# lecDenote: Optional Notes on Denotational Semantics

Dimitrios Economou

April 12, 2022

## 1 What?

This course is about the *semantics*, that is, the meaning of programming languages. There are several ways to give meaning to a programming language. So far, we have focused on the *operational* way, where we describe the meaning of a language by providing an abstract machine for syntactically manipulating terms of the language. There are other approaches, such as *axiomatic*, which defines the meaning of a command in an imperative programming language by describing its effects on logical assertions about the state of the program (see *Hoare logic*). *Denotational semantics* capture the meaning of a language in terms of "standard" mathematical objects, like sets or sets with some additional "well-known" mathematical structure, such as a partial order. This is the perspective we take in these notes, but the terminology in the literature can be confusing: some authors will give denotations of syntax that are themselves syntactic (in the nature of the language itself), not standard mathematical objects separate from but formally related to the language.

We say "standard" and "well-known" because, for example, the reduction and stepping judgments that we see in operational semantics are *in some sense* mathematical objects, but are also relatively alien to mathematicians, in the sense that there isn't a general theory about them commonly used by mathematicians. We suspect that this is partly why operational semantics seems to be the most common approach taken today by researchers of programming language semantics: the mathematical objects one uses to write down an operational semantics are *elementary* in comparison to the relatively heavy machinery of, say, domain theory, category theory, or plain old set theory. But denotational semantics are still commonly used by PL researchers, likely because it has some of its own virtues that are relatively lacking in the operational approach.

## 2 Why?

We want to know important properties of our programming languages (such as type soundness). But our programming languages are often complicated. It would help to translate the complicated language into a simple theory (or at least one well understood by mathematicians working on or with it) where we can prove these kinds of properties. There are good reasons to use ordinary mathematics to specify the semantics of a programming language. If a denotational approach is relatively simple and conceptually clear, then one may prefer to avoid using sophisticated techniques like logical relations—which we don't cover here, but are often needed in the operational approach for minimally realistic functional languages (such as those with recursion). We think this is because denotational semantics tend to be more *compositional* and *extensional* than operational semantics. A semantics is *compositional* if it is directly defined by composition of the meanings of its structurally recursive subparts. A semantics is *extensional* if it only specifies behavior that is externally observable. Extensionality is desirable for specifying the meaning of a programming language, if we expect the specification not to include any artifacts of the language itself, which, after all, is what we are trying to understand in the first place. Operational semantics, on the other hand, are *intensional*: small-step semantics are all about syntactic stepping, and the value of a lambda abstraction in big-step semantics is a syntactic object (not semantic, like a set function) of the programming language. Operational semantics also tend to be less compositional. (This conception of the tradeoffs between denotational and operational semantics comes from Jeremy Siek's unpublished paper "Revisiting Elementary Denotational Semantics" [4].)

## 3 Typing for $L\lambda^-$

To highlight some of the fundamental issues of denotational semantics, let's remove some features from $L\lambda$. Basically, we only remove the base types int and bool and everything associated with them, and leave everything else the same. Now our only base type is unit. (Note that these removals don't actually reduce expressive power, because *Church encodings* exist.)

Below, we give the syntax of $L\lambda^-$. Syntactic values $v$ are not needed for denotational semantics, but we include them here anyway.

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & \texttt{()} \\
 & & | & x \mid \texttt{(Lam } x\ e\texttt{)} \mid \texttt{(Call } e\ e\texttt{)} \\
 & & | & \texttt{(Pair } e\ e\texttt{)} \mid \texttt{(Proj}_1\ e\texttt{)} \mid \texttt{(Proj}_2\ e\texttt{)} \\
 & & | & \texttt{(Inj}_1\ e\texttt{)} \mid \texttt{(Inj}_2\ e\texttt{)} \mid \texttt{(Case } e\ \texttt{(}x\ \texttt{=>}\ e\texttt{)}\ \texttt{(}x\ \texttt{=>}\ e\texttt{))} \\
\text{Values} & v & ::= & \texttt{()} \\
 & & | & x \mid \texttt{(Lam } x\ e\texttt{)} \\
 & & | & \texttt{(Pair } v\ v\texttt{)} \\
 & & | & \texttt{(Inj}_1\ v\texttt{)} \mid \texttt{(Inj}_2\ v\texttt{)}
\end{array}
$$

$$
\begin{array}{lrcll}
\text{Types} & S, T & ::= & \mathsf{unit} & \text{unit type} \\
 & & | & S \rightarrow T & \text{type of functions on } S \text{ that produce } T \\
 & & | & S \times T & \text{type of pairs of one } S \text{ and one } T \\
 & & | & S + T & \text{type of left-injections into } S \text{ and right-injections into } T \\
\text{Typing contexts} & \Gamma & ::= & \emptyset & \text{empty context} \\
 & & | & \Gamma, x : S & x \text{ has type } S
\end{array}
$$

The typing rules for $L\lambda^-$ are given in Figure 1.

