What is the meaning of a Haskell program?

Dustin Mulcahey

Every programming language has *syntax* and *semantics*. The specification of syntax is typically given using BNF notation.

Every programming language has *syntax* and *semantics*. The specification of syntax is typically given using BNF notation. For example, here is the beginning of the Haskell language spec:

```
program ->   {lexeme | whitespace }
lexeme  ->   qvarid | qconid | qvarsym | qconsym
             | literal | special | reservedop | reservedid
literal ->   integer | float | char | string
special ->  ( | ) | , | ; | [ | ] | `| { | }
whitespace  ->   whitestuff {whitestuff}
```

Syntax dictates what strings of text are valid programs. That is, given a syntax specification and a string, one can say whether or not that string is a valid program according to the syntax.

Syntax dictates what strings of text are valid programs. That is, given a syntax specification and a string, one can say whether or not that string is a valid program according to the syntax.

One payoff of specifying your programming language's syntax is that there are programs that take syntax specifications and output programs that check whether or not other strings are programs (according to the input syntax). For example: yacc (yet another compiler compiler).

Syntax dictates what strings of text are valid programs. That is, given a syntax specification and a string, one can say whether or not that string is a valid program according to the syntax.

One payoff of specifying your programming language's syntax is that there are programs that take syntax specifications and output programs that check whether or not other strings are programs (according to the input syntax). For example: yacc (yet another compiler compiler).

As a bonus, these programs typically output an abstract syntax tree (stuffed into an appropriate datatype) that you can then traverse to either interpret the program or output compiled code.

The *semantics* of a programming language specify the *meaning* of individual programs.

The *semantics* of a programming language specify the *meaning* of individual programs.
For example, what is the meaning of the following snippet?

```
f :: Integer -> Integer
f x = x * x
```

Someone might say that the program on the previous slide means this:

```
Dustins-MacBook-Air:haskelltalk dustin$ ghci
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for he
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load square.hs
[1 of 1] Compiling Main                ( square.hs, interprete
Ok, modules loaded: Main.
*Main> f 4
16
```

That is, one could claim that the meaning of the program is precisely what the interpreter does with it. That is, the interpreter gave us a `f` that we could happily apply to things marked as type `Integer`

To get yet another meaning of the program from three slides back, I could spruce it up a bit:

```
import System.Environment

f :: Integer -> Integer
f x = x * x

main = getArgs >>= print . f  . read . head
```

GHC tells me that the meaning of this is:

```
0000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
0000010 11 00 00 00 c8 07 00 00 85 00 20 00 00 00 00 00
0000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
0000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 19 00 00 00 78 02 00 00
0000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
0000080 00 00 00 00 01 00 00 00 00 20 0f 00 00 00 00 00
0000090 00 00 00 00 00 00 00 00 00 20 0f 00 00 00 00 00
00000a0 07 00 00 00 05 00 00 00 07 00 00 00 00 00 00 00
00000b0 5f 5f 74 65 78 74 00 00 00 00 00 00 00 00 00 00
00000c0 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
```

... and on and on

Well, to be more pedantic, it was GHC version 7.4.2 that told me this - when I ran it on the same computer that is projecting these slides right now. I think it might tell me something else on a different machine. Well, a different GHC would... um...

The previous meanings of my little program are all in terms of *operational semantics*. In operational semantics, we say that the meaning of a program is the output of an interpreter. To define the semantics, one must define the interpreter.

The previous meanings of my little program are all in terms of *operational semantics*. In operational semantics, we say that the meaning of a program is the output of an interpreter. To define the semantics, one must define the interpreter.

Now, it wouldn't be very good for the meaning of Haskell programs to be equal to the compiler output on a specific machine! In actual operational semantics, an abstract machine is mathematically defined. So you still cannot escape from math.

As a wannabe mathematician, I would say that the program from many slides ago means this:

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x^2$$

where $\mathbb{Z}$ denotes the set of all integers. Pick your favorite mathematical foundation to construct *this* (more on this later).

As a wannabe mathematician, I would say that the program from many slides ago means this:

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x^2$$

where $\mathbb{Z}$ denotes the set of all integers. Pick your favorite mathematical foundation to construct *this* (more on this later).

To sound even more sophisticated, we could say that the above mathematical object is the *denotation* of my little program.

In *denotational semantics*, we assign a mathematical object to each valid program. In the case of my little program, it was the function that takes an integer and squares it.

As a big of notational pedantry, we like to distinguish the `f` from my code and the f from math. As such, we'll write

$$[\![\texttt{f}]\!] : \mathbb{Z} \to \mathbb{Z}$$
$$[\![\texttt{f}]\!](x) = x^2$$

instead.

