

The lambda calculus

Prolegomenon to functional programming

PATRICK D. ELLIOTT

APRIL 13, 2023

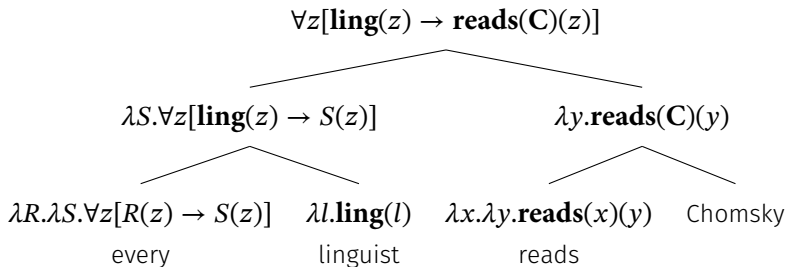
The plan

- **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.

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



Embedding a semantic fragment in 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
```

- A function is a special kind of relation between *inputs* and *outputs*.
- For example, we might imagine 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 $\{A, B, C\}$.

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

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

- 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 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?
 - $\{(\text{Chomsky}, \text{SynStr}), (\text{Reinhart}, \text{Int}), (\text{Chomsky}, \text{Asp})\}$
 - $\{(\text{Ross}, \text{Chomsky}), (\text{Pesetsky}, \text{Chomsky}), (\text{Nevins}, \text{Halle})\}$

- 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:
 - $\{(0, 1), (1, 2), (2, 3), (4, 5), \dots\}$
- Given that the domain and codomain are infinite, this is practically impossible.

- 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

- **Abstractions** are made up of two parts: a *head* and a *body*.
 - $\lambda x.x$
- The head is the λ symbol followed by a variable name. Variables in the body matching the variable name are *bound*.
- The body is a valid expression of the lambda calculus which follows the dot.
- Abstractions in the lambda calculus are interpreted as functions; the head of the abstraction stands in for the input to the function, and the body of the abstraction tells us how we arrive at the output.
- Lambda abstractions allow us to describe what functions do without naming them; we'll sometimes call lambda abstractions *anonymous functions*.
- **Question:** what kind of function is the abstraction above? What does it do?

- So, abstractions are used to express functions. The choice of variable name used in the head is arbitrary - this gives rise to an intuitive notion of sameness: **alpha equivalence**.
- The following expressions are all *alpha equivalent* (that is, they all express the same function):
 - $\lambda x.x$
 - $\lambda d.d$
 - $\lambda z.z$
- The procedure of substituting some expression for an α -equivalent variant is known as **alpha conversion**.

- Beta reduction corresponds to applying a function to an argument, in the lambda calculus.
- A *functional application* is written as $f(x)$ where f is the function, and x is the argument. In anticipation of haskell syntactic conventions, we'll often indicate function application with a space, i.e., $f\ x$
- Beta reduction involves deleting the head, and substituting all occurrences of the bound variable in the body with the function's argument.
 - $(\lambda x.x + 1)\ 2$
- **Question:** What is the result of beta reduction?

- A named function can always be expressed as an anonymous function by applying it to a variable x , and then *abstracting* over that variable with a λx .
- This gives rise to a notion of *eta equivalence*. The following functions are eta equivalent:
 - f
 - $\lambda x.f(x)$
- Simplifying some expression with some eta-equivalent variable is called *eta conversion*; the special case of simplification is called *eta reduction*.

- Nothing stops us from applying a *function* to another *function*:
 - $(\lambda x.x)(\lambda y.y)$
 - $[x := (\lambda y.y)]$
 - $\lambda y.y$
- Note that $[x := \alpha]$ indicates that the variable x is substituted with the expression α in the function body.

- Functional application is *left associative*:
 - $(\lambda x.x)(\lambda y.y)z := ((\lambda x.x)(\lambda y.y))z$

- The previous expression involved a functional application nested within a functional application:
 - $((\lambda x.x)(\lambda y.y))z$
- We typically reduce from the inside out:
 - $[x := (\lambda y.y)]$
 - $(\lambda y.y)z$
 - $[y := z]$
 - z
- If no further reductions are possible, we say that the expression is in **normal form**.

- Sometimes, the body of an abstraction contains variables which aren't bound by the head - these variables are *free* (within the abstraction):
 - $\lambda x.xy$
- Let's try applying an abstraction with free variables to an argument:
