LAMBDA CALCULUS

from Untyped to Simply Typed
The Untyped Lambda Calculus ($\lambda$)

Designed by Alonzo Church (1930s)

- Turing Complete (Turing was his doctoral student!)
- Models functions, always as 1-input
- Definition: terms, values, and evaluation
  - $t := x \mid \lambda x \cdot t \mid (t \ t)$
  - $v := \lambda x \cdot t$

Notes

- terms $t$ are variables, lambdas, or applications
- only lambdas are values.
- this language is untyped!
\( \lambda : \text{Evaluation Semantics} \)

- **E-App1**
  \[
  (t_1 t_2) \rightarrow (t'_1 t_2)
  \]

- **E-App2**
  \[
  (v t) \rightarrow (v t')
  \]

- **E-App-Abs**
  \[
  ((\lambda x . t) v) \rightarrow t[x \mapsto v]
  \]
Sample Expressions

- Consider each expression. Are they already values? If not, show each reduction, and name the rule used.

- \((\lambda e . e + 1)\)
- \(((\lambda x . x + 1) 3)\)
- \(((\lambda z . z * z) 5)\)
- \(((\lambda a . a*a) ((\lambda x . x + 1) 6))\)
- \(((\lambda x . (\lambda y . x – y + 1)) 10) 6)\)
- \(((\lambda x . (\lambda y . x – y + 1)) 10)\)
Encoding Values

• Without any extensions, we can still represent some simple values.
  • we used numbers and mathematical operators on the previous slide, but those aren't actually in our core language!

• Each value shall be represented as a higher-order function

• Note, we will much prefer extending the core language!
Encoding Boolean values/operations

- Encoding 'true' and 'false' as functions:
  - \( \text{true} = \lambda x . \lambda y . x \)
  - \( \text{false} = \lambda x . \lambda y . y \)

- Operator encodings
  - \( \text{not} = \lambda a . (a \ \text{false}) \text{true} \)
  - \( \text{and} = \lambda a . (\lambda b . ((a b) a)) \)
  - \( \text{or} = \lambda a . (\lambda b . ((a a) b)) \)
  - \( \text{if} = \lambda b . (\lambda t . (\lambda e . ((b t) e))) \)

\textit{note: all (})'s are optional on this slide
Extending $\lambda$

Why encode things when we can just extend our language?

• we will add more terms and values, to have more primitives in our language.

### Adding Booleans

<table>
<thead>
<tr>
<th>t  :=</th>
<th>...</th>
<th>true</th>
<th>false</th>
<th>if t t t</th>
</tr>
</thead>
<tbody>
<tr>
<td>v  :=</td>
<td>...</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
Extending evaluation: Booleans

- **E-if**
  
  \[
  (\text{if } t_1 t_2 t_3) \rightarrow (\text{if } t_1' t_2 t_3)
  \]

- **E-if-true**
  
  \[
  (\text{if true } t_2 t_3) \rightarrow t_2
  \]

- **E-if-false**
  
  \[
  (\text{if false } t_2 t_3) \rightarrow t_3
  \]
Implementing the untyped lambda calculus

Investigate implementing the untyped lambda calculus in Haskell:

- data for our terms
- is_value function to check if a term is a value
- eval function to perform evaluation
  - needs substitution capability (subst function)

Until we extend our language, it'll seem ungainly – the only values we have are functions! no booleans, numbers, nothing.

→ see ULC_basic.hs, including booleans and numbers.
Formal Language extension recipe

- Create more terms \( t::= \)
  - both **constructors** and **observers**
- as some new terms (if any) to values \( v::= \)
- add more evaluation rules
  - observers will inspect/consume constructors
Coded Language extension recipe

• add to datatype Tm (extend terms)
• add cases to is_val (extend the values)
• add cases to eval, subst, etc.

→ language is extended! What other features could we add?
Aside: Evaluation Strategy

As written, our evaluation rules require that functions' arguments are evaluated first:

- E-App-Abs

\[
((\lambda x . t) v) \rightarrow t[x \mapsto v]
\]

We could have implemented lazy evaluation (and removed E-App2):

- E-App-Abs-lazy

\[
((\lambda x . t_1) t_2) \rightarrow t_1 [x \mapsto t_2]
\]
Evaluation Strategy

How/where does the chosen evaluation strategy affect:

• your implementation?
• Your language usage?
Extension: Natural Numbers

We create our own numbers as zero, successor of a number, and predecessor of a number.

