean expressions a total function. Fortunately, the particular problem of determining constancy of an IMP arithmetic expression is easily seen to be decidable. This is because any such expression can be be systematically rewritten into an integer polynomial in multiple variables, so after distributing out all multiplications (e.g. $(x-2y)x = x^2-2yx$) and collecting like terms (e.g. $5xy-2yx = 3xy$), it is obvious whether there are any non-constant terms left. However, this simplification process could still require time exponential in the size of the expression, and would require nontrivial effort to formally specify for an executable operational semantics.

On the other hand, many similar-looking properties, such as determining whether an IMP arithmetic expression always evaluates to a non-zero value, can quickly be shown to be undecidable, by reduction from Hilbert's tenth problem (posed in 1900, but only formally shown unsolvable in 1970). Thus, $\mathbf{iscst}(b)$ for *boolean* expressions (e.g., $\mathbf{iscst}(a = \underline{0})$ for some arithmetic expression a) could *not* be effectively defined in a purely semantic way.

7.1.4 Semantics of commands (preliminary formulation)

Intuitively, the meaning of a command should be a *partial* function from states to states, so we would expect to have $\mathcal{C}[\![c]\!] : \Sigma \rightharpoonup \Sigma$. However, since working with partial functions directly is somewhat awkward and notoriously error-prone, we will instead represent partial functions as a particular kind of *total* functions.

Specifically, for any set A, define the *lifted* set $A_\perp = \{\lfloor a \rfloor \mid a \in A\} \cup \{\perp\}$, where $\lfloor a \rfloor$ is some injective function that doesn't have $\perp$ in its range. That is, $A_\perp$ extends A with an additional "undefined" element $\perp$, guaranteed to be different from those in A (even if A itself already contains a $\perp$). (Set-theoretically, we could take $\perp = \emptyset$, and $\lfloor a \rfloor = \{a\}$, but the exact choice of representation doesn't matter.) For example, $\mathbf{T}_\perp = \{\lfloor \mathbf{true} \rfloor, \lfloor \mathbf{false} \rfloor, \perp\}$. Then any partial function from A to B can be seen as a total function from A to $B_\perp$, where a result of $\perp$ represents that the function is undefined.

We also introduce, for any sets A and B, the following auxiliary operations:

$$\begin{array}{rcl} \eta & : & A \to A_\perp \\ \eta\, a & = & \lfloor a \rfloor \\[4pt] (\star) & : & A_\perp \times (A \to B_\perp) \to B_\perp \\ \perp \star f & = & \perp \\ \lfloor a \rfloor \star f & = & f\, a \end{array}$$

These will allow us to concisely and unambiguously write down and reason about applications and compositions of partial functions. (You may recognize $(A_\perp, \eta, \star)$ as a *monad*, representing the computational effect of possible divergence or errors. In fact, the use of monads for uniformly modeling effects such as state, exceptions, or nondeterminism was pioneered in the setting of denotational semantics.)

We are now ready to define the denotational semantics of commands, albeit with one serious hole that will need some more mathematical machinery to fully resolve. We take:

$$\begin{array}{rcl} \mathcal{C}[\![c]\!] & : & \Sigma \to \Sigma_\perp \\[4pt] \mathcal{C}[\![\mathbf{skip}]\!] & = & \lambda\sigma.\, \eta\, \sigma \\ \mathcal{C}[\![X := a]\!] & = & \lambda\sigma.\, \eta\, (\sigma[X \mapsto \mathcal{A}[\![a]\!]\sigma]) \\ \mathcal{C}[\![c_0; c_1]\!] & = & \lambda\sigma.\, (\mathcal{C}[\![c_0]\!]\sigma \star \lambda\sigma_1.\, \mathcal{C}[\![c_1]\!]\sigma_1) \\ \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] & = & \lambda\sigma.\, \mathit{cond}\, (\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma) \\ \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!] & = & \varphi, \text{ where } \varphi = \lambda\sigma.\, \mathit{cond}\, (\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma \star \varphi, \eta\, \sigma) \end{array}$$

In the last line, we have given a *recursive* definition of the function φ. While the definition makes some amount of sense if we think of the semantics as interpreting commands in (or compiling into) a purely functional language, it is far less clear what such recursion should mean in a mathematical definition. In particular, two questions arise: (1) must a function φ with the requested property always exist, and (2) can there in general be multiple functions satisfying φ's defining equation, and if so, which of them do we want? We postpone these questions to the next section.

We will now prove equivalence of the operational and denotational semantics. Unlike the case for expressions, the proof is involved enough that we must split it into two parts. We start with the conceptually simpler direction:

Lemma 7.3 *If $\langle c, \sigma\rangle \downarrow \sigma'$, then $\mathcal{C}[\![c]\!]\sigma = \lfloor\sigma'\rfloor$.*

Proof. By induction on the derivation $\mathcal{E}$ of the assumption:

- Case $\mathcal{E} = \text{EC-Skip}\,\dfrac{}{\langle \mathbf{skip}, \sigma\rangle \downarrow \sigma}$, so $c = \mathbf{skip}$ and $\sigma' = \sigma$.

  Then $\mathcal{C}[\![c]\!]\sigma = \mathcal{C}[\![\mathbf{skip}]\!]\sigma = \eta\,\sigma = \lfloor\sigma\rfloor = \lfloor\sigma'\rfloor$, so we are done.

- Case $\mathcal{E} = \text{EC-Assign}\,\dfrac{\overset{\mathcal{E}_0}{\langle a, \sigma\rangle \downarrow n}}{\langle X := a, \sigma\rangle \downarrow \sigma[X \mapsto n]}$, so $c = (X := a)$ and $\sigma' = \sigma[X \mapsto n]$.

  By Theorem 7.1($\Leftarrow$) on $\mathcal{E}_0$, we get that $\mathcal{A}[\![a]\!]\sigma = n$, and so $\mathcal{C}[\![c]\!]\sigma = \mathcal{C}[\![X := a]\!]\sigma = \eta\,(\sigma[X \mapsto \mathcal{A}[\![a]\!]\sigma]) = \lfloor\sigma[X \mapsto n]\rfloor = \lfloor\sigma'\rfloor$.

