We’ve seen that several language conveniences aren’t strictly necessary

- Multi-argument functions (use currying or tuples)

```ocaml
let fst (x, y) = x
let fst x y = x
let fst x =
match p with
  fun y -> x
(x, y) -> x
```

- Loops (use recursion)

```ocaml
r = 0;
let rec sum i r =
  if i >= n then r
  else sum (i+1) (r+i)
in
sum 0 0
```

Goal: Come up with a “core” language that’s as small as possible and still Turing complete

This will give a way of illustrating important language features and algorithms
Lambda Calculus Syntax

- A lambda calculus expression is defined as
  
  $$\begin{align*}
e & ::= x & \text{variable} \\
   & | \lambda x. e & \text{function} \\
   & | e e & \text{function application}
  \end{align*}$$

- $\lambda x. e$ is like `(fun x -> e)` in OCaml

- That’s it! All there is is higher-order functions

Conventions

- The scope of $\lambda$ extends as far to the right as possible
  
  $$\lambda x. \lambda y. x y \quad \text{is} \quad \lambda x.(\lambda y.(x y))$$

- Function application is left-associative
  
  $$x y z \quad \text{is} \quad (x y) z$$
  
  – Same rule as OCaml

Operational Semantics

- All we’ve got are functions, so all we can do is call them

- To evaluate $(\lambda x. e_1) e_2$
  
  - Evaluate $e_1$ with $x$ bound to $e_2$

  \[ (\lambda x. e_1) e_2 \rightarrow e_1[x/e_2] \]

- $e_1[x/e_2]$ is $e_1$ where occurrences of $x$ are replaced by $e_2$

- Slightly different than the environments we saw for Ocaml
  
  – apply a substitution instead of carry an environment

- We allow reductions to occur anywhere in a term

Examples

- $(\lambda x.x) z \rightarrow z$
- $(\lambda x.y) z \rightarrow y$
- $(\lambda x.x y) z \rightarrow zy$
  
  – A function that applies its argument to $y$

- $(\lambda x.x y) (\lambda z.z) \rightarrow (\lambda z.z) y \rightarrow y$
- $(\lambda x.\lambda y.x y) z \rightarrow \lambda y.z y$
  
  – A curried function of two arguments that applies its first argument to its second

- $(\lambda x.\lambda y.x y) (\lambda z.zz) x \rightarrow \lambda y.(\lambda z.zz)y)x \rightarrow (\lambda z.zz)x \rightarrow xx$
Syntactic sugar for local declarations

\[
\text{let } x = e_1 \text{ in } e_2
\]

can be written as

\[
(\lambda x. e_2) \; e_1
\]

Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - \((\lambda x. (\lambda x. x)) \; z \rightarrow ?\)
    - The rightmost “\(x\)” refers to the second binding
    - This is a function that takes its argument and applies it to the identity function
  - This function is “the same” as \((\lambda x. (\lambda y. y))\)
    - Renaming bound variables consistently is allowed

**alpha-conversion (alpha-renaming)**

Ex. \(\lambda x. y = \lambda z. x \quad \lambda y. \lambda x. y = \lambda z. \lambda x. z\)

Static Scoping (cont’d)

- How about the following?
  - \((\lambda x. \lambda y. x \; y) \; y \rightarrow ?\)
    - When we replace \(y\) inside, we don’t want it to be “captured” by the inner binding of \(y\)
  - This function is “the same” as \((\lambda x. \lambda z. x \; z)\)

Beta-Reduction, Again

- Whenever we do a step of beta reduction...
  - \((\lambda x. e_1) \; e_2 \rightarrow e_1[x/e_2]\)
    - ...alpha-convert variables as necessary

- Examples:
  - \((\lambda x. (\lambda x. x)) \; z = (\lambda x. (\lambda y. y)) \; z \rightarrow z \; (\lambda y. y)\)
  - \((\lambda x. \lambda y. x \; y) \; y = (\lambda x. \lambda z. x \; z) \; y \rightarrow \lambda z. y \; z\)
Encodings

- It turns out that this language is Turing complete
- That means we can encode any computation we want in it
  - ...if we’re sufficiently clever...

Booleans

The lambda calculus was created by logician Alonzo Church in the 1930’s to formulate a mathematical logical system

true = \lambda x.\lambda y.x
false = \lambda x.\lambda y.y
if a then b else c = a b c (the \lambda expression)

- Examples:
  - if true then b else c → (\lambda x.\lambda y.x) b c → (\lambda y.b) c → b
  - if false then b else c → (\lambda x.\lambda y.y) b c → (\lambda y.c) → c

Booleans (continued)

Other Boolean operations:
- not = \lambda x.((x false) true)
- not true → \lambda x.((x false) true) true → ((true false) true) → false
- and = \lambda x.\lambda y.((xy) false)
- or = \lambda x.\lambda y.((x true) y)
- Show not, and and or have the desired properties, …
- Given these operations, can build up a logical inference system

