Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

SIGPLAN

https://www.sigplan.org/

Silver Sponsors

OREGON
PROGRAMMING
LANGUAGES
SUMMER
SCHOOL

Introduction to Programming Language Semantics

Lecture 2: The λ-Calculus

Roadmap

- Lecture 1: Structural operational semantics
  IMP language, small-step rules, big-step rules, evaluation contexts, big-step rules as binary relations, algebraic properties
- Lecture 2: The λ-calculus
  λ-terms, safe substitution, binding and scope, α- and β-reduction, Church-Rosser theorem, data encodings, recursion, simple types, progress and preservation
- Lecture 3: Denotational semantics
  CPOs, least fixpoints, Knaster-Tarski theorem, domain equations
- Lecture 4: Axiomatic semantics, probabilistic semantics
  Weakest preconditions, semantics of Hoare logic, Dynamic logic, probabilistic semantics

Prehistory

- Invented by Alonzo Church in the 1930s to study the interaction of functional abstraction and function application
- One of many related systems proposed in the 1920s and 1930s to capture the general idea of computation

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

- All these systems can simulate each other!
- Church's thesis: This is what we mean by computable

The pure λ-calculus

- Looks like any modern functional language (e.g. OCaml, Haskell), but much simpler
- Everything is a function – there are no other types
- Purely functional – no state or side effects
- Order of evaluation is irrelevant
- All functions are unary

λ-calculus syntax

$$e \ ::= \ x \mid e_1 \, e_2 \mid \lambda x \, . \, e$$

- $e_1 \, e_2$ is function application — a binary operator, written as juxtaposition, non-associative, associates to the left
- $\lambda x \, . \, e$ is function abstraction — represents a function with input parameter x and body e
- $\lambda x \, . \, e$ would be written `fun x -> e` in OCaml
- $\lambda x \, . \, \lambda y \, . \, e_1 \, e_2$ should be read as $\lambda x \, . \, (\lambda y \, . \, (e_1 \, e_2))$
- $\lambda x y \, . \, e$ is short for $\lambda x \, . \, \lambda y \, . \, e$
- Can include other domains and their operations, e.g. $\mathbb{N}$ with $+, \cdot$ and $=, \leq, <$ or the two-element Boolean algebra $\{\texttt{false}, \texttt{true}\}$ with $\wedge, \vee, \neg$, if you want to

Examples

$\lambda x\,.\,x$	the identity function
$\lambda x a\,.\,a$	ignores its argument x and returns the identity function $\lambda a\,.\,a$
$\lambda f\,.\,f a$	takes another function f as an argument and applies it to a
$\lambda v f\,.\,f v$	takes an argument v and returns a function $\lambda f\,.\,f v$
$\lambda f g x\,.\,g(f x)$	takes two functions f and g and returns their composition $g \circ f = \lambda x\,.\,g(f x)$

Binding and Scope

- An occurrence of x in a term is a bound occurrence if it is in the body of an abstraction $\lambda x\,.\,e$, otherwise it is a free occurrence
- The λx in $\lambda x\,.\,e$ is called a binding operator and e is its scope
- An bound occurrence of x in a term is bound to (or by) the λx with the smallest scope in which it occurs
- The λx in $\lambda x\,.\,e$ binds all free occurrences of x in e
- A term is closed if all variables are bound
- Note: The x in λx is not considered an "occurrence"

Example: $(\lambda x\,.\,x\ y\ (\lambda y\,.\,x\ y\ (\lambda x\,.\,x\ y\ z)))\ x$
(the colors show the bindings; free occurrences are black)

λ-calculus semantics

Computation in the λ-calculus is effected by substitution

Two rules:

- α-conversion (renaming bound variables)

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

(e.g. `fun x -> x` is the same as `fun y -> y`)

- β-reduction (substitution rule)

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

λ-calculus semantics

Examples:

- α-conversion (renaming bound variables)

$$\lambda x\,.\,xyz \to \lambda a\,.\,ayz$$
$$\lambda x\,.\,(\lambda x\,.\,xy)\,x \to \lambda a\,.\,(\lambda x\,.\,xy)\,a$$

