Lecture Notes on Denotational Semantics

Jeremy G. Siek

January 19, 2018

Contents

Figure 1: Language of binary arithmetic

Syntax

$$
\begin{array}{lrl}
\text{digit} & d ::= & 0 \mid 1 \\
\text{binary numeral} & b^n \in Bin^n ::= & \epsilon^n \mid (b^n d)^{n+1} \\
\text{expression} & e \in \mathbb{E} ::= & b^{64} \mid e + e \mid e \times e
\end{array}
$$

Semantics

$$
\begin{array}{rl}
 & E : \mathbb{E} \rightharpoonup \mathbb{N}_{64} \\
 & E(b) = N(b) \\
N : Bin^{64} \to \mathbb{N}_{64} & E(e_1 + e_2) = \begin{cases} n_1 + n_2 & \text{if } n_1 + n_2 < 2^{64} \\ \text{undefined} & \text{otherwise} \end{cases} \\
N(\epsilon) = 0 & \text{where } n_1 = E(e_1), n_2 = E(e_2) \\
N(bd) = 2N(b) + d & E(e_1 \times e_2) = \begin{cases} n_1 n_2 & \text{if } n_1 n_2 < 2^{64} \\ \text{undefined} & \text{otherwise} \end{cases} \\
 & \text{where } n_1 = E(e_1), n_2 = E(e_2)
\end{array}
$$

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). In our grammar for binary numerals, ϵ represents the empty string of digits and bd is a number whose least-significant digit is d and whose more-significant digits are given by b. (This is the reverse of Schmidt [1986] but the same direction as Stoy [1977]). We place superscripts on our binary numerals to enfore a bound on their maximum length, restricting them 64 bits. We ommit the superscripts when their length is not of interest.

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. We write $\mathbb{N}_{64}$ for the set of numbers from 0 to $2^{64} - 1$.

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.

Exercise 1 Prove that $N(bin^2(n)) = n$ for any $n < 4$.

Exercise 2 Prove that $N(add(b_1, b_2, c)) = N(b_1) + N(b_2) + c$.

Exercise 3 Prove that $N(mult(b_1, b_2)) = N(b_1)N(b_2)$.

Exercise 4 Prove that the interpreter is correct. That is

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

2 *Loops and Mutable Variables*

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.

2.1 *Semantics of IMP*

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

$$Store = \mathbb{X} \rightharpoonup \mathbb{N}_{64}$$

Figure 2: Binary arithmetic interpreter.

Interpreter

$$
\begin{aligned}
I : \mathbb{E} &\rightharpoonup Bin^{64} \\
I(b) &= b \\
I(e_1 + e_2) &= add(I(e_1), I(e_2), 0) \\
I(e_1 \times e_2) &= mult(I(e_1), I(e_2))
\end{aligned}
$$

Auxiliary Functions

$$
bin^2(n) = \begin{cases} 00 & \text{if } n = 0 \\ 01 & \text{if } n = 1 \\ 10 & \text{if } n = 2 \\ 11 & \text{if } n = 3 \end{cases}
$$

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

$$
\begin{aligned}
add(b_1 d_1, b_2 d_2, c) &= b_3\, d_3 \text{ if } bin^2(d_1 + d_2 + c) = c' d_3, \\
&\qquad add(b_1, b_2, c') = b_3^n, \text{ and } n < 64 \\
add(b_1 d_1, \epsilon, c) &= add(b_1 d_1, \epsilon 0, c) \\
add(\epsilon, b_2 d_2, c) &= add(\epsilon 0, b_2 d_2, c)
\end{aligned}
$$

$$
\begin{aligned}
mult(b_1, \epsilon) &= \epsilon \\
mult(b_1, b_2 0) &= b_3 0 \text{ if } mult(b_1, b_2) = b_3^n \text{ and } n < 64 \\
mult(b_1, b_2 1) &= add(b_1, b_3 0, 0) \\
&\qquad \text{if } mult(b_1, b_2) = b_3^n \text{ and } n < 64
\end{aligned}
$$

Syntax

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

$$\mathit{loop} : (\mathit{Store} \to \mathbb{B}) \to \mathcal{P}(\mathit{Store} \times \mathit{Store}) \to \mathcal{P}(\mathit{Store} \times \mathit{Store})$$

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

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

Semantics

$$P : \mathit{Cmd} \to \mathcal{P}(\mathbb{N}_{64} \times \mathbb{N}_{64})$$
$$P\, c = \{(n, s'\,\mathtt{Z}) \mid (\{\mathtt{A} \mapsto n\}, s') \in C\, c\}$$
$$E : \mathbb{E} \to \mathit{Store} \to \mathbb{N}_{64}$$
$$E\, x\, s = s\, x$$
$$\vdots$$
$$B : \mathit{Cnd} \to \mathit{Store} \to \mathbb{B}$$
$$B\,(e_1 = e_2)\, s = (E\, e_1\, s) = (E\, e_2\, s)$$
$$\vdots$$
$$C : \mathit{Cmd} \to \mathcal{P}(\mathit{Store} \times \mathit{Store})$$
$$C\,\mathtt{skip} = \mathit{id}$$
$$C(x := e) = \{(s, [x \mapsto E\, e\, s]s) \mid s \in \mathit{Store}\}$$
$$C(c_1\,;c_2) = C\, c_2 \circ C\, c_1$$
$$C\begin{pmatrix}\mathtt{if}\ b\ \mathtt{then}\ c_1 \\ \mathtt{else}\ c_2\end{pmatrix} = \left\{ (s,s') \;\middle|\; \begin{array}{l} \mathit{if}\ B\, b\, s\ \mathit{then}\ (s,s') \in C\, c_1 \\ \mathit{else}\ (s,s') \in C\, c_2 \end{array} \right\}$$
$$C(\mathtt{while}\ b\ \mathtt{do}\ c) = \mathit{loop}(B\, b, C\, c)$$

Figure 3: An Imperative Language: IMP

We write $[x \mapsto n]s$ for removing the entry for x is s (if there is one) and then adding the entry $x \mapsto n$, that is, $\{x \mapsto n\} \cup (s|_{\mathrm{dom}(s)-\{x\}})$.

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. 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 ($\mathbb{N}_{64} \rightharpoonup \mathbb{N}_{64}$). Here we shall use the alternate, but equivalent, approach of using a relation, a subset of $\mathbb{N}_{64} \times \mathbb{N}_{64}$. A program is a command and the meaning of commands is given by the semantic function C, which maps a command to a relation on stores, that is, to a subset of $\mathit{Store} \times \mathit{Store}$. The meaning of a program is given by the function P, which initializes the store with the input number in variable `A`. When the program completes, the output is obtained from variable `Z`.

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.

2.2 *Implementation of IMP*

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.

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

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

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

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

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

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 $P(c) = \mathit{eval}(c)$.

Exercise 7 Is the IMP language Turing-complete? Why or why not?

Is the semantic function C sound wrt. contextual equivalence? Is it complete? –Jeremy

2.3 *Recursive Definitions via Least Fixed Points*

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 [Lassez et al., 1982]. 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.

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 and F produces an element that is 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.

