Introduction to Program Semantics — Dexter Kozen

Lecture 2 - The Lambda Calculus *June 26, 2026*

1 Understanding Computation

In the 1920s, Hilbert proposed a program to mechanise mathematics, this proposal led to the creation of a number of different formal systems as given in the table below. Thanks to Church's thesis, we know that all of these systems can simulate one another. In these notes we will focus on the lambda calculus in particular.

Inventor	**System**	**Data**
Alan Turing	Turing machines	finite-length strings
Kurt Gödel	μ-recursive functions	natural numbers $\mathbb{N}$
Emil Post	rewrite systems	finite-length strings
Alonzo Church	λ-calculus	λ-terms
Haskell Curry	combinatory logic	combinator terms

2 Pure Lambda Calculus

The lambda calculus is defined by the following grammar rule:

$$e := x | e_1\ e_2 | \lambda x.e \tag{1}$$

This definitions gives us a language that looks like any modern functional language (e.g. OCaml, Haskell) but is much simpler. Importantly everything is a function and there is no state or side effects. The system is also non-deterministic with no enforced evaluation order on the system as a whole. We interpret the rules in the following way:

- x, is the variable term. We assume we have an infinite list of variables where we can always ask for one more.
- $e_1\ e_2$ is application, a non-associative binary operator that binds to the left.

Compiled By:
Yanjun Chen, Kunha Kim, Jonathon Yallop

- $\lambda x.e$ is abstraction, this binds all occurences of the variable x in the term, e. In OCaml the following syntax would give us the same construction:

```
fun x => e
```

We can always add more data types with their operators from different domains, such as $\mathbb{N}$ with $+, \cdot, =, \leq, <$ or $\{false, true\}$ with $\wedge, \vee, \neg$, but the lambda calculus itself consists solely of the three terms listed above.

To demonstrate exactly what different lambda terms look like, we can construct a number of different examples.

$\lambda x.\, x$	the identity function
$\lambda xa.\, a$	ignores argument x; returns identity $\lambda a.\, a$
$\lambda f.\, f\, a$	takes a function f and applies it to a
$\lambda vf.\, f\, v$	takes v; returns $\lambda f.\, f\, v$
$\lambda fgx.\, g(f\, x)$	takes f, g; returns their composition $g \circ f = \lambda x.\, g(f\, x)$

2.1 Binding and Scope

The λx in $\lambda x.\, e$ is a *binding operator*. An occurrence of x is *bound* if it is in the body of $\lambda x.\, e$, and otherwise it is *free*. The scope of a variable is the expression in the binding operation, in this case e. A bound occurrence is bound by the λx with the *smallest*, (think closest) enclosing scope. A term is *closed* if all variables are bound. Also the x in λx is not itself an "occurrence." In the following example, the colors of the variables have been changed to match which binding operator to which they belong.

$$(\lambda x.\, x\ y\ (\lambda y.\, x\ y\ (\lambda x.\, x\ y\ z)))\ x$$

3 Substitutions and Reductions

Substition is how we perform computation and gives us its semantics. We have two different forms, α and β conversion

3.1 α-conversion

This conversion renames bound variables:

$$\lambda x.\, e \longrightarrow \lambda y.\, (e[y/x])$$

Which we can see demonstrated in the following reductions

$$\begin{aligned} \lambda x.\, xyz &\longrightarrow \lambda a.\, ayz \\ \lambda x.(\lambda x.\, xy)\, x &\longrightarrow \lambda a.(\lambda x.\, xy)\, a \end{aligned}$$

Using α-conversion will also gain a form of equivalence, where two terms will be considered α equivalent if there is some set of alpha conversion that turn one term into the other, for example `fun x -> x` is the same as `fun y -> y`.

