Domain Theory

CS 6520, Spring 2000

1 Function Denotations

From the previous lecture, we know that the denotation of

$$\lambda x.(x+1)$$

might be derived as

$$\underline{\lambda} e \underline{\lambda} v.(v+1)$$

using a semantics with environments, but without continuations. The line beneath the lambda reminds us that it is a mathematical lambda, rather than a source-expression lambda. Since the lambdas are simply shorthand for set descriptions, we might describe the above set more directly as

$$\{\langle e, \{\langle v, v+1\rangle \mid \forall v \in \mathit{Num}\}\rangle \mid \forall e \in \mathit{Env}\}$$

or we could use a mixture of notations:

$$\underline{\lambda} e.\{\langle v, v+1\rangle \mid \forall v \in \mathit{Num}\}$$

Now consider the denotation of

$$\lambda f.f(1)$$

It should be a function that takes a function as its argument:

$$\underline{\lambda} e.\{\langle f, f\underline{(}1\underline{)}\rangle \mid \forall f \in \mathit{Num} \rightarrow \mathit{Num}\}$$

The parentheses are underlined to emphasize that they represent application in the mathematical world. Just as each v in the previous denotation stood for a mathematical number, each f in the above denotation stands for a function from numbers to numbers.

> Note: Every object in the mathematical world belongs to a specific set, and this set is mentioned explicitly in quantifications. In the previous lecture, we omitted most set declarations for simplicity, but this time they are central to the questions at hand.

Once again, we can unwind the function notation to clarify that each f is really a set of number pairs:

$$\underline{\lambda} e.\{\langle f, v\rangle \mid \forall g \in \mathit{Num} \rightarrow \mathit{Num}, 1 \notin \operatorname{dom}(g), f = g \cup \{\langle 1, v\rangle\}\}$$

2 Recursive Functions

What is the denotation of of the following expression?

$$\lambda f.f(f)$$

It would have to be something like

$$\underline{\lambda} e.\{\langle f, f\underline{(}f\underline{)}\rangle \mid \forall f \in\ ? \rightarrow \mathit{Num}\}$$

but we see right away a problem defining the set containing f. It would have to be a set of functions whose domain is the set itself — but set theory disallows sets that contain themselves!

Although the expression $\lambda f.f(f)$ may not be directly useful in practice, we now it as a step towards expressing recursive functions with the **Y** combinator. Indeed, the problem we see above looks similar to the problems that we encountered when trying to define **Y** in a simply-typed lambda calculus. Although our "types" in the mathematical world are fundamentally different from the "types" we studied in the syntactic world, the path to recursion in denotational semantics is again to introduce explicit syntax for recursion that is handled in a special way.

We will introduce a fix form into our language. It takes a function and produces a new function that is its fixpoint. For the purpose of trying out fix, we will also throw in **if** constructs, comparisons, etc. For example, the expression

$$\mathsf{fix}(\lambda f \lambda n\ .\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n-1))$$

denotes the (mathematical) factorial function, which is

$$\underline{\lambda} e.\{\langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,6\rangle, \langle 4,24\rangle, \ldots\}$$

But how do we define the semantics of fix expressions, mathematically, to acheive this result?

In order to define the mathematical meaning of fix, we need new mathematical machinery. Fortunately, Dana Scott developed it for us in the early 70s; he called it *domain theory.*

3 Domain Theory

In domain theory, instead of belonging to a particular set, every object belongs to a particular *domain.* A domain is roughly like a set, but it has additional structure. Specifically, a domain is a *partially-ordered set*, which is defined by the elements of the set, plus an order relation, $\sqsubseteq$. Also, every finite chain of elements in a domain must have a least upper bound, where a *chain* is an ordered set of elements $x_1, x_2, \ldots x_n$, where $x_1 \sqsubseteq x_2 \sqsubseteq \ldots x_n$.

For the purposes of assigning denotations to expressions, we want to think of the order relation $x \sqsubseteq y$ as "x provides no more information than y". To understand this definition of domains, consider the domain of boolean values:

- Clearly, this domain should contain at least **true** and **false**. The information **true** is exclusive to the information **false**, so these two elements are not related by $\sqsubseteq$. In contrast, **true** provides the same amount of information as **true**, so **true** $\sqsubseteq$ **true**, and the least upper bound of the chain $\{\mathbf{true}\}$ is obviously **true**. The same holds for **false**.