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

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 $\bigsqcup S \sqsubseteq z$.*

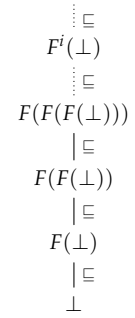

Figure 6: Ascending sequence of F.

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

Figure 7: The lub of partial functions.

In arbitrary posets, a least upper bound may not exist for a 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 sequences, and all the elements in such a sequence are consistent.

Definition 5. *A* **complete partial order (cpo)** *has a least upper bound for every sequence of totally ordered elements.*

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

Proof. Let S be a totally ordered sequence in $(A \rightharpoonup B, \subseteq)$. We claim that the lub of S is 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 sequence S should be the same as taking the lub of the set obtained by mapping F over each element of S.

Definition 6. *A monotonic function $F : A \to B$ on a cpo is* **continuous** *iff for all totally ordered sequences S of A*

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

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

Proof. Let S be a totally ordered sequence 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 sequence of F:*

$$\operatorname{fix} F = \bigsqcup\{F^n(\bot) \mid n \in \mathbb{N}\}$$

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

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

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.

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

Exercise 8 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 totally ordered sequences [Plotkin, 1983, Schmidt, 1986, Winskel, 1993]. The other requires that the poset be directed complete, that is, lubs exists for all directed sets [Gunter

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

$$P' : Cmd \to \mathbb{N} \rightharpoonup \mathbb{N}$$

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

$$C' : Cmd \to Store \rightharpoonup Store$$

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

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 totally ordered sequence in P has a lub [Davey and Priestley, 2002] (Theorem 8.11).

3 *Simply-Typed Functions*

Reading: Gunter [1992] Chapter 2, Chlipala [2007]

The appropriately-named simply-typed λ-calculus (STLC) provides a simple setting in which to study first-class functions. That is, functions that can be used like any other data. In the limited context of the STLC, this just means that they can be passed as parameters and returned from functions. Figure 9 defines the syntax and type system of the STLC. Whenever we discuss expression of the STLC, we always assume that they are well-typed, that is, that they satisfy the type system defined in Figure 9. The types include natural numbers and functions $A \to B$ where A is the type of the input (there is just one) and B is the type of the output.

The purpose of the variables in the STLC is for refering to the parameters of functions. The expression $\lambda x : A.\, e$ creates a function of one parameter named x of type A. The expression e is the body of the function and it may refer to parameter x. The value of e is the return value of the function. The expression $e_1\, e_2$ *applies* the function produced by e_1 to the value of e_2. The type system ensures that we never apply numbers (as if they were functions) or perform arith-

Figure 9: Syntax and types of the simply-typed λ-calculus

Syntax

$$
\begin{array}{lrl}
\text{types} & A, B ::= & \mathit{Nat} \mid A \to B \\
\text{numbers} & n \in & \mathbb{N} \\
\text{variables} & x \in & \mathbb{X} \\
\text{arithmetic} & \oplus ::= & + \mid \times \\
\text{expressions} & e ::= & n \mid e \oplus e \mid x \mid \lambda x\!:\!A.\, e \mid e\, e \\
\text{type env.} & \Gamma ::= & \emptyset \mid \Gamma, x\!:\!A
\end{array}
$$

Type System

$$
\frac{}{\Gamma \vdash n : \mathit{Nat}} \qquad \frac{\Gamma \vdash e_1 : \mathit{Nat} \quad \Gamma \vdash e_2 : \mathit{Nat}}{\Gamma \vdash e_1 \oplus e_2 : \mathit{Nat}}
$$

$$
\frac{\Gamma_n = x\!:\!A}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x\!:\!A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\, e : A \to B} \qquad \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}
$$

metic on functions (as if they were numbers). The type system also prevents the use of undefined variables.

The denotational semantics of the STLC [Henkin, 1950] is simple because STLC functions are total, they always terminate with an answer. So they can be straightforwardly modeled by mathematical (total) functions. But there is one twist, describing the semantics function's type is a bit interesting. The set of values that can be produced by an STLC expression *depends* on the type of the expression. For example, if an expression has type *Nat*, then its meaning will be in $\mathbb{N}$. If an expression has type $\mathit{Nat} \to \mathit{Nat}$, then its meaning will be in $\mathbb{N}^{\mathbb{N}}$ [1]. Thus, we define a mapping $[\![-]\!]$ from each type into a set.

[1] The set-theoretic notation B^A refers to the set of all total functions from set A to set B.

$$
\begin{aligned}
[\![\mathit{Nat}]\!] &= \mathbb{N} \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!]
\end{aligned}
$$

Likewise, we define a mapping from type environments to sets, in particular, to a cartesian product of the types of the variables.

$$
\begin{aligned}
[\![\emptyset]\!] &= \mathit{Unit} \\
[\![\Gamma, x\!:\!A]\!] &= [\![\Gamma]\!] \times [\![A]\!]
\end{aligned}
$$

With these mapping in hand, we can describe the semantic function E for the STLC. Because of the dependency described above, the type of E is interesting. It is a dependent function whose first parameter is some well-typed expression e, so we have $\Gamma \vdash e : A$. The second parameter is a type environment drawn from $[\![\Gamma]\!]$ and the result is in $[\![A]\!]$. The definition of the semantic function E is given in

Figure 10: Semantics of the simply-typed λ-calculus

$$E : \prod(e : \mathbb{E}), [\![\Gamma]\!] \to [\![A]\!] \qquad \text{where } \Gamma \vdash e : A$$

$$\begin{aligned}
E\, n\, \rho &= n \\
E\, (e_1 \oplus e_2)\, \rho &= n_1 \oplus n_2 \\
&\qquad \text{if } n_1 = E\, e_1\, \rho, n_2 = E\, e_2\, \rho \\
E\, x\, \rho &= \rho_n \quad \text{if } \Gamma_n = x : A \\
E\, (\lambda x : A.\, e)\, \rho &= \boldsymbol{\lambda} d.\, E\, e\, (\rho, d) \\
E\, (e_1\, e_2)\, \rho &= (E\, e_1\, \rho)\, (E\, e_2\, \rho)
\end{aligned}$$

Figure 11: Equational Theory of the simply-typed λ-calculus

$$(\text{refl})\frac{}{\vdash e = e} \qquad (\text{sym})\frac{\vdash e_2 = e_1}{\vdash e_1 = e_2} \qquad (\text{trans})\frac{\vdash e_1 = e_2 \quad \vdash e_2 = e_3}{\vdash e_1 = e_3}$$

$$(\text{cong})\frac{\vdash e_1 = e_3 \quad \vdash e_2 = e_4}{\vdash (e_1\, e_2) = (e_3\, e_4)} \qquad (\xi)\frac{\vdash e = e'}{\vdash \lambda x.\, e = \lambda x.\, e'}$$

$$(\beta)\frac{}{\vdash (\lambda x : A.\, e)\, e' = [x := e']e} \qquad (\eta)\frac{x \notin \mathrm{FV}(e)}{\vdash (\lambda x : A.\, e\, x) = e}$$

Figure 10. We interpret each λ as a total function and we interpret application as application. Regarding variables, the environment ρ is a tuple of values (not a mapping from variables to values) so environment lookup is performed by consulting the type environment Γ to determine the position n of a variable x, and then selecting the nth element from ρ.