- β-reduction (substitution rule)

$$(\lambda x\,.\,xy)\,a \to ay$$
$$(\lambda x\,.\,\lambda y\,.\,xy)\,(\lambda a\,.\,ab) \to \lambda y\,.\,(\lambda a\,.\,ab)\,y \to \lambda y\,.\,yb$$

Safe substitution

With substitution, must avoid variable capture, e.g. $(\lambda x\,.\,y)[x/y]$

Rules for safe (capture-avoiding) substitution:

$$
\begin{array}{rcll}
x[e/x] & = & e & \\
y[e/x] & = & y & y \neq x \\
(e_1\ e_2)[e/x] & = & (e_1[e/x])\ (e_2[e/x]) & \\
(\lambda x\,.\,e_0)[e/x] & = & \lambda x\,.\,e_0 & \\
(\lambda y\,.\,e_0)[e/x] & = & \lambda y\,.\,(e_0[e/x]) & y \neq x,\ y \text{ not free in } e \\
(\lambda y\,.\,e_0)[e/x] & = & \lambda z\,.\,(e_0[z/y][e/x]) & y \neq x,\ z \neq x, \\
 & & & z \text{ not free in } e_0 \text{ or } e
\end{array}
$$

Mostly we just use the Barendregt variable convention: rename all bound variables to fresh variables, all different

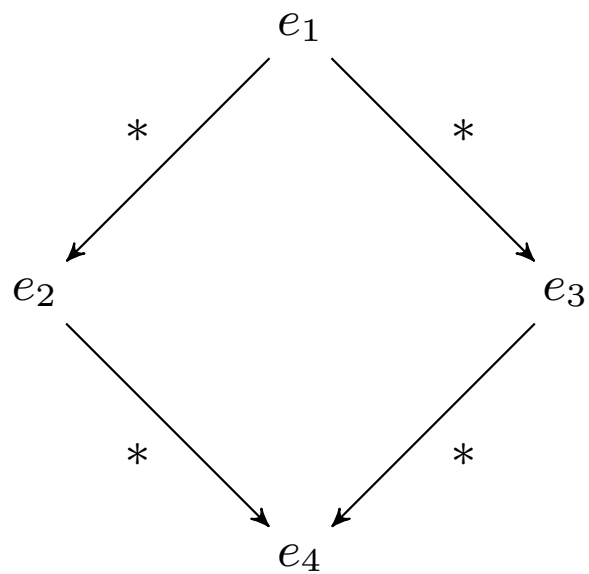
λ-calculus semantics

Conventions:

- an α-redex is a subterm of a λ-term of the form $\lambda x\,.\,e$
- a β-redex is a subterm of a λ-term of the form $(\lambda x\,.\,e_1)\;e_2$
- write $e_1 \xrightarrow{*} e_2$ if e_1 reduces to e_2 in some finite number (zero or more) α- or β-steps
- In the pure λ-calculus, a rule can be performed at any time to any redex — Can we get into trouble?

Church-Rosser theorem

If $e_1 \xrightarrow{*} e_2$ by some sequence of α- or β-steps, and
if $e_1 \xrightarrow{*} e_3$ by some sequence of α- or β-steps, then
there exists e_4 such that $e_2 \xrightarrow{*} e_4$ and $e_3 \xrightarrow{*} e_4$



Values and termination

- A term is in normal form if it has no β-redexes, i.e., no subterm of the form $(\lambda x\,.\,e_1)\;e_2$
- A value is a closed (= no free variables) term in normal form
  Examples:

$$\mathsf{I} = \lambda x\,.\,x \quad \mathsf{K} = \lambda xy\,.\,x \quad \mathsf{B} = \lambda xy\,.\,y \quad \mathsf{S} = \lambda xyz\,.\,xz(yz)$$

- By Church-Rosser, values are unique up to α-conversion
- A term is normalizable (has a value) if some sequence of α- and β-steps leads to a value

Values and termination

- Q: Do computations always terminate in a value?