- The empty chain $\{\}$ also needs a least upper bound. Although **true** and **false** are both upper bounds, neither is less than the other. Hence, our domain needs one more element, written $\perp$ and pronounced "bottom". The new element $\perp$ means "no information." It is defined so that $\perp \sqsubseteq$ **true** and $\perp \sqsubseteq$ **false**. Thus, $\perp$ is the least upper bound of $\{\}$.

To summarize: the domain of booleans, *Bool*, is the set $\{\perp, \mathbf{true}, \mathbf{false}\}$ combined with an order relation $\sqsubseteq$ defined as the transitive-reflexive closure of $\{\langle \perp, \mathbf{true}\rangle, \langle \perp, \mathbf{false}\rangle\}$. This domain is often illustrated with the following diagram:

$$\begin{array}{ccc} \mathbf{false} & & \mathbf{true} \\ \backslash & & / \\ & \perp & \end{array}$$

The diagram shows both the elements of the set and the ordering relation (implicit in the vertical layout of the elements).

The domain of natural numbers, *Nat*, can be similarly illustrated:

$$\begin{array}{cccc} 1 & 2 & 3 & \ldots \\ \diagdown & \backslash & | & \ldots \\ & & \perp & \end{array}$$

Both the domain of booleans and the domain of natural numbers are *flat* (and somewhat boring); every chain contains at most one boolean number or number. But the domain representation of functions is not flat.

The domain of $Bool \rightarrow Bool$ roughly consists of mappings from $Bool$ to $Bool$. The complete set of mappings includes the following:

$$\{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\bot\rangle\}$$
$$\{\langle\bot,\mathbf{false}\rangle,\langle\mathbf{false},\mathbf{false}\rangle,\langle\mathbf{true},\mathbf{false}\rangle\}$$
$$\{\langle\bot,\mathbf{true}\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$
$$\{\langle\bot,\bot\rangle,\langle\mathbf{false},\mathbf{false}\rangle,\langle\mathbf{true},\mathbf{false}\rangle\}$$
$$\{\langle\bot,\bot\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$
$$\{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\mathbf{false}\rangle\}$$
$$\{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$

plus twenty other elements. But not all of these mappings will be included in the domain, as we will see further below.

We can order the mappings by thinking of them as descriptions of how much we know about a function. For example,

$$F_0 = \{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\bot\rangle\}$$

describes a function where we do not know the result for any input. In contrast,

$$F_2 = \{\langle\bot,\mathbf{true}\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$

describes a function that we know always returns **true**. Since F_2 provdies strictly more information than F_0, we order $F_0 \sqsubseteq F_2$. In fact, since F_0 provides no information, $F_0 \sqsubseteq F$ for any F in the domain. In other words, F_0 is the "bottom" for the domain $Bool \rightarrow Bool$, and we sometimes use the symbol $\bot$ to mean F_0 when it is clear that we mean the least element of $Bool \rightarrow Bool$.

There are several elements in the domain between F_0 and F_2. For example,

$$F_1 = \{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$

and

$$F_1' = \{\langle\bot,\bot\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\bot\rangle\}$$

both provide more information than F_0, but less than F_2. Thus, $F_0 \sqsubseteq F_1 \sqsubseteq F_2$ and $F_0 \sqsubseteq F_1' \sqsubseteq F_2$. Note, however, that F_1 and F_1' are incomparable; neither provides all of the information of the other. The domain of $Bool \rightarrow Bool$ is clearly not flat:

$$\begin{array}{ccccccc} & \ldots & & F_2 & & \ldots & \\ & \ldots & / & & \backslash & \ldots & \\ \ldots & & F_1' & & F_1' & & \ldots \\ & \ldots & \backslash & & / & \ldots & \\ & & & F_0 & & & \end{array}$$

Now consider the mapping

$$F_{weird} = \{\langle\bot,\mathbf{true}\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}$$

This mapping says that the function returns **true**, even when we don't know what the argument is (i.e., it is $\bot$). But it also say that we don't know what the function returns for a **false** input. This does not make sense for functions as programs. Intuitively, if a program returns **true** for an unknown input, it must return **true** for any input, including **false**!

