Lecture Notes on Denotational Semantics

Jeremy G. Siek

November 5, 2017

Contents

1 Binary Arithmetic

Reading: Schmidt [1986] Chapter 4
Exercises: 4.2, 4.6

Borrowing and combining elements from Chapter 4 of Schmidt [1986], we consider the language of binary arithmetic specified in Figure 1. The Syntax defines the form of the programs and the Semantics specifies the behavior of running the program. In this case, the behavior is simply to output a number (in decimal). Our grammar for binary numerals departs slightly from Schmidt [1986], with ϵ representing the empty string of digits.

The main purpose of a semantics is communicate in a precise way with other people, primarily language implementers and programmers. Thus, it is incredibly important for the semantics to be written in a way that will make it most easily understood, while still being completely precise. Here we have chosen to give the semantics of binary arithmetic in terms of decimal numbers because people are generally much more familiar with decimal numbers.

When writing down a semantics, one is often tempted to consider the efficiency of a semantics, as if it were an implementation. Indeed, it would be straightforward to transcribe the definitions of E and N in Figure 1 into your favorite programming language and thereby obtain an interpreter. All other things being equal, it is fine to prefer a semantics that is suggestive of an implementation, but one should prioritize ease of understanding first. As a result, some semantics that we study may be more declarative in nature. This is not to say that one should not consider the efficiency of implementations when designing a language. Indeed, the semantics should be co-designed with implementations to avoid accidental designs that preclude the desired level of efficiency. Thus, a recurring theme of these notes will be to consider implementations of languages alongside their semantics.

Figure 2 presents an interpreter for binary arithmetic. This interpreter, in a way reminiscent of real computers, operates on the binary numbers directly. The auxiliary functions *add* and *mult* implement the algorithms you learned in grade school, but for binary instead of decimals.

Syntax

digit	$d ::=$	$0 \mid 1$
binary numeral	$n ::=$	$\epsilon \mid nd$
expression	$e ::=$	$n \mid e + e \mid e \times e$

Semantics

$$N(\epsilon) = 0$$
$$N(nd) = 2N(n) + d$$

$$E(n) = N(n)$$
$$E(e_1 + e_2) = E(e_1) + E(e_2)$$
$$E(e_1 \times e_2) = E(e_1)E(e_2)$$

Figure 1: Language of binary arithmetic

Exercise 1 Prove that $N(add3(d_1, d_2, d_3)) = d_1 + d_2 + d_3$.

Exercise 2 Prove that $N(add(n_1, n_2, c)) = N(n_1) + N(n_2) + c$.

Exercise 3 Prove that $N(mult(n_1, n_2)) = N(n_1)N(n_2)$.

Exercise 4 Prove that the interpreter is correct. That is

$$N(I(e)) = E(e)$$

Figure 2: Binary arithmetic interpreter.

Interpreter

$$
\begin{aligned}
I(n) &= n \\
I(e_1 + e_2) &= add(I(e_1), I(e_2), 0) \\
I(e_1 \times e_2) &= mult(I(e_1), I(e_2))
\end{aligned}
$$

Auxiliary Functions

$$
add3(d_1, d_2, d_3) = \begin{cases} 00 & \text{if } n = 0 \\ 01 & \text{if } n = 1 \\ 10 & \text{if } n = 2 \\ 11 & \text{if } n = 3 \end{cases} \quad \text{where } n = d_1 + d_2 + d_3
$$

$$
add(\epsilon, \epsilon, c) = \begin{cases} \epsilon & \text{if } c = 0 \\ \epsilon 1 & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
add(n_1 d_1, n_2 d_2, c) &= add(n_1, n_2, c')\, d_3 \qquad \text{if } add3(d_1, d_2, c) = c' d_3 \\
add(n_1 d_1, \epsilon, c) &= add(n_1 d_1, \epsilon 0, c) \\
add(\epsilon, n_2 d_2, c) &= add(\epsilon 0, n_2 d_2, c)
\end{aligned}
$$

$$
\begin{aligned}
mult(n_1, \epsilon) &= \epsilon \\
mult(n_1, n_2 0) &= mult(n_1, n_2) 0 \\
mult(n_1, n_2 1) &= add(n_1, mult(n_1, n_2) 0, 0)
\end{aligned}
$$

2 *An Imperative Language: IMP*

Reading: Schmidt [1986] Chapter 5
Exercises: 5.4, 5.5 a, 5.9

Figure 3 defines a language with mutable variables, a variant of the IMP [Plotkin, 1983, Winskel, 1993, Amadio and Pierre-Louis, 1998] and WHILE [Hoare, 1969] languages that often appear in textbooks on programming languages. As a bonus, we include the `while` loop, even though Schmidt [1986] does not cover loops until Chapter 6. The reason for his delayed treatment of `while` loops is that their semantics is typically expressed in terms of fixed points of continuous functions, which takes some time to explain. However, it turns out that the semantics of `while` loops can be defined more simply.

To give meaning to mutable variables, we use a *Store* which is partial function from variables (identifiers) to numbers.

$$Store = Id \rightharpoonup Nat$$

We write $[x \mapsto n]s$ for $\{x \mapsto n\} \cup (s|_{\mathrm{dom}(s)-\{x\}})$.

