Untyped Lambda Calculus

Principles of Programming Languages

CSE 526

Compiled at 08:26 on 2020/03/10

Lambda Calculus

- A formal notation to study computability and programming.
- Can be considered as the smallest universal programming language.
  - *Universal:* Can be used to express any computation that can be performed on a Turing Machine
  - *Small:* Has only two constructs: abstraction and application.
- Brief History:
  - Introduced by Church and Kleene in 1930s.
  - Used by Church to study problems in computability.
  - Concepts have heavily influenced *functional programming*.
  - Used to study *types* and type systems in programming languages

Lambda Terms

Syntax of the λ-calculus

t	::=		Terms
		x	Variable
	$\mid$	$\lambda x.\ t$	Abstraction
	$\mid$	$t\ t$	Application

Textual Representation:

Use parentheses to represent trees as linear text

Informal Semantics

λ-expressions can be considered as expressions in a functional language

- **Abstraction:** $(\lambda x.\ t)$ is a "function" with formal parameter x that returns (the value of) term t.
  - Example 1: $\lambda x.\ x$ is the identity function: one that returns the argument value itself.
  - Example 2: $\lambda x.\lambda y.\ x$ is a function that takes "two arguments x and y and returns the first argument".
    The explanation in blue above is not accurate, but is good enough for government work. We'll see the subtlety shortly.
- **Application:** $(t_1\ t_2)$ is a "function call" where t_1 is a function and t_2 is the supplied argument.
  - Example: $((\lambda x.\ x)\ y)$ supplies y as the argument to the identity function.

Syntactic Conventions and Syntactic Sugar

- Parentheses can be dropped using the following conventions:
  - application is left associative
    e.g. $((f\ f)\ x)$ is same as $f\ f\ x$
  - a λ binds as much as possible to its right.
    e.g $\lambda f.\ \lambda x.\ f\ (f\ x)$ is same as $(\lambda f.(\lambda x.\ f\ (f\ x)))$
- Multiple consecutive abstractions can be combined:
  e.g. $\lambda f.\lambda x.f\ (f\ x)$ is same as $\lambda f\ x.\ f\ (f\ x)$

The Meaning of Lambda Expressions

- Recall: $\lambda x.\ t$ stands for a function with x as the parameter and (the value of) t as the return value.
- $(t_1\ t_2)$ stands for "calling" the function t_1 with t_2 as the parameter.
- Example: Consider the expression

$$((\lambda wyx.\ y\ (w\ y\ x))\quad (\lambda sz.\ z))$$

  This is an instance of an application. The expression in blue is passed as an argument to the function in red.
- The meaning of an application: replace every occurrence of the formal parameter in the body of the function with the given argument.
  In the above example
  1. $\lambda\ yx.\ y\ ((\lambda\ sz.\ z)\ y\ x)$
  2. $\lambda\ yx.\ y\ ((\lambda\ z.\ z)\ x)$
  3. $\lambda\ yx.\ y\ x$

Encoding Booleans in the λ-Calculus

B	λ-calculus
`true`	$\lambda x.\ \lambda y.\ x$
`false`	$\lambda x.\ \lambda y.\ y$
`&&`	$\lambda x.\ \lambda y.\ ((x\ y)\ \texttt{false})$
`||`	$\lambda x.\ \lambda y.\ ((x\ \texttt{true})\ y)$
`!`	$\lambda x.\ ((x\ \texttt{false})\ \texttt{true})$
`if`	$\lambda c.\ \lambda t.\ \lambda e.\ ((c\ t)\ e)$

This is known as the *Church encoding of Booleans*, or simply *Church Booleans*.

Example:

$$(\texttt{true}\ \texttt{\&\&}\ \texttt{false})$$

$$\equiv\ (\lambda x.\ \lambda y.\ ((x\ y)\ \texttt{false}))\ (\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y)$$

$$\to\ (\lambda y.\ (((\lambda x.\ \lambda y.\ x)\ y)\ \texttt{false}))\ (\lambda x.\ \lambda y.\ y)$$

$$\to\ (\ ((\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y))\ \texttt{false})$$