Pairs

(a,b) = \lambda x.\lambda y.(\lambda z.\lambda w.\lambda z.w) x y
fst = \lambda f.f true
snd = \lambda f.f false

- Examples:
  - fst (a,b) = (\lambda f.true) (\lambda z.\lambda w.\lambda z.w) x y → (\lambda x.\lambda z.w) x y → if true then a else b → a
  - snd (a,b) = (\lambda f.false) (\lambda z.\lambda w.\lambda z.w) x y → (\lambda x.\lambda z.w) x y → if false then a else b → b
Natural Numbers (Church*)

*(Named after Alonzo Church, developer of lambda calculus)

0 = \lambda f.\lambda y.y
1 = \lambda f.\lambda y.f y
2 = \lambda f.\lambda y.f (f y)
3 = \lambda f.\lambda y.f (f (f y))

i.e., n = \lambda f.\lambda y.<apply f n times to y>

\text{succ} = \lambda z.\lambda f.\lambda y.f (f z y)
\text{iszero} = \lambda g.\lambda y.\text{false} true

– Recall that this is equivalent to \lambda g.((g (\lambda y.\text{false})) true)

Arithmetic defined

• Addition, if M and N are integers (as \lambda expressions):
  \text{M + N} = \lambda x.\lambda y.((\text{M} x)((N x) y))

  Equivalently: \text{+} = \lambda M.\lambda N.\lambda x.\lambda y.((\text{M} x)((N x) y))

• Multiplication: \text{M * N} = \lambda x.\lambda y.((\text{M} (\text{N} x)) x)

• Prove 1+1 = 2.
  \begin{align*}
  1+1 &= \lambda x.\lambda y.((1 x)(1 x) y) \\
  &= \lambda x.\lambda y.((\lambda x.\lambda y.(\lambda y x y) x) y) \\
  &= \lambda x.\lambda y.((\lambda y x y) y) \\
  &= \lambda x.\lambda y.((\lambda y x)(\lambda y x) y) \\
  &= \lambda x.\lambda y.((\lambda y x) y) \\
  &= \lambda x.\lambda y.x \\
  &= 2
  \end{align*}

• With these definitions, can build a theory of integer arithmetic.

Natural Numbers (cont’d)

• Examples:
  \text{succ} 0 =
  (\lambda z.\lambda f.\lambda y.(z f y)) (\lambda f.\lambda y.y) \rightarrow
  \lambda f.\lambda y.f ((\lambda f.\lambda y.y) f y) \rightarrow
  \lambda f.\lambda f.y = 1

  \text{iszero} 0 =
  (\lambda z.\lambda y.\text{false}) true

  \text{iszero} 1 =
  (\lambda f.\lambda y.\text{false}) true

  \text{iszero} 2 =
  (\lambda y.\text{true}) true

What else?

• What about looping or recursion?

  let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
Looping

• Define \( D = \lambda x. x x \)

• Then
  
  \[ D \: D = (\lambda x. x x) \: (\lambda x. x x) \rightarrow (\lambda x. x x) \: (\lambda x. x x) = D \: D \]

• So \( D \: D \) is an infinite loop
  
  – In general, *self application* is how we get looping

The “Paradoxical” Combinator

\[ Y = \lambda f. (\lambda x. f (x x)) \: (\lambda x. f (x x)) \]

• Then

\[
Y \: F = \\
(\lambda f. (\lambda x. f (x x)) \: (\lambda x. f (x x))) \: F \rightarrow \\
(\lambda x. F (x x)) \: (\lambda x. F (x x)) \rightarrow \\
F \: ((\lambda x. F (x x)) \: (\lambda x. F (x x))) \\
= F \: (Y \: F)
\]

• Thus \( Y \: F = F \: (Y \: F) = F \: (F \: (Y \: F)) = ... \)

Example

\[ \text{fact} = \lambda f. \: \lambda n. \text{if} \: n = 0 \: \text{then} \: 1 \: \text{else} \: n \: * \: (f \: (n-1)) \]

– The second argument to fact is the integer

– The first argument is the function to call in the body
  
  • We’ll use \( Y \) to make this recursively call fact

\[
(Y \: \text{fact}) \: 1 = (\text{fact} \: (Y \: \text{fact})) \: 1 \\
\rightarrow \text{if} \: 1 = 0 \: \text{then} \: 1 \: \text{else} \: 1 \: * \: ((Y \: \text{fact}) \: 0) \\
\rightarrow 1 \: * \: ((Y \: \text{fact}) \: 0) \\
\rightarrow 1 \: * \: (\text{fact} \: (Y \: \text{fact}) \: 0) \\
\rightarrow 1 \: * \: (\text{if} \: 0 = 0 \: \text{then} \: 1 \: \text{else} \: 0 \: * \: ((Y \: \text{fact}) \: (-1))) \\
\rightarrow 1 \: * \: 1 \rightarrow 1
\]

Discussion

• Using encodings we can represent pretty much anything we have in a “real” language
  
  – But programs would be pretty slow if we really implemented things this way

  – In practice, we use richer languages that include built-in primitives

• Lambda calculus shows all the issues with scoping and higher-order functions

• It’s useful for understanding how languages work
The Need for Types

• Consider the untyped lambda calculus
  – false = λx.λy.y
  – 0 = λx.λy.y

• Since everything is encoded as a function...
  – We can easily misuse terms
    • false 0 → λy.y
    • if 0 then ...
    • Everything evaluates to some function

• The same thing happens in assembly language
  – Everything is a machine word (a bunch of bits)
  – All operations take machine words to machine words

What is a Type System?