3.1 *Equational Theory*

Another way to specify the meaning of a language is with an equational theory. Figure 11 gives the standard equational rules for the STLC, and the equational theory is the set of all equalities that can be deduced from these rules. Note that these rules are closely related to the reduction rules of the STLC; they basically specify a bi-directional version of those rules.

UNDER CONSTRUCTION

Theorem 5 (Soundness of Semantics wrt. Equations)**.**
If $\vdash e = e'$*, then* $\forall \rho, E\, e\, \rho = E\, e'\, \rho$*.*

Proof. The proof is by induction on the derivation of $\vdash e_1 = e_2$.

- (refl), (sym), (trans) are immediate.
- (cong) Let ρ be an arbitrary environment. By the induction hypothesis, we have $E\,e_1\,\rho = E\,e_3\,\rho$ and $E\,e_2\,\rho = E\,e_4\,\rho$. Therefore $(E\,e_1\,\rho)\,(E\,e_2\,\rho) = (E\,e_3\,\rho)\,(E\,e_4\,\rho)$.
- (ξ) Let ρ be an arbitrary environment. We need to show that $E\,(\lambda x.\,e)\,\rho = E\,(\lambda x.\,e')\,\rho$. Thus, it suffices to show for an arbitrary d that $E\,e\,\rho(x{\mapsto}d) = E\,e'\,\rho(x{\mapsto}d)$, but that follows from the induction hypothesis.
- (β) Let ρ be an arbitrary environment.

$$\begin{aligned} E\,((\lambda x{:}A.\,e)\,e')\,\rho &= (E\,(\lambda x{:}A.\,e)\,\rho)(E\,e'\,\rho) \\ &= E\,e\,\rho(x{\mapsto}E\,e'\,\rho) \\ &= E\,[x{:=}e']e\,\rho && \text{by Lemma 6} \end{aligned}$$

- (η) Let ρ be an arbitrary environment.

$$\begin{aligned} E\,(\lambda x.\,e\,x)\,\rho &= \lambda d.\,E\,(e\,x)\,\rho(x{\mapsto}d) \\ &= \lambda d.\,(E\,e\,\rho(x{\mapsto}d))\,d \\ &= \lambda d.\,(E\,e\,\rho)\,d && \text{because } x \notin \mathrm{FV}(e) \\ &= E\,e\,\rho \end{aligned}$$

□

Lemma 6 (Substitution). $E\,([x{:=}e']e)\,\rho = E\,e\,\rho(x{\mapsto}E\,e'\,\rho)$

Proof. TODO (page 51 of Gunter) □

4 *The untyped 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 $(e_1\,e_2)$ applies the function produced by e_1 to 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 arithmetic on natural numbers.

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

4.1 *Semantics*

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 4.8.

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 4.6. But first, we study perhaps the most straightforward of the graph models, the recent one by yours truly [Siek, 2017]. (todo: What about filter models? And isn't Siek's in some sense closer to the filter models?)

Domain

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

Ordering

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

Application

$$\begin{array}{c} \_ \cdot \_ : \mathcal{P}(\mathbb{D}) \times \mathcal{P}(\mathbb{D}) \to \mathcal{P}(\mathbb{D}) \\ D_1^* \cdot D_2^* \stackrel{\text{def}}{=} \left\{ d_3 \;\middle|\; \begin{array}{l} \exists t d_2 d d', t \in D_1^*, d_2 \in D_2^*, \\ (d, d') \in t, d \sqsubseteq d_2, d_3 \sqsubseteq d' \end{array} \right\} \end{array}$$

Semantics

$$\begin{array}{rcl} P\, e & \stackrel{\text{def}}{=} & \{d \mid d \in E\, e\, \varnothing\} \\ E & : & \mathbb{E} \to (\mathbb{X} \rightharpoonup \mathbb{D}) \to \mathcal{P}(\mathbb{D}) \\ E\, n\, \rho & = & \{n\} \\ E\, (e_1 + e_2)\, \rho & = & \{n_1 + n_2 \mid n_1 \in E\, e_1\, \rho, n_2 \in E\, e_2\, \rho\} \\ E\, (e_1 \times e_2)\, \rho & = & \{n_1 n_2 \mid n_1 \in E\, e_1\, \rho, n_2 \in E\, e_2\, \rho\} \\ E\, x\, \rho & = & \{d \mid d \sqsubseteq \rho\, x\} \\ E\, (\lambda x.\, e)\, \rho & = & \{t \mid \forall (d, d') \in t, d' \in E\, e\, \rho(x \mapsto d))\} \\ E\, (e_1\, e_2)\, \rho & = & (E\, e_1\, \rho) \cdot (E\, e_2\, \rho) \end{array}$$

Figure 12: Semantics of the λ-calculus

Figure 12 defines the syntax and semantics of a CBV λ-calculus. To enable trying all possible finite subsets of a function's graph, 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 set $\mathbb{D}$ consists of just the natural numbers and finite graphs (association tables) of functions. $\mathbb{D}$ is defined inductively, or equivalently, it is the least solution of

$$\mathbb{D} = \mathbb{N} + \mathcal{P}_f(\mathbb{D} \times \mathbb{D}) \tag{3}$$

Let t range over the finite graphs, that is, $t \in \mathcal{P}_f(\mathbb{D} \times \mathbb{D})$, D range over finite sets of elements from $\mathbb{D}$, and D^* range over possibly infinite sets of elements from $\mathbb{D}$.

The semantics in Figure 12 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, $d' \in E\, e\, \rho(x \mapsto d)$.

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^*, collecting all of the results into a resulting set. A naive definition of the application operator is given below. The idea is to collect up all the results d' that come from matching an argument $d \in D_2^*$ with an entry (d, d') from a table $t \in D_1^*$.

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

The problem with this 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 12. So then, for argument $d_2 \in D_2^*$, we require $d \sqsubseteq d_2$ instead of $d = d_2$.

Say something about d' and d_3 to motivate the following.

$$X \vdash \rho \sqsubseteq \rho' \stackrel{\text{def}}{=} \forall x, x \in X \implies \rho(x) \sqsubseteq \rho'(x)$$

Proposition 7 (Subsumption)**.**
If $e \in E\, e\, \rho$, $d' \sqsubseteq d$, *and* $\mathrm{FV}(e) \vdash \rho \sqsubseteq \rho'$, *then* $d' \in E\, e\, \rho'$.

4.2 *Interpreter*

We turn to an implementation of the λ-calculus, the interpreter in Figure 13. 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.

Figure 13: Interpreter for the λ-calculus

$$\begin{array}{lll} \text{values} & v \in \mathbb{V} ::= & n \mid \langle \lambda x.\, e, \varrho \rangle \\ \text{env.} & \varrho \in \mathbb{X} \rightharpoonup \mathbb{V} & \end{array}$$

$$\begin{aligned} I\, n\, \varrho &= n \\ I\, (e_1 + e_2)\, \varrho &= n_1 + n_2 \\ &\quad \text{if } I\, e_1\, \varrho = n_1, I\, e_2\, \varrho = n_2 \\ I\, (e_1 \times e_2)\, \varrho &= n_1 n_2 \\ &\quad \text{if } I\, e_1\, \varrho = n_1, I\, e_2\, \varrho = n_2 \\ I\, x\, \varrho &= \varrho(x) \\ I\, (\lambda x.\, e)\, \varrho &= \langle \lambda x.\, e, \varrho \rangle \\ I\, (e_1\, e_2)\, \varrho &= I\, e\, [x \mapsto v] \varrho' \\ &\quad \text{if } I\, e_1\, \varrho = \langle \lambda x.\, e, \varrho' \rangle, I\, e_2\, \varrho = v \end{aligned}$$

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 number n, then the interpreter also produces n. The second part is to show that if the interpreter produces an answer n, then the semantics agrees that n should be the answer.