$\boxed{\Gamma \vdash e : T}$ Under assumptions $\Gamma$, expression $e$ has type $T$

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ \text{type-assum} \qquad \frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \texttt{(Lam}\ x\ e\texttt{)} : (S \to T)}\ {\to}\text{Intro} \qquad \frac{\Gamma \vdash e_1 : (S \to T) \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash \texttt{(Call}\ e_1\ e_2\texttt{)} : T}\ {\to}\text{Elim}$$

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}\ \text{unitIntro} \qquad \frac{\Gamma \vdash e_1 : S_1 \qquad \Gamma \vdash e_2 : S_2}{\Gamma \vdash \texttt{(Pair}\ e_1\ e_2\texttt{)} : (S_1 \times S_2)}\ {\times}\text{Intro}$$

$$\frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash \texttt{(Proj}_1\ e\texttt{)} : S_1}\ {\times}\text{Elim1} \qquad \frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash \texttt{(Proj}_2\ e\texttt{)} : S_2}\ {\times}\text{Elim2}$$

$$\frac{\Gamma \vdash e_1 : S_1}{\Gamma \vdash \texttt{(Inj}_1\ e_1\texttt{)} : (S_1 + S_2)}\ {+}\text{Intro1} \qquad \frac{\Gamma \vdash e_2 : S_2}{\Gamma \vdash \texttt{(Inj}_2\ e_2\texttt{)} : (S_1 + S_2)}\ {+}\text{Intro2}$$

$$\frac{\Gamma \vdash e : (S_1 + S_2) \qquad \Gamma, x_1 : S_1 \vdash e_1 : T \qquad \Gamma, x_2 : S_2 \vdash e_2 : T}{\Gamma \vdash \texttt{(Case}\ e\ \texttt{(}x_1\ \texttt{=>}\ e_1\texttt{)}\ \texttt{(}x_2\ \texttt{=>}\ e_2\texttt{))} : T}\ {+}\text{Elim}$$

**Figure 1** **Typing for L$\lambda^-$**

## 4 Sets and types

In $L\lambda^-$, we can model types by their *sets of values* (the kinds of which correspond to the productions of the grammar for $v$, ignoring variables).

We use the letters $A$ and $B$ to name sets, and $a$ and $b$ to name their elements. We overload the symbol "$\in$" with the meaning of *set membership* (also used to express a membership of, say, a typing context): we have $a \in A$ if the element $a$ is a member of the set $A$. The *set builder* or *set comprehension* notation $\{a \mid \varphi\}$ means "the set of elements $a$ such that property $\varphi$ is true of $a$". We use the symbol $\longmapsto$ to define functions *on sets*, a longer version of the symbol $\mapsto$ used for stepping in the operational semantics of $L\lambda^-$ (we emphasize that $\longmapsto$ and $\mapsto$ are two different things). For example, we can define the integer set function *double* : $\mathbb{Z} \to \mathbb{Z}$ that doubles integers by writing *double* $= (x \longmapsto 2x)$, so that *double*$(-2) = -4$ and *double*$(0) = 0$ and *double*$(5) = 10$ and so on, that is, *double* $= \{\ldots, (-2, -4), (-1, 1), (0, 0), (1, 1), (2, 4), \ldots\}$.

To model the unit type of one value, we define a special singleton set, the *unit set*, containing exactly one element, $\star$:

$$\{\star\} = \text{the } \textit{unit set}$$

We model a product type $\times$ by a *Cartesian product* $\times$ (where we overload the symbol $\times$):

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

Given two sets $A_1$ and $A_2$, we define two *projection* functions out of their product $A_1 \times A_2$:

$$\begin{aligned} \textit{proj}_1 &: A_1 \times A_2 \to A_1 \\ \textit{proj}_1(a_1, a_2) &= a_1 \end{aligned}$$

$$\begin{aligned} \textit{proj}_2 &: A_1 \times A_2 \to A_2 \\ \textit{proj}_2(a_1, a_2) &= a_2 \end{aligned}$$

We model a sum type $+$ by a *disjoint union of sets* $+$ (overloading $+$), where we make its two sets disjoint by pairing a 1 with elements of the left set, and pairing a 2 with elements of the right set:

$$A + B = (\{1\} \times A) \cup (\{2\} \times B)$$

where $A \cup B = \{a \mid a \in A \text{ or } a \in B\}$. Given two sets $A_1$ and $A_2$, we define two *injection* functions into their disjoint union $A_1 + A_2$:

$$\begin{aligned} \textit{inj}_1 &: A_1 \to A_1 + A_2 \\ \textit{inj}_1(a_1) &= (1, a_1) \end{aligned}$$

$$\begin{aligned} \textit{inj}_2 &: A_2 \to A_1 + A_2 \\ \textit{inj}_2(a_2) &= (2, a_2) \end{aligned}$$

We model a function type $(\to)$ by a *set of functions* on sets (overloading $\to$), where functions are taken in their ordinary, Cantorian sense of a graph: a *function* from set $A$ to set $B$ is a set of pairs $(a, b) \in A \times B$ such that there is *exactly one* $b \in B$ for *every* $a \in A$.