The syntax of expressions is extended to include variables, so the meaning of expressions must be parameterized on the store. The meaning of a variable x is the associated number in the store s.

$$E\,x\,s = s\,x$$

A program takes a number as input and it may produce a number or it might diverge. Traditional denotational semantics model this behavior with a partial function $Nat \rightharpoonup Nat$. Here we shall use the alternate, but equivalent, approach of using a relation $Nat \times Nat$.

We define the `while` loop using an auxiliary relation named *loop*, which we define inductively in Figure 3. Its two parameters are for the meaning of the condition b and the body c of the loop. If the meaning of the condition m_b is `false`, then the loop is already finished so the starting and finishing stores are the same. If the condition m_b is `true`, then the loop executes the body, relating the starting store s_1 to an intermediate store s_2, and then the loop continues, relating s_2 to the finishing store s_3.

We define an implementation of the imperative language, in terms of an abstract machine, in Figure 4. The machine executes one command at a time, similar to how a debugger such as `gdb` can be used to view the execution of a C program one statement at a time. Each command causes the machine to transition from one state to the next, where a state is represented by a control component and the store. The control component is the sequence of commands that need to be executed, which is convenient to represent as a command of the following form.

$$\text{control} \quad k \quad ::= \quad \texttt{skip} \mid c\ ;\ k$$

Syntax

$$
\begin{array}{ll}
\text{variables} & x \in \mathbb{X} \\
\text{expressions} & e ::= \ldots \mid x \\
\text{conditions} & b ::= \texttt{true} \mid \texttt{false} \mid e = e \mid \\
 & \quad \neg b \mid b \vee b \mid b \wedge b \\
\text{commands} & c ::= \texttt{skip} \mid x := e \mid c \,;\, c \mid \\
 & \quad \texttt{if}\, b\, \texttt{then}\, c\, \texttt{else}\, c \mid \\
 & \quad \texttt{while}\, b\, \texttt{do}\, c
\end{array}
$$

Loops

$$\frac{m_b\, s = \texttt{false}}{(s,s) \in loop(m_b, m_c)}$$

$$\frac{m_b\, s_1 = \texttt{true} \quad (s_1,s_2) \in m_c \quad (s_2,s_3) \in loop(m_b,m_c)}{(s_1,s_3) \in loop(m_b,m_c)}$$

Semantics

$$P\, c = \{(n, s'\, Z) \mid (\{A \mapsto n\}, s') \in C\, c\}$$

$$
\begin{aligned}
C\, \texttt{skip} &= id \\
C(x := e) &= \{(s, [x \mapsto E\, e\, s]s) \mid s \in Store\} \\
C(c_1 \,;\, c_2) &= C\, c_2 \circ C\, c_1 \\
C \begin{pmatrix} \texttt{if}\, b\, \texttt{then}\, c_1 \\ \texttt{else}\, c_2 \end{pmatrix} &= \begin{cases} C\, c_1 & \text{if } B\, b\, s = \texttt{true} \\ C\, c_2 & \text{if } B\, b\, s = \texttt{false} \end{cases} \\
C(\texttt{while}\, b\, \texttt{do}\, c) &= loop(B\, b, C\, c)
\end{aligned}
$$

Figure 3: An Imperative Language: IMP

The partial function *eval*, also defined in Figure 4 in the main entry point for the abstract machine.

$$\boxed{k, s \longrightarrow k', s'}$$

$$
\begin{aligned}
\texttt{skip} \,;\, k,\ s &\longrightarrow k,\ s \\
(x := e) \,;\, k,\ s &\longrightarrow k,\ [x \mapsto E(e)(s)]s \\
(c_1 \,;\, c_2) \,;\, k,\ s &\longrightarrow c_1 \,;\, (c_2 \,;\, k),\ s \\
(\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2) \,;\, k,\ s &\longrightarrow c_1 \,;\, k,\ s && \text{if } B\, b\, s = \texttt{true} \\
(\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2) \,;\, k,\ s &\longrightarrow c_2 \,;\, k,\ s && \text{if } B\, b\, s = \texttt{false} \\
(\texttt{while}\, b\, \texttt{do}\, c) \,;\, k,\ s &\longrightarrow c \,;\, ((\texttt{while}\, b\, \texttt{do}\, c) \,;\, k),\ s && \text{if } B\, b\, s = \texttt{true} \\
(\texttt{while}\, b\, \texttt{do}\, c) \,;\, k,\ s &\longrightarrow k,\ s && \text{if } B\, b\, s = \texttt{false}
\end{aligned}
$$

$$eval(c) = \{(n, n') \mid (c \,;\, \texttt{skip}), \{A \mapsto n\} \longrightarrow^* \texttt{skip}, s' \text{ and } n' = s'(Z)\}$$

Figure 4: Abstract Machine for IMP

Notation: given a relation R, we write $R(a)$ for the image of $\{a\}$ under R. For example, if $R = \{(0,4),(1,2),(1,3),(2,5)\}$, then $R(1) = \{2,3\}$ and $R(0) = \{4\}$.

Exercise 5 Prove that if $k, s \longrightarrow k', s'$, then $C\, k\, s = C\, k'\, s'$.

Exercise 6 Prove that $P(c) = eval(c)$.