As a big of notational pedantry, we like to distinguish the `f` from my code and the f from math. As such, we'll write

$$[\![\texttt{f}]\!] : \mathbb{Z} \to \mathbb{Z}$$
$$[\![\texttt{f}]\!](x) = x^2$$

instead.

That is, we surround any piece of code by double brackets to refer to its denotation.

Why did we just do all of that?

Why did we just do all of that?

Well, now we can do things like this:

Why did we just do all of that?

Well, now we can do things like this:

Theorem

`f` *will never output* 2 *for any input.*

Proof.

Suppose that `f` outputs 2 for an input x. Then $[\![\texttt{f}]\!](x) = 2$ for some $x \in \mathbb{Z}$. Hence, there exists an $x \in \mathbb{Z}$ such that $x^2 = 2$, which is a contradiction.

Okay, that wasn't *that* exciting. The following paper is much more interesting:

Okay, that wasn't *that* exciting. The following paper is much more interesting:

HALO: Haskell to Logic through Denotational Semantics, Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosen, POPL 2013.

Idea:

```
c_head :: Statement
c_head = head ::: CF :&: Pred (not . null) --> CF
```

Here, `c_head` is a contract that says that head takes things that are crash-free and non-empty to things that are crash-free.

Quick overview:

- Translate the Haskell program to a first order logic (FOL) theory. Translate contracts to a FOL statement.

Quick overview:

- Translate the Haskell program to a first order logic (FOL) theory. Translate contracts to a FOL statement.
- Invoke theorem checker to verify that the FOL statement is a logical conclusion of the FOL theory.

Quick overview:

- Translate the Haskell program to a first order logic (FOL) theory. Translate contracts to a FOL statement.
- Invoke theorem checker to verify that the FOL statement is a logical conclusion of the FOL theory.
- Denotational semantics is employed to prove that if the translation of a contract follows from the translation of a program, then the actual program satisfies the actual contract.

Now that we've had an introduction to denotational semantics and its uses, let's get into it in more detail.

Attempt 1: Provied a recursively defined map of Haskell programs to mathematical objects. Here, types go to sets and functions between types go to functions between sets. This is the notion that a type is the set of all things of that type.

For example,

```
f :: Integer -> Integer
f x = x^2
```

would map to

$$[\![\texttt{f}]\!] : \mathbb{Z} \to \mathbb{Z}$$
$$[\![\texttt{f}]\!](x) = x^2$$

since $(*)$ maps to the actual function $* : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

We can simplify our thinking by expressing programs *morphologically*. For instance, I can think of the above program as the composite:

```
diag x = (x,x)
f = (uncurry (*)) . diag
```

We can simplify our thinking by expressing programs *morphologically*. For instance, I can think of the above program as the composite:

```
diag x = (x,x)
f = (uncurry (*)) . diag
```

which, as a category theorist, I would write as:

$$\texttt{Integer} \xrightarrow{\texttt{diag}} \texttt{Integer} \times \texttt{Integer} \xrightarrow{(*)} \texttt{Integer}$$

We can simplify our thinking by expressing programs *morphologically*. For instance, I can think of the above program as the composite:

```
diag x = (x,x)
f = (uncurry (*)) . diag
```

which, as a category theorist, I would write as:

$$\texttt{Integer} \xrightarrow{\texttt{diag}} \texttt{Integer} \times \texttt{Integer} \xrightarrow{(*)} \texttt{Integer}$$

which would then get mapped to this by our denotational semantics:

$$[\![\texttt{Integer}]\!] \xrightarrow{[\![\texttt{diag}]\!]} [\![\texttt{Integer}]\!] \times [\![\texttt{Integer}]\!] \xrightarrow{[\![(*)]\!]} [\![\texttt{Integer}]\!]$$

We can simplify our thinking by expressing programs *morphologically*. For instance, I can think of the above program as the composite:

```
diag x = (x,x)
f = (uncurry (*)) . diag
```

which, as a category theorist, I would write as:

$$\texttt{Integer} \xrightarrow{\texttt{diag}} \texttt{Integer} \times \texttt{Integer} \xrightarrow{(*)} \texttt{Integer}$$

which would then get mapped to this by our denotational semantics:

$$[\![\texttt{Integer}]\!] \xrightarrow{[\![\texttt{diag}]\!]} [\![\texttt{Integer}]\!] \times [\![\texttt{Integer}]\!] \xrightarrow{[\![(*)]\!]} [\![\texttt{Integer}]\!]$$

and then, by an appropriate definition, this would be equal to:

$$\mathbb{Z} \xrightarrow{\Delta} \mathbb{Z} \times \mathbb{Z} \xrightarrow{*} \mathbb{Z}$$

Here, we see that our intended $[\![-]\!]$ maps Haskell types to sets and Haskell programs to functions between sets.