In Figure 2, we summarize our interpretation of types (as sets of values) in terms of a *denotation function* $[\![-]\!]$ from $L\lambda^-$ types to sets, defined by structural induction on types.

$$\begin{aligned}
[\![\mathsf{unit}]\!] &= \{\star\} \\
[\![S \to T]\!] &= [\![S]\!] \to [\![T]\!] \\
[\![S \times T]\!] &= [\![S]\!] \times [\![T]\!] \\
[\![S + T]\!] &= [\![S]\!] + [\![T]\!]
\end{aligned}$$

**Figure 2** **Denotational semantics of types for L$\lambda^-$**

## 5 Interpretation of typing contexts and well-typed expressions

Next, we define a denotation function for *well-typed* expressions in L$\lambda^-$. We also define a denotation function for typing contexts in L$\lambda^-$. In the literature, especially textbooks on denotational semantics, it is common to distinguish the various denotation functions for the various constructs of the programming language. This is notationally heavy. We instead overload the $[\![-]\!]$ symbol; we can tell whether we are interpreting a type, an expression, or a typing context according to the local discussion and by being familiar with the grammars and judgments.

We mentioned there is a correspondence between the sets modeling the various types (Figure 2) and values $v$, *ignoring variables*. We interpret a typing context by an *environment* for it. An environment $\gamma$ is a list defined by the grammar:

$$\gamma ::= \emptyset \mid \gamma, a/x$$

where each $a$ is an element of a set interpreting some L$\lambda^-$ type. We define the metaoperation *dom* that takes a typing context and returns its domain of variables as a set:

$$\begin{aligned}
\mathit{dom}(\emptyset) &= \{\} \\
\mathit{dom}(\Gamma, x : S) &= \mathit{dom}(\Gamma) \cup \{x\}
\end{aligned}$$

In Figure 3, we define a judgment $\vdash \gamma : \Gamma$ that $\gamma$ is an *environment for* $\Gamma$, or a $\Gamma$-*environment*. We emphasize that we maintain the invariant that a variable can appear in a typing context *at most once*, and we follow the convention that variables are always freshened where appropriate (such as in the addition of a variable to a typing context in rule $\to$Intro): hence the third premise of rule Entry$\gamma$.

$\boxed{\vdash \gamma : \Gamma}$ We know that $\gamma$ is an environment for the typing context $\Gamma$

$$\frac{}{\vdash \emptyset : \emptyset}\ \text{Empty}\gamma \qquad\qquad \frac{\vdash \gamma : \Gamma \qquad a \in [\![S]\!] \qquad x \notin \mathit{dom}(\Gamma)}{\vdash \gamma, a/x : \Gamma, x : S}\ \text{Entry}\gamma$$

**Figure 3** **Well-typed environments**

Given $\vdash \gamma : \Gamma$ and a variable $x$, we inductively define *variable lookup* $\gamma(x)$ that looks up the semantic entry for $x$ in the environment $\gamma$ (and returns $x$ if there is no entry for it in $\gamma$):

$$\begin{aligned}(\emptyset)(x) &= x\\ (\gamma_0, a/x)(x) &= a\\ (\gamma_0, a/y)(x) &= \gamma_0(x)\end{aligned}$$

If there is an entry, that is, if $x \in dom(\Gamma)$ and $\vdash \gamma : \Gamma$, then inversion gives us the following facts: there exists a semantic value $a$ and a type $S$ such that $(x : S) \in \Gamma$ and $(a/x) \in \gamma$ and $a \in [\![S]\!]$.