Values and termination

- Q: Do computations always terminate in a value?
- A: No! Let $\Omega = (\lambda x\,.\,xx)\,(\lambda x\,.\,xx)$; then $\Omega \to \Omega \to \Omega \to \ldots$

Values and termination

- Q: Do computations always terminate in a value?
- A: No! Let $\Omega = (\lambda x\,.\,xx)\,(\lambda x\,.\,xx)$; then $\Omega \to \Omega \to \Omega \to \ldots$
- Q: If a term is normalizable, does every sequence of reductions lead to a value?

Values and termination

- Q: Do computations always terminate in a value?
- A: No! Let $\Omega = (\lambda x\,.\,xx)\,(\lambda x\,.\,xx)$; then $\Omega \to \Omega \to \Omega \to \ldots$
- Q: If a term is normalizable, does every sequence of reductions lead to a value?
- A: No! $(\lambda xy\,.\,y)\,\Omega$

Reduction Strategies

A reduction strategy is a specification that tells which of the possible β-reductions to perform next

The λ-calculus does not specify a reduction strategy; it is nondeterministic

In real programming languages, a reduction strategy is needed to resolve the nondeterminism

Some common reduction strategies:

- Normal order (leftmost outermost)
- Applicative order (leftmost innermost)
- Call by name (normal order, but disallowing reductions in the body of a λ)
- Call by value (applicative order, but disallowing reductions in the body of a λ)

Reduction Strategies

Most functional programming languages use CBV, with the notable exception of Haskell

Under CBV, functions may only be called on values

It is known that if a term is normalizable, then normal order will terminate — also true of CBN

Under CBN, evaluation of arguments are deferred until as late as possible — this is known as lazy evaluation

CBV	CBN
$(\lambda x\,.\,\mathsf{inc}\,x)\,((\lambda y\,.\,\mathsf{inc}\,y)\,3)$	$(\lambda x\,.\,\mathsf{inc}\,x)\,((\lambda y\,.\,\mathsf{inc}\,y)\,3)$
$\rightarrow (\lambda x\,.\,\mathsf{inc}\,x)\,(\mathsf{inc}\,3)$	$\rightarrow \mathsf{inc}\,((\lambda y\,.\,\mathsf{inc}\,y)\,3)$
$\rightarrow (\lambda x\,.\,\mathsf{inc}\,x)\,4$	$\rightarrow \mathsf{inc}\,(\mathsf{inc}\,3)$
$\rightarrow \mathsf{inc}\,4$	$\rightarrow \mathsf{inc}\,4$
$\rightarrow 5$	$\rightarrow 5$

Reduction Strategies

We can formalize CBV and CBN in small-step style

In both cases, values v are closed λ-terms

CBV

$$(\lambda x\,.\,e)\;v \to e[v/x] \qquad\qquad C \;::=\; \texttt{[]} \mid C\;e \mid v\;C$$

CBN

$$(\lambda x\,.\,e_1)\;e_2 \to e_1[v/e_2] \qquad\qquad C \;::=\; \texttt{[]} \mid C\;e$$

Encoding datatypes

Booleans

$$\texttt{true} = \lambda xy.x \qquad \texttt{false} = \lambda xy.y$$

$$\texttt{if-then-else} = \lambda xyz.xyz$$

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 = (\lambda xyz.xyz)\ e_1\ e_2\ e_3$$

$$\texttt{if true then } e_2 \texttt{ else } e_3 = (\lambda xyz.xyz)\ (\lambda ab.a)\ e_2\ e_3$$

$$\rightarrow (\lambda ab.a)\ e_2\ e_3 \rightarrow e_2 \rightarrow \ldots$$

$$\texttt{if false then } e_2 \texttt{ else } e_3 = (\lambda xyz.xyz)\ (\lambda ab.b)\ e_2\ e_3$$

$$\rightarrow (\lambda ab.b)\ e_2\ e_3 \rightarrow e_3 \rightarrow \ldots$$

Encoding datatypes

Other Boolean operations

$$\texttt{and} = \lambda xy.xyx \qquad \text{or} = \lambda xy.xxy$$
$$\texttt{not} = \lambda x.x\ \texttt{false}\ \texttt{true}$$

$$\texttt{and}\ \texttt{true}\ \texttt{true} = (\lambda xy.xyx)\ (\lambda ab.a)\ (\lambda cd.c)$$
$$\rightarrow (\lambda ab.a)\ (\lambda cd.c)\ (\lambda ab.a) \rightarrow \lambda cd.c = \texttt{true}$$

$$\texttt{and}\ \texttt{false}\ \texttt{true} = (\lambda xy.xyx)\ (\lambda ab.b)\ (\lambda cd.c)$$
$$\rightarrow (\lambda ab.b)\ (\lambda cd.c)\ (\lambda ab.b) \rightarrow \lambda ab.b = \texttt{false}$$