$$\to\ (\ (\lambda y.\ (\lambda x.\ \lambda y.\ y))\ \texttt{false})$$

$$\to\ (\lambda x.\ \lambda y.\ y)$$

$$\equiv\ \texttt{false}$$

Encoding Natural Numbers in the λ-Calculus

N	λ-calculus
0	$\lambda s.\ \lambda z.\ z$
1	$\lambda s.\ \lambda z.\ (s\ z)$
2	$\lambda s.\ \lambda z.\ (s\ (s\ z))$
3	$\lambda s.\ \lambda z.\ (s\ (s\ (s\ z)))$
$\vdots$	
inc	$\lambda n.\ \lambda s.\ \lambda z.\ (s\ ((n\ s)\ z))$
plus	$\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ ((m\ s)\ ((n\ s)\ z))$
times	$\lambda m.\ \lambda n.\ ((m\ (\mathit{plus}\ n))\ 0)$
iszero	$\lambda m.\ ((m\ (\lambda x.\ \texttt{false}))\ \texttt{true})$
$\vdots$	

This is known as the *Church encoding of Naturals*, or simply *Church Numerals*.

Encoding Data Structures in the λ-Calculus

pair	$\lambda f.\ \lambda s.\ \lambda c.\ ((c\ f)\ s)$
fst	$\lambda p.\ (p\ \texttt{true})$
snd	$\lambda p.\ (p\ \texttt{false})$

Example: Let φ_1 and φ_2 be two arbitrary expressions.

$$\begin{array}{ll} & pair\ \varphi_1\ \varphi_2 \\ \equiv & (\ (\lambda f.\ \lambda s.\ \lambda c.\ ((c\ f)\ s)\ \varphi_1)\ \varphi_2) \\ \to^* & \lambda c.\ ((c\ \ \varphi_1)\ \varphi_2) \end{array}$$

$$\begin{array}{ll} & fst\ (pair\ \varphi_1\ \varphi_2) \\ \equiv & (\lambda p.\ (p\ \texttt{true}))\ (pair\ \varphi_1\ \varphi_2) \\ \to & (pair\ \varphi_1\ \varphi_2)\ \texttt{true} \\ \to^* & (\lambda c.\ ((c\ \ \varphi_1)\ \varphi_2))\ \ \texttt{true} \\ \to & ((\texttt{true}\ \varphi_1)\ \varphi_2) \\ \to & \varphi_1 \end{array}$$

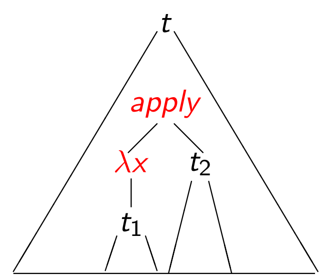
$$\begin{array}{ll} & snd\ (pair\ \varphi_1\ \varphi_2) \\ \equiv & (\lambda p.\ (p\ \texttt{false}))\ (pair\ \varphi_1\ \varphi_2) \\ \to^* & ((\texttt{false}\ \varphi_1)\ \varphi_2) \\ \to & \varphi_2 \end{array}$$

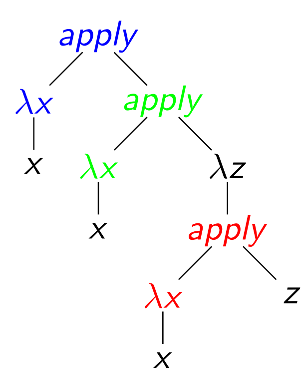
Evaluating Lambda Expressions: An Informal Intro.

Basic reduction: $(\lambda x.\ t_1)\ t_2\ \ \to\ \ [x \mapsto t_2]t_1$, where

$[x \mapsto t_2]t_1$ be the term obtained by replacing all "free" occurrences of x in t_1 by t_2.

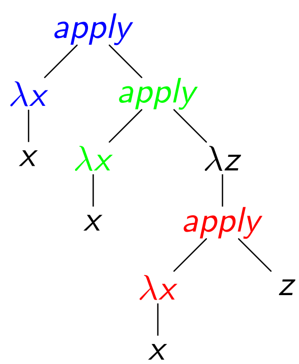
- A sub-term of t of the form $\boldsymbol{(\lambda x.\ t_1)\ t_2}$ is called a **redex** of t.
- One step in evaluating a λ-term t is replacing some redex in t according to the above reduction schema.
- In general, there may be many redexes in a term.
  Example: Let $id = (\lambda x.\ x)$ in term $id\ (id\ (\lambda z.\ id\ z))$





Reduction Strategies

A reduction strategy is used to **choose** a redex where the basic reduction step will be done.