3 *Recursive Definitions via Least Fixed Points*

Reading: Schmidt [1986] Chapter 6
Exercises: 6.2 a, 6.6, 8

We shall revisit the semantics of the imperative language, this time taking the traditional but more complex approach of defining `while` with a recursive equation and using least fixed points to solve it. Recall that the meaning of a command is a partial function from stores to stores, or more precisely,

$$C\,c : \mathit{Store} \rightharpoonup \mathit{Store}$$

The meaning of a loop $(\texttt{while}\,b\,\texttt{do}\,c)$ is a solution to the equation

$$w = \lambda s.\, \mathit{if}\ B\,b\,s\ \mathit{then}\ w(C\,c\,s)\ \mathit{else}\ s \tag{1}$$

In general, one can write down recursive equations that do not have solutions, so how do we know whether this one does? When is there a unique solution? The theory of least fixed points provides answers to these questions.

Definition 1. *A* **fixed point** *of a function is an element that gets mapped to itself, i.e.,* $x = f(x)$.

In this case, the element that we're interested in is w, which is itself a function, so our f will be higher-order function. We reformulate Equation 1 as a fixed point equation by abstracting away the recursion into a parameter r.

$$w = F_{b,c}(w) \qquad \text{where } F_{b,c}\,r\,s = \mathit{if}\ B\,b\,s\ \mathit{then}\ r(C\,c\,s)\ \mathit{else}\ s \tag{2}$$

There are quite a few theorems in the literature that guarantee the existence of fixed points, but with different assumptions about the function and its domain. For our purposes, the CPO Fixed-Point Theorem will do the job. The idea of this theorem is to construct an infinite sequence that provides increasingly better approximations of the least fixed point. The union of this sequence will turn out to be the least fixed point.

The CPO Fixed-Point Theorem is quite general; it is stated in terms of a function F over partially ordered sets with a few mild conditions. The ordering captures the notion of approximation, that is, we write $x \sqsubseteq y$ if x approximates y.

Definition 2. *A* **partially ordered set (poset)** *is a pair* $(L, \sqsubseteq)$ *that consists of a set* L *and a partial order* $\sqsubseteq$ *on* L.

For example, consider the poset $(\mathbb{N} \rightharpoonup \mathbb{N}, \subseteq)$ of partial functions on natural numbers. Some partial function f is a better approximation than another partial function g if it is defined on more inputs, that is, if the graph of f is a subset of the graph of g. Two partial functions are incomparable if neither is a subset of the other.

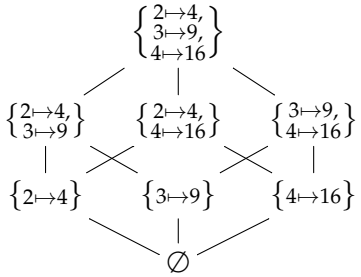

Figure 5: A poset of partial functions.

The sequence of approximations will start with the worst approximation, a bottom element, written $\bot$, that contains no information. (For example, $\emptyset$ is the $\bot$ for the poset of partial functions.) The sequence proceeds to by applying F over and over again, that is,

$$\bot \quad F(\bot) \quad F(F(\bot)) \quad F(F(F(\bot))) \quad \cdots \quad F^i(\bot) \quad \cdots$$

But how do we know that this sequence will produce increasingly better approximations? How do we know that

$$F^i(\bot) \sqsubseteq F^{i+1}(\bot)$$

We can ensure this by requiring the output of F to improve when the input improves, that is, require F to be monotonic.

Definition 3. *Given two partial orders $(A, \sqsubseteq)$ and $(B, \sqsubseteq)$, $F : A \to B$ is* **monotonic** *iff for any $x, y \in A$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(x)$.*

Proposition 1. *The functional $F_{b,c}$ for* `while` *loops* (2) *is monotonic.*

Proof. Let $f, g : \mathit{Store} \rightharpoonup \mathit{Store}$ such that $f \subseteq g$. We need to show that $F_{b,c}(f) \subseteq F_{b,c}(g)$. Let s be an arbitrary state. Suppose $B\,b\,s = \texttt{true}$.

$$F_{b,c}\,f\,s = f(C\,c\,s) \subseteq g(C\,c\,s) = g(C\,c\,s) = F_{b,c}\,g\,s$$

So we have $F_{b,c}(f) \subseteq F_{b,c}(g)$. Next suppose $B\,b\,s = \texttt{false}$.

$$F_{b,c}\,f\,s = s = F_{b,c}\,g\,s$$

So again we have $F_{b,c}(f) \subseteq F_{b,c}(g)$. Having completed both cases, we conclude that $F_{b,c}(f) \subseteq F_{b,c}(g)$. □

We have $\bot \sqsubseteq F(\bot)$ because $\bot$ is less or equal to everything. Then we apply monotonicity to obtain $F(\bot) \sqsubseteq F(F(\bot))$. Continuing in this way we obtain the sequence of increasingly better approximations in Figure 6. If at some point the approximation stops improving, but instead F produces an element that is merely equal to the last one, then we have found a fixed point. However, because we are interested in elements that are partial functions, which are infinite, the sequences of approximations will also be infinite. So we'll need some other way to go from the sequences of approximations to the actual fixed point.

$$\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq F^3(\bot) \sqsubseteq \cdots$$

Figure 6: Ascending chain of F.

The solution is to take the union of all the approximations. The analogue of union for an arbitrary partial order is least upper bound.

$$\bigsqcup \left\{ \begin{array}{l} \{2 \mapsto 4, 3 \mapsto 9\}, \\ \{3 \mapsto 9, 4 \mapsto 16\} \end{array} \right\} = \left\{ \begin{array}{l} 2 \mapsto 4, \\ 3 \mapsto 9, \\ 4 \mapsto 16 \end{array} \right\}$$

Figure 7: The lub of partial functions.