- Case $\mathcal{E} = \text{EC-Seq}\,\dfrac{\overset{\mathcal{E}_0}{\langle c_0, \sigma\rangle \downarrow \sigma''} \qquad \overset{\mathcal{E}_1}{\langle c_1, \sigma''\rangle \downarrow \sigma'}}{\langle c_0; c_1, \sigma\rangle \downarrow \sigma'}$, so $c = (c_0; c_1)$.

  By IH on $\mathcal{E}_0$, we get $\mathcal{C}[\![c_0]\!]\sigma = \lfloor\sigma''\rfloor$; and by IH on $\mathcal{E}_1$, $\mathcal{C}[\![c_1]\!]\sigma'' = \lfloor\sigma'\rfloor$. But then, $\mathcal{C}[\![c]\!]\sigma = \mathcal{C}[\![c_0]\!]\sigma \star \lambda\sigma_1.\,\mathcal{C}[\![c_1]\!]\sigma_1 = \lfloor\sigma''\rfloor \star \lambda\sigma_1.\,\mathcal{C}[\![c_1]\!]\sigma_1 = (\lambda\sigma_1.\,\mathcal{C}[\![c_1]\!]\sigma_1)\,\sigma'' = \mathcal{C}[\![c_1]\!]\sigma'' = \lfloor\sigma'\rfloor$.

- Case $\mathcal{E} = \text{EC-IfT}\,\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \mathbf{true}} \qquad \overset{\mathcal{E}_1}{\langle c_0, \sigma\rangle \downarrow \sigma'}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \downarrow \sigma'}$, so $c = \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$.

  By Theorem 7.2($\Leftarrow$) on $\mathcal{E}_0$, we get $\mathcal{B}[\![b]\!]\sigma = \mathbf{true}$; and by IH on $\mathcal{E}_1$, $\mathcal{C}[\![c_0]\!]\sigma = \lfloor\sigma'\rfloor$. Thus $\mathcal{C}[\![c]\!]\sigma = \mathit{cond}\,(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma) = \mathit{cond}\,(\mathbf{true}, \lfloor\sigma'\rfloor, \mathcal{C}[\![c_1]\!]\sigma) = \lfloor\sigma'\rfloor$, as required. (Note that, according to the definition of *cond*, it doesn't matter if $\mathcal{C}[\![c_1]\!]\sigma = \bot$.) The case where $\mathcal{E}$ ends in EC-IfF is completely analogous.

- Case $\mathcal{E} = \text{EC-WhileF}\,\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \mathbf{false}}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c_0, \sigma\rangle \downarrow \sigma}$, so $c = \mathbf{while}\ b\ \mathbf{do}\ c_0$ and $\sigma' = \sigma$.

  Like in the case for EC-IfF, by Theorem 7.2 on $\mathcal{E}_0$ we have $\mathcal{B}[\![b]\!]\sigma = \mathbf{false}$. Thus, assuming the function φ in $\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]$ exists at all, we have $\mathcal{C}[\![c]\!]\sigma = \varphi\,\sigma = \mathit{cond}\,(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma \star \varphi, \eta\,\sigma) = \mathit{cond}\,(\mathbf{false}, \mathcal{C}[\![c_0]\!]\sigma \star \varphi, \lfloor\sigma\rfloor) = \lfloor\sigma\rfloor = \lfloor\sigma'\rfloor$.

- Case $\mathcal{E} = \text{EC-WhileT}\,\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \mathbf{true}} \qquad \overset{\mathcal{E}_1}{\langle c_0, \sigma\rangle \downarrow \sigma''} \qquad \overset{\mathcal{E}_2}{\langle \mathbf{while}\ b\ \mathbf{do}\ c_0, \sigma''\rangle \downarrow \sigma'}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c_0, \sigma\rangle \downarrow \sigma'}$, so again $c = \mathbf{while}\ b\ \mathbf{do}\ c_0$.

Here, by Theorem 7.2 on $\mathcal{E}_0$ we have $\mathcal{B}[\![b]\!]\sigma = \mathbf{true}$; by IH on $\mathcal{E}_1$, $\mathcal{C}[\![c_0]\!]\sigma = \lfloor\sigma''\rfloor$; and by IH on $\mathcal{E}_2$, $\mathcal{C}[\![c]\!]\sigma'' = \lfloor\sigma'\rfloor$.

So, again assuming the existence of φ, we have $\mathcal{C}[\![c]\!]\sigma = \varphi\,\sigma = \mathit{cond}\,(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma \star \varphi, \eta\,\sigma) = \mathit{cond}\,(\mathbf{true}, \lfloor\sigma''\rfloor \star \varphi, \lfloor\sigma\rfloor) = \lfloor\sigma''\rfloor \star \varphi = \varphi\,\sigma'' = \mathcal{C}[\![c]\!]\sigma'' = \lfloor\sigma'\rfloor$.

■

For the opposite direction, we can show:

Lemma 7.4 *If* $\mathcal{C}[\![c]\!]\sigma = \lfloor\sigma'\rfloor$*, then* $\langle c, \sigma\rangle \downarrow \sigma'$*.*

Proof. By structural induction on c:

- Case $c = \mathbf{skip}$. Then we have $\lfloor\sigma'\rfloor = \mathcal{C}[\![c]\!]\sigma = \mathcal{C}[\![\mathbf{skip}]\!]\sigma = \eta\,\sigma = \lfloor\sigma\rfloor$, so $\sigma' = \sigma$ (because $\lfloor - \rfloor$ was injective). And thus, we can just use EC-Skip to construct the required derivation of $\langle \mathbf{skip}, \sigma\rangle \downarrow \sigma$.