- **Full β-reduction:** Pick a redex *non-deterministically*
- **Normal Order:** choose the left-most, outer-most redex.
- **Call-By-Name:** like normal-order, but ignore redexes inside abstractions.
- **Call-By-Value:** choose the right-most, inner-most redex that is not inside an abstraction.

Evaluating Lambda Expressions

- The key step in evaluating an application then is: *replace every occurrence of a formal parameter with the actual argument*.

  Example: $((\lambda x.(\lambda z.\ x\ z))\quad y) \quad \rightarrow \quad (\lambda z.\ y\ z)$
- We can formalize the meaning of application by introducing a function, called *substitution* that maps terms to terms:

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

- The central problem now is how we define this substitution function.

Substitutions (1^{st} attempt)

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \lambda y.\ [x \mapsto s]t & \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

- Appears to be correct.
  Example: $[x \mapsto y](\lambda z.\ x\ z) = (\lambda z.\ y\ z)$
  Use: $(\lambda x.\ (\lambda z.\ x\ z))\quad y) \quad \rightarrow \quad (\lambda z.\ y\ z)$
- But is incorrect!
  Example: $[x \mapsto y](\lambda x.\ x) = (\lambda x.\ y)$
  Use: $((\lambda x.(\lambda x.\ x))\quad y) \quad \rightarrow \quad (\lambda x.\ y)$

Substitutions (2^{nd} attempt)

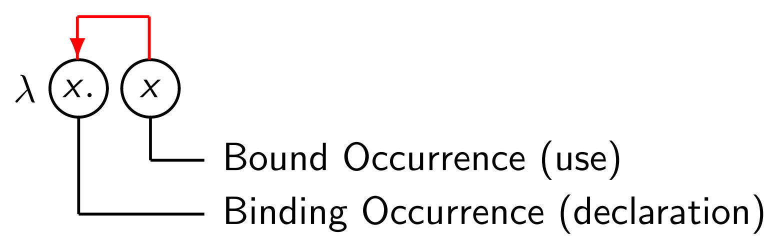
$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \begin{cases} \lambda y.\ t & \text{if } x = y \\ \lambda y.\ [x \mapsto s]t & \text{if } x \neq y \end{cases} & \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

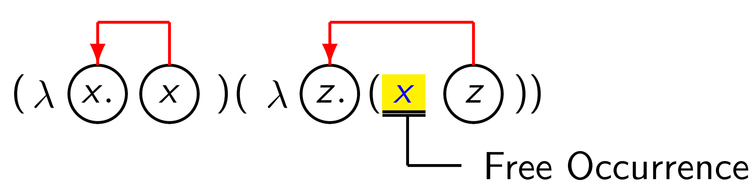
- $[x \mapsto y](\lambda x.\ x) = (\lambda x.\ x)$
- But is still incorrect! e.g. $[x \mapsto y](\lambda y.\ x\ y) = (\lambda y.\ y\ y)$
- In the result of the above example, one y is local to the function while the other y is not local.
- But going by our definition, there is no way to distinguish between the two y's!
- **Solution:** We should get $(\lambda w.\ y\ w)$ instead (by suitably renaming "local" variables).

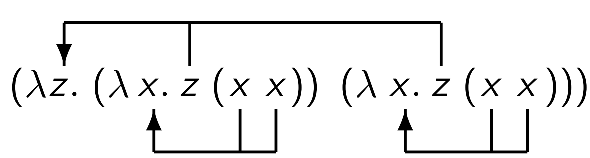
Bound and Free Variables: An Informal Intro.

- Variable $\boldsymbol{x}$ in λ-expression $\lambda x.\ t$ is said to be **bound**.
  - Example 1: x in $\lambda x.\ x$ is a bound variable.
  - Example 2: in $\lambda x.(x\ y)$, x is bound but y is not bound.
  - Rough meaning: parameters are local to a function definition.
- A variable that is not bound is said to be **free**.
  - Example 2: in $\lambda x.(x\ y)$, y is free.
  - Rough meaning: free variables in a function definition are analogous to non-local variables.

