Simply Typed Lambda Calculus, Progress and Preservation

**Announcements**
- HW3 due today
- HW5 is out and due on Tuesday after the break

**Outline**
- Pure lambda calculus (catch up)
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Normal forms
  - Reduction strategies
- Lambda calculus interpreters
- Coding them in Haskell

**Rules of Lambda Calculus: Exercises**
- Use $\alpha$-conversion and/or $\beta$-reduction:
  - $(\lambda x. x) y \rightarrow_\alpha$ ?
  - $(\lambda x. x) (\lambda y. y) \rightarrow_\alpha$ ?
  - $(\lambda x. \lambda y. \lambda z. x z (y z)) (\lambda u. u) (\lambda v. v) \rightarrow_\alpha$

  Notation: $\rightarrow_\alpha$ denotes that expression on the left reduces to the expression on the right, through a sequence $\alpha$-conversions and $\beta$-reductions.

**Definitions of Normal Form**
- Normal form (NF): a term without redexes
- Head normal form (HNF)
  - $x$ is in HNF
  - $(\lambda x. E)$ is in HNF if $E$ is in HNF
  - $(x E_1 E_2 \ldots E_n)$ is in HNF
- Weak head normal form (WHNF)
  - $x$ is in WHNF
  - $(\lambda x. E)$ is in WHNF
  - $(x E_1 E_2 \ldots E_n)$ is in WHNF

**Reductions**
- An expression $(\lambda x. E) M$ is called a redex (for reducible expression)
- An expression is in normal form if it cannot be $\beta$-reduced
- The normal form is the meaning of the term, the “answer”
Questions

- \( \lambda z. z \) is in NF, HNF, or WHNF?
- \( \lambda z. z \) \((\lambda x. x)\) is in?
- \( \lambda x. \lambda y. \lambda z. x \) \((y \ (\lambda u. u))\) is in?
- \((\lambda x. \lambda y. x)\) \(z\) \((\lambda x. z \) \((\lambda x. z))\) is in?
- \(z \ ((\lambda x. z \) \((\lambda x. z))\) is in?
- \((\lambda z. (\lambda x. \lambda y. x)\) \((\lambda x. z \) \((\lambda x. z)\))\) is in?

Exercise

- \( S = \lambda x. \lambda y. \lambda z. x \) \((y z)\)
- \( I = \lambda x. x\)
- What is \( S I I I\)?

\[ \begin{align*}
\text{Path 1: } & (\lambda x. \lambda y. \lambda z. x \) \((y z)\) \(\rightarrow_{\beta} (\lambda y. \lambda z. (I \ z) \ (I I)) \rightarrow_{\beta} (\lambda x. (\lambda v. v) \ I) \rightarrow_{\beta} I \)
\end{align*} \]

Simple Reduction Exercise

- \( C = \lambda x. \lambda y. \lambda f. f \) \(x y\)
- \( H = \lambda f. f \) \((\lambda x. \lambda y. x)\)
- \( T = \lambda f. f \) \((\lambda x. \lambda y. y)\)

Outlines

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
- Normal form
- Reduction strategies
- Lambda calculus interpreters
- Coding them in Haskell

Reduction Strategy

- Let us look at \((\lambda x. \lambda y. \lambda z. \) \(x \) \(y z)\) \((\lambda u. u)\) \((\lambda v. v)\)

  - Actually, there are (at least) two “reduction paths”:
    - Path 1: \((\lambda x. \lambda y. \lambda z. \) \(x z \) \((y z))\) \((\lambda u. u)\) \((\lambda v. v)\) \(\rightarrow_{\beta} (\lambda y. \lambda z. \) \((\lambda u. u)\) \((y z))\) \(\rightarrow_{\beta} (\lambda z. \) \((\lambda v. v)\) \((y z))\) \(\rightarrow_{\beta} \lambda z. z\)
    - Path 2: \((\lambda x. \lambda y. \lambda z. \) \(x z \) \((y z))\) \((\lambda u. u)\) \((\lambda v. v)\) \(\rightarrow_{\beta} (\lambda y. \lambda z. \) \((\lambda u. u)\) \((y z))\) \(\rightarrow_{\beta} (\lambda z. \) \((\lambda v. v)\) \((y z))\) \(\rightarrow_{\beta} \lambda z. z\)

- A reduction strategy (also called evaluation order) is a strategy for choosing redexes
  - How do we arrive at the normal form (answer)?
  - Applicative order reduction chooses the leftmost-innermost redex in an expression
    - Also referred to as call-by-value reduction
  - Normal order reduction chooses the leftmost-outermost redex in an expression
    - Also referred to as call-by-name reduction
Reduction Strategy: Examples