- Case $c = (X := a)$. Then $\lfloor\sigma'\rfloor = \mathcal{C}[\![X := a]\!]\sigma = \lfloor\sigma[X \mapsto \mathcal{A}[\![a]\!]\sigma]\rfloor$, so $\sigma' = \sigma[X \mapsto n]$, where $n = \mathcal{A}[\![a]\!]\sigma$. By Theorem 7.1($\Rightarrow$) on a, we get a derivation $\mathcal{E}_0$ of $\langle a, \sigma\rangle \downarrow n$, and EC-Assign on $\mathcal{E}_0$ gives us the required $\langle X := a, \sigma\rangle \downarrow \sigma[X \mapsto n]$.

- Case $c = (c_0; c_1)$. Here, $\lfloor\sigma'\rfloor = \mathcal{C}[\![c_0; c_1]\!]\sigma = \mathcal{C}[\![c_0]\!]\sigma \star \lambda\sigma_1.\,\mathcal{C}[\![c_1]\!]\sigma_1$. By the definition of $\star$, this can only happen if $\mathcal{C}[\![c_0]\!]\sigma = \lfloor\sigma''\rfloor$ for some σ'', and $\mathcal{C}[\![c_1]\!]\sigma'' = \lfloor\sigma'\rfloor$. But then by IH on c_0, we get a derivation $\mathcal{E}_0$ of $\langle c_0, \sigma\rangle \downarrow \sigma''$; and by IH on c_1, a derivation $\mathcal{E}_1$ of $\langle c_1, \sigma''\rangle \downarrow \sigma'$. And then, EC-Seq on $\mathcal{E}_0$ and $\mathcal{E}_1$ gives us precisely the required derivation of $\langle c_0; c_1, \sigma\rangle \downarrow \sigma'$.

- Case $c = \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$. We have $\lfloor\sigma'\rfloor = \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \mathit{cond}\,(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma)$. We know that $\mathcal{B}[\![b]\!]\sigma$ must be one of the two truth values. Suppose $\mathcal{B}[\![b]\!]\sigma = \mathbf{true}$; then by Theorem 7.2($\Rightarrow$), we get a derivation $\mathcal{E}_0$ of $\langle b, \sigma\rangle \downarrow \mathbf{true}$. Further, $\lfloor\sigma'\rfloor = \mathit{cond}\,(\mathbf{true}, \mathcal{C}[\![c_0]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma) = \mathcal{C}[\![c_0]\!]\sigma$, and thus by IH on c_0, we get an $\mathcal{E}_1$ of $\langle c_0, \sigma\rangle \downarrow \sigma'$. Finally, by EC-IfT on $\mathcal{E}_0$ and $\mathcal{E}_1$, we get the required derivation of $\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \downarrow \sigma'$. The subcase where $\mathcal{B}[\![b]\!]\sigma = \mathbf{false}$ is completely analogous.

- Case $c = \mathbf{while}\ b\ \mathbf{do}\ c_0$. Postponed to Section 7.2.2. ■

This is about as far as we can get with our preliminary semantics. To proceed, we'll need to make a slight change of setting.

7.2 A refined framework for denotational semantics

7.2.1 Basic domain theory

A fundamental concept in domain theory is the notion of comparing the "definedness" of two partial functions, and characterizing a (total or partial) function as a *limit* (in the mathematical sense) of a sequence of more and more defined approximations of that function. Let us introduce some machinery for this.

A *partially ordered set* (or *poset*) is a set A equipped with a distinguished relation $(\sqsubseteq_A) : A \times A \to \Omega$ satisfying three conditions:

1. For all $a \in A$, $a \sqsubseteq_A a$. (reflexivity)

2. If $a \sqsubseteq_A a'$ and $a' \sqsubseteq_A a''$, then $a \sqsubseteq_A a''$. (transitivity)

3. If $a \sqsubseteq_A a'$ and $a' \sqsubseteq_A a$, then $a = a'$. (antisymmetry)

(When the poset A is clear from the context, we'll write just $\sqsubseteq$ for $\sqsubseteq_A$.) Although the integers or reals with the ordinary arithmetic ordering ($\leq$) are technically posets, they are not good examples, because they are actually *totally* ordered sets (i.e., for any two numbers n and n', either $n \leq n'$, $n' \leq n$, or both). Rather, the prototypical example of a "proper" poset is the set of all subsets of some other set X, i.e., $A = \mathcal{P}(X)$, with subset inclusion ($\subseteq$) as the ordering relation. A bit more subtly, the set of partial functions from X to Y, $A = X \rightharpoonup Y$, is also a poset when ordered by $f \sqsubseteq_{X \rightharpoonup Y} f' \Leftrightarrow \forall x \in \operatorname{dom} f.\, x \in \operatorname{dom} f' \wedge f'(x) = f(x)$. (Equivalently, we may say that $f \sqsubseteq_{X \rightharpoonup Y} f' \Leftrightarrow gr(f) \subseteq gr(f')$, where the *graph* of a partial function is the set of pairs given by $gr(f) = \{(x, y) \in X \times Y \mid x \in \operatorname{dom} f \wedge f(x) = y\}$.)

An *ω-chain* (frequently just called a *chain*) in a poset is an infinite, (not necessarily strictly) increasing sequence of elements from A: $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots$. (We use ω as another name for the natural numbers $\mathbb{N}$, when emphasizing just their order structure, not the full complement of arithmetic operations like $+$ or $\times$.) A poset is said to be *complete* if every chain $(a_i)^{i \in \omega}$ has a *least upper bound* (or *supremum*), i.e., an element $a \in A$ such that:

1. $\forall i \in \omega.\, a_i \sqsubseteq a$. (upper bound)

2. if also $\forall i \in \omega.\, a_i \sqsubseteq a'$ for some a', then $a \sqsubseteq a'$. (least among upper bounds)

Note that the supremum of a chain is unique, i.e., if both a and a' are *least* upper bounds, they must necessarily be equal by antisymmetry. We can therefore unambiguously write $\bigsqcup_{i \in \omega} a_i$ for the supremum of the chain $(a_i)^{i \in \omega}$. A complete poset is commonly referred to as an (ω-)*cpo* or *predomain.*

In particular, the poset of partial functions from X to Y is a cpo, because we can take $\bigsqcup_{i \in \omega} f_i$ to be the partial function f such that $f(x) = f_i(x)$ if $x \in \operatorname{dom} f_i$ for some i, and $f(x)$ undefined otherwise. Note that the choice of i doesn't matter, because if $j \geq i$, then $f_j(x) = f_i(x)$ by the assumption that the $(f_i)^{i \in \omega}$ form a chain.