Definition 4. *Given a subset S of a partial order $(L, \sqsubseteq)$, an* **upper bound** *of S is an element y such that for all $x \in S$ we have $x \sqsubseteq y$. The* **least upper bound (lub)** *of S, written $\bigsqcup S$, is the least of all the upper bounds of S, that is, given any upper bound z of S, we have $\bigsqcup S \sqsubseteq z$.*

In arbitrary posets, a least upper bound may not exist for an arbitrary subset. In particular, for the poset $(\mathbb{N} \rightharpoonup \mathbb{N}, \subseteq)$, two partial functions do not have a lub if they are inconsistent, that is, if they map the same input to different outputs, such as $\{3 \mapsto 8\}$ and $\{3 \mapsto 9\}$. However, the CPO Fixed-Point Theorem will only need to consider totally ordered subsets, i.e., chains, and all the elements in a chain are consistent.

Definition 5. *A* **chain** *is a totally ordered subset of a poset. A* **chain-complete partial order (cpo)** *has a least upper bound for every chain.*

Proposition 2. *The poset of partial functions* $(A \rightharpoonup B, \subseteq)$ *is a cpo.*

Proof. Let S be a chain in $(A \rightharpoonup B, \subseteq)$. We claim that the lub of S is just the union of all the elements of S, that is $\bigcup S$. Recall that $\forall xy, (x, y) \in \bigcup S$ iff $\exists f \in S, (x, y) \in f$. We first need to show that $\bigcup S$ is an upper bound of S. Suppose $f \in S$. We need to show that $f \subseteq \bigcup S$. Consider $(x, y) \in f$. Then $(x, y) \in \bigcup S$. So indeed, $\bigcup S$ is an upper bound of S. Second, consider another upper bound g of S. We need to show that $\bigcup S \subseteq g$. Suppose $(x, y) \in \bigcup S$. Then $(x, y) \in h$ for some $h \in S$. Because g is an upper bounf of S, we have $h \subseteq g$ and therefore $(x, y) \in g$. So we conclude that $\bigcup S \subseteq g$. □

The last ingredient required in the proof of the fixed point theorem is that the output of F should only depend on a finite amount of information from the input, that is, it should be continuous. For example, if the input to F is itself a function g, F should only need to apply g to a finite number of different values. This requirement is at the core of what it means for a function to be computable [Gunter et al., 1990]. So applying F to the lub of a directed set X (an infinite thing) should be the same as taking the lub of the set obtained by mapping F over the elements of X (finite things).

Definition 6. *A monotonic function* $F : A \to B$ *on a cpo is* **continuous** *iff for all chains* X *of* A

$$F(\bigsqcup X) = \bigsqcup \{F(x) \mid x \in X\}$$

Proposition 3. *The functional* $F_{b,c}$ *for* `while` *loops* (2) *is continuous.*

Proof. Let S be a chain in $(A \rightharpoonup B, \subseteq)$. We need to show that

$$F_{b,c}(\bigcup S) = \bigcup \{F_{b,c}(f) \mid f \in S\}$$

We shall prove this equality by showing that each graph is a subset of the other. We assume $(s, s'') \in F_{b,c}(\bigcup S)$. Suppose $B\, b\, s = \texttt{true}$. Then $(s, s'') \in (\bigcup S) \circ (C\, c)$, so $(s, s') \in C\, c$ and $(s', s'') \in g$ for some s' and $g \in S$. So $(s, s'') \in g \circ (C\, c)$. Therefore $(s, s'') \in \bigcup \{F_{b,c}(f) \mid f \in S\}$.

So we have shown that $F_{b,c}(\bigcup S) \subseteq \bigcup\{F_{b,c}(f) \mid f \in S\}$. Next suppose $B\,b\,s = \texttt{false}$. Then $F_{b,c}(\bigcup S) = id = \bigcup\{F_{b,c}(f) \mid f \in S\}$.

For the other direction, we assume $(s, s'') \in \bigcup\{F_{b,c}(f) \mid f \in S\}$. So $(s, s'') \in F_{b,c}(g)$ for some $g \in S$. Suppose $B\,b\,s = \texttt{true}$. Then $(s, s') \in C\,c$ and $(s', s'') \in g$ for some s'. So $(s', s'') \in \bigcup S$ and therefore $(s, s'') \in F_{b,c}(\bigcup S)$. So we have shown that $\bigcup\{F_{b,c}(f) \mid f \in S\} \subseteq F_{b,c}(\bigcup S)$. Next suppose $B\,b\,s = \texttt{false}$. Then again we have both graphs equal to the identity relation. □

We now state the fixed point theorem for cpos.