Here, we see that our intended $[\![-]\!]$ maps Haskell types to sets and Haskell programs to functions between sets.

Additionally, we want $[\![-]\!]$ to respect composites (which I sort of did implicitely on the last slide). More explicitely:

$$\begin{aligned}
[\![\texttt{f}]\!] &= [\![\texttt{(uncurry (*)) . diag}]\!] \\
&= [\![\texttt{(uncurry (*))}]\!] \circ [\![\texttt{diag}]\!] \\
&= * \circ \Delta
\end{aligned}$$

This means that $[\![-]\!]$ ought to be a *functor*. What's that again?

This means that $[\![-]\!]$ ought to be a *functor*. What's that again?

Definition

A category $\mathcal{C}$ consists of objects and morphisms. Given morphisms $f : X \to Y$ and $g : Y \to Z$, there is a composite $g \circ f : X \to Z$. This is an associative operation. Additionally, every object X has an identity map $1_X : X \to X$ which acts as a unit for composition, that is, $f \circ 1_X = f = 1_Y \circ f$.

Instead of giving tons of examples, I want to focus on two that are relevant to the discussion (with another to come soon!).

- `Hask` - the Haskell category whose objects are Haskell types and morphisms are Haskell programs.

Instead of giving tons of examples, I want to focus on two that are relevant to the discussion (with another to come soon!).

- `Hask` - the Haskell category whose objects are Haskell types and morphisms are Haskell programs.
- Set - the category of sets, whose objects are sets and morphisms are functions between sets.

Definition

A functor $F : \mathcal{C} \to \mathcal{D}$ is a mapping of the objects of $\mathcal{C}$ to the objects of $\mathcal{D}$ and the morphisms of $\mathcal{C}$ to the morphisms of $\mathcal{D}$. Additionally, F takes identities to identities and composites to composites, that is, $F(1_X) = 1_{F(X)}$ and $F(g \circ f) = F(g) \circ F(f)$.

You already know several examples of functors, but I am guessing that most of them are *endofunctors* on the category `Hask`. (*endofunctor* is a fancy word for a functor that goes from a category to itself.)

You already know several examples of functors, but I am guessing that most of them are *endofunctors* on the category `Hask`. (*endofunctor* is a fancy word for a functor that goes from a category to itself.)

Examples:

- [] : `Hask` $\rightarrow$ `Hask` which takes a type `a` to `[a]` and a morphism `f :: a -> b` to `fmap f :: [a] -> [b]`

You already know several examples of functors, but I am guessing that most of them are *endofunctors* on the category `Hask`. (*endofunctor* is a fancy word for a functor that goes from a category to itself.)

Examples:

- [] : `Hask` $\rightarrow$ `Hask` which takes a type `a` to `[a]` and a morphism `f :: a -> b` to `fmap f :: [a] -> [b]`
- `Maybe` : `Hask` $\rightarrow$ `Hask`

You already know several examples of functors, but I am guessing that most of them are *endofunctors* on the category `Hask`. (*endofunctor* is a fancy word for a functor that goes from a category to itself.)

Examples:

- [] : `Hask` $\rightarrow$ `Hask` which takes a type `a` to `[a]` and a morphism `f :: a -> b` to `fmap f :: [a] -> [b]`
- `Maybe` : `Hask` $\rightarrow$ `Hask`
- `IO` : `Hask` $\rightarrow$ `Hask`

However, my dear audience, it is time to leave $\mathtt{Hask}$! It seems that denotational semantics is a functor:

$$[\![-]\!] : \mathtt{Hask} \to \mathsf{Set}$$

This is all well and good for my little program that squares integers. However, what about this monstrosity?

```
g :: Integer -> Integer
g x = (g x) + 1
```

This is all well and good for my little program that squares integers. However, what about this monstrosity?

```
g :: Integer -> Integer
g x = (g x) + 1
```

The problem:

- We would like $[\![\texttt{Integer}]\!] = \mathbb{Z}$

This is all well and good for my little program that squares integers. However, what about this monstrosity?

```
g :: Integer -> Integer
g x = (g x) + 1
```

The problem:

- We would like $[\![\texttt{Integer}]\!] = \mathbb{Z}$
- That would imply that $[\![\text{g}]\!] : \mathbb{Z} \to \mathbb{Z}$

This is all well and good for my little program that squares integers. However, what about this monstrosity?

```
g :: Integer -> Integer
g x = (g x) + 1
```

The problem:

- We would like $[\![\texttt{Integer}]\!] = \mathbb{Z}$
- That would imply that $[\![\texttt{g}]\!] : \mathbb{Z} \to \mathbb{Z}$
- However, there is no actual function $\mathbb{Z} \to \mathbb{Z}$ that satisfies the definition of g! If there was, then that would imply that $0 = 1$ in $\mathbb{Z}$ (this only happens in $\mathbb{Z}/1$, which is a pretty stupid group)

As we know, g diverges for any input. Yet it is still a program and therefore should still have meaning.