Finally, we define the denotation $[\![\Gamma]\!]$ of a typing context $\Gamma$ to be the set of all $\Gamma$-environments:

$$[\![\Gamma]\!] = \{\gamma \mid \vdash \gamma : \Gamma\}$$

We are now prepared to interpret *well-typed* expressions, specifically, their typing derivations. In Figure 4, we interpret a derivation $\mathcal{D}$ of $\Gamma \vdash e' : S$, by structural induction on $\mathcal{D}$, as a set function $[\![\Gamma]\!] \to [\![S]\!]$. As is usual for defining set functions, to define such a function, we say what semantic value it outputs for any input $\gamma \in [\![\Gamma]\!]$. We write the application of the denotation $[\![\Gamma \vdash e : S]\!]$ of a well-typed expression $\Gamma \vdash e : S$ to a $\Gamma$-environment $\gamma$ as $[\![\Gamma \vdash e : S]\!]\,(\gamma)$.

$$\begin{aligned}
[\![\Gamma \vdash x : S]\!] &= \gamma \longmapsto \gamma(x)\\
[\![\Gamma \vdash (\texttt{Lam}\ x\ e) : (S \to T)]\!] &= \gamma \longmapsto \big(a \longmapsto [\![\Gamma, x : S \vdash e : T]\!]\,(\gamma, a/x)\big)\\
[\![\Gamma \vdash (\texttt{Call}\ e_1\ e_2) : T]\!] &= \gamma \longmapsto \Big([\![\Gamma \vdash e_1 : S \to T]\!]\,(\gamma)\Big)\big([\![\Gamma \vdash e_2 : S]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash () : \mathsf{unit}]\!] &= \gamma \longmapsto \star\\
[\![\Gamma \vdash (\texttt{Pair}\ e_1\ e_2) : (S_1 \times S_2)]\!] &= \gamma \longmapsto \big([\![\Gamma \vdash e_1 : S_1]\!]\,(\gamma), [\![\Gamma \vdash e_2 : S_2]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash (\texttt{Proj}_1\ e) : S_1]\!] &= \gamma \longmapsto proj_1\big([\![\Gamma \vdash e : (S_1 \times S_2)]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash (\texttt{Proj}_2\ e) : S_2]\!] &= \gamma \longmapsto proj_2\big([\![\Gamma \vdash e : (S_1 \times S_2)]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash (\texttt{Inj}_1\ e_1) : (S_1 + S_2)]\!] &= \gamma \longmapsto inj_1\big([\![\Gamma \vdash e_1 : S_1]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash (\texttt{Inj}_2\ e_2) : (S_1 + S_2)]\!] &= \gamma \longmapsto inj_2\big([\![\Gamma \vdash e_2 : S_2]\!]\,(\gamma)\big)\\
[\![\Gamma \vdash (\texttt{Case}\ e\ (x_1\ \texttt{=>}\ e_1)\ (x_2\ \texttt{=>}\ e_2)) : T]\!] &=\\
\gamma \longmapsto &\begin{cases}[\![\Gamma, x_1 : S_1 \vdash e_1 : T]\!]\,(\gamma, a_1/x_1) & \text{if } [\![\Gamma \vdash e : (S_1 + S_2)]\!]\,(\gamma) = inj_1(a_1)\\ [\![\Gamma, x_2 : S_2 \vdash e_2 : T]\!]\,(\gamma, a_2/x_2) & \text{if } [\![\Gamma \vdash e : (S_1 + S_2)]\!]\,(\gamma) = inj_2(a_2)\end{cases}
\end{aligned}$$

**Figure 4** **Denotational semantics of typing derivations in L$\lambda^-$**

## 6 (Denotational) semantic type soundness of L$\lambda^-$

We prove that L$\lambda^-$is sound with respect to its set-theoretic denotational semantics.

**Theorem 1** (Semantic Type Soundness of L$\lambda^-$)**.** *If* $\vdash \gamma : \Gamma$ *and* $\mathcal{D}$ *derives* $\Gamma \vdash e : S$, *then* $[\![\Gamma \vdash e : S]\!]\,(\gamma) \in [\![S]\!]$.

*Proof.* By structural induction on the derivation $\mathcal{D}$ of $\Gamma \vdash e : S$. (**Induction hypothesis**: If $\vdash \gamma' : \Gamma'$ and $\mathcal{D}'$ derives $\Gamma' \vdash e' : S'$ and $\mathcal{D}'$ is a proper subderivation of $\mathcal{D}$, then $[\![\Gamma' \vdash e' : S']\!]\,(\gamma') \in [\![S']\!]$.)