3.2 β-conversion

This is the substitution rule, and it has the following form:

$$(\lambda x.\, e_1)\, e_2 \longrightarrow e_1[e_2/x]$$

We will often refer to the use of this rule as reduction, and we can see how it works in the following examples:

$$\begin{aligned} (\lambda x.\, xy)\, a &\longrightarrow ay \\ (\lambda x.\lambda y.\, xy)\, (\lambda a.\, ab) &\longrightarrow \lambda y.\, (\lambda a.\, ab)\, y \longrightarrow \lambda y.\, yb \end{aligned}$$

3.3 Safe Substitution

Substitution is in fact very hard to get entirely correct. In our definition, we have to be carfeul to avoid variable capture, which can lead to a semantically different term. As an example imagine the function, $(\lambda x.\, y)[x/y]$. After this substitution, the function would becomes, $\lambda x.x$, which is different from our original term, and is behavior we would like to avoid. In order to do this, we have the following rules for safe, capture-avoiding substituion:

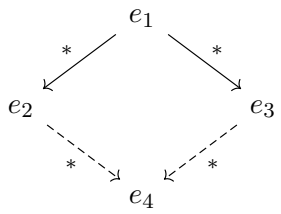
$$\begin{aligned} x[e/x] &= e \\ y[e/x] &= y \quad (y \neq x) \\ (e_1\, e_2)[e/x] &= (e_1[e/x])\, (e_2[e/x]) \\ (\lambda x.\, e')[e/x] &= \lambda x.\, e' \\ (\lambda y.\, e')[e/x] &= \lambda y.\, (e'[e/x]) \quad (y \neq x,\ y \notin \mathrm{fv}(e)) \\ (\lambda y.\, e')[e/x] &= \lambda z.\, (e'[z/y][e/x]) \quad (y \neq x,\ z \neq x,\ z \notin \mathrm{fv}(e') \cup \mathrm{fv}(e)) \end{aligned}$$

To avoid the difficulty of variable capture, we can also rely on something called normal sets, which require more set up but lead to a simpler definition of capture-avoiding substition. In practice, we also rely on the Barendregt variable convention which renames all bound variables to fresh, distinct names.

3.4 Church-Rosser theorem

The Church-Rosser theorem, otherwise known as confluence, is one of the most important theorems in the lambda calculus. It states that if we choose to perform substitions in two different subterms, we will always be able to reduce to a common term.

Theorem 1 *(Confluence) If* $e_1 \xrightarrow{*} e_2$ *and* $e_1 \xrightarrow{*} e_3$ *by some sequences of* α*- or* β*-steps, then there exists* e_4 *such that* $e_2 \xrightarrow{*} e_4$ *and* $e_3 \xrightarrow{*} e_4$*.*



(2)

3.5 Values and termination

A term is in *normal form* if it has no β-redexes. This means we have no more reductions to perform. A *value* is a closed term in normal form. Importantly, by Church-Rosser, values are unique up to α-conversion. We call a term *normalizable* if some sequence of steps leads to a value. The following examples are all in normal form:

$$I = \lambda x.\, x \qquad K = \lambda xy.\, x \qquad B = \lambda xy.\, y \qquad S = \lambda xyz.\, xz(yz)$$

Remark. The above S, K, I, B have some very interesting properties. Using them, we can form a complete system for deductive propositional logic, and since I and B can be deduced from S and K, we only need S and K to define this system. This means S and K are *combinatorially complete*, meaning we could define a rewrite system such that $S\ x\ y\ z \to xz\ (y\ z)$ and $K\ x\ y \to x$, and we could use those two terms to rearrange an element in any way you could want.

Not all computations lead to a value. Consider the following:
Let $\Omega = (\lambda x.\, xx)\,(\lambda x.\, xx)$
then $\Omega \to \Omega \to \Omega \to \cdots$

Reduction of the Ω term leads to an infinite chain of reductions, and in fact there are many such terms. Even if a term is normalizable, every reduction sequence will not lead to a normal form. Consider the following term:
$(\lambda xy.\, y)\,\Omega$
We can continue to reduce on the Ω term in an infinite sequence. It would only be by performing the outer β-reduction that the term would lead to the normal form, y.

