Introduction

- Loops (use recursion)
- Side-effects

\[
\begin{align*}
  & r = 0; \\
  & \text{for } (i = 0; i < n; i++) \\n  & \quad r += i; \\
\end{align*}
\]

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

\[
\begin{align*}
  e ::= & x & \text{variable} \\
  & \lambda x. e & \text{function} \\
  & e \: e & \text{function application} \\
  & \lambda x. e & \text{like } (\text{fun } x \to e) \text{ in OCaml} \\
  & \text{That’s it! All there is is higher-order functions}
\end{align*}
\]

Conventions

- The scope of \( \lambda \) extends as far to the right as possible

\[
\lambda x. \lambda y. x y \text{ is } \lambda x. (\lambda y. (x y))
\]

- Function application is left-associative

\[
\text{Same rule as OCaml}
\]

\[
\begin{align*}
  & \text{let } \text{fst } (x, y) \equiv x \\
  & \text{let } \text{fst } x \equiv x \\
  & \text{let } \text{fst } p = \\
  & \quad \text{match } p \text{ with} \\
  & \quad \quad (x, y) \to x \\
  & \quad \quad \text{fun } y \to x
\end{align*}
\]
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 \(e_2\)

\[
\text{beta-reduction}
\]

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

- \(e_1[x/e_2]\) 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

- \((\lambda \, x. x) \, z \rightarrow z\)
- \((\lambda \, x. y) \, z \rightarrow y\)
- \((\lambda \, x. x \, y) \, z \rightarrow z \, y\)
  - 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
- \((\lambda \, x. \, \lambda \, y. x \, y) \, (\lambda \, z. zz) \) \rightarrow \lambda \, y.((\lambda \, z. zz)y) \rightarrow (\lambda \, z. zz)x \rightarrow xx\)

Syntactic sugar for local declarations

- let \(x = e_1\) 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

\[
\text{alpha-conversion (alpha-renaming)}
\]

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

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

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:
  
  \((\lambda \, x. x \, (\lambda \, x. x)) \, z = (\lambda \, x. x \, (\lambda \, y. y)) \, z \rightarrow (\lambda \, y. y)\)
  
  \((\lambda \, x. \, \lambda \, y. x \, y) \, y = (\lambda \, x. \, \lambda \, z. x \, z) \, y \rightarrow \lambda \, z. y \, z\)
**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** = λx. λy. x
- **false** = λx. λy. y
- **if a then b else c** = a b c (the λ expression)

- **Examples:**
  - if true then b else c = (λx. λy. x) b c
  - if false then b else c = (λx. λy. y) b c

**Booleans (continued)**

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

**Pairs**

**(a,b) = λx. if x then a else b**

**fst** = λ f. f true
**snd** = λ f. f false

- **Examples:**
  - **fst** (a,b) = (λ f. f true) (λ x. if x then a else b) →
    (λ x. if x then a else b) true → a
  - **snd** (a,b) = (λ f. f false) (λ x. if x then a else b) →
    (λ x. if x then a else b) false → b

**Natural Numbers (Church*)**

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

- 0 = λ f. f y
- 1 = λ f. f y y
- 2 = λ f. f (f y)
- 3 = λ f. f (f (f y))
  - i.e., n = λ f. f y. <apply f n times to y>

**succ** = λ z. λ f. λ y. f (z f y)
**iszero** = λ g. g (λ y. false) true
  - Recall that this is equivalent to λ g. ((g (λ y. false)) true)

**Natural Numbers (cont’d)**

- **Examples:**
  - **succ** 0 =
    (λ z. λ f. λ y. f (z f y)) (λ f. λ y. y) →
    λ f. λ y. f (λ f. λ y. y) →
    λ f. λ y. y = 1
  - **iszero** 0 =
    (λ z. λ f. (λ y. false) true) (λ f. λ y. y) →
    (λ f. λ y. y) (λ y. false) true →
    (λ f. lambda calculus)**
Arithmetic defined

- Addition, if $M$ and $N$ are integers (as $\lambda$ expressions):
  $$M + N = \lambda x. \lambda y.((M x)(N x)y)$$
  Equivalently: $+ = \lambda M. \lambda N. \lambda x. \lambda y.((M x)(N x)y)$
- Multiplication: $M \times N = \lambda x.((M x)(N x)x)$
- Prove $1+1 = 2$.
  $$1+1 = \lambda x. \lambda y.((\lambda y.x y)((\lambda x.\lambda y.x y)x)y)\rightarrow$$
  $$\lambda x. \lambda y.((\lambda y.x y)((\lambda x.\lambda y.x y)x)y)\rightarrow$$
  $$\lambda x. \lambda y.((\lambda y.x y)((\lambda x.\lambda y.x y)x)y)\rightarrow$$
  $$\lambda x. \lambda y.x((\lambda y.x y)y)\rightarrow$$
  $$\lambda x. \lambda y.x x = 2$$
- With these definitions, can build a theory of integer arithmetic

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
  - $D D = ((\lambda x.\lambda x.x x)(\lambda x.x x))\rightarrow (\lambda x.\lambda x.x x)(\lambda x.x x) = D D$
- So $D D$ is an infinite loop
  - In general, self application is how we get looping

The “Paradoxical” Combinator

- Define $Y = \lambda f.((\lambda x.f (x x)) (\lambda x.f (x x)))$
- 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

- Define $fact = \lambda f. \lambda n. if n = 0 then 1 else n * (f (n-1))$
  - 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
  $$(Y \text{fact}) 1 = (\text{fact} (Y \text{fact})) 1$$
    - if $1 = 0$ then 1 else 1 * ($(Y \text{fact}) 0)$
    - 1 * ($(Y \text{fact}) 0)$
    - 1 * (fact (Y fact) 0)
    - 1 * (if 0 = 0 then 1 else 0 * ($(Y \text{fact}) (-1)$)
    - 1 * 1 = 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 = \( \lambda x. \lambda y.y \)
  – 0 = \( \lambda x. \lambda y.y \)
• Since everything is encoded as a function...
  – We can easily misuse terms
    • false 0 \( \rightarrow \lambda y.y \)
    • 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 \( \rightarrow \) well typed
  – false 0 \( \rightarrow \) 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

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

• 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

\[ t ::= \text{int} | t \rightarrow t \]

• \text{int} is the type of integers
• \( t_1 \rightarrow t_2 \) is the type of a function that takes arguments of type \( t_1 \) and returns a result of type \( t_2 \)
• \( t_1 \) is the domain and \( t_2 \) 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

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

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
- \( \ast \) 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

Example

\[
A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int}  \\
B = A, x : \text{int}  \\
B \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
B \vdash x : \text{int}  \\
B \vdash x + 3 : \text{int}  \\
B \vdash (\lambda x : \text{int}. x + 3) : \text{int} \rightarrow \text{int} \\
B \vdash \lambda x : \text{int}. (x + 3) : \text{int} \rightarrow \text{int} \\
A \vdash \lambda x : \text{int}. (x + 3) 4 : \text{int}  \\
A \vdash + (\lambda x : \text{int}. (x + 3) 4) 5 : \text{int}  \\
A \vdash \lambda x : \text{int}. (x + 3) 4 + 5 : \text{int}  \\
A \vdash \text{int}  \\
\]

Type Rules

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

\[
A \vdash n : \text{int}  \\
A \vdash x : A(x)  \\
\]

\[
A, x : t \vdash e : t'  \\
A \vdash \lambda x : t. e : t'  \\
A \vdash e e' : t'  \\
\]

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!)

\[
\text{TypeCheck : type env} \times \text{expression} \rightarrow \text{type}
\]

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

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