Consider cases for the rule concluding $\mathcal{D}$:

- **Case** $\dfrac{(x : S) \in \Gamma}{\Gamma \vdash x : S}$ type-assum

$$\begin{array}{rl}
(x : S) \in \Gamma & \text{Premise} \\
\vdash \gamma : \Gamma & \text{Given} \\
(a/x) \in \gamma & \text{By inversion on above lines} \\
a \in [\![S]\!] & '' \\
\\
[\![\Gamma \vdash x : S]\!]\,(\gamma) = \gamma(x) & \text{By definition of } [\![-]\!] \\
= a & \text{By definition of variable lookup} \\
\in [\![S]\!] & \text{Above}
\end{array}$$

- **Case** $\dfrac{\Gamma, x : S_0 \vdash e_0 : T}{\Gamma \vdash (\mathtt{Lam}\ x\ e_0) : (S_0 \rightarrow T)}$ $\rightarrow$Intro

Let $a \in [\![S_0]\!]$.

$$\begin{array}{rl}
\vdash \gamma : \Gamma & \text{Given} \\
\Gamma, x : S_0 \vdash e_0 : T & \text{Subderivation} \\
x \notin \mathit{dom}(\Gamma) & \text{By variable renaming convention} \\
\vdash (\gamma, a/x) : (\Gamma, x : S_0) & \text{By Entry}\gamma
\end{array}$$

$$\begin{array}{rll}
\big([\![\Gamma \vdash (\mathtt{Lam}\ x\ e_0) : (S_0 \rightarrow T)]\!]\,(\gamma)\big)(a) & = [\![\Gamma, x : S_0 \vdash e_0 : T]\!]\,(\gamma, a/x) & \text{By definition of } [\![-]\!] \\
& \in [\![T]\!] & \text{By induction hypothesis} \\
[\![\Gamma \vdash (\mathtt{Lam}\ x\ e_0) : (S_0 \rightarrow T)]\!]\,(\gamma) & \in [\![S_0]\!] \rightarrow [\![T]\!] & \text{By defn. of set function} \\
& = [\![S_0 \rightarrow T]\!] & \text{By definition of } [\![-]\!]
\end{array}$$

- The remaining cases are an **exercise**. □

## 7 Relating operational and denotational semantics

In previous lectures, we use the small-step operational semantics of $\mathrm{L}\lambda$ to prove *syntactic* type soundness (progress and preservation). Because the semantics are small-step, we prove progress and preservation lemmas. In combination, these lemmas say that a closed, well-typed expression $e$ either steps to an expression $e'$ of the same type, or is a syntactic value $v$.

Our denotational semantics are only about resulting *semantic* values, and not about the steps taken to get to their *syntactic* equivalent. Our semantic type soundness result (Thm. 1) only says the semantic values of well-typed expressions means what we say they mean (that is, what they denote). But the actual evaluation of function application necessarily involves syntactic substitution (beta- or $\beta$-*reduction*). It is important that syntactic substitution and semantic substitution (semantic substitution is the application of the denotation of a typing derivation to an environment) are compatible in some way, so that the computation of expressions is sound with respect to our *denotational* semantics. A common way to express this is by giving an equational theory $\Gamma \vdash e \equiv e' : S$

that includes (among other rules) a rule for beta-equality (this one is not confined to call-by-value)

$$\frac{\Gamma \vdash (\texttt{Call}\ (\texttt{Lam}\ x\ e_1)\ e_2) : T}{\Gamma \vdash (\texttt{Call}\ (\texttt{Lam}\ x\ e_1)\ e_2) \equiv [e_2/x]e_1 : T}\ \text{expr-equiv-beta}$$

and proving that the equational theory is sound with respect to the denotational semantics: if $\Gamma \vdash e \equiv e' : S$ and $\vdash \gamma : \Gamma$, then $[\![\Gamma \vdash e : S]\!]\,(\gamma) = [\![\Gamma \vdash e' : S]\!]\,(\gamma)$. We instead consider our small-step (call-by-value) semantics for $L\lambda^-$ (a subset of that of $L\lambda$), which has the following rule:

$$\frac{}{(\texttt{Call}\ (\texttt{Lam}\ x\ e)\ v) \mapsto_R [v/x]e}\ \text{red-beta}$$

In proving the soundness of reduction with respect to our denotational semantics, we need to consider the denotation of the right hand side of $\mapsto_R$ in red-beta, which has a syntactic substitution ($[v/x]e$). Therefore, we need to interpret syntactic substitutions in our denotational semantics. Given $\emptyset \vdash v : S$ and $\Gamma = (\emptyset, x : S)$, we define the environment $\left([\![v/x]\!]\,(\emptyset)\right) \in [\![\Gamma]\!]$ corresponding to the syntactic substitution $v/x$ by $[\![v/x]\!]\,(\emptyset) = \left(\emptyset, \left([\![\emptyset \vdash v : S]\!]\,(\emptyset)\right)/x\right)$. Note that this definition can be generalized to non-values (which, as mentioned in lec12, we'd want to do for call-by-name semantics), and to well-typed substitutions for entire typing contexts, but we're keeping our discussion simple and consistent with lec12 (at least for now).

We want to prove the following conjecture:

**Conjecture 1** (Semantic Soundness of Reduction in $L\lambda^-$). *If* $\emptyset \vdash e : S$ *and* $e \mapsto_R e'$, *then* $[\![e]\!]\,(\emptyset) = [\![e']\!]\,(\emptyset)$.

