LAMBDA CALCULUS

(untyped)
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
  - $t ::= x | \lambda x . t | (t t)$
  - $v ::= \lambda x . t$

Notes

- terms $t$ are variables, lambdas, or applications
- only lambdas are values.
- this language is untyped!
### λ: Evaluation Semantics

Evaluation: applying these rules to simplify your term until you have a value (no more evaluation possible).

- **E-App1**: $t_1 \rightarrow t_1'$
  
  $(t_1 \ t_2) \rightarrow (t_1' \ t_2)$

- **E-App2**: $t \rightarrow t'$
  
  $(v \ t) \rightarrow (v \ t')$

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

---

**E-App1**: "If I know that a term $t_1$ can be evaluated to $t_1'$, then when I've got an application $(t_1 \ t_2)$, I can evaluate just $t_1$ to $t_1'$, and then I've got $(t_1' \ t_2)$ left over."

---

**E-App2**: "When an application has a value first and a term second, and that term can evaluate further, we are allowed to evaluate that second term."

---

**E-App-Abs**: "When a lambda term is applied to a value $v$, it can simplify down to the body of the lambda, $t$, with all occurrences of the lambda parameter $x$ being replaced with the argument $v$. We don't need any simplifications available (nothing above the line)."
Sample Expressions

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

  - $\left(\left(\lambda x \cdot x + 1\right) 3\right)$
  - $\left(\lambda e \cdot e + 1\right)$
  - $\left(\left(\lambda z \cdot z \cdot z\right) 5\right)$

  - $\left(\left(\left(\lambda a \cdot (\lambda b \cdot a)\right) 10\right) 20\right)$
  - $\left(\left(\lambda a \cdot a \cdot a\right) \left(\left(\lambda x \cdot x + 1\right) 6\right)\right)$
  - $\left(\left(\left(\lambda x \cdot (\lambda y \cdot x - y + 1)\right) 10\right) 6\right)$
  - $\left(\left(\lambda x \cdot (\lambda y \cdot x - y + 1)\right) 10\right)$
Think in Trees

• Drawing out an abstract syntax tree for terms can help understand them:

\(((\lambda a \cdot (\lambda b \cdot a)) \ 10) \ 20\)
\[ \lambda : \text{Evaluation Rules as Trees} \]

E-App1

\[ t_1 \rightarrow t_1' \]

E-App2

\[ t_2 \rightarrow t_2' \]
$\lambda : \text{Evaluation Rules as Trees}$

E-App-Abs

Diagram:

- $\lambda x$ (lambda abstraction)
- $\nu$ (variable)
- $t$ (term)
- $t[x \mapsto \nu]$ (substitution)

Arrow from $\lambda x$ and $\nu$ to $t[x \mapsto \nu]$.
Reading Expressions

• the body of a lambda expression grabs as much as it can (it reads through until it hits a close-parenthesis it didn't open, or the end of the expression). We can remove some parentheses for slightly easier reading.

• These are equivalent:
  
  • \((\lambda x. (\lambda y. x+y))\)
  • \(\lambda x. \lambda y. x+y\)

• These are not:
  
  • \(((\lambda x. x+1) 3)\)
  • \((\lambda x. x+1 3)\)
Extending λ

• we will add more terms and values, to have more primitives in our language.
• The core lambda calculus is actually pretty painful/nearly useless on its own (we already sneaked numbers/ops in!)

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

Add these rules:

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

- **E-if-true**
  \[
  \frac{t_1 \rightarrow \ t_2}{(if \ true \ t_2 \ t_3) \rightarrow t_2}
  \]

- **E-if-false**
  \[
  \frac{t_1 \rightarrow \ t_3}{(if \ false \ t_2 \ t_3) \rightarrow t_3}
  \]
Sample Expressions - Booleans

Consider each expression. Are they already values? If not, show each reduction, and name the rule used. Show what happens in each step.

- (if true 5 10)
- (if false true 20)
- (if true 5 (3/0))

- (λ x . if x 3 6)
- ((λ x . if x 4 7) false)
- ((λ z . if true z 9) 5)

- (((λ a . (λ b . if b 100 200)) (2+3)) (if (a>4) true false))
Using Definitions

We can name expressions and use them. Using these definitions:

- definition: \( \text{not} = (\lambda \ x \ . \ \text{if} \ x \ \text{false} \ \text{true}) \)
- definition: \( \text{and} = (\lambda \ a \ . \ (\lambda \ b \ . \ \text{if} \ a \ b \ \text{false})) \)

Simplify these expressions.

- \((\text{not} \ \text{true})\)
- \(((\lambda \ x \ . \ x \ \text{and} \ \text{true}) \ \text{true})\)
- \(((\lambda \ x \ . \ x \ \text{and} \ \text{true}) \ \text{false})\)

Define these:

- or
- xor
Extension: Natural Numbers

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

<table>
<thead>
<tr>
<th>Adding Naturals</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::= ...</td>
</tr>
<tr>
<td>v ::= ...</td>
</tr>
</tbody>
</table>
Extending evaluation: Naturals

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

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

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

We add primitive support for paired values.

<table>
<thead>
<tr>
<th>Adding Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::= ...</td>
</tr>
<tr>
<td>v ::= ...</td>
</tr>
</tbody>
</table>

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

- **E-pair2**
  \[
  t_2 \rightarrow t_2' \quad \text{pair } v t_2 \rightarrow \text{pair } v t_2'
  \]
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
  - \( \text{snd } (\text{pair } t_1 \ t_2) \rightarrow t_2 \)
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

Start again with only the core untyped lambda calculus – no true/false values, no if-term, no E-Bool rules.

• Encoding 'true' and 'false' as functions:
  • true = λ x . λ y . x
  • false = λ x . λ y . y

• Operator encodings
  • not = λ a . (a false) true
  • and = λ a . (λ b . ((a b) a))
  • or = λ a . (λ b . ((a a) b))
  • if = λ b . (λ t . (λ e . ((b t) e)))

*note: all ()'s are optional on this slide*
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_bools_nums.hs, which includes booleans and numbers.
Formal Language extension recipe

• Create more terms \( t ::= \ldots \)
  • both **constructors** and **observers**

• add some new terms (if any) to be values \( v ::= \ldots \)

• add more evaluation rules
  • enough that all "proper" terms can become values
  • observers will inspect/consume constructors
Coded Language extension recipe

(implementing in Haskell)

• 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 can we add?
Aside: Evaluation Strategy

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

- **E-App-Abs**

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

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

  - **E-App-Abs-lazy**

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

How/where does the chosen evaluation strategy affect:

• your implementation?
• Your language usage?
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.
Choosing a value space

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

• every single expression must result in a value of this type!

→ 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)

<table>
<thead>
<tr>
<th>The y combinator</th>
</tr>
</thead>
</table>
| ycomb = \f . ( (\x . (f (\y . x x y)))
  (\x . (f (\y . x x y))))) |
| evenF = Lam "self" $ Lam "n"
  $ If (Equal vn (Num 0)) Tru
  $ If (Equal vn (Num 1)) Fls
  $ App (Var "self")
  (Sub vn (Num 2)) |
| iseven = App ycomb 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

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

E-fix

\[
\text{fix } (\lambda x . t) \rightarrow t [x \mapsto \text{fix } (\lambda x . t)]
\]
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 for free!