Nonsensical elements such as F_{weird} are not part of the domain $Bool \rightarrow Bool$. They do not "respect the structure" of the input and output domains. More technically, they are not *montonic*. A monotonic function is one where $f(a) \sqsubseteq f(b)$ when $a \sqsubseteq b$. In F_{weird}, we see that $F_{weird}(\bot) \not\sqsubseteq F_{weird}(\mathbf{false})$, even though $\bot \sqsubseteq \mathbf{false}$. In contrast, F_0, F_2, F_1, and F_1' are all monotonic.

The domain $Nat \rightarrow Nat$ can be derived in the same way as $Bool \rightarrow Bool$, since the monotonicity requirement is abstract with respect to the set and its $\sqsubseteq$ relation. Furthermore, we can use the same techniques to define domains like $Bool \rightarrow Bool \rightarrow Nat$, and so on.

4 Fixpoints

The key to assigning denotations to fix expressions is to map expressions to elements of domains, and then exploit a process that is guaranteed to find a fixpoints of functions.

First, let's look at the denotation of a few expressions in the domain $Bool \to Bool$:[1]

$$\begin{aligned}
[\![\lambda b.\mathbf{true}]\!] &= \{\langle\bot,\mathbf{true}\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}\\
[\![\lambda b.\bot]\!] &= \{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\bot\rangle\}\\
[\![\lambda b.b]\!] &= \{\langle\bot,\bot\rangle,\langle\mathbf{false},\mathbf{false}\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}\\
[\![\lambda b.\mathbf{if}\ b\ \mathbf{then\ true\ else}\ \bot]\!] &= \{\langle\bot,\bot\rangle,\langle\mathbf{false},\bot\rangle,\langle\mathbf{true},\mathbf{true}\rangle\}\\
[\![\lambda b.\mathbf{if}\ b\ \mathbf{then\ false\ else\ true}]\!] &= \{\langle\bot,\bot\rangle,\langle\mathbf{false},\mathbf{true}\rangle,\langle\mathbf{true},\mathbf{false}\rangle\}
\end{aligned}$$

In the last example, $\bot$ maps to $\bot$ because the function will not know what to do with an unknown argument. In the first example, though, every argument is mapped to **true**, so the function could even return a value for an unknown argument. Note that all of the mappings above are monotonic. (Don't fall into the trap of thinking that $\mathbf{false} \sqsubseteq \mathbf{true}$ or $1 \sqsubseteq 2$. The objects **false**, **true**, 1, and 2 are all incomparable with $\sqsubseteq$.)

Now consider the expression

$$\mathsf{fix}(\lambda n.5)$$

How do we assign a denotation to the expression? It is the fixpoint of the denotation of $(\lambda n.5)$. In this case, the denotation of fix's argument is obvious, but suppose it was not. We know that it could be an element in the domain $Nat \to Nat$, but which one? A first guess might be

$$\{\langle\bot,\bot\rangle,\langle 0,\bot\rangle,\langle 1,\bot\rangle,\langle 2,\bot\rangle,\ldots\}$$

This is clearly an approximation of the right answer, because it says "we don't know what the function will return for any input". We might refine this guess by testing $(\lambda n.5)$ on an input — assuming the least information input! So we provide $\bot$ to the function, and discoever that it returns 5. Thus, we have an improvement in our approximation for the denotation of $(\lambda n.5)$:

$$\{\langle\bot,5\rangle,\langle 0,\bot\rangle,\langle 1,\bot\rangle,\langle 2,\bot\rangle,\ldots\}$$

But this mapping is not in the domain, since it isn't monotonic. We know that the function must return at least as much information as 5 (and 5 is maximal information) if we give it more infomation than $\bot$ (i.e., an number). Thus, the correct denotation must be

$$\{\langle\bot,5\rangle,\langle 0,5\rangle,\langle 1,5\rangle,\langle 2,5\rangle,\ldots\}$$

If we then consider applying $(\lambda n.5)$, which was the result we obtained by trying $\bot$, we get 5 back. So 5 is a fixpoint of the denotation of $(\lambda n.5)$, and therefore 5 is the denotation of $\mathsf{fix}(\lambda n.5)$.