- Evaluate \((\lambda x. x x) (\lambda y. y) (\lambda z. z)\)
- Using applicative order reduction:
  \[(\lambda x. x x) (\lambda y. y) (\lambda z. z)\]
  \[\rightarrow (\lambda z. z) (\lambda z. z)\]
  \[\rightarrow (\lambda z. z)\]

- Using normal order reduction:
  \[(\lambda x. x x) (\lambda y. y) (\lambda z. z)\]
  \[\rightarrow (\lambda y. y) (\lambda z. z)\]
  \[\rightarrow (\lambda y. y) (\lambda z. z)\]
  \[\rightarrow (\lambda y. y) (\lambda z. z)\]
  \[\rightarrow (\lambda z. z)\]

Church-Rosser Theorem

- Normal form implies that there are no more reductions possible
- Church-Rosser Theorem, informally:
  - If normal form exists, then it is unique (i.e., result of computation does not depend on the order that reductions are applied; i.e., no expression can have two distinct normal forms)
  - If normal form exists, then normal order will find it

Exercises

- Evaluate \((\lambda x. \lambda y. x y) ((\lambda z. z) w)\)
  - Using applicative order reduction
  - Using normal order reduction

Reduction Strategy

- In our examples, both strategies produced the same result. Is this always the case?
  - First, look at expression \((\lambda x. x x) (\lambda x. x x)\). What happens when we apply \(\beta\)-reduction to this expression?
    - Then look at \((\lambda z. y) ((\lambda x. x x) (\lambda x. x x))\)
      - Applicative order reduction – what happens?
      - Normal order reduction – what happens?

Interpreters

- An interpreter for the lambda calculus is a program that reduces lambda expressions to “answers”
  - We must specify
    - The definition of “answer”. Which normal form?
    - The reduction strategy. How do we choose redexes in an expression?
An Interpreter

Definition by cases on $E ::= x \mid \lambda x. E_1 \mid E_1 E_2$

- $\text{interpret}(x) = x$
- $\text{interpret}(\lambda x. E_1) = \lambda x. \text{interpret}(E_1)$
- $\text{interpret}(E_1 E_2) =$ let $f = \text{interpret}(E_1)$
  in case $f$ of
  $\lambda x. E_3 -> \text{interpret}(E_3[E_2/x])$
  $- -> f E_2$

What normal form: Weak head normal form
What strategy: Normal order

Another Interpreter

Definition by cases on $E ::= x \mid \lambda x. E_1 \mid E_1 E_2$

- $\text{interpret}(x) = x$
- $\text{interpret}(\lambda x. E_1) = \lambda x. \text{interpret}(E_1)$
- $\text{interpret}(E_1 E_2) =$ let $f = \text{interpret}(E_1)$
  in case $f$ of
  $\lambda x. E_3 -> \text{interpret}(E_3[a/x])$
  $- -> f a$

What normal form: Weak head normal form
What strategy: Applicative order

Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Reduction strategies
  - Normal form
- Lambda calculus interpreters
- Coding them in Haskell

Coding them in Haskell

- In HW5 you will code interpreters in Haskell
- Haskell
  - A functional programming language
- Key ideas
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching
  - Monads ... and more

Lazy Evaluation

Unlike Scheme (and most programming languages)
Haskell does lazy evaluation, i.e., normal order reduction
- It won't evaluate an argument expr. until it is needed

> f x = [] // f takes x and returns the empty list
> f (repeat 1) // returns?
> []
> head (tail [1..]) // returns?
> 2 // [1..] is infinite list of integers
- Lazy evaluation allows us to work with infinite structures!

Static Typing and Type Inference

Unlike Scheme, which is dynamically typed, Haskell is statically typed!

Unlike Java/C++ we don’t always have to write type annotations. Haskell infers types!
- A lot more on type inference later!

> f x = head x // f returns the head of list x
> f True // returns?
- Couldn't match expected type '[a]' with actual type 'Bool'
- In the first argument of 'f', namely 'True'
  In the expression: f True …
Algebraic Data Types

- Algebraic data types are **tagged unions** (aka **sums**) of **products** (aka **records**)

```haskell
data Shape = Line Point Point |
            Triangle Point Point Point |
            Quad Point Point Point Point
```

Haskell keyword: `data` (aka `typedef` in C)

- Constructors create new values

```haskell
> e1 = Var "x" // Lambda term x
> e2 = Lambda "x" e1 // Lambda term λx.x
```

Examples of Algebraic Data Types

- Polymorphic types.
  - `a` is a type parameter!

```haskell
data Bool = True | False
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

data List a = Nil |
            Cons a (List a)

data Tree a = Leaf a |
              Node (Tree a) (Tree a)

data Maybe a = Nothing |
               Just a
```

- Maybe type denotes that result of computation can be `a` or Nothing. Maybe is a **monad**.