Encoding datatypes

Pairs and projections

$$\texttt{pair} = \lambda xyf\,.\,fxy \quad \texttt{fst} = \lambda p.p\,\texttt{true} \quad \texttt{snd} = \lambda p.p\,\texttt{false}$$

$$\texttt{pair}\;e_1\;e_2 = (\lambda xyf\,.\,fxy)\;e_1\;e_2 \to \lambda f\,.\,f\;e_1\;e_2$$

$$\begin{aligned}&\texttt{fst}\;(\texttt{pair}\;e_1\;e_2) = (\lambda p.p\,\texttt{true})\;(\lambda f\,.\,f\;e_1\;e_2)\\ &\to (\lambda f\,.\,f\;e_1\;e_2)\;(\lambda xy\,.\,x) \to (\lambda xy\,.\,x)\;e_1\;e_2 \to e_1\end{aligned}$$

$$\begin{aligned}&\texttt{snd}\;(\texttt{pair}\;e_1\;e_2) = (\lambda p.p\,\texttt{false})\;(\lambda f\,.\,f\;e_1\;e_2)\\ &\to (\lambda f\,.\,f\;e_1\;e_2)\;(\lambda xy\,.\,y) \to (\lambda xy\,.\,y)\;e_1\;e_2 \to e_2\end{aligned}$$

Numbers and arithmetic

Church numerals:

$$\overline{0} = \lambda f x\,.\,x = \lambda f x\,.\,f^0 x$$
$$\overline{1} = \lambda f x\,.\,f x = \lambda f x\,.\,f^1 x$$
$$\overline{2} = \lambda f x\,.\,f(f x) = \lambda f x\,.\,f^2 x$$
$$\overline{3} = \lambda f x\,.\,f(f(f x)) = \lambda f x\,.\,f^3 x$$
$$\ldots$$
$$\overline{n} = \lambda f x\,.\,\underbrace{f(\ldots(f(f(f}_{n}\,x)))\ldots) = \lambda f x\,.\,f^n x$$
$$\ldots$$

Numbers and arithmetic

Successor:

$$\texttt{inc} = \lambda n f x \,.\, f\,(n\,f\,x)$$

$$\begin{aligned}
\texttt{inc}\,\overline{n} &= (\lambda n f x \,.\, f\,(n\,f\,x))\;(\lambda g y \,.\, g^n\, y) \\
&\to \lambda f x \,.\, f\,((\lambda g y \,.\, g^n\, y)\;f\;x) \\
&\to \lambda f x \,.\, f\,(f^n\, x) \\
&= \lambda f x \,.\, f^{n+1}\, x \\
&= \overline{n+1}
\end{aligned}$$

Numbers and arithmetic

Addition:

$$\texttt{add} = \lambda nm.m\ \texttt{inc}\ n$$

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

Numbers and arithmetic

Multiplication:

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

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

Predecessor

Let $$g = \lambda z.\texttt{pair}\,(\texttt{snd}\,z)\,(\texttt{inc}\,(\texttt{snd}\,z))$$
$$= \lambda(x,y).(y, \texttt{inc}\,y)$$

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

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

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

Recursive functions

Recursive functions typically must refer to themselves in the body

$$\texttt{let rec fact}\ x\ =$$
$$\quad \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \texttt{fact}\ (x - 1)$$

This doesn't seem possible in the λ-calculus, because there are no names — all functions are anonymous

We want `fact` to be a solution of the equation

$$\texttt{fact} = \lambda n.(\texttt{if-then-else}\ (\texttt{isZero}\ n)\ 1\ (\texttt{mul}\ n\ (\texttt{fact}\ (\texttt{dec}\ n))))$$

Recursive functions

We want `fact` to be a solution of the equation

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

In other words, let

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

We want `fact` to be a fixpoint of h:

$$\texttt{fact} = h\ \texttt{fact}$$

Recursive functions

But we can construct a fixpoint of any term!