As we know, g diverges for any input. Yet it is still a program and therefore should still have meaning.

Moreover, we have the general program of providing meaning to recursive definitions that do not necessarily diverge. After all,

```
fact 0 = 1
fact n = n * (fact (n - 1))
```

does not break down into a compisite of functions for which we already have semantics. In other words, how can we define $[\![\texttt{fact}]\!]$ without appealing to $[\![\texttt{fact}]\!]$?
(We can solve this syntatically with the Y-combinator, but we are not going to do that.)

We have to “dress up” the sets that we map our types to. That is, we need a different *semantic domain*. Towards this, consider the elephant in the room:

```
Prelude> :type undefined
undefined :: forall a. a
```

We have to "dress up" the sets that we map our types to. That is, we need a different *semantic domain*. Towards this, consider the elephant in the room:

```
Prelude> :type undefined
undefined :: forall a. a
```

That is, `undefined` can be considered a member of any type whatsoever. So `Integer` "contains" not only things like $0, -1, 1, \ldots$ but also `undefined`. Don't believe me?

We have to "dress up" the sets that we map our types to. That is, we need a different *semantic domain*. Towards this, consider the elephant in the room:

```
Prelude> :type undefined
undefined :: forall a. a
```

That is, `undefined` can be considered a member of any type whatsoever. So `Integer` "contains" not only things like $0, -1, 1, \ldots$ but also `undefined`. Don't believe me?

```
h :: Integer -> Integer
h 0 = undefined
h x = x + 1
```

With this in mind, we are now going to spruce up $\mathbb{Z}$ a little bit. We are simply going to add an element $\perp$ to it! We will denote our new and improved $\mathbb{Z}$ as $\mathbb{Z}_{\perp}$. This process is called *l*ifting.

The idea is that $\perp$ corresponds to computations that are undefined or diverge.

With this in mind, we are now going to spruce up $\mathbb{Z}$ a little bit. We are simply going to add an element $\perp$ to it! We will denote our new and improved $\mathbb{Z}$ as $\mathbb{Z}_\perp$. This process is called *lifting*.

The idea is that $\perp$ corresponds to computations that are undefined or diverge.

Returning to our completely divergent g, we can denote this as:
$[\![\mathrm{g}]\!] : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$
$[\![\mathrm{g}]\!](x) = \perp$
Our partially divergent `h` can be denoted similarly!

Adding $\perp$ is only part of the "lifting" process. We are also going to give every set a *partial ordering*. Let's see what this means for $\mathbb{Z}$.

Recall that we started with $\mathbb{Z}$ and added $\perp$ to get $\mathbb{Z}_\perp$. We are now going to declare that $\perp$ is less than every integer in $\mathbb{Z}$. That is, any number has "more information" than $\perp$.

Recall that we started with $\mathbb{Z}$ and added $\bot$ to get $\mathbb{Z}_\bot$. We are now going to declare that $\bot$ is less than every integer in $\mathbb{Z}$. That is, any number has "more information" than $\bot$.

Formally, this is achieved by defining a binary relation on $\mathbb{Z}_\bot$ which we write as $\sqsubseteq$. The relation is defined as follows:

$$x \sqsubseteq y \text{ iff } x = y \text{ or } x = \bot$$

Recall that we started with $\mathbb{Z}$ and added $\bot$ to get $\mathbb{Z}_\bot$. We are now going to declare that $\bot$ is less than every integer in $\mathbb{Z}$. That is, any number has "more information" than $\bot$.

Formally, this is achieved by defining a binary relation on $\mathbb{Z}_\bot$ which we write as $\sqsubseteq$. The relation is defined as follows:

$$x \sqsubseteq y \text{ iff } x = y \text{ or } x = \bot$$

n.b.: Do not confuse this with the usual ordering on $\mathbb{Z}$! For instance, it is not the case that $1 \sqsubseteq 2$. In fact, they are incomparable under this ordering (hence the *partial* in partial ordering).

As long as I've given you the axioms for categories and functors, I might as well give you the ones for posets (partially ordered sets)!

As long as I've given you the axioms for categories and functors, I might as well give you the ones for posets (partially ordered sets)!

Definition

A relation $\sqsubseteq$ on a set X is a partial ordering if:

- $\forall x(x \sqsubseteq x)$

As long as I've given you the axioms for categories and functors, I might as well give you the ones for posets (partially ordered sets)!

Definition

A relation $\sqsubseteq$ on a set X is a partial ordering if:

- $\forall x(x \sqsubseteq x)$
- $\forall x, y(x \sqsubseteq y \text{ and } y \sqsubseteq x \text{ implies } x = y)$

As long as I've given you the axioms for categories and functors, I might as well give you the ones for posets (partially ordered sets)!