• A type system is some mechanism for distinguishing good programs from bad
  – Good = well typed
  – Bad = ill typed or not typable; has a type error

• Examples
  – 0 + 1 // well typed
  – false 0 // ill-typed; can’t apply a boolean

Static versus Dynamic Typing

• In a static type system, we guarantee at compile time that all program executions will be free of type errors
  – OCaml and C have static type systems

• In a dynamic type system, we wait until runtime, and halt a program (or raise an exception) if we detect a type error
  – Ruby has a dynamic type system

• Java, C++ have a combination of the two

Simply-Typed Lambda Calculus

• e ::= n | x | λx:te | ee

  – We’ve added integers n as primitives
  – Without at least two distinct types (integer and function), can’t have any type errors

  – Functions now include the type of their argument

• t ::= int | t → t

  – int is the type of integers
  – t1 → t2 is the type of a function that takes arguments of type t1 and returns a result of type t2
  – t1 is the domain and t2 is the range

  – Notice this is a recursive definition, so that we can give types to higher-order functions
Looping?

- Simply-typed lambda calculus disallows looping

\[ D = \lambda x : ?. \quad x \quad x \]

Type Judgments

- We will construct a type system that proves judgments of the form

\[ A \vdash e : t \]

- “In type environment \( A \), expression \( e \) has type \( t \)”

- If for a program \( e \) we can prove that it has some type, then the program type checks
  - Otherwise the program has a type error, and we’ll reject the program as bad

Type Environments

- A type environment is a map from variables names to their types
  - Just like in our operational semantics for Scheme

- is the empty type environment

- \( A, x : t \) is just like \( A \), except \( x \) now has type \( t \)

- When we see a variable in the program, we’ll look up its type in the environment

Type Rules

\[ e ::= n \mid x \mid \lambda x : t . e \mid e \ e \]

\[ \begin{align*}
A \vdash n : \text{int} \\
A \vdash x \in \text{dom}(A) \\
A \vdash x : A(x) \\
A, x : t \vdash e : t' \\
A \vdash \lambda x : t . e : t' \\
A \vdash e : t \rightarrow t' \\
A \vdash e' : t \\
A \vdash e \ e' : t'
\end{align*} \]
Example

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
\[ B = A, x : \text{int} \]

\[ \frac{\text{B} \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int}}{\text{B} \vdash x : \text{int}} \]
\[ \frac{\text{B} \vdash + x : \text{int} \rightarrow \text{int}}{\text{B} \vdash 3 : \text{int}} \]
\[ \frac{\text{B} \vdash + 3 : \text{int}}{\text{A} \vdash (\lambda x : \text{int} \rightarrow x + 3) : \text{int} \rightarrow \text{int}} \]
\[ \frac{\text{A} \vdash 4 : \text{int}}{\text{A} \vdash (\lambda x : \text{int} \rightarrow x + 3) 4 : \text{int}} \]

Discussion

- The type rules are a kind of logic for reasoning about types of programs
  - The tree of judgments we just saw is a kind of proof in this logic that the program has a valid type
- So the type checking problem is like solving a jigsaw puzzle
  - Can we apply the rules to a program in such a way as to produce a typing proof?
  - It turns out we can easily decide whether or not we can do this.

An Algorithm for Type Checking

(Write this in OCaml!)

TypeCheck : type env \times expression \rightarrow type

\[ \text{TypeCheck}(A, n) = \text{int} \]
\[ \text{TypeCheck}(A, x) = \text{if } x \in \text{dom}(A) \text{ then } A(x) \text{ else fail} \]
\[ \text{TypeCheck}(A, \lambda x : t. e) = \]
\[ \quad \text{let } t' = \text{TypeCheck}((A, x : t), e) \text{ in } t \rightarrow t' \]
\[ \text{TypeCheck}(A, e1 e2) = \]
\[ \quad \text{let } t1 = \text{TypeCheck}(A, e1) \text{ in} \]
\[ \quad \text{let } t2 = \text{TypeCheck}(A, e2) \text{ in} \]
\[ \quad \text{if } \text{dom}(t1) = t2 \text{ then } \text{range}(t1) \text{ else fail} \]

Type Inference

- We could extend the rules to show how a language could figure out, even if types aren’t specified, what the types of everything are in a program
  - Can you believe there are languages which can actually do this?
- We could do these things, but we actually won’t.
Summary

• Lambda calculus shows all the issues with scoping and higher-order functions

• It's useful for understanding how languages work