 - $(\lambda x.xy)z$
 - $[x := z]$
 - zy
- Note that alpha equivalence doesn't apply to free variables: $\lambda x.xy$ and $\lambda x.xz$ are different expressions, because y and z might be assigned distinct values.

- Each λ can only bind one parameter and can only accept one argument.
- Multiple arguments are encoded by multiple λ s (this is called *currying*; semanticists call it *Schönfinkelization*).
 - $\lambda xy.xy := \lambda x.(\lambda y.xy)$
- N.b. in haskell we'll be able to express functions that take tuples of arguments by using something called *pattern matching*.

1. $\lambda xy.xy$
2. $(\lambda xy.xy) 1 2$
3. $(\lambda x.(\lambda y.xy)) 1 2$
4. $[x := 1]$
5. $(\lambda y.1 y) 2$
6. $[y := 2]$
7. $1 2$

We've reached normal form, since we can't apply 1 to 2.

1. $(\lambda xy.xy)(\lambda z.a) 1$
2. $(\lambda x.(\lambda y.xy))(\lambda z.a) 1$
3. $[x := (\lambda z.a)]$
4. $(\lambda y.(\lambda z.a) y) 1$
5. $[y := 1]$
6. $(\lambda z.a) 1$
7. $[z := 1]$
8. a

- Simplification rules for the lambda calculus:
 - β -reduction
 - α -conversion
 - η -reduction
- The lambda calculus can be thought of as a *logic*, and these rules constitute its proof theory.
- We've left the (denotational) semantics of the lambda calculus implicit, but intuitively β -reduction is licit *because* applications are interpreted by applying the function to the argument, etc.
- The variant of the lambda calculus we're considering here is *Turing complete*, which means that it can be used to simulate an arbitrary Turing machine.

- How do we go about reducing this expression?
 - $(\lambda xy. xxy)(\lambda x. xy)(\lambda x. xz)$

- Remember, when we can no longer simplify an expression, the result is said to be in **normal form** (N.b., there are different kinds of normal forms, but the differences aren't relevant to us).
- In programming terms, this corresponds to a **fully executed program**.
- Arithmetic expressions can be thought of as a simple logic/programming language.
 - $(10 + 2) * 100 / 2$
- What's the normal form of this arithmetic expression?
- Remember that complex expressions can nevertheless be in normal form, such as $\lambda x.x$.

- Combinators are special kinds of lambda expressions with no free variables.
- Which of the following are (not) combinators?
 - $\lambda x.x$
 - $\lambda xy.x$
 - $\lambda xyz.xz(yz)$
 - $\lambda y.x$
 - $\lambda x.xz$
- As the name suggests, combinators serve to combine their arguments.

- A combinator which we'll encounter quite a lot is *function composition*:
 - $\lambda f.\lambda g.\lambda x.g(f(x))$
- Simplify the following expression.
 - $(\lambda f.\lambda g.\lambda x.g(f(x)))(\lambda n.n/2)(\lambda z.z * 12) 100$
- The *composition* of two functions is often abbreviated using dot notation:
 - $g \cdot f := \lambda x.g(f(x))$
- This means we could write the previous expression as:
 - $((\lambda z.z * 12) \cdot (\lambda n.n/2)) 100$

- Ordinarily, reducing a lambda expression *converges* to normal form.
- Not every *reducible* lambda expression reduces to normal form; some lambda expressions *diverge*.
- This underlies the Turing-completeness of the simple lambda calculus.
- Reduce the following expression (called omega) until you're satisfied it doesn't converge:
 - $(\lambda x.xx)(\lambda x.xx)$
- Diverging expressions correspond to non-terminating programs.

- **Obligatory:** Do the *chapter exercises* from chapter 1 of **Haskell programming from first principles** (p17-18). If you get stuck somewhere, send me a note before next week's class.
- Optionally, do either of the following:
 - Read chapter 1 of **Haskell programming from first principles**.
 - Re-read the slides from today's class at your own pace.
- If you have time, you can start setting up a haskell development environment; instructions here:
<https://www.haskell.org/get-started/>, but
<https://play.haskell.org/> will be sufficient for the first few weeks.

- For an in-depth introduction to the *simply-typed* lambda calculus, from a logical perspective, read (Carpenter, Bob, 1998)

Fin

Carpenter, Bob (1998). *Type-Logical Semantics*, MIT Press.

van Eijck, Jan and Unger, Christina (2010). *Computational Semantics with Functional Programming*, Cambridge University Press.