Let's start proving this conjecture by induction on the given reduction derivation $e \mapsto_R e'$. Let's case analyze rules concluding the given reduction derivation. We are especially interested in the red-beta case:

- **Case**

$$\frac{}{(\texttt{Call}\ (\texttt{Lam}\ x\ e_0)\ v) \mapsto_R [v/x]e_0}\ \text{red-beta}$$

| | |
|---|---|
| $e = (\texttt{Call}\ (\texttt{Lam}\ x\ e_0)\ v)$ | By inversion on reduction |
| $e' = [v/x]e_0$ | " |
| $\emptyset \vdash (\texttt{Lam}\ x\ e_0) : S_0 \to S$ | By inversion on typing |
| $\emptyset \vdash v : S_0$ | " |
| $\emptyset, x : S_0 \vdash e_0 : S$ | By inversion on typing |
| $\emptyset \vdash [v/x]e_0 : S$ | By lec12, Lemma 2 (Substitution (Generalized)) |

| | |
|---|---|
| $[\![\emptyset \vdash e : S]\!]\,(\emptyset)$ | |
| $= [\![\emptyset \vdash (\texttt{Call}\ (\texttt{Lam}\ x\ e_0)\ v) : S]\!]\,(\emptyset)$ | Current case |
| $= \left([\![\emptyset \vdash (\texttt{Lam}\ x\ e_0) : S_0 \to S]\!]\,(\emptyset)\right)\left([\![\emptyset \vdash v : S_0]\!]\,(\emptyset)\right)$ | By defn. of $[\![-]\!]$ |
| $= [\![\emptyset, x : S_0 \vdash e_0 : S]\!]\left(\emptyset, \left([\![\emptyset \vdash v : S_0]\!]\,(\emptyset)\right)/x\right)$ | By defn. of $[\![-]\!]$ |
| $= [\![\emptyset, x : S_0 \vdash e_0 : S]\!]\left([\![v/x]\!]\,(\emptyset)\right)$ | By defn. of $[\![-]\!]$ |
| $\overset{?}{=} [\![\emptyset \vdash [v/x]e_0 : S]\!]\,(\emptyset)$ | **I don't know** |
| $= [\![\emptyset \vdash e' : S]\!]\,(\emptyset)$ | Current case |

How do we justify the equality with a question mark ($\overset{?}{=}$), above? We need a key lemma stating that, the denotation of the result of a syntactic substitution is equal to the denotation of the expression into which we syntactically substitute (that is, the conclusion of lec12, Lemma 2 (Substitution (Generalized))) *at the denotation of the syntactic substitution itself*, that is, syntactic substitution and semantic substitution commute. However, because the conclusion of lec12, Lemma 2 (Substitution (Generalized)) has the contexts $\Gamma_L$ and $\Gamma_R$, we need to generalize syntactic substitutions and their denotations to entire typing contexts. Below, we define generalized call-by-value syntactic substitutions $\rho$:

$$\frac{}{\Gamma_0 \vdash \emptyset : \emptyset}\ \text{Empty}\rho \qquad \frac{\Gamma_0 \vdash \rho : \Gamma \quad \Gamma_0 \vdash v : T \quad x \notin \mathit{dom}(\Gamma)}{\Gamma_0 \vdash (\rho, v/x) : (\Gamma, x : T)}\ \text{Entry}\rho$$

Given $\Gamma \vdash e : S$ and $\Gamma_0 \vdash \rho : \Gamma$, we define the application $[\rho]e$ of $\rho$ to $e$ inductively by

$$[\emptyset]e = e$$
$$[\rho_0, v/x]e = [\rho_0]([v/x]e)$$

where $[v/x]e$ is our usual substitution operation. We can prove a generalized substitution lemma: if $\Gamma \vdash e : S$ and $\Gamma_0 \vdash \rho : \Gamma$, then $\Gamma_0 \vdash [\rho]e : S$. Given a typing context $\Gamma$, define the identity syntactic substitution $id_\Gamma$ by induction on $\Gamma$:

$$id_\emptyset = \emptyset$$
$$id_{(\Gamma, x:T)} = (id_\Gamma), x/x$$

It is straightforward to prove that $\Gamma \vdash id_\Gamma : \Gamma$, as well as that the identity syntactic substitution is an identity operation on well-typed expressions: if $\Gamma \vdash e : S$, then $[id_\Gamma]e = e$. Finally, given $\Gamma_0 \vdash \rho : \Gamma$ and $\vdash \gamma_0 : \Gamma_0$ we define the denotation $[\![\rho]\!](\gamma_0)$ of $\rho$ at $\gamma_0$ by structural induction on $\rho$:

$$[\![\emptyset]\!](\gamma_0) = \emptyset$$
$$[\![\rho_0, v/x]\!](\gamma_0) = \big([\![\rho_0]\!](\gamma_0)\big), ([\![v]\!]\gamma_0)/x$$

where we understand $[\![v]\!]\gamma_0$ to denote the typing derivation of $v$ (obtained by inversion on the derivation of the syntactic substitution typing) at environment $\gamma_0$.