Definition

A relation $\sqsubseteq$ on a set X is a partial ordering if:

- $\forall x(x \sqsubseteq x)$
- $\forall x, y(x \sqsubseteq y \text{ and } y \sqsubseteq x \text{ implies } x = y)$
- $\forall x, y, z(x \sqsubseteq y \text{ and } y \sqsubseteq z \text{ implies } x \sqsubseteq z)$

Thus we move from the category of sets to the category of posets. The morphisms also change. Instead of arbitraty functions of sets, we demand that the functions be *monotonic*.

Thus we move from the category of sets to the category of posets. The morphisms also change. Instead of arbitraty functions of sets, we demand that the functions be *monotonic*.

Definition

A function $f : X \to Y$, where X, Y are posets, is monotonic if for all $x_1, x_2 \in X$:

$$x_1 \sqsubseteq_X x_2 \text{ implies } f(x_1) \sqsubseteq_Y f(x_2)$$

Thus we move from the category of sets to the category of posets. The morphisms also change. Instead of arbitraty functions of sets, we demand that the functions be *monotonic*.

Definition

A function $f : X \to Y$, where X, Y are posets, is monotonic if for all $x_1, x_2 \in X$:

$$x_1 \sqsubseteq_X x_2 \text{ implies } f(x_1) \sqsubseteq_Y f(x_2)$$

Intuition: A computable function preserves relative information content.

The preceeding slides seem to suggest that $[\![-]\!]$ is actually a functor from `Hask` to the category of partially ordered sets (morphisms are monotonic functions).

This still isn't quite right!

Strategy for denoting recursive functions:

Strategy for denoting recursive functions:

- For a recursively defined f, express f as the fixed point of a higher order function Φ. That is, $f = \Phi(f)$.

Strategy for denoting recursive functions:

- For a recursively defined f, express f as the fixed point of a higher order function Φ. That is, $f = \Phi(f)$.
- To do this, adjust our domains so that Φ always exists and has a "unique" fixed point.

Before we get into the hairy details, let's consider an example:
Consider our old friend:

```
fact 0 = 1
fact n = n * (fact (n-1))
```

Let's go ahead and define

$$\Phi(f) = \lambda n. \begin{cases} 1 \text{ if } n = 0 \\ n * f(n-1) \text{ otherwise} \end{cases}$$

Before we get into the hairy details, let's consider an example:
Consider our old friend:

```
fact 0 = 1
fact n = n * (fact (n-1))
```

Let's go ahead and define

$$\Phi(f) = \lambda n.\ \begin{cases} 1 \text{ if } n = 0 \\ n * f(n-1) \text{ otherwise} \end{cases}$$

Convince yourself that $\Phi([\![\texttt{fact}]\!]) = [\![\texttt{fact}]\!]$.

Those following along at home will note that $\Phi : \mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}} \to \mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}}$.

(Recall that for sets X, Y, X^Y is the set of all functions from X to Y)

Problem: what is the partial ordering on $\mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}}$? We would like an ordering such that $[\![\texttt{fact}]\!]$ is the *least* fixed point of Φ.

Those following along at home will note that $\Phi : \mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}} \to \mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}}$.

(Recall that for sets X, Y, X^Y is the set of all functions from X to Y)

Problem: what is the partial ordering on $\mathbb{Z}_{\perp}^{\mathbb{Z}_{\perp}}$? We would like an ordering such that $[\![\texttt{fact}]\!]$ is the *least* fixed point of Φ.

Then, the denotation of *any* recursive function will be the least fixed point of an appropriate Φ.

Theorem

Let X and Y be posets. Consider the sets

- $X \times Y = \{(x, y) | x \in X, y \in Y\}$
- $X^Y = \{f : X \to Y | f \text{ monotonic}\}$

These can be given partial orderings induced by the orderings on X and Y.

Theorem

Let X and Y be posets. Consider the sets

- $X \times Y = \{(x, y) | x \in X, y \in Y\}$
- $X^Y = \{f : X \to Y | f \text{ monotonic}\}$

These can be given partial orderings induced by the orderings on X and Y.

In fact, these become the universal *product* and *exponent* in the category of partially ordered sets. The universality is similar to the universality of `(a,b)` and `a->b` for types `a` and `b`.

Definition

Let $X \subseteq Y$ be a subset of a poset Y. An element $y \in Y$ is a least upper bound (lub) for X iff

- $\forall x \in X(x \sqsubseteq y)$ (y is an upper bound)
- If y' is another element with this property, then $y \sqsubseteq y'$ (y is the least upper bound)

Definition

Let $X \subseteq Y$ be a subset of a poset Y. An element $y \in Y$ is a least upper bound (lub) for X iff

- $\forall x \in X(x \sqsubseteq y)$ (y is an upper bound)
- If y' is another element with this property, then $y \sqsubseteq y'$ (y is the least upper bound)

Theorem

If X has a lub y, then it is unique. We denote it by $\bigsqcup X$.