Theorem 4 (CPO Fixed-Point Theorem)**.** *Suppose $(L, \sqsubseteq)$ is a cpo and let $F : L \to L$ be a continuous function. Then F has a least fixed point, written* fix F*, which is the least upper bound of the ascending chain of F:*

$$\text{fix}\,F = \bigsqcup\{F^n(\bot) \mid n \in Nat\}$$

Proof. Note that $\bot$ is an element of L because it is the lub of the empty chain. We first prove that fix F is a fixed point of F.

$$\begin{aligned}
F(\text{fix}\,F) &= F(\bigsqcup\{F^n(\bot) \mid n \in Nat\}) \\
&= \bigsqcup\{F(F^n(\bot)) \mid n \in Nat\} && \text{by continuity} \\
&= \bigsqcup\{F^{n+1}(\bot)) \mid n \in Nat\} \\
&= \bigsqcup\{F^n(\bot)) \mid n \in Nat\} && \text{because } F^0(\bot) = \bot \sqsubseteq F^1(\bot) \\
&= \text{fix}\,F
\end{aligned}$$

Next we prove that fix F is the least of the fixed points of F. Suppose e is an arbitrary fixed point of F. By the monotonicity of F we have $F^i(\bot) \sqsubseteq F^i(e)$ for all i. And because e is a fixed point, we also have $F^i(e) = e$, so e is an upper bound of the ascending chain, and therefore fix $F \sqsubseteq e$. □

Returning to the semantics of the `while` loop, we give the least fixed-point semantics of an imperative language in Figure 8. We have already seen that the poset of partitial functions is a cpo and that $F_{b,c}$ is continuous, so fix $(F_{b,c})$ is well defined and is the least fixed-point of $F_{b,c}$. Thus, we can define the meaning of the `while` loop as follows.

$$C(\texttt{while}\,b\,\texttt{do}\,c)\,s = \text{fix}(F_{b,c})\,s$$

$$P'\,c\,n = (C'\,c\,\{A \mapsto n]\})\,Z$$

$$\begin{aligned}
C'\,\texttt{skip}\,s &= s \\
C'(x := e)\,s &= [x \mapsto E\,e\,s]s \\
C'(c_1\,;\,c_2)\,s &= C'\,c_2(C'\,c_1\,s) \\
C'\begin{pmatrix}\texttt{if}\,b\,\texttt{then}\,c_1\\ \texttt{else}\,c_2\end{pmatrix} s &= \begin{cases} C'\,c_1\,s & \text{if } B\,b\,s = \texttt{true} \\ C'\,c_2\,s & \text{if } B\,b\,s = \texttt{false} \end{cases} \\
C'(\texttt{while}\,b\,\texttt{do}\,c)\,s &= \text{fix}(F_{b,c})\,s
\end{aligned}$$

Figure 8: Least Fixed-Point Semantics of an Imperative Language

Exercise 7 Prove that the semantics in Figure 3 is equivalent to the least fixed-point semantics in Figure 8, i.e., $(n, n') \in P\,c$ iff $P\,c\,n = n'$.

In the literature there are two schools of thought regarding how to define complete partial orders. There is the one presented above, that requires lubs to exists for all chains [Plotkin, 1983, Schmidt,

1986, Winskel, 1993]. The other requires that the poset be directed complete, that is, lubs exists for all directed sets [Gunter et al., 1990, Mitchell, 1996, Amadio and Pierre-Louis, 1998]. The two schools of thought are equivalent, i.e., a poset P with a least element is directed-complete iff every chain in P has a lub [Davey and Priestley, 2002] (Theorem 8.11).

4 A Functional Language: the λ-calculus

Reading: Siek [2017]

We now turn to the prototypical functional language, the λ-calculus. The main feature of this language is a function, created by lambda abstractions $\lambda x.\, e$. The application expression $(e_1\, e_2)$ calls the function produced by e_1 with the argument produced by e_2. We study here the call-by-value (CBV) version of the λ-calculus. As a secondary feature, we include binary arithmetic (Section 1).

In the λ-calculus, functions are first-class entities, so they can be passed to other functions and returned from them. This introduces a difficulty in trying to give a semantics in terms of regular mathematical sets and functions. It seems that we need a set $\mathbb{D}$ that solves the following equation.

$$\mathbb{D} = \mathbb{N} + (\mathbb{D} \rightharpoonup \mathbb{D})$$

But such a set cannot exist because the size of $\mathbb{D} \rightharpoonup \mathbb{D}$ is necessarily larger than $\mathbb{D}$!

There are several ways around this problem. One approach is to consider only continuous functions, which cuts down the size enough to make it possible to solve the equation. The first model of the λ-calculus, D_∞ of Scott [1970], takes this approach. We shall study D_∞ in Section 7.

Another approach does not require solving the above equation, but recognizes that when passing a function to another function, one doesn't need to pass the entirety of the function (which is infinite), but one can instead pass a finite subset of the function's graph. The trouble of deciding which finite subset to pass can be sidestepped by trying all possible subsets. This would of course be prohibitive if our current goal was to implement the λ-calculus, but this "inefficiency" does not pose a problem for a semantics, a *specification*, of the λ-calculus. The various semantics that take this approach are called *graph models*. This first such model was T_C^* of Plotkin [1972]. Then came $\mathcal{P}(\omega)$ of Scott [1976] and the simpler D_A of Engeler [1981]. We shall study these three models in Section 5. But first, we study perhaps the most straightforward of the graph models, the recent one by yours truly [Siek, 2017].