Claim: $(\lambda x\,.\,h(xx))\;(\lambda x\,.\,h(xx))$ is a fixpoint of h

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

More simply, let $Y = \lambda t\,.\,(\lambda x\,.\,t(xx))\;(\lambda x\,.\,t(xx))$

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

So Yh is a fixpoint of h — we can define `fact` $= Yh$!

Simply typed λ-calculus

Let's add types and some primitive data constructs

$$
\begin{array}{llll}
\text{values} & v & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \text{null} \mid \lambda x : \tau \,.\, e \\
\text{terms} & e & ::= & v \mid x \mid e_1\ e_2 \\
\text{types} & \tau & ::= & \text{int} \mid \text{bool} \mid \text{unit} \mid \tau_1 \to \tau_2
\end{array}
$$

Church-style = functions have explicit type annotations $\lambda x : \tau \,.\, e$
Curry-style = no type annotations $\lambda x \,.\, e$

Simply typed λ-calculus

Type judgments $\Gamma \vdash e : \tau$
says that e has type τ under assumptions Γ

where Γ is a type environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$
specifies types for the free variables of e

Simply typed λ-calculus

Typing rules

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash \texttt{true} : \mathsf{bool} \qquad \Gamma \vdash \texttt{false} : \mathsf{bool}$$

$$\Gamma \vdash \mathsf{null} : \mathsf{unit} \qquad \Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_0 : \sigma \to \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash e_0\, e_1 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma\, .e) : \sigma \to \tau}$$

Simply typed λ-calculus

An example type derivation

$$\dfrac{\dfrac{\dfrac{x : \mathsf{int},\ y : \mathsf{bool} \vdash x : \mathsf{int}}{x : \mathsf{int} \vdash (\lambda y : \mathsf{bool}\,.\,x) : \mathsf{bool} \to \mathsf{int}}}{\vdash (\lambda x : \mathsf{int}\,.\,\lambda y : \mathsf{bool}\,.\,x) : \mathsf{int} \to \mathsf{bool} \to \mathsf{int}} \qquad \vdash 2 : \mathsf{int}}{\vdash ((\lambda x : \mathsf{int}\,.\,\lambda y : \mathsf{bool}\,.\,x)\ 2) : \mathsf{bool} \to \mathsf{int}} \qquad \vdash \mathtt{true} : \mathsf{bool}}{\vdash ((\lambda x : \mathsf{int}\,.\,\lambda y : \mathsf{bool}\,.\,x)\ 2\ \mathtt{true}) : \mathsf{int}}$$

Simply typed λ-calculus

Why Typing?

- Avoid runtime type errors
- Well-typed programs cannot get "stuck"

Progress and Preservation

- Progress: If $\vdash e : \tau$, then e is either a value or has a β-redex
- Preservation: If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$

Simply typed λ-calculus

Expressive Power

The simply typed λ-calculus is less expressive than the full λ-calculus — e.g. Ω and Y are not typeable

Most importantly, we cannot write loops:

Theorem

The STLC is strongly normalizing: every reduction sequence terminates.

Proof technique: logical relations

Next time — Denotational semantics!

- Lecture 1: Structural operational semantics
  IMP language, small-step rules, big-step rules, evaluation contexts, big-step rules as binary relations, algebraic properties
- Lecture 2: The λ-calculus
  λ-terms, safe substitution, binding and scope, α- and β-reduction, Church-Rosser theorem, data encodings, recursion, simple types, progress and preservation
- Lecture 3: Denotational semantics
  CPOs, least fixpoints, Knaster-Tarski theorem, domain equations
- Lecture 4: Axiomatic semantics, probabilistic semantics
  Weakest preconditions, semantics of Hoare logic, Dynamic logic, probabilistic semantics

Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

SIGPLAN

https://www.sigplan.org/

Silver Sponsors