A cpo D is called *pointed*, a *cppo*, or a *domain*, if it also has a *least element*, i.e., a $d \in D$ such that for all $d' \in D$, $d \sqsubseteq_D d'$. Again, by antisymmetry of $\sqsubseteq$, a least element is necessarily unique, and we may unambiguously refer to it as $\bot_D$ when it exists.

Finally, we say that a function f between posets A and B is *order-preserving*, or *monotone*, if $\forall a, a' \in A.\, a \sqsubseteq_A a' \Rightarrow f(a) \sqsubseteq_B f(a')$. Note that when f is monotone and $(a_i)^{i \in \omega}$ is a chain in A, then $(f(a_i))^{i \in \omega}$ is a chain in B. A monotone function between two cpos is then said to be *continuous* if it preserves suprema, i.e., if $f(\bigsqcup_{i \in \omega} a_i) = \bigsqcup_{i \in \omega} f(a_i)$. And a function between pointed posets D and E is called *strict* if it also preserves least elements, i.e., if $f(\bot_D) = \bot_E$.

Many familiar constructions of sets and functions have direct analogs in the world of cpos and continuous functions. We state without separate proof a number of useful results:

- The identity function $id_A = \lambda a.\, a : A \to A$ on any cpo A is continuous; and if $f : A \to B$ and $g : B \to C$ are continuous, then so is $g \circ f = \lambda a.\, g(f(a)) : A \to C$. When all the cpos are pointed, evidently *id* is strict, and so is $g \circ f$ for strict f and g. Also, for any $b_0 \in B$, the constant function $\lambda a.\, b_0 : A \to B$ is continuous, but it is only strict if $b_0 = \bot_B$.

- Any set X (e.g., $\mathbb{Z}$ or $\mathbf{T}$) can be seen as a *discrete cpo*, where we take the ordering relation $\sqsubseteq_X$ to be simply the equality relation on X. This clearly satisfies the requirements for a poset, and also those for a cpo, since all chains in X must by definition be constant. For the same reason, any function $f : X \to A$ is trivially continuous, regardless of the ordering on A. X is technically pointed iff it has exactly one element, but we will not normally use that, since we generally reserve $\bot$ for representing errors or nontermination, which is only meaningful if there is at least one other element in the poset.

- For any poset A, we can partially order the lifted set $A_\bot = \{\lfloor a \rfloor \mid a \in A\} \cup \{\bot\}$ by

  $$d \sqsubseteq_{A_\bot} d' \iff d = \bot \vee \exists a, a' \in A.\, d = \lfloor a \rfloor \wedge d' = \lfloor a' \rfloor \wedge a \sqsubseteq_A a' .$$

  $A_\bot$ is also easily seen to be a cpo whenever A is one, and $A_\bot$ is pointed by construction, with $\bot_A = \bot$. Further, the inclusion function $\eta : A \to A_\bot$ is continuous; and for a continuous $f : A \to B_\bot$, the *strict extension* of f, given by $\lambda d.\, d \star f : A_\bot \to B_\bot$, is continuous and strict.

- For posets A and B, we can partially order their cartesian product $A \times B = \{(a, b) \mid a \in A, b \in B\}$ by

  $$(a, b) \sqsubseteq_{A \times B} (a', b') \iff a \sqsubseteq_A a' \wedge b \sqsubseteq_B b' .$$

  If A and B are cpos, then so is $A \times B$. (And if X and Y are discrete cpos, then so is $X \times Y$; this means that functions like $(+) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, or $leq : \mathbb{Z} \times \mathbb{Z} \to \mathbf{T}$ are trivially continuous.) Moreover, the evident *projection* functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are continuous; and if $f : C \to A$ and $g : C \to B$ are continuous, then so is the *pairing* function $\langle f, g \rangle = \lambda c.\, (f(c), g(c)) : C \to A \times B$. (These constructions obviously generalize to n-ary products, including for $n = 0$, meaning that the unique function $\langle\rangle : C \to \mathbf{1}$, where $\mathbf{1} = \{*\}$ is the one-point cpo, is continuous.) Finally, if D and E are pointed, then so is $D \times E$, with $\bot_{D \times E} = (\bot_D, \bot_E)$, and the projection functions are strict.

- For posets A and B, we can order their *sum*, or *disjoint union*, $A + B = \{(1, a) \mid a \in A\} \cup \{(2, b) \mid b \in B\}$ by

  $$(i, u) \sqsubseteq_{A+B} (i', u') \iff (i = i' = 1 \wedge u \sqsubseteq_A u') \vee (i = i' = 2 \wedge u \sqsubseteq_B u') .$$

  When A and B are cpos, so is $A + B$. Moreover, the *injection* functions $\iota_1 = \lambda a.(1, a) : A \to A + B$ and $\iota_2 = \lambda b.(2, b) : B \to A + B$ are continuous; and if $f : A \to C$ and $g : B \to C$ are continuous, then so is the *casing* function $[f, g] : A + B \to C$, given by $[f, g]\,(1, a) = f(a)$ and $[f, g]\,(2, b) = g(b)$. (Again, these constructions generalize to n-ary sums, including for $n = 0$.) However, $D + E$ is *not* pointed even when both D and E are, because no single element in the disjoint union is less-than-or-equal-to all the others.