Figure 9 defines the syntax and semantics of a CBV λ-calculus. To enable trying all possible finite subsets of a function's graph,

Syntax

$$\begin{array}{lll} \text{vars.} & & x \in \mathbb{X} \\ \text{expr.} & e \in \mathbb{E} ::= & n \mid e + e \mid e \times e \mid x \mid \lambda x.\, e \mid (e\, e) \end{array}$$

Domain

$$\begin{array}{lll} & & i \in \mathbb{N} \\ \text{elts.} & d \in \mathbb{D} ::= & i \mid \{(d_1, d_1'), \ldots, (d_n, d_n')\} \\ \text{env.} & \rho \in \mathbb{X} \rightharpoonup \mathbb{D} & \end{array}$$

Ordering

$$n \sqsubseteq n \qquad \frac{t \subseteq t'}{t \sqsubseteq t'}$$

Semantics

$$P\, e = \{d \mid \exists d'.\, d \sqsubseteq d' \text{ and } d' \in E\, e\, \emptyset\}$$

$$\begin{aligned} E\, n\, \rho &= \{N(n)\} \\ E\, (e_1 + e_2)\, \rho &= \{n_1 + n_2 \mid n_1 \in E\, e_1\, \rho, n_2 \in E\, e_2\, \rho\} \\ E\, (e_1 \times e_2)\, \rho &= \{n_1 n_2 \mid n_1 \in E\, e_1\, \rho, n_2 \in E\, e_2\, \rho\} \\ E\, x\, \rho &= \{d \mid d = \rho\, x\} \\ E\, (\lambda x.\, e)\, \rho &= \{t \mid \forall (d, d') \in t, d' \in E\, e\, [x \mapsto d]\rho\} \\ E\, (e_1\, e_2)\, \rho &= (E\, e_1\, \rho) \cdot (E\, e_2\, \rho) \end{aligned}$$

$$D_1 \cdot D_2 \stackrel{\text{def}}{=} \left\{ d' \;\middle|\; \begin{array}{l} \exists t d d_2.\, t \in D_1, d_2 \in D_2, \\ (d, d') \in t, d \sqsubseteq d_2 \end{array} \right\}$$

Figure 9: λ-calculus

the semantics is non-deterministic. That is, the semantic function E maps programs to sets of elements, instead of a single element. In the case when an expression produces a natural number, the set is just a singleton of that number. However, when the expression produces a function, the set contains all finite approximations of the function.

The domain $\mathcal{D}$ consists of just the natural numbers and finite graphs (association tables) of functions. $\mathcal{D}$ is defined inductively, or equivalently, it is the least solution of

$$\mathbb{D} = \mathbb{N} + \mathcal{P}_f(\mathbb{D} \times \mathbb{D})$$

Let t range over the finite graphs, that is, $t \in \mathcal{P}_f(\mathbb{D} \times \mathbb{D})$.

The semantics in Figure 9 says the meaning of an abstraction $(\lambda x.\, e)$ is the set of all tables t such that each input-output entry (d, d') makes sense for the λ. That is, $d' \in E\, e\, [x \mapsto d]\rho$. The semantics of application is defined in terms of the auxilliary application operator $D_1 \cdot D_2$, which applies all the tables in D_1 to all the elements (arguments) in D_2. A naive definition of the application operator is just table lookup in a non-deterministic setting.

$$D_1 \cdot D_2 = \{d' \mid \exists t d.\, t \in D_1, d \in D_2, (d, d') \in t\} \qquad \text{(naive)}$$

The idea is to collect up all the results d' that come from matching up an argument $d \in D_2$ with an entry (d, d') from a table $t \in D_1$. The problem with this naive version is that it prohibits self application, which is an important part of the λ-calculus. For example, it is necessary to define the Y combinator and thereby express general recursion. The problem is that in self application, one would need a table to be inside itself, that is, $(t, d') \in t$. But that can't happen because the domain $\mathbb{D}$ is inductively defined. The solution is to allow

the argument to be a larger table than the input d of the table entry. We define the approximation ordering $\sqsubseteq$ in Figure 9. So then, for argument $d_2 \in D_2$, we require $d \sqsubseteq d_2$ instead of $d = d_2$.

We turn to an implementation of the λ-calculus, the interpreter in Figure 10. The interpreter is based on the notion of a closure, which pairs a λ abstraction with its environment to ensure that the free variables get their definitions from the lexical scope.

We prove the correctness of the interpreter in two steps. First we show that if the semantics says the result of a program should be an integer i, then the interpreter produces a binary numeral n such that $N(n) = i$. The second part is to show that if the interpreter produces an answer n, then the semantics agrees that $N(n)$ should be the answer.

For the first part, we need to relate denotations (elements d) to the values v used by the interpreter.

$$\mathcal{G}(i) = \{n \mid N(n) = i\}$$

$$\mathcal{G}(t) = \left\{ \langle \lambda x.\, e, \varrho \rangle \;\middle|\; \begin{array}{l} \forall (d, d') \in t, v \in \mathcal{G}(d), \\ \exists v'. I\, e\, [x \mapsto v]\varrho = v' \text{ and } v' \in \mathcal{G}(d') \end{array} \right\}$$

Similarly, we related semantic environments to the interpreter's environments.

$$\frac{}{\mathcal{G}(\emptyset, \emptyset)} \qquad \frac{v \in \mathcal{G}(d) \quad \mathcal{G}(\rho, \varrho)}{\mathcal{G}([x \mapsto d]\rho, [x \mapsto v]\varrho}$$

Leading up to the first theorem, we establish the following lemmas.