Bound and Binding Occurrences

- 

- 

- 

Bound Variables

Formal definition: $bv(t)$, the set of all bound variables of t, is such that:

- t is an abstraction of the form $\lambda x.t'$:
  - $bv(t) = bv(t') \cup \{x\}$
- t is an application of the form $t_1\ t_2$:
  - $bv(t) = bv(t_1) \cup bv(t_2)$
- Example:
  $bv(\ (\lambda x.\ x)\ (\lambda z.\ (x\ z))\ )$
  $= bv(\lambda x.\ x) \cup bv(\lambda z.\ (x\ z))$
  $= \{x\} \cup \{z\} = \{x, z\}$

Free Variables

Formal definition: $fv(t)$, the set of all free variables of t, is such that:

- t is a variable of the form x:
  - $fv(t) = \{x\}$
- t is an abstraction of the form $\lambda x.t'$:
  - $fv(t) = fv(t') - \{x\}$
- t is an application of the form $t_1\ t_2$:
  - $fv(t) = fv(t_1) \cup fv(t_2)$
- Example:
  $fv(\ (\lambda x.\ x)\ (\lambda z.\ (x\ z))\ )$
  $= fv(\lambda x.\ x) \cup fv(\lambda z.\ (x\ z))$
  $= \{\ \} \cup \{x\} = \{x\}$

α-Conversion (Renaming)

- *Intuition:* We can rename a bound variable as long as
  - the new name is not also the name of a free variable, and
  - we replace every occurrence of the bound variable
- Example 1: $(\lambda y.\ x\ y)$ is equivalent to $(\lambda z.\ x\ z)$
- Example 2: $(\lambda y.\ x\ y)$ is *not* equivalent to $(\lambda x.\ x\ x)$ (the name of new variable is same as that of a free variable)
- Example 3: $(\lambda y.\ x\ y)$ is *not* equivalent to $(\lambda y.\ x\ z)$ (not every occurrence of y has been replaced).
- Two terms t and t' are said to be "α-equivalent" (denoted by $t \equiv_\alpha t'$) if they are identical modulo the names of bound variables.

Substitutions (3^{rd} attempt)

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \lambda y.\ [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin fv(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

- The definition is now incomplete! e.g. $[x \mapsto y](\lambda y.\ x\ y)\ = ??$
- This drawback is not serious:
- We can apply a substitution on an α-equivalent term instead.
- E.g. $[x \mapsto y](\lambda z.\ x\ z)\ = (\lambda z.\ y\ z)$

Operational Semantics: Full β-Reduction

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \quad \text{E-APP1}$$

$$\frac{t_2 \to t_2'}{t_1\ t_2 \to t_1\ t_2'} \quad \text{E-APP2}$$

$$\frac{t \to t'}{\lambda x.\ t \to \lambda x.\ t'} \quad \text{E-ABS}$$

$$(\lambda x.\ t_1)\ t_2 \to [x \mapsto t_2]t_1 \qquad \text{E-APPABS}$$

Operational Semantics: Call-By-Value

$$\begin{array}{lll} t & ::= & \dots \\ v & ::= & \lambda x.\ t \end{array}$$

Terms (all λ-terms)

Values

Evaluation:

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \quad \text{E-APP1}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \quad \text{E-APP2}$$

$$(\lambda x.\ t_1)\ v_2 \to [x \mapsto v_2]t_1 \qquad \text{E-APPABS}$$

- In an application of the form $(t_1\ t_2)$, if t_1 is a λ-abstraction, then t_2 has to be reduced to a value before the application is done.
- This corresponds to Call-By-Value parameter passing: evaluate the actual arguments first before passing them as parameters to a called function.

Operational Semantics: Call-By-Name

t	$::=$	$\ldots$	Terms (all λ-terms)
v	$::=$	$\lambda x.\ t$	Values

Evaluation:

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \quad \text{E-APP}$$

$$(\lambda x.\ t_1)\ t_2 \rightarrow [x \mapsto t_2]t_1 \quad \text{E-APPABS}$$

- In an application of the form $(t_1\ t_2)$, if t_1 is a λ-abstraction, then t_1 has to be reduced to a value before the application is done.
- In terms of familiar languages, the actual arguments are passed *unevaluated* to the called function. They will be evaluated in the called function if needed.