3.6 Reduction Strategies

A *reduction strategy* specifies which β-redex to reduce next. The λ-calculus is nondeterministic and so for a real language, we need a fixed strategy. Some common strategies are as follows:

- **Normal order** (leftmost outermost)
- **Applicative order** (leftmost innermost)
- **Call by name (CBN)** — normal order, but no reductions inside a λ-body
- **Call by value (CBV)** — applicative order, but no reductions inside a λ-body

Most functional languages use CBV. Haskell uses CBN (lazy evaluation). If a term is normalizable, normal order and CBN will find the value. We can see the difference in these two strategies in the following reductions:
CBV

$$\begin{aligned} & (\lambda x.\, inc\ x)\,((\lambda y.\, inc\ y)\,3) \\ \to\ & (\lambda x.\, inc\ x)\,(inc\ 3) \\ \to\ & (\lambda x.\, inc\ x)\,4 \\ \to\ & inc\ 4 \\ \to\ & 5 \end{aligned}$$

CBN

$$\begin{aligned} & (\lambda x.\, inc\ x)\,((\lambda y.\, inc\ y)\,3) \\ \to\ & inc\,((\lambda y.\, inc\ y)\,3) \\ \to\ & inc\,(inc\ 3) \\ \to\ & inc\ 4 \\ \to\ & 5 \end{aligned}$$

We can also define reduction strategies using evaluation contexts. Call by value would have the following grammar (where v denotes a value):

$$C \ ::= \ [\,] \ \mid \ C\,e \ \mid \ v\,C$$

Whereas call by name would look like the following:

$$C \ ::= \ [\,] \ \mid \ C\,e$$

4 Encoding Datatypes

Using the lambda calculus, we can encode a number of standard data types

4.1 Booleans

Boolean terms would take the following form:

$$\begin{aligned} \mathit{true} &= \lambda xy.\, x \\ \mathit{false} &= \lambda xy.\, y \\ \mathit{if\text{-}then\text{-}else} &= \lambda xyz.\, xyz \end{aligned}$$

These definitions lead to the following examples of if-then-else statements: **if true then** e_2 **else** e_3**:**

$$(\lambda xyz.\, xyz)\,(\lambda ab.\, a)\,e_2\,e_3 \to (\lambda ab.\, a)\,e_2\,e_3 \to e_2 \to \cdots$$

if false then e_2 **else** e_3**:**

$$(\lambda xyz.\, xyz)\,(\lambda ab.\, b)\,e_2\,e_3 \to (\lambda ab.\, b)\,e_2\,e_3 \to e_3 \to \cdots$$

We can also define logical predicates:

$$\begin{aligned} \mathit{and} &= \lambda xy.\, xyx \\ \mathit{or} &= \lambda xy.\, xxy \\ \mathit{not} &= \lambda x.\, x\,\mathit{false}\,\mathit{true} \end{aligned}$$

And confirm that they are correct by seeing that **and true true** reduces to true,

$$(\lambda xy.\, xyx)\,(\lambda ab.\, a)\,(\lambda cd.\, c) \to (\lambda ab.\, a)\,(\lambda cd.\, c)\,(\lambda ab.\, a) \to \lambda cd.\, c = \mathit{true}$$

and that **and false true** reduces to false,

$$(\lambda xy.\, xyx)\,(\lambda ab.\, b)\,(\lambda cd.\, c) \to (\lambda ab.\, b)\,(\lambda cd.\, c)\,(\lambda ab.\, b) \to \lambda ab.\, b = \mathit{false}$$

4.2 Pairs

We can also define pairs in the language.

$$\begin{aligned} \mathit{pair} &= \lambda xyf.\, fxy \\ \mathit{fst} &= \lambda p.\, p\, \mathit{true} \\ \mathit{snd} &= \lambda p.\, p\, \mathit{false} \end{aligned}$$

The following examples demonstrate how these constructions work:

$$\begin{aligned} \mathit{pair}\, e_1\, e_2 &= (\lambda xyf.\, fxy)\, e_1\, e_2 \to \lambda f.\, f\, e_1\, e_2 \\ \mathit{fst}\, (\mathit{pair}\, e_1\, e_2) &\to (\lambda f.\, f\, e_1\, e_2)\, (\lambda xy.\, x) \to (\lambda xy.\, x)\, e_1\, e_2 \to e_1 \\ \mathit{snd}\, (\mathit{pair}\, e_1\, e_2) &\to (\lambda f.\, f\, e_1\, e_2)\, (\lambda xy.\, y) \to (\lambda xy.\, y)\, e_1\, e_2 \to e_2 \end{aligned}$$