Lemma 5 ($\mathcal{G}$ is downward closed)**.** *If $v \in \mathcal{G}(d)$ and $d' \sqsubseteq d$, then $v \in \mathcal{G}(d')$.*

Lemma 6. *If $\mathcal{G}(\rho, \varrho)$, then $\varrho(x) \in \mathcal{G}(\rho(x))$*

Lemma 7. *If $d \in E\, e\, \rho$ and $\mathcal{G}(\rho, \varrho)$, then $I\, e\, \varrho = v$, $v \in \mathcal{G}(d)$ for some v.*

Theorem 8 (Adequacy)**.** *If $E\, e\, \emptyset = E\, i\, \emptyset$, then $I\, e\, \emptyset = n$ and $N(n) = i$.*

UNDER CONSTRUCTION

values $\quad v \in \mathbb{V} ::= \quad n \mid \langle \lambda x.\, e, \rho \rangle$
env. $\quad \varrho \in \mathbb{X} \rightharpoonup \mathbb{V}$

$$\begin{aligned} I\, n\, \varrho &= n \\ I\, (e_1 + e_2)\, \varrho &= add(I\, e_1\, \varrho, I\, e_2\, \varrho, 0) \\ I\, (e_1 \times e_2)\, \varrho &= mult(I\, e_1\, \varrho, I\, e_2\, \varrho) \\ I\, (\lambda x.\, e)\, \varrho &= \langle \lambda x.\, e, \varrho \rangle \\ I\, (e_1\, e_2)\, \varrho &= I\, e\, [x \mapsto I\, e_2\, \varrho]\varrho' \\ &\qquad \text{if } I\, e_1\, \varrho = \langle \lambda x.\, e, \varrho' \rangle \end{aligned}$$

Figure 10: Interpreter for the λ-calculus

5 *Graph models of λ-calculus $(T^*_C, D_A, \mathcal{P}(\omega))$*

UNDER CONSTRUCTION

6 *Filter models of λ-calculus*

UNDER CONSTRUCTION

7 *D_∞ model of λ-calculus*

UNDER CONSTRUCTION

Answers to Exercises

Answer of Exercise 1

d_1, d_2, d_3	$add3(d_1, d_2, d_3)$	$d_1 + d_2 + d_3$
0, 0, 0	00	0
0, 0, 1	01	1
0, 1, 0	01	1
0, 1, 1	10	2
1, 0, 0	01	1
1, 0, 1	10	2
1, 1, 0	10	2
1, 1, 1	11	3

Answer of Exercise 2

The proof is by induction on add.

- Case $N(add(\epsilon, \epsilon, 0)) = N(\epsilon) = 0 = N(\epsilon) + N(\epsilon) + 0$
- Case $N(add(\epsilon, \epsilon, 1)) = N(\epsilon 1) = 1 = N(\epsilon) + N(\epsilon) + 1$
- Case $add(n_1' d_1, n_2' d_2, c) = add(n_1, n_2, c') d_3$
  where $add3(d_1, d_2, c) = c' d_3$.

$$\begin{aligned} N(add(n_1' d_1, n_2' d_2, c)) &= N(add(n_1', n_2', c') d_3) \\ &= N(add(n_1', n_2', c')) \times 2 + d_3 \\ &= (N(n_1') + N(n_2') + c') \times 2 + d_3 \\ &= 2N(n_1') + 2N(n_2') + c' \times 2 + d_3 \\ &= 2N(n_1') + 2N(n_2') + N(c' d_3) \\ &= 2N(n_1') + d_1 + 2N(n_2') + d_2 + c \quad \text{by Ex. 1} \\ &= N(n_1' d_1) + N(n_2' d_2) + c \end{aligned}$$

- Case $add(n_1 d_1, \epsilon, c) = add(n_1 d_1, \epsilon 0, c)$

$$\begin{aligned} N(add(n_1 d_1, \epsilon, c)) &= Nadd(n_1 d_1, \epsilon 0, c) \\ &= N(n_1 d_1) + N(\epsilon 0) + c \qquad \text{by I.H.} \\ &= N(n_1 d_1) + N(\epsilon) + c \end{aligned}$$

- Case $add(\epsilon, n_2 d_2, c) = add(\epsilon 0, n_2 d_2, c)$

$$\begin{aligned} N(add(\epsilon, n_2 d_2, c)) &= N(add(\epsilon 0, n_2 d_2, c)) \\ &= N(\epsilon 0) + N(n_2 d_2) + c \\ &= N(\epsilon) + N(n_2 d_2) + c \end{aligned}$$

Answer of Exercise 3

By induction on *mult*.

- Case $mult(n_1, \epsilon) = \epsilon$:

$$N(mult(n_1, \epsilon)) = N(\epsilon) = 0 = N(n_1)N(\epsilon)$$

- Case $mult(n_1, n_2'0) = mult(n_1, n_2')0$:

$$\begin{aligned} N(mult(n_1, n_2'0)) &= N(mult(n_1, n_2')0) \\ &= 2N(mult(n_1, n_2')) \\ &= 2N(n_1)N(n_2') && \text{by I.H.} \\ &= N(n_1)N(n_2'0) \end{aligned}$$

- Case $mult(n_1, n_2'1) = add(n_1, mult(n_1, n_2')0)$:

$$\begin{aligned} N(mult(n_1, n_2'1)) &= N(add(n_1, mult(n_1, n_2')0, 0)) \\ &= N(n_1) + N(mult(n_1, n_2')0) + 0 && \text{by Ex. 2} \\ &= N(n_1) + 2N(mult(n_1, n_2')) \\ &= N(n_1) + 2N(n_1)N(n_2') && \text{by I.H.} \\ &= N(n_1)(2N(n_2') + 1) \\ &= N(n_1)N(n_2'1) \end{aligned}$$

Answer of Exercise 4

The proof is by induction on e.