Infinite and Diverging Computations in the λ-Calculus

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

Evaluation:

$$\begin{array}{ll} & omega \\ \equiv & (\lambda x.\ x\ x)\ \ (\lambda x.\ x\ x) \\ \rightarrow & (\lambda x.\ x\ x)\ \ (\lambda x.\ x\ x) \\ \equiv & omega \\ \rightarrow & omega \\ \vdots & \end{array}$$

inf : $(\lambda x.\ (x\ x)\ x)$

Evaluation:

$$\begin{array}{ll} & (inf\ \ inf) \\ \equiv & (\lambda x.\ (x\ x)\ x)\ inf \\ \rightarrow & (inf\ \ inf)\ inf \\ \rightarrow & (\ (inf\ \ inf)\ inf)\ inf \\ \rightarrow & \ldots \\ \vdots & \end{array}$$

Recursive Functions in the λ-Calculus —(1)

- Consider the function to compute *factorial* of a natural number, written as follows:

$$fact \equiv \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (fact\ (dec\ n))\ )\ )$$

  where *dec* is the function that decrements a number by 1.
- Note this is not a proper encoding: *fact* is being defined in terms of itself!
- The solution is to "lift" factorial into a *functional*:

$$F \equiv \lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ )$$

- Note that F is well-defined.
- F is a very special function, as we'll see in the next...

Recursive Functions in the λ-Calculus —(2)

$$F \equiv \lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ )$$

- Consider $fact_0 \equiv F\ omega$:

$$\begin{aligned} fact_0 &\equiv F\ omega \\ &\equiv (\lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ ))\ omega \\ &\rightarrow \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (omega\ (dec\ n))\ )\ ) \end{aligned}$$

- *When non-strict evaluation is used,* $fact_0$ computes the same as *fact* for 0, but diverges elsewhere.

Recursive Functions in the λ-Calculus —(3)

$$F \equiv \lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ )$$

- Now consider $fact_1 \equiv F\ fact_0$:
  $fact_1 \equiv F\ fact_0$
  $\equiv (\lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ ))\ fact_0$
  $\rightarrow \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (fact_0\ (dec\ n))\ )\ )$
- *When non-strict evaluation is used,* $fact_1$ computes the same as *fact* for 0 and 1, but diverges elsewhere.

Recursive Functions in the λ-Calculus —(4)

- Consider the sequence of functions $fact_0, fact_1, fact_2, \ldots$ such that $fact_0 = omega$, and $fact_{n+1} = (F\ fact_n)$.
- None of these functions is same as *fact*, but as we construct more and more members of this sequence, we get functions that approximate *fact* closer and closer.
- *fact* is indeed the limit of this sequence of functions!
- If only we had a way, in the λ-calculus, to generate such a sequence...

The Y-Combinator

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

- Consider $(Y\ F)$:
  $$\begin{array}{lll} (Y\ F) & \equiv & (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ F \\ \to & & (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \\ \to & & F\ (\ (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))\ ) \\ \cong & & F\ (Y\ F) \end{array}$$
- Recall $F \equiv \lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ )$.
- Putting it all together:
  $$\begin{array}{lll} (Y\ F) & \cong & F\ (Y\ F) \\ \equiv & & (\lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ ))\ (Y\ F) \\ \to & & \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ ((Y\ F)\ (dec\ n))\ )\ ) \end{array}$$
- $(Y\ F)$ looks like the mythical function *fact*.

The Z-Combinator

- $(Y\ F) \cong F\ (Y\ F)$
- With *call-by-name* evaluation strategy, the next steps in reduction will first substitute the formal parameter of F with $(Y\ F)$.
- With *call-by-value* strategy, $F\ (Y\ F)$ will first reduce $(Y\ F)$, which result in:
  $$\begin{array}{ll} \to^* & F\ (F\ (Y\ F)) \\ \to^* & F\ (F\ (F\ (Y\ F))) \\ \to^* & \ldots \end{array}$$
- For *call-by-value* strategy, we should use the Z combinator instead:

$$Z = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

Recursive Functions in the λ-Calculus —(5)