Definition

Let $X \subseteq Y$ be a subset of a poset Y. An element $y \in Y$ is a least upper bound (lub) for X iff

- $\forall x \in X(x \sqsubseteq y)$ (y is an upper bound)
- If y' is another element with this property, then $y \sqsubseteq y'$ (y is the least upper bound)

Theorem

If X has a lub y, then it is unique. We denote it by $\bigsqcup X$.

Proof.

Write down two lubs for X and invoke the antisymmetry axiom for $\sqsubseteq$.

$\square$

Intuition: y has all the information in X and nothing else!

Intuition: y has all the information in X and nothing else!
Example: let $\mathbb{B} = \{\text{true}, \text{false}\}$ and consider $\mathbb{B}_+ \times \mathbb{B}_+$.

We have the following:

$$\begin{array}{ll} (\bot, \text{false}) \coprod (\text{true}, \bot) & = (\text{true}, \text{false}) \\ (\text{false}, \text{false}) \coprod (\text{true}, \text{true}) & \text{does not exist} \end{array}$$

Another example:

Let $f, g \in \mathbb{Z}_+^{\mathbb{Z}_+}$ defined by:

$f(x) = x + 1$ if x is odd, $\perp$ otherwise
$g(x) = x + 1$ if x is even, $\perp$ otherwise

Pop Quiz: What is $f \coprod g$?

Another example:

Let $f, g \in \mathbb{Z}_+^{\mathbb{Z}_+}$ defined by:

$f(x) = x + 1$ if x is odd, $\perp$ otherwise
$g(x) = x + 1$ if x is even, $\perp$ otherwise

Pop Quiz: What is $f \coprod g$?

$(f \coprod g)(x) = x + 1$ for all $x \in \mathbb{Z}$

Definition

A subset X of a poset Y is directed if for all $x_1, x_2 \in X$, $x_1 \coprod x_2$ exists.

Intuition: Given any two things in X, the information is compatible and they can be combined into one thing.

Definition

A complete partial order (cpo) is a poset Y such that

- Y has a bottom $\bot$
- $\coprod X$ exists for all directed $X \subseteq Y$

Definition

A complete partial order (cpo) is a poset Y such that

- Y has a bottom $\bot$
- $\coprod X$ exists for all directed $X \subseteq Y$

Intuition: A set of things that can be pairwise combined can be mushed together into one thing.

Examples

- Any finite poset with a bottom is a cpo.
- Given any unordered set X, $X_\perp$ is a cpo.
- $[0, 1] \in \mathbb{R}$ given the standard ordering on $\mathbb{R}$ is a cpo

But $[0, 1]$ in $\mathbb{Q}$ is not!

A function $f : Y \to Z$ of cpos is continuous if

- f is monotonic
- f preserves lubs. That is, for any directed set $X \subseteq Y$, $f(\coprod X) = \coprod f(X)$.

A function $f : Y \to Z$ of cpos is continuous if

- f is monotonic
- f preserves lubs. That is, for any directed set $X \subseteq Y$, $f(\coprod X) = \coprod f(X)$.

Intuition: f preserves things that are bigger, and those big things don't go too far away!

A function $f : Y \to Z$ of cpos is continuous if

- f is monotonic
- f preserves lubs. That is, for any directed set $X \subseteq Y$, $f(\coprod X) = \coprod f(X)$.

Intuition: f preserves things that are bigger, and those big things don't go too far away!

A function $f : Y \to Z$ of cpos is continuous if

- f is monotonic
- f preserves lubs. That is, for any directed set $X \subseteq Y$, $f(\coprod X) = \coprod f(X)$.

Intuition: f preserves things that are bigger, and those big things don't go too far away!

Theorem

If $f : Y \to Z$ is a monotonic function between cpos, then if Y is finite then f is continuous.

Big Idea: computable functions are continuous maps of cpos.

Implication: the proper target for $[\![-]\!]$ is the category of cpos and continuous maps.