- Case $e = n$: $N(I(n)) = N(n) = E(n)$.
- Case $e = e_1 + e_2$:

$$\begin{aligned} N(I(e_1 + e_2)) &= N(add(I(e_1), I(e_2), 0)) \\ &= N(I(e_1)) + N(I(e_2)) && \text{by Ex. 2} \\ &= E(e_1) + E(e_2) && \text{by the I.H.} \\ &= E(e_1 + e_2) \end{aligned}$$

- Case $e = e_1 \times e_2$:

$$\begin{aligned} N(I(e_1 + e_2)) &= N(mult(I(e_1), I(e_2))) \\ &= N(I(e_1)) \times N(I(e_2)) && \text{by Ex. 3} \\ &= E(e_1) \times E(e_2) && \text{by the I.H.} \\ &= E(e_1 \times e_2) \end{aligned}$$

Answer of Exercise 5

The proof is by cases on $k, s \longrightarrow k', s'$.

- Case $\boxed{\texttt{skip} \,;\, k,\ s \longrightarrow k,\ s}$

$$C(\texttt{skip} \,;\, k)(s) = (C(k) \circ C(\texttt{skip}))(s) = (C(k) \circ id)(s) = C(k)(s)$$

- Case $\boxed{(x := e) \,;\, k,\ s \longrightarrow k,\ [x \mapsto E(e)(s)]s}$

$$\begin{aligned} C((x := e) \,;\, k)(s) &= (C(k) \circ C(x := e))(s) \\ &= C(k)(C(x := e)(s)) \\ &= C(k)([x \mapsto E(e)(s)]s) \end{aligned}$$

- Case $\boxed{(c_1 \,;\, c_2) \,;\, k,\ s \longrightarrow c_1 \,;\, (c_2 \,;\, k),\ s}$

$$\begin{aligned} C((c_1 \,;\, c_2) \,;\, k)(s) &= (C(k) \circ (C(c_2) \circ C(c_1)))(s) \\ &= ((C(k) \circ C(c_2)) \circ C(c_1))(s) \\ &= C(c_1 \,;\, (c_2 \,;\, k))(s) \end{aligned}$$

- Case $\boxed{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \,;\, k,\ s \longrightarrow c_1 \,;\, k,\ s \text{ and } B\,b\,s = \texttt{true}}$

$$\begin{aligned} C((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \,;\, k)(s) &= (C(k) \circ C(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2))(s) \\ &= (C(k) \circ C(c_1))(s) \\ &= C(c_1 \,;\, k)(s) \end{aligned}$$

- Case $\boxed{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \,;\, k,\ s \longrightarrow c_2 \,;\, k,\ s \text{ and } B\,b\,s = \texttt{false}}$

$$\begin{aligned} C((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \,;\, k)(s) &= (C(k) \circ C(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2))(s) \\ &= (C(k) \circ C(c_2))(s) \\ &= C(c_2 \,;\, k)(s) \end{aligned}$$

- Case $\boxed{(\texttt{while}\ b\ \texttt{do}\ c) \,;\, k,\ s \longrightarrow c \,;\, ((\texttt{while}\ b\ \texttt{do}\ c) \,;\, k),\ s \text{ and } B\,b\,s = \texttt{true}}$

$$\begin{aligned} C((\texttt{while}\ b\ \texttt{do}\ c) \,;\, k)(s) &= (C(k) \circ C(\texttt{while}\ b\ \texttt{do}\ c))(s) \\ &= (C(k) \circ (C(\texttt{while}\ b\ \texttt{do}\ c) \circ C(c)))(s) \\ &= ((C(k) \circ C(\texttt{while}\ b\ \texttt{do}\ c)) \circ C(c))(s) \\ &= C(c \,;\, ((\texttt{while}\ b\ \texttt{do}\ c) \,;\, k))(s) \end{aligned}$$

- Case $\boxed{(\texttt{while}\ b\ \texttt{do}\ c) \,;\, k,\ s \longrightarrow k,\ s \text{ and } B\,b\,s = \texttt{false}}$

$$\begin{aligned} C((\texttt{while}\ b\ \texttt{do}\ c) \,;\, k)(s) &= (C(k) \circ C(\texttt{while}\ b\ \texttt{do}\ c))(s) \\ &= (C(k) \circ id)(s) \\ &= C(k)(s) \end{aligned}$$

Answer of Exercise 6

TODO

Answer of Exercise 7

TODO

References

Roberto M. Amadio and Curien Pierre-Louis. *Domains and Lambda-Calculi*. Cambridge University Press, 1998.

B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

Erwin Engeler. Algebras and combinators. *algebra universalis*, 13(1): 389–392, Dec 1981. ISSN 1420-8911. DOI: 10.1007/BF02483849. URL https://doi.org/10.1007/BF02483849.

Carl A. Gunter, Peter D. Mosses, and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. 1990.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

John C. Mitchell. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-13321-0.

Gordon D. Plotkin. A set-theoretical definition of application. Technical Report MIP-R-95, University of Edinburgh, 1972.

Gordon D. Plotkin. Domains. course notes, 1983.

David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986. ISBN 0-697-06849-2.

Dana Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford University, November 1970.

Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3): 522–587, 1976.

Jeremy G. Siek. Revisiting elementary denotational semantics. *CoRR*, abs/1707.03762(4), 2017. URL http://arxiv.org/abs/1707.03762.

Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993.