Adding Naturals

| t  := ... | zero | succ t | pred t |
| v  := ... | zero | succ v |
Extending evaluation: Naturals

- \( E\text{-succ} \)
  \[ t_1 \rightarrow t_1' \]
  \( \text{succ } t_1 \rightarrow \text{succ } t_1' \)

- \( E\text{-pred} \)
  \[ t_1 \rightarrow t_1' \]
  \( \text{pred } t_1 \rightarrow \text{pred } t_1' \)

- \( E\text{-succ-pred} \)
  \[ \text{succ } (\text{pred } t) \rightarrow t \]
Extension: Pairs

We add primitive support for paired values.

Adding Pairs

| t := ... | pair t t | fst t | snd t |
| v := ... | pair v v |

- E-pair1
  \[ t_1 \rightarrow t_1' \]
  \[ \text{pair } t_1 t_2 \rightarrow \text{pair } t_1' t_2 \]

- E-pair2
  \[ t_2 \rightarrow t_2' \]
  \[ \text{pair } t_1 t_2 \rightarrow \text{pair } t_1 t_2' \]
Extending evaluation: Pairs

- **E-fst**
  \[ t \rightarrow t' \]
  \[ \text{fst } t \rightarrow \text{fst } t' \]

- **E-snd**
  \[ t \rightarrow t' \]
  \[ \text{snd } t \rightarrow \text{snd } t' \]

- **E-pair-fst**
  \[ \text{fst } (\text{pair } t_1 t_2) \rightarrow t_1 \]

- **E-pair-snd**
  \[ \text{snd } (\text{pair } t_1 t_2) \rightarrow t_2 \]
Building an Interpreter

Start with simple core features
- define terms, values, evaluation
- expand with more features

Implementation choice:
- **Domain Specific Language (DSL)**
  - a language designed/dedicated to one task or domain of knowledge
  - write all tools, e.g. parser/compiler
- **Embedded DSL (EDSL)**
  - DSL that is implemented as a library directly in some other language.
  - All of the host's features are directly available: we're actually writing code in the host language that heavily uses the library definitions
  → we're exploring an EDSL.
Task: Write a mathematical expression EDSL

Calculator EDSL

• write Haskell code that provides a way to represent mathematical calculations, including three basic operations (add, sub, mul)
• choose a value representation and provide evaluation.
Choosing a value space

We can choose any type that's already available in the host language, like Int.

• note, every single expression must result in one of these values!

→ see ExprLang1.hs

We can make our own data type for the value space

→ see ExprLang2.hs
Choosing evaluation semantics

Once we include some notion of functions in our code, we can then choose calling conventions.

• how can we introduce functions?
• where do declarations go?
• what kinds of declarations are allowed? (recursive?)

• we can implement any evaluation strategy, such as pass-by-value, pass-by-name, simply by changing our eval definition.
Fun diversion

The $\omega$-combinator always diverges.

$$\omega = (\lambda x . x x) (\lambda x . x x)$$

Try performing the application. What do you get?
Representing Recursion

In the untyped lambda calculus, we can represent recursion directly or with a language extension. (see `ycomb.txt`)

The y combinator

\[ ycomb = \lambda f . ( (\lambda x . (f (\lambda y . x x y))) (\lambda x . (f (\lambda y . x x y)))) \]

\[ evenF = \lambda \text{"self"} \; \lambda \text{"n"} \]
\[ \quad \lambda \text{If (Equal vn (Num 0)) Tru} \]
\[ \quad \lambda \text{If (Equal vn (Num 1)) Fls} \]
\[ \quad \lambda \text{App (Var "self")} \]
\[ \quad \lambda (\text{Sub vn (Num 2)}) \]

\[ \text{iseven} = \text{App} \; ycomb \; \text{evenF} \]

personally, I prefer extending the language. The y-combinator is a real headache to watch in action!
Providing primitive recursion

We can provide primitive definitions for recursion.

Adding Fix
\[
\text{E-fix} \\
\text{fix} (\lambda x . t) \rightarrow t [x \mapsto (\text{fix} (\lambda x . t))]
\]

\[
t := \ldots | \text{fix } t
\]
Other features

How might we introduce each of the following?

• case statements
• let expressions
• records
• abstract data types
• variable assignments
• classes and objects
• types
• type inference

What else would you want to add to your language?
Valuable resource

To get a much more thorough treatment of writing interpreters for more advanced language features, look for this book:

Types and Programming Languages, by Benjamin Pierce.