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

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

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

Lemma 9. *If $\mathcal{G}(\rho, \varrho)$, then $\varrho(x) \in \mathcal{G}(\rho(x))$*

Lemma 10. *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 11 (Adequacy)**.** *If $E\, e\, \emptyset = E\, n\, \emptyset$, then $I\, e\, \emptyset = n$.*

[The rest of this subsection is under construction! -Jeremy]

Theorem 12 (Completeness wrt. Interpreter). *If $I\,e\,\emptyset = n$, then $E\,e\,\emptyset = E\,n\,\emptyset$.*

Toward proving completeness, we introduce a memoizing interpreter, that is, an interpreter that records the input and output of every function call in a table.

$$\begin{array}{lrl}
\text{fun. id's} & f \in \mathbb{N} & \\
\text{values} & v \in \mathbb{V} ::= & n \mid f \\
\text{env.} & \varrho \in \mathbb{X} \rightharpoonup \mathbb{V} & \\
& t ::= & \{(v_1, v_1'), \ldots, (v_n, v_n')\} \\
\text{memo} & m \in \mathbb{M} ::= & \langle \lambda x.\, e, \varrho, t \rangle \\
\text{memo table} & T \in & \mathbb{N} \rightharpoonup \mathbb{M}
\end{array}$$

The following produces cyclic tables. How do we convert these into non-cyclic tables? Or can we generate non-cyclic tables to begin with? Or put another way, how to we record the callee-view of a memo. table? -Jeremy

$$\begin{aligned}
I'\,n\,\varrho\,T &= (n, T) \\
I'\,(e_1 + e_2)\,\varrho\,T &= (n_1 + n_2, T_2) \\
&\quad \text{if } I'\,e_1\,\varrho\,T = (n_1, T_1), I'\,e_2\,\varrho\,T_1 = (n_2, T_2) \\
I'\,(e_1 \times e_2)\,\varrho\,T &= (n_1 n_2, T_2) \\
&\quad \text{if } I'\,e_1\,\varrho\,T = (n_1, T_1), I'\,e_2\,\varrho\,T_1 = (n_2, T_2) \\
I'\,x\,\varrho\,T &= (\varrho(x), T) \\
I'\,(\lambda x.\, e)\,\varrho\,T &= (|T|, T(|T| \mapsto \langle \lambda x.\, e, \varrho, \emptyset \rangle)) \\
I'\,(e_1\,e_2)\,\varrho\,T &= (v', T_3(f \mapsto \langle \lambda x.\, e, \varrho', \{(v, v')\} \cup t \rangle)) \\
&\quad \text{if } I'\,e_1\,\varrho\,T = (f, T_1), I'\,e_2\,\varrho\,T_1 = (v, T_2), \\
&\quad T(f) = \langle \lambda x.\, e, \varrho', t \rangle, I'\,e\,[x \mapsto v]\varrho'\,T_2 = (v', T_3)
\end{aligned}$$

4.3 *Reduction Semantics*

[TODO: define reduction semantics]

Lemma 13 (Invariance under Substitution).

$$E\,[x := v]e\,\rho = \bigcup_{d' \in E\,v\,\emptyset} E\,e\,\rho(x \mapsto d')$$

Proposition 14 (Invariance under reduction).
If $e \longrightarrow e'$, then $\forall \rho, E\,e\,\rho = E\,e'\,\rho$.

Theorem 15 (Completeness wrt. reduction).
If $e \longrightarrow^* n$, then $E\,e\,\emptyset = E\,n\,\emptyset$.

Definition 7 (Contexts)**.**

$$C ::= \Box \mid C + e \mid e + C \mid C \times e \mid e \times C \mid \lambda x.\, C \mid C\, e \mid e\, C$$

Proposition 16 (Congruence)**.**
If $\forall \rho, E\, e\, \rho = E\, e'\, \rho$*, then* $\forall \rho, E\, C[e]\, \rho = E\, C[e']\, \rho$ *for any* C*.*

We write $e \Downarrow$ to mean that e terminates, i.e., it reaches a value or gets stuck after some number of reduction steps.

Theorem 17 (Sound wrt. contextual equivalence)**.**
If $\forall \rho, E\, e\, \rho = E\, e'\, \rho$*, then* $C[e] \Downarrow$ *iff* $C[e'] \Downarrow$ *for any closing context* C*.*

4.4 Equational Theory and Models

Figure 14 defines the equational theory of two variants of the λ-calculus, the full λ-calculus with the rule β, and the call-by-value calculus with the rule β_v.

Figure 14: Equational Theory of the λ-calculus

$$(\text{refl}) \vdash e = e \quad (\text{sym}) \frac{\vdash e_1 = e_2}{\vdash e_2 = e_1} \quad (\text{trans}) \frac{\vdash e_1 = e_2 \quad \vdash e_2 = e_3}{\vdash e_1 = e_3}$$

$$(\text{cong}) \frac{\vdash e_1 = e_3 \quad \vdash e_2 = e_4}{\vdash e_1\, e_2 = e_3\, e_4} \quad (\xi) \frac{\vdash e = e'}{\vdash \lambda x.\, e = \lambda x.\, e'}$$

$$(\beta) \vdash (\lambda x.\, e_1)\, e_2 = [x := e_2] e_1$$

$$(\beta_v) \vdash (\lambda x.\, e)\, v = [x := v] e$$

Direct Proof [TODO: direct proofs that E satisfies the equational theory based on prior theorems about invariance under reduction and congruence.]

Proof via λ-model The following notion of λ-model is used in the literature that captures a set of conditions that are sufficient for a model to satisfy the equational theory of the λ-calculus.

Definition 8 (λ-models)**.** *An* R-model of the λ-calculus *is a triple* $\langle D, \cdot, [\![\ ]\!] \rangle$ *such that* D *is a set,* $\_ \cdot \_ : D \times D \to D$*, and* $[\![\_]\!] : \mathbb{E} \to (\mathbb{X} \rightharpoonup D) \to D$*, and the following conditions hold.*

1. $[\![x]\!]\rho = \rho(x)$,
2. $[\![e_1\, e_2]\!]\rho = [\![e_1]\!]\rho \cdot [\![e_2]\!]\rho$,

3. *if* $(\lambda x.\, e)\, e'$ *is a R-redex,* $[\![(\lambda x.\, e)]\!]\rho \cdot [\![e']\!]\rho = [\![e]\!]\rho(x \mapsto [\![e']\!]\rho)$, *alternative:* $[\![(\lambda x.\, e)]\!]\rho \cdot d = [\![e]\!]\rho(x \mapsto d)$
4. $\forall x \in \mathrm{FV}(e)$, *if* $\rho(x) = \rho'(x)$, *then* $[\![e]\!]\rho = [\![e]\!]\rho'$.
5. *If* $\forall d \in D, [\![e]\!]\rho(x \mapsto d) = [\![e']\!]\rho(x \mapsto d)$, *then* $[\![\lambda x.\, e]\!]\rho = [\![\lambda x.\, e']\!]\rho$.