Here is another example:

$$\mathbf{let}\ G = (\lambda f \lambda n\ .\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n-1))\ \mathbf{in}\ \mathsf{fix}G$$

This example is more complex. The denotation of G resides in the domain $(Nat \to Nat) \to (Nat \to Nat)$, and the denotation of $\mathsf{fix}G$ resides in $Nat \to Nat$. But we can use the same stategy as before; initially, we guess an approximation of G that has no information:

$$\{\langle\bot,\bot\rangle,\langle f_1,\bot\rangle,\langle f_2,\bot\rangle,\ldots\}$$

where $f_1, f_2, \ldots$ are elements of the domain $Nat \to Nat$. If we supply $\bot$ to G, we get a function that is at least defined to return 1 for $n = 0$. Thus, we have

$$\{\langle\bot,F_0\rangle,\langle f_1,F_0\rangle,\langle f_2,F_0\rangle,\ldots\}$$

as an approximation of G, where

$$F_0 = \{\langle\bot,\bot\rangle,\langle 0,1\rangle,\langle 1,\bot\rangle,\langle 2,\bot\rangle,\ldots$$

[1] For now, we ignore the leading $\underline{\lambda}e$.

Remember that we are trying to find a fixpoint of G, which is a function F such that the denotation of $(\lambda n\ .\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * F(n-1))$ is the same as F. Whatever this F is, we know it must be approximated by F_0, because that's what we got without knowing anything about the argument to G.

So, if we know at least F_0, what if we supply F_0 to G? Of course, we get a function that is still defined on 0, but it's also defined on 1, since it calls F_0 on $n-1$. Now, G is approximated slightly better:

$$\{\langle\perp, F_0\rangle, \langle f_1, F_0\rangle, \langle f_2, F_0\rangle, \ldots, \langle F_0, F_1\rangle, \langle f_3, F_1\rangle, \langle f_2, F_1\rangle, \ldots, \ldots\}$$

assuming $F_0 \not\sqsubseteq f_1, f_2, \ldots$ and $F_0 \sqsubseteq f_3, f_4, \ldots$, and

$$F_1 = \{\langle\perp,\perp\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 2,\perp\rangle, \ldots$$

This F_1 is an improved approximation of a fixpoint of G. We can keep going, and we get a sequence of Fs:

$$F_3 = \{\langle\perp,\perp\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,\perp\rangle, \langle 4,\perp\rangle, \ldots\}$$

$$F_4 = \{\langle\perp,\perp\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,6\rangle, \langle 4,\perp\rangle, \ldots\}$$

$$F_5 = \{\langle\perp,\perp\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,6\rangle, \langle 4,24\rangle, \ldots\}$$

Each F_n provides more information than it's predecessors. We could keep going to get as much information as we want. The denotation of $\mathsf{fix}G$, then, is the the limit of the sequence of F_ns.

5 Least Fixpoints

Now try

$$\mathsf{fix}\lambda f \lambda n. f(n)$$

What is it's denotation? We can start with an approximation for the denotation of the function

$$[\![\lambda f \lambda n. f(n)]\!] = \{\langle\perp,\perp\rangle, \langle f_1,\perp\rangle, \langle f_2,\perp\rangle, \ldots\}$$

but this time, supplying $\perp$ as f produces $\perp$ — so we've found a fixpoint immediately! $[\![\mathsf{fix}\lambda f \lambda n. f(n)]\!] = \perp$. To connect intution with the theory, we see that $\perp$ corresponds to an infinite loop; this makes sense, because we never know what an infinite loop will return.

But wait, the identity function — $\{\langle\perp,\perp\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \ldots\}$ — is also a fixpoint of $\lambda f \lambda n. f(n)$. In fact, any function in $Nat \to Nat$ is a fixpoint. So why should $[\![\mathsf{fix}\lambda f \lambda n. f(n)]\!]$ be $\perp$ instead of any other function?

The answer is that $\perp$ is the *least* fixpoint. It satisfies the constraints of $\mathsf{fix}\lambda f \lambda n. f(n)$ in a minimal way, without inventing new information. Intuitively, this is the meaning we want, because it's the meaning we can implement in a programming languages.

Fortunately, the algorithm we sketched for finding a fixpoint always produces the least fixpoint, and a fixpoint always exists, provably. Therefore, we can define the semantics of a fix expression by

$$[\![\mathsf{fix}\, a]\!] \quad = \quad \underline{\lambda} e.\text{LeastFixpoint}([\![a]\!]e)$$