Note that there is a subtle difference between $A_\perp$ and $A+\{*\}$: while both extend the poset A with an extra "undefined" element, distinct from all the elements of A, the ordering is different: we have $\perp \sqsubseteq_{A_\perp} \lfloor a \rfloor$, but $(2,*) \not\sqsubseteq_{A+\{*\}} (1,a)$. This means that the function $\mathit{isdiv} : A_\perp \to \mathbf{T}$, given by $\mathit{isdiv}(\lfloor a \rfloor) = \mathbf{false}$ and $\mathit{isdiv}(\perp) = \mathbf{true}$, is *not* monotone (let alone continuous), while $\mathit{iserr} = [\lambda a.\, \mathbf{false}, \lambda e.\, \mathbf{true}] : A + \{*\} \to \mathbf{T}$ is fine. In a denotational semantics, we can thus distinguish formally between *divergence* (or *fatal errors*), which are by definition unrecoverable; and *exceptions* (or *non-fatal errors*) which may be caught and handled. And we can of course model an A-returning computation that can potentially also diverge or signal a non-fatal error by $(A + \{*\})_\perp$.

- For cpos A and B, the set of *continuous* functions from A to B, written $[A \to B]$, can be organized as a cpo by taking

$$f \sqsubseteq_{[A\to B]} f' \Leftrightarrow \forall a \in A.\, f(a) \sqsubseteq_B f'(a)\,.$$

(The ordering relation on A doesn't matter in this definition.) Note in particular that when X is discretely ordered, $[X \to B] = X \to B$, because all functions from X are continuous. (This gives us an easy way to show that $\mathit{cond} : \mathbf{T} \times A \times A \to A$ is continuous, because it can be written as $\mathit{cond}\,(t, a_1, a_2) = \mathit{prj}(t)\,(a_1, a_2)$ where $\mathit{prj} : \mathbf{T} \to [A \times A \to A]$, given by $\mathit{prj}(\mathbf{true}) = \pi_1$ and $\mathit{prj}(\mathbf{false}) = \pi_2$, is automatically continuous because $\mathbf{T}$ is discrete.) Further, the general *application* operation $\mathit{ap} = \lambda(f, a).\, f\,a : [A \to B] \times A \to B$ is continuous; and for a continuous $f : C \times A \to B$, the *currying* of f, $\mathit{cur}(f) = \lambda c.\, \lambda a.\, f(c, a) : C \to [A \to B]$ is continuous. Finally, when E is pointed, then so is $[A \to E]$, with $\perp_{[A\to E]} = \lambda a.\, \perp_E$. We observe that our previously defined ordering on the cpo of partial functions between two sets, $X \rightharpoonup Y$, coincides with the derived ordering relation on $[X \to Y_\perp]$.

The above properties mean that our semantics of IMP, though originally expressed with sets and general set-theoretic functions can also be seen as working with cpos and continuous functions. This matters, because we will then be able to employ the following important theorem:

Theorem 7.5 (Existence of fixed points) *Let $f : D \to D$ be a continuous function on a pointed cpo D, and define $\mathit{fix}(f) = \bigsqcup_{i\in\omega} f^i(\perp_D)$, where f^n is the n-fold composition of f with itself (i.e., $f^0 = id$ and $f^{n+1} = f \circ f^n$). Then $d = \mathit{fix}(f)$ is a* fixed point *of f, i.e., $f(d) = d$.*

Proof. We first check that the sequence $(f^i(\perp_D))^{i\in\omega}$, is actually a chain in D, i.e., that for all i, $f^i(\perp_D) \sqsubseteq f^{i+1}(\perp_D)$. This follows by a simple mathematical induction on i: the base case $i = 0$ is immediate because $f^0(\perp_D) = \perp_D \sqsubseteq f(\perp_D) = f^1(\perp_D)$, simply by $\perp_D$ being the least element of D. And for the case $i = i' + 1$, where we can inductively assume $f^{i'}(\perp_D) \sqsubseteq f^{i'+1}(\perp_D)$, we exploit monotonicity of f to get $f^i(\perp_D) = f^{i'+1}(\perp_D) = f(f^{i'}(\perp_D)) \sqsubseteq f(f^{i'+1}(\perp_D)) = f(f^i(\perp_D)) = f^{i+1}(\perp_D)$, as required.

Now, since D is a cpo, this chain has a supremum, so let us take $d = \bigsqcup_{i\in\omega} f^i(\bot)$. We must then establish that $f(d) = d$. By continuity of f, we have $f(d) = f(\bigsqcup_{i\in\omega} f^i(\bot_D)) = \bigsqcup_{i\in\omega} f(f^i(\bot_D)) = \bigsqcup_{i\in\omega} f^{i+1}(\bot_D) =^{\dagger} \bigsqcup_{i\in\omega} f^i(\bot_D) = d$, where the equation marked † holds because the chains $f^1(\bot_D) \sqsubseteq f^2(\bot_D) \sqsubseteq \cdots$ and $f^0(\bot_D) \sqsubseteq f^1(\bot_D) \sqsubseteq f^2(\bot_D) \sqsubseteq \cdots$ clearly have the same supremum. ∎

One can also show that $\mathit{fix} : [D \to D] \to D$ is itself a *continuous* function, i.e., that the fixpoint of the supremum of a chain of functions is equal to the supremum of the fixpoints of the individual functions in the chain.

It is instructive to analyze the structure of the fixpoint found by Theorem 7.5. Consider the following definition of $\varphi \in [\mathbb{N} \to \mathbb{N}_\bot]$

$$\varphi = \mathit{fix}(\underbrace{\lambda\phi.\, \lambda n.\, \mathit{cond}(\mathit{eq}\,(n, 0), \eta\, 1, \phi\,(n-1) \star \lambda r.\, \eta\,(n \times r))}_{\Phi:[\mathbb{N}\to\mathbb{N}_\bot]\to[\mathbb{N}\to\mathbb{N}_\bot]})$$