The semantic function E given in Figure 12 does not directly fit the above definition of λ-model because its environment maps variables to $\mathbb{D}$ instead of $\mathcal{P}(\mathbb{D})$. However, this difference can be bridged with the following alternative semantic function, based on an analogous construction for filter models [Alessi et al., 2006], which we discuss in Section 4.5.

$$E'\, e\, \rho' = \{d \mid \exists \rho, d \in E\, e\, \rho \text{ and } \rho \models \rho'\}$$

where

$$\rho \models \rho' \stackrel{\text{def}}{=} \forall x \in \mathrm{dom}(\rho), \rho(x) \in \rho'(x)$$

Actually, for the proof to go through, we can't let environments map variables to arbitrary sets, many of which couldn't possibly be the denotation of an expression. Such sets are missing two important properties of actual denotations; they are 1) downward closed with respect to $\sqsubseteq$, and 2) closed under $\sqcup$. A set with these closure conditions is called an *ideal*. We write $\mathcal{I}(\mathbb{D})$ for the ideals over $\mathbb{D}$. So E' has the following type.

$$E' : \mathbb{E} \to (\mathbb{X} \rightharpoonup \mathcal{I}(\mathbb{D})) \to \mathcal{I}(\mathbb{D})$$

Lemma 18. $\langle \mathcal{I}(\mathbb{D}), \cdot, E' \rangle$ *is a* β_v*-model of the* λ*-calculus.*

Proof.

1. We need to show that $E'\, x\, \rho' = \rho'(x)$.

$$\begin{aligned} E'\, x\, \rho' &= \{d \mid \exists \rho, d \in E\, x\, \rho, \rho \models \rho'\} \\ &= \{d \mid \exists \rho, \rho(x) = d, d \in \rho'(x)\} \\ &= \{d \mid d \in \rho'(x)\} \\ &= \rho'(x) \end{aligned}$$

2. We need to show that $E'\, (e_1\, e_2)\, \rho' = (E'\, e_1\, \rho') \cdot (E'\, e_2\, \rho')$.

$$\begin{aligned} E'\, (e_1\, e_2)\, \rho' &= \{d' \mid \exists \rho, d' \in E\, (e_1\, e_2)\, \rho, \rho \models \rho'\} \\ &= \{d' \mid \exists \rho, d' \in (E\, e_1\, \rho) \cdot (E\, e_2\, \rho), \rho \models \rho'\} \\ &= \{d' \mid \exists t d d_2 \rho, t \in E\, e_1\, \rho, d_2 \in E\, e_2\, \rho, (d, d') \in t, d \sqsubseteq d_2, \rho \models \rho'\} \\ &= \{d' \mid \exists t d d_2, (\exists \rho, t \in E\, e_1\, \rho, \rho \models \rho'), (\exists \rho, d_2 \in E\, e_2\, \rho, \rho \models \rho'), (d, d') \in t, d \sqsubseteq d_2\} \\ &\qquad \text{by Proposition 7 and Lemma 19} \\ &= \{d' \mid \exists t d d_2, t \in E'\, e_1\, \rho', d_2 \in E'\, e_2\, \rho', (d, d') \in t, d \sqsubseteq d_2\} \\ &= (E'\, e_1\, \rho') \cdot (E'\, e_2\, \rho') \end{aligned}$$

3.

$$\begin{aligned} E'\,(\lambda x.\,e)\,\rho' \cdot D^* &= \{t \mid \exists \rho, \rho \models \rho', \forall (d,d) \in t, d' \in E\,e\,\rho(x{\mapsto}d)\} \cdot D^* \\ &= \{d_3 \mid \exists d_2 d d' \rho, \rho \models \rho', d' \in E\,e\,\rho(x{\mapsto}d), d_2 \in D^*, d \sqsubseteq d_2, d_3 \sqsubseteq d'\} \\ &= \{d \mid \exists \rho, d \in E\,e\,\rho, \rho \models \rho'(x{\mapsto}D^*)\} \\ &\qquad \text{because } \rho(x{\mapsto}d) \models \rho'(x{\mapsto}D^*), d_3 \in E\,e\,\rho(x{\mapsto}d) \\ &= E'\,e\,\rho'(x{\mapsto}D^*) \end{aligned}$$

4. TODO

5. TODO

□

Lemma 19. *If $\rho_1 \models \rho'$, $\rho_2 \models \rho'$, then $\rho_1 \sqcup \rho_2 \models \rho'$.*

Proof. TODO This is where we probably need the requirements of being closed under union (ideal), dual to filters. □

Lemma 20 (Replace Variable)**.** *Suppose $\langle D, \cdot, [\![\ ]\!] \rangle$ is an R-model of the λ-calculus and $y \notin \mathrm{FV}(e)$.*

$$[\![e]\!]\rho(x{\mapsto}d) = [\![[x{:=}y]e]\!]\rho(y{\mapsto}d)$$

Proof.

$$\begin{aligned} [\![e]\!]\rho(x{\mapsto}d) &= [\![\lambda x.\,e]\!]\rho \cdot d && \text{by condition 3} \\ &= [\![\lambda y.\,[x{:=}y]e]\!]\rho \cdot d && \alpha \text{ conversion} \\ &= [\![[x{:=}y]e]\!]\rho(y{\mapsto}d) && \text{by condition 3} \end{aligned}$$

□

Lemma 21 (Substitution)**.** *(5.3.3 of Barendregt [1984]) Suppose $\langle D, \cdot, [\![\ ]\!] \rangle$ is an R-model of the λ-calculus.*

1. *If $z \notin \mathrm{FV}(e)$, then $\forall \rho, [\![[x{:=}z]e]\!]\rho = [\![e]\!]\rho(x{\mapsto}[\![z]\!]\rho)$.*
2. *If $\forall \rho, [\![[x{:=}e']e]\!]\rho = [\![e]\!]\rho(x{\mapsto}[\![e']\!]\rho)$,*
   then $\forall \rho, [\![[x{:=}e']\lambda y.\,e]\!]\rho = [\![\lambda y.\,e]\!]\rho(x{\mapsto}[\![e']\!]\rho)$.
3. *$\forall \rho, [\![[x{:=}e']e]\!]\rho = [\![e]\!]\rho(x{\mapsto}[\![e']\!]\rho)$.*

Proof. We prove part 1, then part 2 using part 1, and finally part 3 using part 2 of this lemma.

1. Assume $z \notin \mathrm{FV}(e)$. Let $e \equiv e(x)$.

$$\begin{aligned} [\![e(z)]\!]\rho &= [\![e(z)]\!][z{:=}\rho(z)]\rho && \text{by condition 4} \\ &= [\![e(x)]\!][x{:=}\rho(z)]\rho && \text{by Lemma 20} \\ &= [\![e(x)]\!][x{:=}[\![z]\!]\rho]\rho && \text{by condition 1} \end{aligned}$$

2. To prove this implication, we assume its premise:

$$\forall \rho, [\![[x := e']e]\!]\rho = [\![e]\!]\rho(x \mapsto [\![e']\!]\rho) \tag{4}$$

Let ρ be an arbitrary environment. We consider two cases, whether $x \in \mathrm{FV}(e')$ or not.

