The lambda calculus

Prolegomenon to functional programming

Patrick D. Elliott

April 13, 2023

The plan

- Today: theoretical preliminaries to programming in haskell - functions and the (simple, untyped) lambda calculus.

- Week 2:

    - Setting up a haskell dev environment.

    - Getting started with haskell - basic concepts and syntax.

- Week 3: strings and lists.

- Week 4: datatypes, typeclasses, etc.

- In subsequent weeks, once we have a grasp of functional programming basics, we'll start to tackle linguistics-specific topics, using (van Eijck, Jan and Unger, Christina, 2010).

Why study the lambda calculus

- The lambda calculus is a formal logic for reasoning about computation; the simple untyped lambda calculus is Turing complete, and can therefore be used to reason about *any* computation.

- Haskell is based on a more restrictive, but still extremely expressive variant of the lambda calculus called System F (i.e., the polymorphic lambda calculus).

- Moreover, the lambda calculus undergirds the functional programming paradigm more generally (see, e.g., one of the many variants of *lisp*).

- The lambda calculus is a common formal tool in theoretical linguistics; more specifically, it is a *lingua franca* in compositional semantics.

Semantic computation as a program

- If you’ve taken a formal semantics class before, the following might look familiar:

$$\forall z[\mathbf{ling}(z) \to \mathbf{reads}(\mathbf{C})(z)]$$

$$\lambda S.\forall z[\mathbf{ling}(z) \to S(z)] \qquad \lambda y.\mathbf{reads}(\mathbf{C})(y)$$

$$\lambda R.\lambda S.\forall z[R(z) \to S(z)] \qquad \lambda l.\mathbf{ling}(l) \qquad \lambda x.\lambda y.\mathbf{reads}(x)(y) \qquad \text{Chomsky}$$

every linguist reads

Embedding a semantic fragment in haskell

```haskell
data E = Chomsky | Reinhart | Borer | ...

_people :: [E]
_people = [Chomsky,Reinhart,Borer,...]

_reads :: E -> E -> Bool

_everyone :: (E -> Bool) -> Bool
_everyone f = all f _people

-- >>> (_everyone (_reads Chomsky)) :: Bool
-- >>> ((_reads Chomsky) _Borer) :: Bool
-- >>> (everyone (\x -> (_reads x Borer))) :: Bool
```

Functions

- A function is a special kind of relation between *inputs* and *outputs*.
- For example, we might imaging a function f that defines the following relations:
    - $f(1) = A$
    - $f(2) = B$
    - $f(3) = C$
- The input set is $\{1, 2, 3\}$
- and the output set is $\{A, B, C\}$.

Determinacy

- Is f in the following a valid function?
  - $f(1) = A$
  - $f(1) = B$
  - $f(2) = C$

Uniqueness

- Is f in the following a valid function?
  - $f(1) = A$
  - $f(2) = A$
  - $f(3) = B$

Function terminology

- We call the set of values from which a function draws its inputs the domain of the function.

- We call the set of values from which a function draws its outputs the codomain.

- A function always maps *every* member of the domain to a member of the codomain, but not every member of the codomain is necessarily paired with an input. We call the subset of values in the codomain paired with inputs the image of the function.

Functions as relations

- Functions can be represented as *relations*, i.e., sets of ordered pairs.
- For example, the following is a valid function:
    - $\{(1, A), (2, B), (3, C)\}$
- A relation R is *functional* iff $\forall (x, y), (x', y'), x = x' \rightarrow y = y'$).
- Which of the following relations are functional?
    - $\{(\text{Chomsky}, \text{SynStr}), (\text{Reinhart}, \text{Int}), (\text{Chomsky}, \text{Asp})\}$
    - $\{(\text{Ross}, \text{Chomsky}), (\text{Pesetsky}, \text{Chomsky}), (\text{Nevins}, \text{Halle})\}$

Extension vs. intension

- The intuition behind functions is that they define determinate *procedures* for getting from an input to a fixed output.

- Sometimes we can simply list the input-output pairings defined by the function (this is called the function's *extension*).

- Most of the time this either isn't useful or it's impossible, rather we describe the procedure - this is called giving the function's *intension*. One famous function is the *successor function*.

    - $f(x) = x + 1$

- We could try giving the extension:

    - $\{(0, 1), (1, 2), (2, 3), (4, 5), \ldots\}$

- Given that the domain and codomain are infinite, this is practically impossible.

Lambda expressions

- The lambda calculus is used as a logic used to reason about functions, how they compose, and computation more generally.

- Valid expressions of the lambda calculus can be variables, abstraction, or combinations of both; variables have no intrinsic meaning, they're just names for possible inputs to functions.

Structure of an abstraction