Then we have (using the equations at the end of Section 7.1.2):

$$\begin{array}{rcl}
\varphi_0 & = & \Phi^0(\bot_{[\mathbb{N}\to\mathbb{N}_\bot]}) = \lambda n.\, \bot \\
\varphi_1 & = & \Phi^1(\bot_{[\mathbb{N}\to\mathbb{N}_\bot]}) = \Phi(\varphi_0) = \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \varphi_0\,(n-1) \star \lambda r.\, \eta\,(n \times r)) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \bot \star \lambda r.\, \eta\,(n \times r)) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \bot) \\
\varphi_2 & = & \Phi^2(\bot_{[\mathbb{N}\to\mathbb{N}_\bot]}) = \Phi(\varphi_1) = \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \varphi_1\,(n-1) \star \lambda r.\, \eta(n \times r)) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \mathit{cond}(\mathit{eq}(n-1,0), \eta\, 1, \bot) \star \lambda r.\, \eta(n \times r)) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \mathit{cond}(\mathit{eq}(n,1), \eta\, 1 \star \lambda r.\, \eta(n \times r), \bot \star \lambda r.\, \eta(n \times r))) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \mathit{cond}(\mathit{eq}(n,1), \eta\,(1 \times 1), \bot)) \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta 1, \mathit{cond}(\mathit{eq}(n,1), \eta\, 1, \bot)) \\
\varphi_3 & = & \Phi^3(\bot_{[\mathbb{N}\to\mathbb{N}_\bot]}) = \Phi(\varphi_2) = \cdots \\
& = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 1, \mathit{cond}(\mathit{eq}(n,1), \eta\, 1, \mathit{cond}(\mathit{eq}(n,2), \eta\, 2, \bot))) \\
& \vdots & \\
\varphi_i & = & \lambda n.\, \mathit{cond}(\mathit{eq}(n,0), \eta\, 0!, \mathit{cond}(\mathit{eq}(n,1), \eta\, 1!, \ldots, \mathit{cond}(\mathit{eq}(n,i-1), \eta\,(i-1)!, \bot))) \\
& = & \lambda n.\, \mathit{cond}(\mathit{leq}(i,n), \bot, \eta\, n!)
\end{array}$$

That is, the functions $\varphi_0, \varphi_1, \varphi_2, \ldots$ form better and better (i.e., more widely defined) approximations of the factorial function. And the least upper bound of this chain, $\varphi = \bigsqcup_{i\in\omega} \varphi_i = \lambda n.\, \eta\,(n!)$, is the actual factorial function, because for any n, there exists an i such that $\mathit{leq}(i,n) = \mathbf{false}$ (i.e. $i > n$), and therefore φ_i (and all subsequent φ_j for $j \geq i$) give the correct result for input n.

Theorem 7.5 tells us that we can always solve equations of the form $\varphi = \Phi(\varphi)$ (as long as Φ is continuous and the codomain of φ is a pointed cpo), as implicitly assumed in the preliminary semantics of **while**-commands. However, merely finding *some* solution is not always enough. Consider $\mathcal{C}[\![\mathbf{while\ true\ do\ skip}]\!] = \varphi$, where φ must satisfy the equation marked with '?':

$$\begin{aligned}
\varphi &\overset{?}{=} \lambda\sigma.\mathit{cond}(\mathcal{B}[\![\mathbf{true}]\!]\sigma, \mathcal{C}[\![\mathbf{skip}]\!]\sigma \star \varphi, \eta\,\sigma) \\
&= \lambda\sigma.\mathit{cond}(\mathbf{true}, \eta\,\sigma \star \varphi, \eta\,\sigma) \\
&= \lambda\sigma.\lfloor\sigma\rfloor \star \varphi \\
&= \lambda\sigma.\varphi\,\sigma \\
&= \varphi
\end{aligned}$$

That is, the equation does not constrain φ at all in this case, while we clearly want φ to be the constant-$\bot$ function, representing divergence from every state. In the following, we will show that $\mathit{fix}(f)$ is actually the *least* of all fixed points of f (when there are more than one), and give a general technique, *fixpoint induction* for proving additional properties of $\mathit{fix}(f)$.

Mirroring the notion of continuous functions between cpos, we first define a notion of subsets of cpos closed under suprema: a subset $P \subseteq A$ of a cpo A is said to be *admissible* if, for all chains $(a_i)^{i\in\omega}$ such that $\forall i \in \omega.\, a_i \in P$, we also have $\bigsqcup_{i\in\omega} a_i \in P$.

Unlike for continuity, where most "sensible" functions one could define are in fact continuous, admissible subsets are a little harder to come by. The following are some important general ways of constructing admissible subsets:

- Any subset of a discrete cpo is admissible. More generally, any subset of a cpo of *finite height* (i.e., a cpo that does not have any infinitely *strictly* increasing chains) is admissible.
- The subset $\{(a, a') \mid a \sqsubseteq_A a'\}$ of $A \times A$ is admissible.
- If $Q \subseteq B$ is admissible and $f : A \to B$ is a continuous function, then the *inverse image* of Q by f, i.e., $P = f^{-1}(Q) = \{a \in A \mid f(a) \in Q\}$ is admissible.
- If for all $x \in X$ (where X is any set, finite or infinite), $P_x \subseteq A$ is admissible, then so is their intersection $P = \bigcap_{x\in X} P_x = \{a \in A \mid \forall x \in X.\, a \in P_x\}$.

We can now state the following general proof principle:

Lemma 7.6 (Fixpoint induction) *Let D be a pointed cpo, $f : D \to D$ a continuous function, and let $P \subseteq D$ be an admissible subset. If $\bot_D \in P$, and $\forall d \in D.\, d \in P \Rightarrow f(d) \in P$, then $\mathit{fix}(f) \in P$.*

Proof. The result follows almost directly from the definitions. We first see that, for all $i \in \omega$, $f^i(\bot_D) \in P$; this is by a simple mathematical induction on i, using the assumptions on P and f. Since P was required to be admissible, this means that also $\bigsqcup_{i\in\omega} f^i(\bot_D) \in P$, but by definition of *fix*, this says precisely that $\mathit{fix}(f) \in P$. ■

In particular, we can use fixpoint induction to show that, if f happens to have multiple fixed points, then $\mathit{fix}(f)$ is the *least* of them. In fact, it is the least among the even larger set of *pre-fixed* points of f:

Lemma 7.7 *Let $f : D \to D$ be a continuous function on a pointed cpo D. If for some $d' \in D$, $f(d') \sqsubseteq d'$ (including, in particular, if $f(d') = d'$), then $\mathit{fix}(f) \sqsubseteq d'$.*

Proof. Let $P = \{d \in D \mid \forall d'. f(d') \sqsubseteq d' \Rightarrow d \sqsubseteq d'\}$, i.e., the set of all elements in D below all pre-fixed points of f. Then P is admissible, because it can be expressed as the intersection of all $P_x = \{d \mid d \sqsubseteq x\}$ for $x \in X = \{d' \in D \mid f(d') \sqsubseteq d'\}$; and each P_x is admissible because it is the inverse image by the continuous function $f_x(d) = (d, x)$ of the admissible subset $\{(d, d') \in D \times D \mid d \sqsubseteq d'\}$.