- Assume $x \in \mathrm{FV}(e')$. Let z be a fresh variable.

$$\begin{aligned}
[\![[x := e']\lambda y.\, e]\!]\rho &= [\![[z := e'][x := z]\lambda y.\, e]\!]\rho \\
&= [\![[x := z]\lambda y.\, e]\!]\rho(z \mapsto [\![e']\!]\rho) && ?? \\
&= [\![\lambda y.\, e]\!]\rho(z \mapsto [\![e']\!]\rho)(x \mapsto [\![e']\!]\rho) && \text{by part 1} \\
&= [\![\lambda y.\, e]\!]\rho(x \mapsto [\![e']\!]\rho)
\end{aligned}$$

- Assume $x \notin \mathrm{FV}(e')$. Let $d \in D$ be an arbitrary element.

$$\begin{aligned}
[\![[x := e']e]\!]\rho(x \mapsto [\![e']\!]\rho)(y \mapsto d) &= [\![[x := e']e]\!]\rho(y \mapsto d) && \text{cond. 4} \\
&= [\![e]\!]\rho(y \mapsto d)(x \mapsto [\![e']\!]\rho) && \text{by eq. (4)} \\
&= [\![e]\!]\rho(x \mapsto [\![e']\!]\rho)(y \mapsto d) && x \neq y
\end{aligned} \tag{5}$$

$$\begin{aligned}
[\![[x := e']\lambda y.\, e]\!]\rho &= [\![[x := e']\lambda y.\, e]\!]\rho(x \mapsto [\![e']\!]\rho) && \text{condition 4} \\
&= [\![\lambda y.\, [x := e']e]\!]\rho(x \mapsto [\![e']\!]\rho) \\
&= [\![\lambda y.\, e]\!]\rho(x \mapsto [\![e']\!]\rho) && \text{cond. 5 and eq. (5)}
\end{aligned}$$

3. The proof is by induction on e.

Case $e = y$: If $y \neq x$, then $[\![[x := e']y]\!]\rho = \rho(y) = [\![y]\!]\rho(x \mapsto [\![e']\!]\rho)$.
If $y = x$, then $[\![[x := e']y]\!]\rho = [\![e']\!]\rho = [\![y]\!]\rho(x \mapsto [\![e']\!]\rho)$.

Case $e = \lambda y.\, e_1$: By the IH we have

$$\forall \rho, [\![[x := e']e_1]\!]\rho = [\![e_1]\!]\rho(x \mapsto [\![e']\!]\rho)$$

Then by part 2 of this lemma we conclude that

$$\forall \rho, [\![[x := e']\lambda y.\, e_1]\!]\rho = [\![\lambda y.\, e_1]\!]\rho(x \mapsto [\![e']\!]\rho)$$

Case $e = e_1\, e_2$: Let ρ be an arbitrary environment.

$$\begin{aligned}
[\![[x := e'](e_1\, e_2)]\!]\rho &= [\![[x := e']e_1\ [x := e']e_2]\!]\rho \\
&= [\![[x := e']e_1]\!]\rho \cdot [\![[x := e']e_2]\!]\rho && \text{by cond. 2} \\
&= [\![e_1]\!]\rho(x \mapsto [\![e']\!]\rho) \cdot [\![e_2]\!]\rho(x \mapsto [\![e']\!]\rho) && \text{by IH} \\
&= [\![e_1\, e_2]\!]\rho(x \mapsto [\![e']\!]\rho) && \text{by cond. 2}
\end{aligned}$$

□

Lemma 22 (Soundess of R-models wrt. λ theory)**.** *Suppose $\langle D, \cdot, [\![\ ]\!] \rangle$ is an R-model of the λ-calculus.*

$$\text{if } R \vdash e = e', \text{ then } \forall \rho, [\![e]\!]\rho = [\![e']\!]\rho$$

Proof. The proof is by induction on $R \vdash e = e'$.

Case (refl): We have $\forall \rho, [\![e]\!]\rho = [\![e]\!]\rho$ because $[\![\ ]\!]$ is a function.

Case (sym): By the IH, we have $\forall \rho, [\![e_1]\!]\rho = [\![e_2]\!]\rho$. So it trivially follows that $\forall \rho, [\![e_2]\!]\rho = [\![e_1]\!]\rho$.

Case (trans): By the IH, we have $\forall \rho, [\![e_1]\!]\rho = [\![e_2]\!]\rho$ and $\forall \rho, [\![e_2]\!]\rho = [\![e_3]\!]\rho$. Thus, we immediately have $\forall \rho, [\![e_1]\!]\rho = [\![e_3]\!]\rho$.

Case (cong): Let ρ be an arbitrary environment. We need to show that $[\![(e_1\, e_2)]\!]\rho = [\![(e_3\, e_4)]\!]\rho$.

$$\begin{aligned} [\![e_1\, e_2]\!]\rho &= [\![e_1]\!]\rho \cdot [\![e_2]\!]\rho && \text{condition 2} \\ &= [\![e_3]\!]\rho \cdot [\![e_4]\!]\rho && \text{by IH} \\ &= [\![e_3\, e_4]\!]\rho && \text{condition 2} \end{aligned}$$

Case ξ: We need to show $\forall \rho, [\![\lambda x.\, e]\!]\rho = [\![\lambda x.\, e']\!]\rho$. Let ρ be an arbitrary environment. By condition 5 it suffices to show that $[\![e]\!]\rho(x \mapsto d) = [\![e']\!]\rho(x \mapsto d)$, and that follows from the IH.

Case R: Let ρ be an arbitrary environment. We need to show that $[\![(\lambda x.\, e)\, e']\!]\rho = [\![[x := e']e]\!]\rho$, where $(\lambda x.\, e)\, e'$ is an R-redex.

$$\begin{aligned} [\![(\lambda x.\, e)\, e']\!]\rho &= [\![\lambda x.\, e]\!]\rho \cdot [\![e']\!]\rho && \text{condition 2} \\ &= [\![e]\!]\rho(x \mapsto [\![e']\!]\rho) && \text{condition 3} \\ &= [\![[x := e']e]\!]\rho && \text{by Lemma 21} \end{aligned}$$

□

Theorem 23. *$\langle \mathcal{I}(\mathbb{D}), \cdot, E' \rangle$ is a model of the CBV λ-calculus. That is,*

$$\text{if } \beta_v \vdash e = e', \text{ then } \forall \rho, E'\, e\, \rho = E'\, e'\, \rho$$

Resources:

- Barendregt [1984] (Section 2.1 for equational theory and Section 5.3 for (syntactical) λ-models).
- Alessi et al. [2006]
- Hindley and Longo [1980]

4.5 *Intersection Types and Filter models*

The addition of intersection types to a type system dramatically increases its precision. In fact, intersection types are fully precise; an intersection type system specifies the dynamic semantics of a program. There are a large number of different but related intersection type systems. Alessi et al. [2006] give a survey of them and prove general results that relate them to equational theories of λ-calculi. Here we focus on just one of them, a variant of the $\mathcal{EHR}$ system for the call-by-value λ-calculus of Egidi et al. [1992]. Figure 15 defines types and the subtyping and type equivalence rules. With respect to Egidi et al. [1992], here we have added a singleton type n for natural numbers.