4.3 Church numerals

Importantly we would also like to encode numbers in lambda terms. We achieve this using repeated function application where the number of function applications is the number represented by the term. This leads to the following definition:

$$\begin{aligned} \overline{0} &= \lambda fx.\, x = \lambda fx.\, f^0 x \\ \overline{1} &= \lambda fx.\, fx = \lambda fx.\, f^1 x \\ \overline{2} &= \lambda fx.\, f(fx) = \lambda fx.\, f^2 x \\ \overline{3} &= \lambda fx.\, f(f(fx)) = \lambda fx.\, f^3 x \\ &\vdots \\ \overline{n} &= \lambda fx.\, \underbrace{f(\cdots(f(f(fx)))\cdots)}_{n} = \lambda fx.\, f^n x \end{aligned}$$

Using this definition we can define a number of standard arithmetic operations, like inc, add and mult.

$$\mathit{inc} = \lambda nfx.\, f(nfx)$$

$$\begin{aligned} \mathit{inc}\, \overline{n} &= (\lambda nfx.\, f(nfx))\, (\lambda gy.\, g^n y) \\ &\to \lambda fx.\, f\big((\lambda gy.\, g^n y)\, f\, x\big) \\ &\to \lambda fx.\, f(f^n x) \\ &= \lambda fx.\, f^{n+1} x \\ &= \overline{n+1} \end{aligned}$$

$$add = \lambda nm.\, m\, inc\, n$$

$$\begin{aligned} add\ \overline{n}\ \overline{m} &\to \overline{m}\, inc\, \overline{n} \\ &= (\lambda fx.\, f^m x)\, inc\, \overline{n} \quad \text{(apply } inc \text{ } m \text{ times to } \overline{n}) \\ &= inc^m\, \overline{n} \\ &= \overline{n+m} \end{aligned}$$

$$mul = \lambda nm.\, m\, (add\ n)\ \overline{0}$$

$$\begin{aligned} mul\ \overline{n}\ \overline{m} &\to \overline{m}\, (add\ n)\ \overline{0} \\ &= (\lambda fx.\, f^m x)\, (add\ n)\ \overline{0} \quad \text{(apply } add\ n\ m \text{ times to } \overline{0}) \\ &\to (add\ n)^m\ \overline{0} \\ &= \overline{nm} \end{aligned}$$

Using pairs we are even able to define the predecessor function: Let

$$g = \lambda z.\, pair\, (snd\ z)\, (inc\, (snd\ z)) \;=\; \lambda(x,y).\, (y,\ inc\ y)$$

$$(0,0) \xrightarrow{g} (0,1) \xrightarrow{g} (1,2) \xrightarrow{g} (2,3)$$

$$dec = \lambda n.\, fst\, (n\ g\ (pair\ \overline{0}\ \overline{0}))$$

$$\begin{aligned} dec\ \overline{3} &\to fst\, (\overline{3}\ g\ (pair\ \overline{0}\ \overline{0})) \\ &\to fst\, (g^3\, (pair\ \overline{0}\ \overline{0})) \\ &\to fst\, (pair\ \overline{2}\ \overline{3}) \\ &\to \overline{2} \end{aligned}$$

5 Recursive functions

To define recursive functions, we will have to come up with something clever. As recursive functions typically refer to themselves in the body,

```
let rec fact x =
  if x = 0 then 1 else x * fact (x - 1)
```

We run into a problem as the lambda caluculus has no names, all its functions are anonymous. So we want *fact* to satisfy:

$$fact = \lambda n.\, \big(if\text{-}then\text{-}else\ (isZero\ n)\ \overline{1}\ (mul\ n\ (fact\ (dec\ n)))\big)$$

This is equivalent to the following definition,

$$h = \lambda f.\lambda n.\,\big(\mathit{if\text{-}then\text{-}else}\ (\mathit{isZero}\ n)\ \overline{1}\ (\mathit{mul}\ n\ (f\ (\mathit{dec}\ n)))\big)$$

In order to introduce recursion, we will want *fact* to be a *fixpoint* of h: $\mathit{fact} = h\ \mathit{fact}$. Luckily in the lambda calculus we have a term that is the fixpoint for any term.