This big idea is reinforced by the

Theorem (CPO fixpoint theorem)

A continuous function $f : X \to X$ on a cpo has a least fixpoint, written $fix(x)$, which can be computed as the lub of the chain

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq f(f(f\bot)) \sqsubseteq \cdots$$

This big idea is reinforced by the

Theorem (CPO fixpoint theorem)

A continuous function $f : X \to X$ on a cpo has a least fixpoint, written $fix(x)$, which can be computed as the lub of the chain

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq f(f(f\bot)) \sqsubseteq \cdots$$

Another way of writing this is:

$$\text{fix}(f) = \coprod_{n \in \mathbb{N}} \{f^n(\bot)\}$$

Little example:
Suppose $f : \mathbb{B}_+ \to \mathbb{B}_+$ maps everything to true. Then:

$$\begin{aligned}
&\bot && = \bot \\
&f(\bot) && = \mathsf{true} \\
&f(f(\bot)) = f(\mathsf{true}) && = \mathsf{true} \\
&f^3(\bot) = f(f^2(\bot)) = f(\mathsf{true}) && = \mathsf{true} \\
&\vdots
\end{aligned}$$

We can see that this sequence converges to true, and that is precisely the least (and only) fixpoint of f.

More interesting example:

```
ones = 1 : ones
```

More interesting example:

```
ones = 1 : ones
```

Now let
$f(x) = 1 : x$ where $f : [\mathbb{N}_\perp] \to [\mathbb{N}_\perp]$

More interesting example:

```
ones = 1 : ones
```

Now let
$f(x) = 1 : x$ where $f : [\mathbb{N}_\perp] \to [\mathbb{N}_\perp]$

We have the following sequence:

$$\begin{array}{ll} \perp & = \perp \\ f(\perp) & = 1 : \perp \\ f(f(\perp)) & = 1 : 1 : \perp \\ \vdots & \end{array}$$

The colimit of this is $1 : 1 : 1 : \cdots$

Convince yourself that this is the least (and only) fixpoint of f.

Now let us recall our

$$\Phi(f) = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

Then,

$$\bot \qquad = \lambda n.\bot$$

Now let us recall our

$$\Phi(f) = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

Then,

$$\begin{array}{ll} \bot & = \lambda n.\bot \\ \Phi(\bot) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } \bot \end{array}$$

Now let us recall our

$$\Phi(f) = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

Then,

$$\begin{array}{ll} \bot & = \lambda n.\bot \\ \Phi(\bot) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } \bot \\ \Phi(\Phi(\bot)) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (\ \text{if } n - 1 = 0 \text{ then } 1 \text{ else } \bot) \end{array}$$

Now let us recall our

$$\Phi(f) = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

Then,

$$\begin{array}{ll}
\bot & = \lambda n.\bot \\
\Phi(\bot) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } \bot \\
\Phi(\Phi(\bot)) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (\ \text{if } n-1 = 0 \text{ then } 1 \text{ else } \bot) \\
\Phi^3(n) & = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } \ \text{if } n-1 = 0 \text{ then } 1 \\
 & \quad \text{else } (n-1) * (\ \text{if } n-2 = 0 \text{ then } 1 \text{ else } \bot)
\end{array}$$

In fact, $\Phi^k(\perp) = \lambda n.n!$ if $n < k$ and $\perp$ otherwise.

Therefore $\text{fix}(\Phi) = \bigsqcup\{\Phi^k(\perp)\}$ is our old friend the factorial function, and thus the denotation of

```
fact 0 = 1
fact n = n * (fact (n-1))
```

Phew! We finally have an idea of what $[\![-]\!]$ actually is: a functor from the category `Hask` to the category of complete cpos and continuous maps. Assuming that we can define the denotation of "primitive" objects and morphisms in `Hask`, we can extend recursively by appealing to the fixpoint theorem.

Let's look at our category of cpos in more detail. The neat thing is that the category of cpos gives us laziness "out of the box".

Consider

```
Prelude> let f x = 1
Prelude> f undefined
1
```

Consider

```
Prelude> let f x = 1
Prelude> f undefined
1
```

We see that $[\![\texttt{f}]\!](\bot) = 1$. That is, by our definition of continuity, $\bot$ need not be mapped to $\bot$ by every continuous function. This is the denotational semantics version of laziness!

Now consider

```
Prelude> let g (x,y) = x
Prelude> g (1, head [])
1
Prelude> head []
*** Exception: Prelude.head: empty list
```

According to our definition of products, $(1, \bot) \neq (\bot, \bot) = \bot$. This also allows laziness.

By contrast, to have strictness, we would do two things:

- Require continuous maps $f : X \to Y$ to map $\bot_X$ to $\bot_Y$
- In the product cpo $X \times Y$, identify $(\bot_X, *)$ with $(*, \bot_Y)$ with $(\bot_X, \bot_Y) = \bot_{X \times Y}$.

Thus, we have different semantic domains for laziness and strictness.

- Laziness - the category of cpos
- Strictness - the category of “pointed” cpos

Thus, we have different semantic domains for laziness and strictness.

- Laziness - the category of cpos
- Strictness - the category of "pointed" cpos

Note: This is eerily similar to the distinction between the category of topological spaces and the category of pointed topological spaces from the study of topology.