The intuition behind the typing rules for intersection types is that if we think of types as sets of values, subtyping acts like set inclusion ($\subseteq$), and intersection is like set intersection. So, for example, the rule (incl-L) corresponds to an obviously true statement about sets, that $A \cap B \subseteq A$.

The type ν characterizes all functions, so the (ν) subtyping rule says that any function type is a subtype of ν. The rule ($\to \ -\cap$) captures the intuition that if you call a function of intersection type $(A \to B) \cap (A \to C)$ with an argument of type A, we can use $A \to B$ to deduce that the result is in B and use $A \to C$ to deduce that the result is also in C. Thus, the result must be in $B \cap C$. The (η) subtyping rule is the standard one for function types.

Figure 15: Types, subtyping, and equivalence

$$\text{types} \quad A, B, C \in \mathbb{T} ::= \quad n \mid \nu \mid A \to B \mid A \cap B$$

$$\text{(refl)} \quad A <: A \qquad\qquad \text{(idem)} \quad A <: A \cap A$$

$$\text{(incl-L)} \quad A \cap B <: A \qquad\qquad \text{(incl-R)} \quad A \cap B <: B$$

$$\text{(mon)} \quad \frac{A <: A' \qquad B <: B'}{A \cap B <: A' \cap B'} \qquad\qquad \text{(trans)} \quad \frac{A <: B \qquad B <: C}{A <: C}$$

$$(\nu) \quad A \to B <: \nu$$

$$(\to -\cap) \quad (A \to B) \cap (A \to C) <: A \to (B \cap C) \qquad\qquad (\eta) \quad \frac{A' <: A \qquad B <: B'}{A \to B <: A' \to B'}$$

$$A \sim B \stackrel{\text{def}}{=} A <: B \text{ and } B <: A$$

Figure 16 defines the intersection type system. It includes the introduction rule for intersection, ($\cap$-intro), which assigns expression e the type $A \cap B$ if e can be typed with both A and B. The introduction rule for ν says that every λ has type ν, even if the body of the λ is ill typed! The subsumpton rule (sub), as usual, enables implicit up-casts with respect to subtyping. The rules for variables (var), λ ab-

straction ($\to$-intro), and application ($\to$-elim), are the same as in the simply-typed λ-calculus. The (nat) rule is typical of singleton types.

Figure 16: An Intersection Type System

$$(\text{var})\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad (\text{nat})\frac{}{\Gamma \vdash n : n} \qquad (\nu\text{-intro})\frac{}{\Gamma \vdash \lambda x.\, e : \nu}$$

$$(\to\text{-intro})\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B} \qquad (\to\text{-elim})\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}$$

$$(\cap\text{-intro})\frac{\Gamma \vdash e : A \quad \Gamma \vdash e : B}{\Gamma \vdash e : A \cap B} \qquad (\text{sub})\frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B}$$

Proposition 24 (Subject Reduction and Expansion)**.**

- *If $e \longrightarrow e'$ and $\Gamma \vdash e : A$, then $\Gamma \vdash e' : A$.*
- *If $e \longrightarrow e'$ and $\Gamma \vdash e' : A$, then $\Gamma \vdash e : A$.*

The intersection type system can be made into a λ-model similar to the way we made E into a λ-model by constructing E'. Subtyping plays the same role as $\sqsubseteq$, but inverted. So instead of working with ideals, we work with filters.

Definition 9 (Filter)**.** *A subset S of types $\mathbb{T}$ is a* filter *if it is closed with respect to subtyping and intersection. That is,*

- *if $A \in S$ and $A <: B$, then $B \in S$, and*
- *if $A \in S$ and $B \in S$, then $A \cap B \in S$.*

The set of all filters is $\mathbb{F}$. We define the upward closure of a set $\uparrow S$ as follows.

$$\frac{A \in S}{A \in \uparrow S} \qquad \frac{A \in \uparrow S \quad A <: B}{B \in \uparrow S} \qquad \frac{A \in \uparrow S \quad B \in \uparrow S}{A \cap B \in \uparrow S}$$

Definition 10. *The* filter λ-structure *is the triple $\langle \mathbb{F}, \cdot, F \rangle$ where*

$$\begin{aligned} \_ \cdot \_ &: \mathbb{F} \times \mathbb{F} \to \mathbb{F} \\ S_1 \cdot S_2 &\stackrel{\text{def}}{=} \uparrow\{B \mid \exists A \in S_2, A \to B \in S_1\} \\ \Gamma \models \rho &\stackrel{\text{def}}{=} \forall x{:}B \in \Gamma, B \in \rho(x) \\ F &: \mathbb{E} \to (\mathbb{X} \rightharpoonup \mathbb{F}) \to \mathbb{F} \\ F\, e\, \rho &\stackrel{\text{def}}{=} \{A \mid \exists \Gamma, \Gamma \models \rho \text{ and } \Gamma \vdash e : A\} \end{aligned}$$

Is the upward closure really needed in the definition of application? We are missing that for E and E'. TODO: mechanize this λ-model stuff to make sure one way or the other. -Jeremy

Lemma 25. *The filter λ-structure $\langle \mathbb{F}, \cdot, F \rangle$ is a β_v-model of the λ-calculus.*

Theorem 26. *The filter λ-structure $\langle \mathbb{F}, \cdot, F \rangle$ is a model of the CBV λ-calculus. That is,*

$$\text{if } \beta_v \vdash e = e', \text{ then } \forall \rho, F\, e\, \rho = F\, e'\, \rho$$

Figure 17: The T_C^* model

$$
\begin{aligned}
T_C &= C + \mathcal{P}_f(T_C) \times \mathcal{P}_f(T_C) \\
T_C^* &= \mathcal{P}(T_C) \\
\lambda^P &: [T_C^* \to T_C^*] \to T_C^* \\
\lambda^P f &= \{(D, D') \mid D' \subseteq f\, D\} \\
- \cdot_P - &: T_C^* \to T_C^* \to T_C^* \\
D_1^* \cdot_P D_2^* &\stackrel{\text{def}}{=} \bigcup \{D' \mid \exists D, (D, D') \in D_1^*, D \subseteq D_2^*\} \\
E_P &: \mathbb{E} \to (\mathbb{X} \rightharpoonup T_C^*) \to T_C^* \\
E_P\, x\, \rho &= \rho(x) \\
E_P\, (\lambda x.\, e)\, \rho &= \lambda^P (\lambda D^*.\, E_P\, e\, [x \mapsto D^*] \rho) \\
E_P\, (e_1\, e_2)\, \rho &= (E_P\, e_1\, \rho) \cdot_P (E_P\, e_2\, \rho)
\end{aligned}
$$

4.6 *Graph models of λ-calculus (T_C^*, D_A, and $\mathcal{P}(\omega)$)*

The T_C^* model of Plotkin [1972] Let D range over finite sets of elements from T_C and D^* range over possibly infinite sets. Figure 17 defines the semantics.