**Conjecture 2** (Syntactic and Semantic Substitution Commute).
*If* $\Gamma_L, x : T, \Gamma_R \vdash e_1 : S$ *and* $\emptyset \vdash v_2 : T$ *and* $\emptyset : \gamma_L : \Gamma_L$ *and* $\emptyset : \gamma_R : \Gamma_R$*, then*

$$[\![\Gamma_L, \Gamma_R \vdash [id_{\Gamma_L}, v_2/x, id_{\Gamma_R}]e_1 : S]\!](\gamma_L, \gamma_R) = [\![\Gamma_L, x : T, \Gamma_R \vdash e_1 : S]\!]([\![id_{\Gamma_L}, v_2/x, id_{\Gamma_R}]\!](\gamma_L, \gamma_R))$$

We can use Conjecture 2 to justify the $\overset{?}{=}$ appearing in the above attempt to prove Conjecture 1. I haven't completed the proofs of these conjectures (if they exist), but I expect the conjectures to be true, and their proofs straightforward, if a bit tedious. However, Conjectures 1 and 2 can be generalized (see Conjectures 3 and 4 below) and extended to stepping (see Conjecture 5 below), so we might as well prove these ones instead. **Exercise**: prove Conjectures 3, 4, and 5 below.

**Conjecture 3** (Syntactic and Semantic Substitution Commute (Generalized)).
*If* $\Gamma \vdash e : S$ *and* $\Gamma_0 \vdash \rho : \Gamma$ *and* $\vdash \gamma_0 : \Gamma_0$*, then* $[\![\Gamma_0 \vdash [\rho]e : S]\!](\gamma_0) = [\![\Gamma \vdash e : S]\!]\big([\![\rho]\!](\gamma_0)\big)$.

**Conjecture 4** (Semantic Soundness of Reduction in L$\lambda^-$(Generalized)).
*If* $\Gamma \vdash e : S$ *and* $e \mapsto_R e'$ *and* $\vdash \gamma : \Gamma$*, then* $[\![e]\!](\gamma) = [\![e']\!](\gamma)$.

**Conjecture 5** (Semantic Soundness of Stepping in L$\lambda^-$).
*If* $\Gamma \vdash e : S$ *and* $e \mapsto e'$ *and* $\vdash \gamma : \Gamma$*, then* $[\![e]\!](\gamma) = [\![e']\!](\gamma)$.

# 8 Recursion and domain theory

When we add recursive expressions to our language, we enter a fallen world. We can no longer simply interpret types as sets of values. Why? Because there are recursive expressions that do not terminate to a value. The standard denotational way to model non-termination is *domain theory*. We will not get into the technical weeds, but briefly explain some of the key concepts.

Because some expressions may not terminate, we need to model the gradual approximation of the denotation of a well-typed expression $\Gamma \vdash e : S$ (applied to a $\Gamma$-environment), built up from a special element $\bot_{[\![S]\!]}$ associated with type $S$, called the *bottom element* of $S$, that represents *non-terminating* expressions of type $S$. This element $\bot_{[\![S]\!]}$ represents a semantic value with the least possible amount of information, and semantic values having its information, *but possibly more*, lie above it in a *partial ordering* $\sqsubseteq_{[\![S]\!]}$ (if $d \sqsubseteq_D d'$, then "$d'$ lies above $d$" in partial ordering $\sqsubseteq_D$). A (binary) relation $\sqsubseteq_D \subseteq D \times D$ is defined to be a *partial order* on set $D$ if it is reflexive ($d \sqsubseteq_D d$ for all $d \in D$), anti-symmetric ($d \sqsubseteq_D d'$ and $d' \sqsubseteq_D d$ implies $d = d'$) and transitive ($d_0 \sqsubseteq_D d_1$ and $d_1 \sqsubseteq_D d_2$ implies $d_0 \sqsubseteq_D d_2$). Given a chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots$ in partially ordered set $D$, we define a *least upper bound* of the chain to be an element $d \in D$ such that, for all $k \in \mathbb{N}$, we have $d_k \sqsubseteq_D d$ and, for any $d' \in D$ such that (for all $k \in \mathbb{N}$) $d_k \sqsubseteq_D d'$, we have $d \sqsubseteq_D d'$. A *domain* is defined to be a triple $(D, \sqsubseteq_D, \bot_D)$ where $(D, \sqsubseteq_D)$ is a chain-complete partially ordered set (that is, is a partially ordered set such that every chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D d_2 \sqsubseteq_D \cdots$ in $D$ has a least upper bound $d$ in $D$) and $\bot_D \in D$ and, for all $d \in D$, we have $\bot_D \sqsubseteq_D d$.