I've implicitely required $[\![-]\!]$ to preserve products and exponentials throughout this talk.

I've implicitely required $[\![-]\!]$ to preserve products and exponentials throughout this talk.

A fancy way of saying this is that $[\![-]\!]$ is a cartesian functor.

I've implicitely required $[\![-]\!]$ to preserve products and exponentials throughout this talk.

A fancy way of saying this is that $[\![-]\!]$ is a cartesian functor.

Except that `Hask` is not cartesian closed for technical reasons - due to the distinction between `undefined` and `()` and `(undefined, undefined)` . So instead, I have to require that $[\![-]\!]$ be cartesian on an appropriate cartesian closed subcategory of `Hask`. (Beyond the scope of this talk)

As long as we've done all this work, I can't resist talking a little about *categorical semantics*. You should pay attention because they appear in Moggi's paper on monads and computation (`http://www.disi.unige.it/person/MoggiE/ftp/lics89.pdf`).

Astute observer's will have noted that the process of taking a set X to the poset $X_\perp$ (lifting) is *functorial*. That is, maps between sets turn into maps between partial orders. If $f : X \to Y$ is a function, then there is a $f_\perp : X_\perp \to Y_\perp$ where $f_\perp(\perp_X) = \perp_Y$.

Astute observer's will have noted that the process of taking a set X to the poset $X_\perp$ (lifting) is *functorial*. That is, maps between sets turn into maps between partial orders. If $f : X \to Y$ is a function, then there is a $f_\perp : X_\perp \to Y_\perp$ where $f_\perp(\perp_X) = \perp_Y$.

In fact, there is a *monad* structure on $(-)_\perp$. After all, we can map $X_{\perp_\perp}$ to $X_\perp$ by identifying the two bottoms! There is also a trivial map $X \to X_\perp$.

Our $[\![-]\!]$ functor can be thought of as taking values in the Kleisli category of the monad $(-)_\perp$ on the category of sets. Well, not really! That only works for call-by-value languages!

After all, by making a program correspond to a morphism $X \to Y_\perp$, you are not allowing things like

```
f x = 1
y = f (head [])
```

Moggi's insight was to allow the semantics functor to take values in the Kleisli category for a wider class of monads.

Moggi's insight was to allow the semantics functor to take values in the Kleisli category for a wider class of monads.

For instance, using the powerset monad corresponds to non-deterministic computations (a function maps a value to a set of possible values).

Moggi's insight was to allow the semantics functor to take values in the Kleisli category for a wider class of monads.

For instance, using the powerset monad corresponds to non-deterministic computations (a function maps a value to a set of possible values).

One can also construct monads that correspond to computations with state.

These are sources that I shamelessly ripped off of and are great for further reading:
`http://www.cs.nott.ac.uk/~gmh/domains.html`
`http://en.wikibooks.org/wiki/Haskell/Denotational_semantics`
`http://www.haskell.org/haskellwiki/Hask`

Once you've got the basics, you should be able to at least skim this:
`http://research.microsoft.com/en-us/people/dimitris/hcc-popl.pdf`
(Admittedly, a little knowledge of first order logic will also help!)

Summary:

- Operational semantics - “How does this program run?”
- Denotational semantics - “To what mathematical object does this program correspond?”
- Axiomatic semantics - “What logical properties must any interpretation of this program satisfy?”

A few more from Claus Reinke (`http://www.haskell.org/pipermail/haskell-cafe/2011-January/088315.html`):

- anecdotal semantics: "you know, once I wrote this program, and it just fried the printer.."
- barometric semantics: I think it is getting clearer..
- conventional semantics: usually, this means..
- detonational semantics: what does this button do?
- existential semantics: I'm sure it means something.
- forensic semantics: I think this was meant to prevent..
- game semantics: let the dice decide
- historical semantics: I'm sure this used to work
- idealistic semantics: this can only mean..
- jovial semantics: oh, sure, it can mean that.
- knotty semantics: hm, this part over here probably means..
- looking glass semantics: when I use a program, it means just what I choose it to mean, neither more nor less

- musical semantics: it don't mean a thing if it ain't got that swing.
- nihilistic semantics: this means nothing.
- optimistic semantics: this could mean..
- probabilistic semantics: often, this means that..
- quantum semantics: you can't ask what it means.
- reactionary semantics: this means something else.
- sherlockian semantics: since it cannot possibly mean anything else, ..
- transitional semantics: for the moment, this means that..
- utilitarian semantics: this means we can use it to..
- venerable semantics: this has always meant..
- weary semantics: ¡sigh¿ I guess that means..
- xenophobic semantics: for us here, this means..
- yogic semantics: we shall meditate on the meaning of this.
- zen semantics: ah!

Thanks!