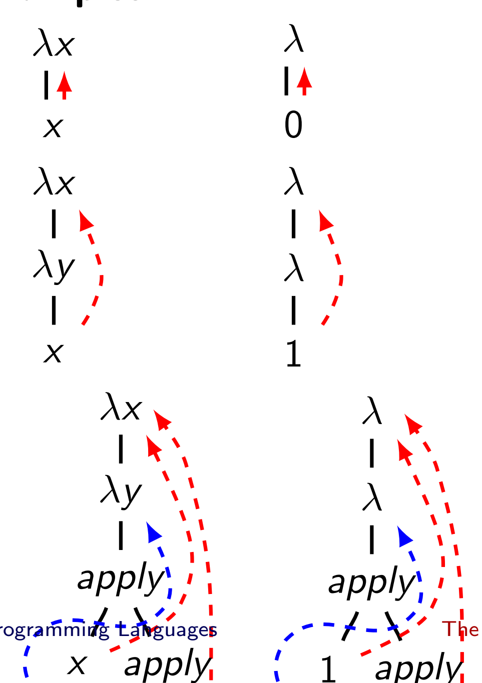
$$\begin{aligned} Y &= \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) \\ F &= \lambda f.\ \lambda n.\ (if\ (iszero\ n)\ 1\ (times\ n\ (f\ (dec\ n))\ )\ ) \\ fact &= (Y\ F) \end{aligned}$$

- Note that the definitions of Y, F and *fact* are all non-recursive.
- The above recipe can be used for writing any recursive function.
- Say, we have a mythical recursive definition $f = \lambda x.\ e$ where e uses f.
- We simply rewrite the definition as $f = (Y\ (\lambda f.\ \lambda x.\ e))$.

Nameless Representation of Terms

- Consider variables in a λ-term as named “holes” to be filled in.
- Instead of using symbolic names for variables, one can name the holes w.r.t. the λ that binds them.

Examples:



- $\lambda x.\ x$ can be written as $\lambda.\ 0$
- $\lambda x.\ \lambda y.\ x$ can be written as $\lambda.\ \lambda.\ 1$
- $\lambda x.\ \lambda y.\ x\ (y\ x)$ can be written as $\lambda.\ \lambda.\ 1\ (0\ 1)$

n-Terms

De Bruijn terms are defined by a family of sets (each set being a set of terms) $\{\mathcal{T}_0, \mathcal{T}_1, \ldots\}$ such that $\mathcal{T}_n$ represents λ-terms with n or fewer free variables

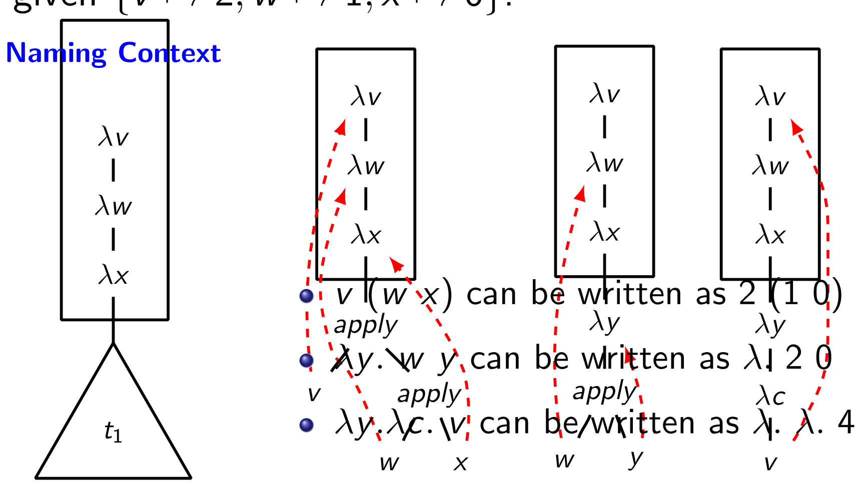
Formally, $\mathcal{T}$ is the smallest family of sets $\{\mathcal{T}_0, \mathcal{T}_1, \ldots\}$ such that

- $k \in \mathcal{T}_n$ whenever $0 \leq k < n$
- if $t_1 \in \mathcal{T}_n$ then $\lambda.\ t_1 \ \in \mathcal{T}_{n-1}$
- if $t_1, t_2 \in \mathcal{T}_n$ then $(t_1\ \ t_2) \in \mathcal{T}_n$

α-equivalent closed λ-terms will have the same de Bruijn representation.

Naming Context

- When a λ-term has free variables, we need information on their relative positions.
- E.g. given $\{v \mapsto 2, w \mapsto 1, x \mapsto 0\}$:



- $v\ (w\ \ x)$ can be written as $2\ (1\ 0)$
- $\lambda y.\ w\ \ y$ can be written as $\lambda.\ 2\ 0$
- $\lambda y. \lambda c. v$ can be written as $\lambda.\ \lambda.\ 4$

- Naming contexts are often written as a sequence, where $x_n, x_{n-1}, \ldots, x_1, x_0$, represents a context where each x_i has de Bruijn index i.