The D_A model of Engeler [1981] Let d range over elements of B_A. Let D range over finite sets of elements from B_A and D^* range over possibly infinite sets. Figure 18 gives the semantics, following the presentation of Barendregt [1984] (Section 5.4).

D_A ordered by set inclusion $\subseteq$ is a cpo.

Definition 11. *A cpo D is* reflexive *if $[D \to D]$ is a retract of D. That is to say, there are continuous maps $F : D \to [D \to D]$ and $G : [D \to D] \to D$ such that for any $f \in [D \to D]$,*

$$(F \circ G)\, f = f$$

Lemma 27. *D_A is a reflexive cpo by choosing $F = \lambda xy.\, x \cdot_E y$ and $G = \lambda^E$. Therefore D_A is a λ-model.*

Proof. We need to show that $(F \circ G)\, f = f$ for any $f \in [D_A \to D_A]$.

$$
\begin{aligned}
(F \circ G)\, f &= \lambda x.(\lambda^E f) \cdot_E x \\
&= \lambda x.\{d' \mid \exists D, D \subseteq x \wedge d' \in f\, D\} \\
&= \lambda x.\bigcup \{f\, D \mid D \subseteq x\} \\
&= \lambda x.\, f\, x && \text{by continuity of } f \\
&= f
\end{aligned}
$$

Figure 18: The D_A model

$$
\begin{aligned}
B_A &= A + \mathcal{P}_f(B_A) \times B_A \\
D_A &= \mathcal{P}(B_A) \\
\lambda^E &: [D_A \to D_A] \to D_A \\
\lambda^E f &= \{(D, d') \mid d' \in f\, D\} \\
- \cdot_E - &: D_A \to D_A \to D_A \\
D_1^* \cdot_E D_2^* &= \{d' \mid \exists D, D \subseteq D_2^* \wedge (D, d') \in D_1^*\} \\
\\
E_E &: \mathbb{E} \to (\mathbb{X} \rightharpoonup D_A) \to D_A \\
E_E\, x\, \rho &= \rho(x) \\
E_E\, (\lambda x.\, e)\, \rho &= \lambda^E(\lambda D^*.\, E_E\, e\, [x \mapsto D^*]\rho) \\
E_E\, (e_1\, e_2)\, \rho &= (E_E\, e_1\, \rho) \cdot_E (E_E\, e_2\, \rho)
\end{aligned}
$$

□

Resources about D_A:

- Barendregt [1984] (Section 5.4),
- Gunter [1992] (Section 8.1), and
- Engeler [1981].

The $\mathcal{P}(\omega)$ model of Scott [1976] Figure 19 defines a semantics for the λ-calculus using the $\mathcal{P}(\omega)$ model of Scott [1976]. However, here we use the formulation of $\mathcal{P}(\omega)$ given by Barendregt [1984].

Figure 19: λ-calculus in $\mathcal{P}(\omega)$

$$
\begin{aligned}
m, n &\in \mathbb{N} \\
\langle n, m \rangle &= \frac{1}{2}(n+m)(n+m+1) + m \\
e_n &= \{k_0, k_1, \ldots, k_{m-1}\} \quad \text{where } k_i < k_{i+1} \text{ and } n = \sum_{i<m} 2^{k_i} \\
\\
E_S &: \mathbb{E} \to (\mathbb{X} \rightharpoonup \mathcal{P}(\mathbb{N})) \to \mathcal{P}(\mathbb{N}) \\
E_S\, x\, \rho &= \rho\, x \\
E_S\, (\lambda x.\, e)\, \rho &= \{\langle n, m \rangle \mid m \in E_S\, e\, [x \mapsto e_n]\rho\} \\
E_S\, (e_1\, e_2)\, \rho &= \{m \mid \exists e_n.\, \langle n, m \rangle \in E_S\, e_1\, \rho \text{ and } e_n \subseteq E_S\, e_2\, \rho\}
\end{aligned}
$$

4.7 Untyped as a uni-typed

types	$A, B ::=$	$nat \mid A \to B \mid A \times B \mid dyn$
type tag	$t ::=$	$fun \mid nat$
coercion	$c ::=$	$t! \mid t?$
expr.	$e \in \mathbb{E} ::=$	$n \mid e + e \mid e \times e \mid x \mid \lambda x\!:\!A.\, e \mid (e\, e) \mid e\langle c \rangle$

$$
\begin{aligned}
U(n) &= n\langle nat! \rangle \\
U(e_1 + e_2) &= (U(e_1)\langle nat? \rangle + U(e_2)\langle nat? \rangle)\langle nat! \rangle \\
U(x) &= x \\
U(\lambda x.\, e) &= (\lambda x\!:\!dyn.\, U(e))\langle fun! \rangle \\
U(e_1\, e_2) &= U(e_1)\langle fun? \rangle\, U(e_2)
\end{aligned}
$$

Equational theory: STLC's +

$$e\langle t! \rangle \langle t? \rangle = e$$

Theorem 28. *If* $\vdash e = e'$, *then* $\vdash U(e) = U(e')$.

Proof. The proof is by induction on the derivation of $\vdash e = e'$.

Case $\vdash (\lambda x.\, e_1)\, e_2 = [x := e_2]e_1$

$$
\begin{aligned}
U((\lambda x.\, e_1)\, e_2) &= (U(\lambda x.\, e_1))\langle fun? \rangle\, U(e_2) \\
&= (\lambda x.\, U(e_1))\langle fun! \rangle \langle fun? \rangle\, U(e_2) \\
&= (\lambda x.\, U(e_1))\, U(e_2) \\
&= [x := U(e_2)]U(e_1) \\
&= U([x := e_2]e_1)
\end{aligned}
$$

Case $\vdash n_1 + n_2 = n_3$ where n_3 is $n_1 + n_2$

$$
\begin{aligned}
U(n_1 + n_2) &= (n_1\langle nat! \rangle \langle nat? \rangle + n_2\langle nat! \rangle \langle nat? \rangle)\langle nat! \rangle \\
&= (n_1 + n_2)\langle nat! \rangle \\
&= n_3\langle nat! \rangle \\
&= U(n_3)
\end{aligned}
$$

TODO

□

4.8 D_∞ model of λ-calculus

UNDER CONSTRUCTION

Answers to Exercises

References

Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. Intersection types and lambda models. *Theoretical Compututer Science*, 355(2):108–126, 2006.

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

H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 54–65. ACM Press, 2007. ISBN 978-1-59593-633-2.

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

Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. Operational, denotational and logical descriptions: A case study. *Fundam. Inf.*, 16(2):149–169, February 1992. ISSN 0169-2968. URL `http://dl.acm.org/citation.cfm?id=161643.161646`.

Erwin Engeler. Algebras and combinators. *algebra universalis*, 13(1): 389–392, Dec 1981.

Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-07143-6.

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.

Leon Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15(2):81–91, 1950. ISSN 00224812. URL `http://www.jstor.org/stable/2266967`.

R. Hindley and G. Longo. Lambda-calculus models and extensionality. *Mathematical Logic Quarterly*, 26, 1980.

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

J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112

– 116, 1982. ISSN 0020-0190. DOI: https://doi.org/10.1016/0020-0190(82)90065-5. URL http://www.sciencedirect.com/science/article/pii/0020019082900655.

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.

Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977. ISBN 0262191474.

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