CMPS 116: Fall 2019

Introduction to Functional Programming

Course review

Owen Arden
UC Santa Cruz

Based on course materials developed by Ranjit Jhala

The Lambda Calculus

- Lambda calculus terms
  - variables, abstractions, & applications
- Variable scope
  - Free vs bound variables
- Evaluation
  - Alpha renaming
  - Beta reduction
  - Normal form
- Church encodings
  - numbers, booleans, etc
- Recursion
  - Fixed-point combinator

Haskell

- A **typed**, **lazy**, **purely functional** programming language
  - Haskell = λ-calculus +
    - Better syntax
    - Types
    - Built-in features
      - Booleans, numbers, characters
      - Records (tuples)
      - Lists
      - Recursion
      - …

Haskell topics

- Haskell's type system
  - Recognizing / understanding relationship between Haskell expressions and their types
- Algebraic data types
  - Records
  - Sum types
  - Recursive ADTs
- Pattern matching
  - Overlapped / missing patterns
- Writing algorithms on (recursive) ADTs
  - Base cases + inductive cases

Higher Order Functions

Iteration patterns over collections:

- **Filter** values in a collection given a *predicate*
- **Map** (iterate) a given *transformation* over a collection
- **Fold** (reduce) a collection into a value, given a *binary operation* to combine results

Useful helper HOFs:

- **Flip** the order of function's (first two) arguments
- **Compose** two functions

Evaluating Nano1

Back to our expressions… now with environments!

```
data Expr = Num Int             -- number
          | Var Id              -- variable
          | Bin Binop Expr Expr -- binary expression
          | Let Id Expr Expr    -- let expression
```

Static vs Dynamic Scoping

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding *during program execution*
- can't tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!!
res2 - res1
```


Static vs Dynamic Scoping

What we want:

```
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 84
```

Lexical (or **static**) scoping:

- each occurrence of a variable refers to the most recent binding *in the program text*
- definition of each variable is unique and known *statically*
- good for readability and debugging: don't have to figure out where a variable got "assigned"

Static vs Dynamic Scoping

What we **don't** want:

```
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 10
```

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding *during program execution*
- can't tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

Static vs Dynamic Scoping

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding *during program execution*
- can't tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!!
res2 - res1
```


Closures

To implement lexical scoping, we will represent function values as *closures*

Closure = *lambda abstraction* (formal + body) + *environment* at function definition

```
data Value = VNum Int
           | VClos Env Id Expr -- env + formal + body
```


Formalizing Nano

Goal: we want to guarantee properties about programs, such as:

- evaluation is deterministic
- all programs terminate
- certain programs never fail at run time
- etc.

To prove theorems about programs we first need to define formally

- their *syntax* (what programs look like)
- their *semantics* (what it means to run a program)

Type system for Nano2

A **type system** defines what types an expression can have

To define a type system we need to define:

- the *syntax* of types: what do types look like?
- the *static semantics* of our language (i.e. the typing rules): assign types to expressions

```
G |- e :: T
```

An expression `e` **has type** `T` in `G` if we can derive `G |- e :: T` using these rules

An expression `e` is **well-typed** in `G` if we can derive `G |- e :: T` for some type `T`

- and **ill-typed** otherwise

Double identity

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)
```

Intuitively this program looks okay, but our type system *rejects* it:

- in the first application, `id` needs to have type `Int -> Int`
- in the second application, `id` needs to have type `(Int -> Int) -> (Int -> Int)`
- the type system forces us to pick *just one type* for each variable, such as `id` :(

What can we do?

Inference with polymorphic types

With polymorphic types, we can derive `e :: Int -> Int` where `e` is

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)
```

At a high level, inference works as follows:

1. When we have to pick a type `T` for `x`, we pick a **fresh type variable** `a`
2. So the type of `\x -> x` comes out as `a -> a`
3. We can **generalize** this type to `forall a . a -> a`
4. When we apply `id` the first time, we **instantiate** this polymorphic type with `Int`
5. When we apply `id` the second time, we **instantiate** this polymorphic type with `Int ->Int`

Let's formalize this intuition as a type system!

Typing rules

We need to change the typing rules so that:

1. Variables (and their definitions) can have polymorphic types

```
[T-Var]  G |- x :: S          if x:S in G

        G |- e1 :: S   G, x:S |- e2 :: T
[T-Let] ------------------------------------
           G |- let x = e1 in e2 :: T
```


Typing rules

2. We can *instantiate* a type scheme into a type

```
         G |- e :: forall a . S
[T-Inst] ----------------------
          G |- e :: [a / T] S
```

3. We can *generalize* a type with free type variables into a type scheme

```
             G |- e :: S
[T-Gen] ---------------------- if not (a in FTV(G))
        G |- e :: forall a . S
```


Typing rules

The rest of the rules are the same:

```
[T-Num]  G |- n :: Int

         G |- e1 :: Int   G |- e2 :: Int
[T-Add]  -------------------------------
               G |- e1 + e2 :: Int

          G, x:T1 |- e :: T2
[T-Abs] ------------------------
        G |- \x -> e :: T1 -> T2

        G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App] -----------------------------------
                 G |- e1 e2 :: T2
```

Nano1: Operational Semantics

We define the step relation *inductively* through a set of *rules*:

```
              e1 => e1'        -- premise
[Add-L]   --------------------
          e1 + e2 => e1' + e2   -- conclusion

             e2 => e2'
[Add-R]   --------------------
          n1 + e2 => n1 + e2'

[Add]     n1 + n2 => n        where n == n1 + n2

                       e1 => e1'
[Let-Def] --------------------------------------
          let x = e1 in e2 => let x = e1' in e2

[Let]     let x = v in e2 => e2[x := v]
```


Operational semantics

We need to extend our reduction relation with rules for abstraction and application:

```
           e1 => e1'
[App-L] ----------------
        e1 e2 => e1' e2

          e => e'
[App-R] ------------
        v e => v e'

[App]   (\x -> e) v => e[x := v]
```


Spring 19 final review

Now what?

Did you like what you learned here? Want to learn more?

- **CSE 114 (not 116) Functional Programming**
  - Someday?
- **CSE 110A Fundamentals of Compiler Design**
  - Fall 2019, Spring 2020, Wesley Mackey
  - Winter 2020, me
- **CSE 210A: Programming languages**
  - Winter 2020, Cormac Flanagan
- **CSE 210B: Adv. Programming languages**
  - Spring 2020, me

Thanks and good luck!