Pattern Matching

- **Pattern Matching**

```haskell
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
  Line p1 p2 -> p1
  Triangle p3 p4 p5 -> p3
  Quad p6 p7 p8 p9 -> p6
```

- Two key points here
  - Test: does the given value match this pattern?
  - Binding: if value matches, **deconstruct** and bind corresponding values of `s` and pattern

Pattern Matching in HW5

- `isFree` takes a **variable name** and an **expression** `E`, and returns true if variable **name** is **free** in `E`.

```haskell
isFree :: Name -> Expr -> Bool
isFree v e =
  case e of
    Var n -> if (n == v) then True else False
    Lambda ... | Haskell keyword: `type` (aka `typedef` in C)
```

Monads

- A way to cleanly compose computations
  - E.g., `f` may return a value of type `a` or Nothing
  - Composing computations becomes tedious:
    ```haskell
    > e1 = Var "x" // Lambda term x
    > e2 = Lambda "x" e1 // Lambda term λx.x
    ```

- `Monad` and `Maybe` as monads
  - In Haskell, monads model IO and other imperative features
An Example: Cloned Sheep

```haskell
type Sheep = ...
father :: Sheep \rightarrow Either Sheep (Maybe Sheep)
mother :: Sheep \rightarrow Either Sheep (Maybe Sheep)

(Note: a sheep has both parents; a cloned sheep has one)
maternalGrandfather :: Sheep \rightarrow Either Sheep (Maybe Sheep)

maternalGrandfather s = case mother s of
  Nothing \rightarrow Nothing
  Just m \rightarrow case (father m) of
    Nothing \rightarrow Nothing
    Just gf \rightarrow father gf
```

The Monad Class

- Haskell’s Monad `type class` requires 2 operations, `>>=` (bind) and `return`:

```haskell
class Monad m where
    -- >>= (the bind operation) takes a monad m a, and a function that takes a and turns it into a monad m b, and returns m b
    (>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
    -- return encapsulates a value into the monad
    return :: a \rightarrow m a
```

The Maybe Monad

```haskell
instance Monad Maybe where
    Nothing >> f = Nothing
    (Just x) >> f = f x
    return = Just
```

- Back to our example:

```haskell
mothersPaternalGrandfather s = case mother s of
    Nothing \rightarrow Nothing
    Just m \rightarrow case (father m) of
      Nothing \rightarrow Nothing
      Just gf \rightarrow father gf
```

The List Monad

- The List type constructor is a monad:

```haskell
li >> f = concat (map f li)
return x = [x]
```

- Note: concat::[[a]] \rightarrow [a]
e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]
- Use any f s.t. f::a\rightarrow[b]. f may return a list of 0,1,2,… elements of type b, e.g.,
  > f x = [x+1]
  > [1,2,3] >>= f // returns [2,3,4]
```

The List Monad

```haskell
parents :: Sheep \rightarrow [Sheep]
parents s = MaybeToList (father s) ++ MaybeToList (mother s)