Clearly $\perp_D \in P$, because $\perp_D \sqsubseteq d'$ for any d', even ones that are not pre-fixed points of f. Next, assuming that $d \in P$, we will show $f(d) \in P$. That is, given any d' such that $f(d') \sqsubseteq d'$, we must show $f(d) \sqsubseteq d'$. By the assumption $d \in P$, we know that $d \sqsubseteq d'$, so by monotonicity also $f(d) \sqsubseteq f(d') \sqsubseteq d'$, as required. And thus, by Lemma 7.6, we have $\mathit{fix}(f) \in P$, i.e., $\forall d'. f(d') \sqsubseteq d' \Rightarrow \mathit{fix}(f) \sqsubseteq d'$. ■

Note, in particular, that any $d \in D$ is a fixed point of id_D. As expected, $\mathit{fix}(id_D)$ gives us precisely the least one, namely $\perp_D$. More generally, the least fixed point of any *strict* function $f : D \to D$ is evidently just $\perp_D$.

7.2.2 Semantics of commands (final formulation)

As already mentioned, the previous definition of $\mathcal{C}[\![-]\!]$ still works, when we consider all the sets to be cpos. However, we can now also give a proper, non-recursive formulation of the last equation of the semantics:

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!] \quad = \quad \mathit{fix}(\underbrace{\lambda\phi.\, \lambda\sigma.\, \mathit{cond}\,(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma \star \phi, \eta\,\sigma)}_{\Phi:[\Sigma\to\Sigma_\perp]\to[\Sigma\to\Sigma_\perp]})$$

Note that $[\Sigma \to \Sigma_\perp]$ is a pointed cpo, and the function Φ is continuous (because it's expressed entirely in terms of continuous base functions and continuity-preserving constructs), so $\mathit{fix}(\Phi)$ is defined by Theorem 7.5. Also, note that if we let $\varphi = \mathit{fix}(\Phi)$, then by the fixed-point property of φ,

$$\varphi = \Phi(\varphi) = \lambda\sigma.\, \mathit{cond}(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_0]\!]\sigma \star \varphi, \eta\,\sigma)$$

Thus, the φ postulated in the preliminary definition of $\mathcal{C}[\![-]\!]$ (and the proof of Lemma 7.3) does in fact exist.

It remains to show Lemma 7.4 for the case of **while**-loops. Recall that we are showing $\mathcal{C}[\![c]\!]\sigma = \lfloor\sigma'\rfloor \Rightarrow \langle c, \sigma\rangle \downarrow \sigma'$, by induction on the structure of c.

- Case $c = \mathbf{while}\ b\ \mathbf{do}\ c_0$. We will show this case by an inner fixpoint induction. Let $P = \{\phi \in [\Sigma \to \Sigma_\perp] \mid \forall\sigma''.\, \phi\,\sigma'' = \lfloor\sigma'\rfloor \Rightarrow \langle c, \sigma''\rangle \downarrow \sigma'\}$; we want to show that $\mathcal{C}[\![c]\!] = \mathit{fix}(\Phi) \in P$, which will directly give us what we need, by taking $\sigma'' = \sigma$.

  First, we must check that P is indeed admissible. P can be expressed as an intersection, namely $P = \bigcap_{\sigma''\in\Sigma} Q_{\sigma''}$, where $Q_{\sigma''} = \{\phi \mid \phi\,\sigma'' = \lfloor\sigma'\rfloor \Rightarrow \langle c, \sigma''\rangle \downarrow \sigma'\}$, so it suffices to show that for every σ'', $Q_{\sigma''}$ is admissible. Let σ'' be arbitrary, and let the continuous function $ap_{\sigma''} : [\Sigma \to \Sigma_\perp] \to \Sigma_\perp$ be given by $ap_{\sigma''} = \lambda\phi.\phi\,\sigma''$. Then $Q_{\sigma''}$ can be expressed as an inverse image, namely $Q_{\sigma''} = \{\phi \mid ap_{\sigma''}(\phi) \in R_{\sigma''}\}$, where $R_{\sigma''} = \{d \in \Sigma_\perp \mid d = \lfloor\sigma'\rfloor \Rightarrow \langle c, \sigma''\rangle \downarrow \sigma'\}$. So it remains to show that $R_{\sigma''}$ is an admissible subset of $\Sigma_\perp$. But $\Sigma_\perp$ is a cpo of finite height (because Σ is discretely ordered, so a chain in $\Sigma_\perp$ can at most strictly increase once), and therefore $R_{\sigma''}$ is automatically admissible, regardless of its definition.