To define the denotation of a recursive expression (typing derivation)

$$\frac{\Gamma, x : S \vdash e : S}{\Gamma \vdash (\mathrm{Rec}\ x\ e) : S}$$

at $\gamma \in [\![\Gamma]\!]$, we compute the fixed point of a certain function, namely,

$$\xi = \big(d \longmapsto [\![\Gamma, x : S \vdash e : S]\!]\,(\gamma, d/x)\big)$$

that is known to have a fixed point because it is proven *continuous* (and that continuous functions have fixed points). That continuous functions on domains have a fixed point ($d$ is said to be a *fixed point* of $f$ if $f(d) = d$) is usually called a *fixed point theorem*. A function on domains is defined to be *continuous* if it *both* respects partial orders (that is, if it is *monotone*: the function $f : (D, \sqsubseteq_D) \to (E, \sqsubseteq_E)$ on partially ordered sets is defined to be *monotone* if $f(d) \sqsubseteq_E f(d')$ whenever $d \sqsubseteq_D d'$) *and* commutes with least upper bounds of chains. A domain function $f : (D, \sqsubseteq_D, \bot_D) \to (E, \sqsubseteq, \bot_E)$ is defined to *commute with least upper bounds of chains* if, for any chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D d_2 \sqsubseteq_D \cdots$ in $D$, applying $f$ to this chain's least upper bound is equivalent to taking the least upper bound of the chain $f(d_0) \sqsubseteq_E f(d_1) \sqsubseteq_E f(d_2) \sqsubseteq_E \cdots$ in $E$. The fixed point of $\xi$ above can be constructed by taking the least upper bound of the following chain:

$$\begin{aligned}
\underbrace{\bot_{[\![S]\!]}}_{d_0} \sqsubseteq_{[\![S]\!]} & \underbrace{[\![\Gamma, x : S \vdash e : S]\!]\,(\gamma, d_0/x)}_{d_1} \\
\sqsubseteq_{[\![S]\!]} & \underbrace{[\![\Gamma, x : S \vdash e : S]\!]\,(\gamma, d_1/x)}_{d_2} \\
\sqsubseteq_{[\![S]\!]} & \cdots \\
\sqsubseteq_{[\![S]\!]} & \underbrace{[\![\Gamma, x : S \vdash e : S]\!]\,(\gamma, d_n/x)}_{d_{n+1}} \\
\sqsubseteq_{[\![S]\!]} & \cdots
\end{aligned}$$

An important property (among others not mentioned in these notes) that PL researchers using both denotational and operational semantics commonly prove is *computational adequacy*, which states that the denotational semantics are sound with respect to the operational semantics. This basically says that our denotational semantics have appropriate computational meaning.

An excellent source on domain-theoretic denotational semantics, that follows a similar line to these lecture notes, is Carl Gunter's textbook "Semantics of Programming Languages" [1]. The classic papers that inaugurated this area of study are by "A type-theoretical alternative to ISWIM, CUCH, OWHY" by Scott (1969) [3] and "LCF considered as a programming language" by Plotkin (1977) [2].

## 9 Category theory

We secretly gave our denotational semantics a fairly *categorical* flavor, in the sense of the mathematical field of *category theory*. Category theory emphasizes *arrows*, and indeed we interpret typing derivations as set *arrows* (functions on sets). We may think of our set-theoretic interpretation of $L\lambda^{-}$as understanding it in terms of the "mathematical world" that is the category of sets and functions. Similarly, we understand $L\lambda^{-}$ plus recursion in the "world" given by the category of domains and continuous functions. When we try to understand these different languages and prove properties of them, we travel between these different mathematical worlds. In this way, category theory helps us organize the semantics of various programming languages and what it means, mathematically speaking, to add new features. When we add fancier features, we often need to travel to new worlds with fancier tools with which we can understand and prove properties of the new system.

## References

[1] C. A. Gunter. *Semantics of programming languages - structures and techniques*. Foundations of computing. MIT Press, 1993. ISBN 978-0-262-07143-7.

[2] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3): 223–255, 1977.

[3] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Written in 1969.

[4] J. G. Siek. Revisiting elementary denotational semantics. *CoRR*, abs/1707.03762, 2017. URL `http://arxiv.org/abs/1707.03762`.