Substitution

- Term $(\lambda y.\ \lambda z.\ (x\ y)\ (w\ z))$ under naming context v, w, x has the following de Bruijn representation:

$$\lambda.\ \lambda.\ (2\ 1)\ (3\ 0)$$

- Term $(v\ w)$ under naming context v, w, x has the following de Bruijn representation:

$$(2\ 1)$$

- Substitution $[x \mapsto (v\ w)](\lambda y.\ \lambda z.\ (x\ y)\ (w\ z))$ will yield the term

$$\lambda y.\ \lambda z.\ ((v\ w)\ y)\ (w\ z)$$

- Assuming the naming context is v, w, x, the above term has the following de Bruijn representation: $(\lambda.\ \lambda.\ ((4\ 3)\ 1)\ (3\ 0))$
- Hence, when carrying out substitution, we need to renumber the indices of free variables in the replacement term, and retain the indices of bound variables.
  This will be done using the shifting operation, defined next.

Shifting

For substitution, we need to

- renumber the indices of free variables (say, by d), and
- retain the indices of bound variables (say, those numbered below c).

This is done using the *shifting* operation, defined as follows:

$$\begin{aligned}
\uparrow_c^d (k) &= \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases} \\
\uparrow_c^d (\lambda.\ t_1) &= \lambda.\ \uparrow_{c+1}^d (t_1) \\
\uparrow_c^d (t_1\ t_2) &= (\uparrow_c^d t_1\ \uparrow_c^d t_2) \\
\uparrow^d (t) &= \uparrow_0^d (t)
\end{aligned}$$

Examples

- $\uparrow^2 (\lambda.\ \lambda.\ 1\ (0\ 2)) = \lambda.\ \lambda.\ 1\ (0\ 4)$
- $\uparrow^2 (\lambda.\ 0\ 1\ (\lambda.\ 0\ 1\ 2)) = \lambda. 0\ 3\ (\lambda.\ 0\ 1\ 4)$

Substitution using Shifting

$$\begin{aligned}
[j \mapsto s]k &= \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases} \\
[j \mapsto s](\lambda.\ t_1) &= \lambda.\ [j+1 \mapsto \uparrow^1 (s)]t_1 \\
[j \mapsto s](t_1\ t_2) &= ([j \mapsto s]t_1\ [j \mapsto s]t_2)
\end{aligned}$$

Examples:

- $[0 \mapsto 1](0\ (\lambda.\ \lambda.\ 2)) = 1\ (\lambda.\ \lambda.\ 3)$
- $[0 \mapsto (1\ (\lambda.\ 2))](0\ (\lambda.\ 1)) = (1\ (\lambda.\ 2))\ (\lambda(2\ (\lambda.\ 3)))$
- $[0 \mapsto 1](\lambda.\ (0\ 2)) = \lambda.\ (0\ 2)$

Evaluation

In the calculus with symbolic term representation:

$$(\lambda x.\ t_1)\ t_2 \rightarrow [x \mapsto t_2]t_1 \qquad \text{E-AppAbs}$$

In the calculus with de Bruijn representation:

$$(\lambda.\ t_1)\ t_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^1 (t_2)]t_1) \qquad \text{E-AppAbs}$$

- The outer λ is removed after application, so the indices have to shift *down* by 1.
- Indices in argument (t_2) should *not* be changed in the end, so we shifting them *up* by 1 first.
  - Consider $(\lambda x.\ w\ x\ v)\ (\lambda y.\ (w\ y))$, whose de Bruijn representation is $(\lambda.\ 1\ 0\ 2)\ (\lambda.\ 1\ 0)$ (assuming naming context v, w).
  - The result of the application is $w\ (\lambda y.\ w\ y)\ v$.
  - $\uparrow^1 (\lambda.\ 1\ 0) = \lambda.\ 2\ 0$
  - $[0 \mapsto (\lambda.\ 2\ 0)](1\ 0\ 2) = 1\ (\lambda.\ 2\ 0)\ 2$
  - $\uparrow^{-1} (1\ (\lambda.\ 2\ 0)\ 2) = 0\ (\lambda.\ 1\ 0)\ 1$