grandParents :: Sheep \rightarrow [Sheep]
grandParents s = (parents s) >>= parents
```
Monad Quote

“A monad is just a monoid in the category of endofunctors, what’s the problem?”

- Monad type class and the monad laws
- Maybe monad
- List monad
- IO monad
- State monad

Outline

- Applied lambda calculus
- Introduction to types and type systems
- The simply typed lambda calculus
- Syntax
- Dynamic semantics
- Static semantics
- Type safety

Reading

- “Types and Programming Languages” by Benjamin Pierce, Chapters 8 and 9
- Lecture notes based on Pierce and notes by Dan Grossman, UW

Applied Lambda Calculus (from Sethi)

- E ::= c | x | (λx.E₁) | (E₁ E₂)
  
  Augments the pure lambda calculus with constants. An applied lambda calculus defines its set of constants and reduction rules. For example:

  Constants: if, true, false
  (all these are λ terms, e.g., true=λx.λy. x)
  0, iszero, pred, succ

  Reduction rules:
  if true M N →₈ M
  if false M N →₈ N
  iszero 0 →₈ true
  iszero (succ 0) →₈ false, k>0
  iszero (pred 0) →₈ false, k>0
  succ (pred M) →₈ M
  pred (succ M) →₈ M

From an Applied Lambda Calculus to a Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied λ-Calculus</th>
<th>A Language (ML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Constant</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>Application</td>
<td>M N</td>
<td>M N</td>
</tr>
<tr>
<td>Abstraction</td>
<td>λx.M</td>
<td>fun x =&gt; M</td>
</tr>
<tr>
<td>Integer</td>
<td>succ⁻¹ 0, k&gt;0</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>pred⁻¹ 0, k&gt;0</td>
<td>-k</td>
</tr>
<tr>
<td>Conditional</td>
<td>if P M N</td>
<td>if P then M else N</td>
</tr>
<tr>
<td>Let</td>
<td>let val x = N in M end</td>
<td></td>
</tr>
</tbody>
</table>

Aside: The Fixed-Point Operator

- One more constant, and one more rule:
  - fix M →₈ M (fix M)
  - fix M

- Needed to define recursive functions:
  - plus x y = if x = 0 then y else plus (pred x) (succ y)

- Therefore:
  - plus = λx.λy. if (iszero x) y (plus (pred x) (succ y))
Aside: The Fixed-Point Operator

But how do we define \textit{plus}?

Define \textit{plus} = \textbf{fix} $M$, where

$M = \lambda f. \lambda x. \lambda y. \text{if } (\text{iszero } x) y (f \ (\text{pred} \ x) \ (\text{succ} \ y))$

We must show that

$\text{fix } M \equiv \beta \lambda x. \lambda y. \text{if } (\text{iszero } x) y ((\text{fix} \ M) \ (\text{pred} \ x) \ (\text{succ} \ y))$

Aside: The Y Combinator

\textbf{fix} is, of course, a lambda expression!

One possibility, the famous Y-combinator:

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

Show that $Y \ M$ indeed beta-reduces to $M \ (Y \ M)$

Types!

- Constants add power
- But they raise problems because they permit “bad” terms such as
  - $\text{if } (\lambda x. x) \ y \ z$ (arbitrary function values are not permitted as predicates, only true/false values)
  - $0 \ x$ (0 does not apply as a function)
  - $\text{succ } \text{true}$ (undefined in our language)
  - $\text{plus } \text{true } 0$ etc.

Types!

- Why types?
  - Safety. Catch semantic errors early
  - Data abstraction. Simple types and ADTs
  - Documentation (statically-typed languages only)
    - Type signature is a form of specification!
  - Statically typed vs. dynamically typed languages
  - Type annotations vs. type inference
  - Type safe vs. type unsafe

Types!

- Important subarea of programming languages, program analysis
- Related to abstract interpretation, although…
  - AI is framework of choice for reasoning about imperative languages
  - Type systems is framework of choice for reasoning about functional languages
  - Type systems and extensions to reason about imperative programs
Type System

- Syntax
- Dynamic semantics (aka concrete semantics). In type theory, it is
  - A sequence of reductions
- Static semantics (aka abstract semantics). In type theory, it is defined in terms of
  - Type environment
  - Typing rules, also called type judgments
  - This is typically referred to as the type system

Example, The Static Semantics. More On This Later!

\[
\begin{align*}
\Gamma &\vdash x : \tau & \text{(Variable)} \\
\Gamma &\vdash E_1 : \sigma \rightarrow \tau, \Gamma &\vdash E_2 : \sigma & \text{(Application)} \\
\Gamma &\vdash \lambda x : \sigma. E_1 : \sigma \rightarrow \tau & \text{(Abstraction)} \\
\end{align*}
\]

Stuck States

- Informally, a term is “stuck” if it cannot be further reduced and it is not a value
  - E.g., 0 x
- “Stuck states” characterize runtime errors
- In real programming languages “stuck states” correspond to forbidden errors such as seg faults, execution of operation on illegal arguments, etc.
- We will define “stuck states” formally for the simply typed lambda calculus, in just awhile

Stuck States Examples

- E.g., c (\lambda x. x), where c is an int constant, is a “stuck state”, i.e., a meaningless state
- E.g., if c E_1 E_2 where c is an int constant, is a “stuck state”
  - Clearly not a value and clearly no rule applies!
  - Because the evaluation rules for if-then-else are:
    - if true E_1 \rightarrow E_1
    - if false E_2 \rightarrow E_2

Type Soundness

- Remember, a type system accepts or rejects terms
- A sound type system never accepts a term that can get stuck
- A complete type system never rejects a term that cannot get stuck
- Typically, whether a term gets stuck is undecidable
- Type systems choose type soundness
Safety = Progress + Preservation

- **Progress**: A well-typed term is not stuck (i.e., either it is a value, or there is an evaluation step that applies)
- **Preservation**: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed

**Soundness follows:**
- Each state reached by program is well-typed (by Preservation)
- A well-typed state is not stuck (by Progress)
- Thus, each state reached by the program is not stuck