For the base case of the fixpoint induction, we just need to show that $\perp_{[\Sigma\to\Sigma_\perp]} = \lambda\sigma.\perp \in P$, which is immediate, because $(\lambda\sigma.\perp)\,\sigma'' = \lfloor\sigma'\rfloor$ can never be true, so the implication in P vacuously holds for $\phi = \lambda\sigma.\perp$. For the inductive step, assume $\phi \in P$; we must show $\Phi(\phi) \in P$, i.e, that for all σ'',

$$\mathit{cond}(\mathcal{B}[\![b]\!]\sigma'', \mathcal{C}[\![c_0]\!]\sigma'' \star \phi, \eta\,\sigma'') = \lfloor\sigma'\rfloor \Rightarrow \langle c, \sigma''\rangle \downarrow \sigma' \,.$$

Assume the LHS of the implication holds. Then there are two subcases, according to the value of $\mathcal{B}[\![b]\!]\sigma''$. Let us start with the simpler one:

- Case $\mathcal{B}[\![b]\!]\sigma'' = \mathbf{false}$. Then Theorem 7.2($\Rightarrow$) gives us a derivation $\mathcal{E}_0$ of $\langle b, \sigma''\rangle \downarrow \mathbf{false}$. Also, the equation assumed above then says that $\lfloor\sigma'\rfloor = \mathit{cond}(\mathbf{false}, \mathcal{C}[\![c_0]\!]\sigma'' \star \phi, \eta\,\sigma'') = \eta\,\sigma'' = \lfloor\sigma''\rfloor$, i.e., $\sigma' = \sigma''$. We can then apply EC-WhileF on $\mathcal{E}_0$ to get the required derivation of $\langle c, \sigma''\rangle \downarrow \sigma'$.
- Case $\mathcal{B}[\![b]\!]\sigma'' = \mathbf{true}$. Again, by Theorem 7.2 we have a derivation $\mathcal{E}_0$ of $\langle b, \sigma''\rangle \downarrow \mathbf{true}$. The LHS equation tells us that $\mathcal{C}[\![c_0]\!]\sigma'' \star \phi = \lfloor\sigma'\rfloor$, which can only happen if (1) $\mathcal{C}[\![c_0]\!]\sigma'' = \lfloor\sigma'''\rfloor$ for some σ''', and (2) $\phi\,\sigma''' = \lfloor\sigma'\rfloor$. From (1), by the (outer) IH on c_0, we get a derivation $\mathcal{E}_1$ of $\langle c_0, \sigma''\rangle \downarrow \sigma'''$. And from (2) and the fixpoint IH, i.e., $\phi \in P$, we get an $\mathcal{E}_2$ of $\langle c, \sigma'''\rangle \downarrow \sigma'$. Finally, putting $\mathcal{E}_0$, $\mathcal{E}_1$, and $\mathcal{E}_2$ together with EC-WhileT gives us the required derivation of $\langle c, \sigma''\rangle \downarrow \sigma'$.

Taking Lemmas 7.3 and 7.4 together, we immediately get the main result:

Theorem 7.8 *For any command c, $\mathcal{C}[\![c]\!]\sigma = \lfloor\sigma'\rfloor \Leftrightarrow \langle c, \sigma\rangle \downarrow \sigma'$.*

And in particular, we again have $c \sim c' \Leftrightarrow \mathcal{C}[\![c]\!] = \mathcal{C}[\![c']\!]$. This means that we can substitute semantically equivalent commands for each other in all contexts. For example, we immediately get that, if $c_0 \sim c_0'$, then also **while** b **do** $(c_0; c_1) \sim$ **while** b **do** $(c_0'; c_1)$. This is because, in *any* compositional semantics, we must have

$$\begin{aligned}\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ (c_0; c_1)]\!] &= \mathit{ws}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c_0]\!], \mathcal{C}[\![c_1]\!]) \\ &= \mathit{ws}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c_0']\!], \mathcal{C}[\![c_1]\!]) \;=\; \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ (c_0'; c_1)]\!]\,,\end{aligned}$$

where *ws* is some mathematical function of the meanings of b, c_0, and c_1. Showing such a *congruence* property would require a separate induction on derivations in the original operational semantics. This is quite straightforward for IMP, but already for a very simple functional language like FUN, it's quite challenging both to properly define a good notion of semantic equivalence with respect to the operational semantics, and to show that two terms are indeed equivalent in this sense. On the other hand, in a denotational semantics, many program equivalences can be easily verified by simple equational reasoning about their denotations.

7.3 Exercises

7.1. Prove Theorem 7.2 (equivalence of operational and denotational semantics for boolean expressions).

7.2. Show the following equivalences in the denotational semantics *directly* (i.e., without using Theorem 7.8 or its two constituent lemmas). Remember that showing $\mathcal{C}[\![c]\!] = \mathcal{C}[\![c']\!]$ amounts to showing $\mathcal{C}[\![c]\!]\sigma = \mathcal{C}[\![c']\!]\sigma$ for all σ.

(a) $\mathcal{C}[\![c_0; (c_1; c_2)]\!] = \mathcal{C}[\![(c_0; c_1); c_2]\!]$.
Hint: consider the possible values that $\mathcal{C}[\![c_0]\!]\,\sigma$ can take.

(b) $\mathcal{C}[\![(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1); c_2]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } (c_0; c_2) \textbf{ else } (c_1; c_2)]\!]$.
Hint: consider the possible values that $\mathcal{B}[\![b]\!]\,\sigma$ can take.

(c) $\mathcal{C}[\![\textbf{while } b \textbf{ do } c_0]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } (c_0; \textbf{while } b \textbf{ do } c_0) \textbf{ else skip}]\!]$.
Hint: exploit that φ is a fixed point of Φ in the *final* formulation of the semantics for **while**-loops.

7.3. Extend the *final* formulation of $\mathcal{C}[\![-]\!]$ with a suitable equation for **repeat**–**until** loops (as previously seen), and extend the proofs of Lemmas 7.3 and 7.4 correspondingly. *Hint:* change only the definition of Φ from the semantics for **while**-loops.

7.4. Verify some of the unproven claims in the notes, such as:

(a) Show that if A is a cpo then so is $A_\perp$ (i.e., that if the ordering relation on A is reflexive, transitive, antisymmetric, and complete, then so is the ordering on $A_\perp$), and that $\eta : A \to A_\perp$ and $\lambda d.\, d \star f : A_\perp \to B_\perp$ are continuous, for any continuous $f : A \to B_\perp$.

(b) Show that the inverse image by a continuous function of an admissible subset of a cpo is also admissible, and that any intersection of admissible subsets is admissible.