→ you can view it electronically through our library's website!
The Simply Typed Lambda Calculus

Type checking is usually a single static phase that happens before any evaluation

- one step in ensuring the expression is well-formed and worth evaluating
- relying on parsing into our host language also performs some well-formedness checks

Option: implement `typecheck :: Expr -> Ty`, where the target is another datatype that represents your language's types
The simply-typed lambda calculus ($\lambda \rightarrow$)

- We can introduce types to the lambda calculus.
- We can choose whether these types are explicit or implicit
  - For now, we will consider explicit typing
- We must ascribe types on our lambdas
- Other values also have known types:
  - true :: Bool
  - 5 :: Int
  - (Pair true 5) :: TyPair Bool Int

→ see STLC.hs
Notes on $\lambda$→

The simply typed lambda calculus is strongly normalizing

• no unbounded calculation – always halts!

• we lose the ability to do any recursion by giving terms types (why? try writing a type for the y- and $\omega$-combinators.)

• we can regain recursion by extending the language as before.
Maintaining an environment

Typechecking a variable (Var) requires us to remember what type it was ascribed when introduced.

As many lambdas as we've descended through, we must remember all of those variables and their types.

- We can keep a list of [(String, Ty)] pairs to look up variables' types.
- We call this listing the "environment", \( \text{Gamma} (\Gamma) \).
A Simple Starting Point

• We consider the following definition for our simply typed lambda calculus, including the definition of allowed types

• then we explore typing rules for the language. They are quite similar in form to the evaluation rules, which will also need to be updated.

```
t ::= λx:T.t | (t t) | x
    | true | false | if t t t
    | num # | add t t | sub t t | mul t t

T ::= T→T | TyNum | TyBool

v ::= λx:T.t | true | false | num #
```
Basic typing rules

- **Ty-var**  
  \[(v, ty) \in \Gamma \]  
  \[\Gamma \vdash v : ty\]

- **Ty-λ**  
  \[(\Gamma, (v, ty)) \vdash t_b : ty_b\]  
  \[\Gamma \vdash (\lambda v : ty \cdot t_b) : ty \rightarrow ty_b\]

- **Ty-app**  
  \[\Gamma \vdash t_1 : ty_d \rightarrow ty_r\]  
  \[\Gamma \vdash t_2 : ty_d\]  
  \[\Gamma \vdash (t_1 \ t_2) : ty_r\]

*If \(\Gamma\) has \((v, ty)\) in it, then \(\Gamma\) can look up variable \(v\) and find its type, \(ty\).*

*If we extend \(\Gamma\) with \((v, ty)\) and that finds \(tb\)'s type to be \(ty_b\), then the original \(\Gamma\) derives the type of \((\lambda v : ty \cdot ty_b)\) to be \(ty \rightarrow ty_b\).*

*If \(\Gamma\) finds \(t_1 : ty_d \rightarrow ty_r\), and \(\Gamma\) finds \(t_2 : ty_d\), then \(\Gamma\) derives the overall type of \((t_1 \ t_2)\) to be \(ty_r\).*
Typing rules: Booleans

• Ty-true

  \[\Gamma \vdash true : TyBool\]

• Ty-false

  \[\Gamma \vdash false : TyBool\]

• Ty-if

  \[\Gamma \vdash a : TyBool \quad \Gamma \vdash b : ty \quad \Gamma \vdash c : ty\]

  \[\Gamma \vdash if \ a \ b \ c : ty\]

without an environment, we know the type of true is TyBool.

same for false.

If \(\Gamma\) derives \(a\)'s type as TyBool, and \(\Gamma\) derives that both \(b\) and \(c\) have type \(ty\), then \(\Gamma\) derives the overall type of \((if \ a \ b \ c)\) as \(ty\).
Typing rules: Numbers

- **Ty-num**
  
  \[
  \Gamma \vdash (\text{Num } n) : \text{TyNum}
  \]

- **Ty-add**
  
  \[
  \Gamma \vdash a : \text{TyNum} \quad \Gamma \vdash b : \text{TyNum} \\
  \Gamma \vdash (\text{Add } a \ b) : \text{TyNum}
  \]

- **Ty-sub, Ty-mul**: similar to Ty-add.

Without using a \( \Gamma \), we know any number is of type TyNum.

If \( \Gamma \) derives that \( a \) is of type TyNum, and \( \Gamma \) derives that \( b \) is of type TyNum, then \( \Gamma \) derives (Add \( a \ b \)) to be of type TyNum.
Typed primitive recursion

We again add Fix to the language, only now it involves typed lambdas.

Adding Fix

\[ t := \ldots \mid \text{fix } t \]

E-fix

\[
\text{fix } (\lambda x : ty . t_2) \rightarrow [x \mapsto (\text{fix } (\lambda x : ty . t_2))] t_2
\]