5.1 The Y combinator

The term $(\lambda x.\, h(xx))\,(\lambda x.\, h(xx))$ is a fixpoint of any h and is called the Y combinator. We can see it act as a fixpoint in the following reduction:

$$(\lambda x.\, h(xx))\,(\lambda x.\, h(xx)) \to h\big((\lambda x.\, h(xx))\,(\lambda x.\, h(xx))\big)$$

So we will define the following:

$$Y = \lambda t.\,(\lambda x.\, t(xx))\,(\lambda x.\, t(xx))$$

Which when we apply to any function, h, gives us:

$$\begin{aligned} Yh &\to (\lambda x.\, h(xx))\,(\lambda x.\, h(xx)) \\ &\to h\big((\lambda x.\, h(xx))\,(\lambda x.\, h(xx))\big) \\ &= h\,(Yh) \end{aligned}$$

So Yh is a fixpoint of h; we can define $\mathit{fact} = Yh$.

6 Simply Typed Lambda Calculus

To avoid runtime type errors, we introduce primitive types and constants into the language, ensuring well-typed programs do not get "stuck".

6.1 Syntax and Types

The Simply Typed Lambda Calculus (STLC) extends the pure lambda calculus with explicit types and base values:

- **Types (τ):**
$$\tau ::= \text{int} \mid \text{bool} \mid \text{unit} \mid \tau_1 \to \tau_2$$
- **Values (v):**
$$v ::= n \mid \text{true} \mid \text{false} \mid \text{null} \mid \lambda x : \tau.\, e$$

- **Terms (e):**

$$e ::= v \mid x \mid e_1\ e_2$$

There are two main styles of defining functions with types:

- **Church-style:** Functions contain explicit type annotations ($\lambda x : \tau.\, e$).
- **Curry-style:** Functions are written without type annotations ($\lambda x.\, e$).

6.2 Type Judgments and Typing Rules

Type assignments are represented as a type judgment:

$$\Gamma \vdash e : \tau$$

This states that under the assumptions of the type environment Γ, the term e has type τ. The environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ maps free variables to their respective types.

The formal static semantics of STLC is defined by the following deductive rules:

$$\Gamma \vdash n : \text{int} \qquad \Gamma \vdash \text{true} : \text{bool} \qquad \Gamma \vdash \text{false} : \text{bool} \qquad \Gamma \vdash \text{null} : \text{unit}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{Var}) \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.\, e) : \sigma \to \tau}\ (\text{Abs})$$

$$\frac{\Gamma \vdash e_0 : \sigma \to \tau \qquad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash e_0\ e_1 : \tau}\ (\text{App})$$

6.3 Example Type Derivation

The following derivation tree demonstrates that the term $((\lambda x : \text{int}.\, \lambda y : \text{bool}.\, x)\ 2\ \text{true})$ is well-typed and has the type int:

$$\dfrac{\dfrac{\dfrac{\dfrac{x{:}\text{int},\, y{:}\text{bool} \vdash x{:}\text{int}}{x{:}\text{int} \vdash (\lambda y{:}\text{bool}.\, x){:}\text{bool}\to\text{int}}}{\vdash (\lambda x{:}\text{int}.\, \lambda y{:}\text{bool}.\, x){:}\text{int}\to\text{bool}\to\text{int}} \quad \vdash 2{:}\text{int}}{\vdash ((\lambda x{:}\text{int}.\, \lambda y{:}\text{bool}.\, x)\, 2){:}\text{bool}\to\text{int}} \quad \vdash \text{true} : \text{bool}}{\vdash ((\lambda x : \text{int}.\, \lambda y : \text{bool}.\, x)\, 2\ \text{true}) : \text{int}}$$

6.4 Properties of STLC

The safety of STLC is established by two foundational properties, together showing that well-typed programs cannot crash or get into an undefined state:

- **Progress:** If $\vdash e : \tau$, then e is either a final value or can perform a β-reduction step ($e \longrightarrow e'$).
- **Preservation:** If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.

6.4.1 Expressive Power and Normalization

Introducing a strict type system limits the expressive capability compared to the untyped λ-calculus:

- Non-terminating terms like $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ and fixed-point operators like the Y combinator cannot be assigned a valid type under STLC rules.
- Consequently, it is impossible to write non-terminating loops in pure STLC.

Theorem (Strong Normalization): The Simply Typed Lambda Calculus is strongly normalizing; every sequence of reductions is finite and eventually terminates in a normal form. *Proof technique:* This theorem is proved using the method of logical relations.