CS 4110

Programming Languages & Logics

Lecture 15

De Bruijn, Combinators, Encodings

Review: λ-calculus

Syntax

$$\begin{array}{rcl} e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \\ v & ::= & \lambda x.\, e \end{array}$$

Semantics

$$\frac{e_1 \rightarrow e_1'}{e_1\, e_2 \rightarrow e_1'\, e_2} \qquad \frac{e \rightarrow e'}{v\, e \rightarrow v\, e'}$$

$$\frac{}{(\lambda x.\, e)\, v \rightarrow e\{v/x\}}\ \beta$$

Rewind: Currying

This is just a function that returns a function:

$$\text{ADD} \triangleq \lambda x.\, \lambda y.\, x + y$$

$$\text{ADD } 38 \rightarrow \lambda y.\, 38 + y$$

$$\text{ADD } 38\ 4 = (\text{ADD } 38)\ 4 \rightarrow 42$$

Informally, you can think of it as a *curried* function that takes two arguments, one after the other.

But that's just a way to get intuition. The λ-calculus only has one-argument functions.

de Bruijn Notation

Another way to avoid the tricky issues with substitution is to use a *nameless* representation of terms.

$$e ::= n \mid \lambda.e \mid e\ e$$

de Bruijn Notation

Another way to avoid the tricky issues with substitution is to use a *nameless* representation of terms.

$$e ::= n \mid \lambda.e \mid e\ e$$

Abstractions have lost their variables!

Variables are replaced with numerical indices!

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\, x$	$\lambda.\, 0$
$\lambda z.\, z$	

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\ x$	$\lambda.\ 0$
$\lambda z.\ z$	$\lambda.\ 0$
$\lambda x.\ \lambda y.\ x$	

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\ x$	$\lambda.\ 0$
$\lambda z.\ z$	$\lambda.\ 0$
$\lambda x.\ \lambda y.\ x$	$\lambda.\ \lambda.\ 1$
$\lambda x.\ \lambda y.\ \lambda s.\ \lambda z.\ x\, s\, (y\, s\, z)$	

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\ x$	$\lambda.\ 0$
$\lambda z.\ z$	$\lambda.\ 0$
$\lambda x.\ \lambda y.\ x$	$\lambda.\ \lambda.\ 1$
$\lambda x.\ \lambda y.\ \lambda s.\ \lambda z.\ x\, s\, (y\, s\, z)$	$\lambda.\ \lambda.\ \lambda.\ \lambda.\ 3\ 1\ (2\ 1\ 0)$
$(\lambda x.\ x\, x)\ (\lambda x.\ x\, x)$	

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\ x$	$\lambda.\ 0$
$\lambda z.\ z$	$\lambda.\ 0$
$\lambda x.\ \lambda y.\ x$	$\lambda.\ \lambda.\ 1$
$\lambda x.\ \lambda y.\ \lambda s.\ \lambda z.\ x\, s\, (y\, s\, z)$	$\lambda.\ \lambda.\ \lambda.\ \lambda.\ 3\ 1\ (2\ 1\ 0)$
$(\lambda x.\ x\, x)\ (\lambda x.\ x\, x)$	$(\lambda.\ 0\ 0)\ (\lambda.\ 0\ 0)$
$(\lambda x.\ \lambda x.\ x)\ (\lambda y.\ y)$	

Examples

Here are some terms written in standard and de Bruijn notation:

Standard	**de Bruijn**
$\lambda x.\ x$	$\lambda.\ 0$
$\lambda z.\ z$	$\lambda.\ 0$
$\lambda x.\ \lambda y.\ x$	$\lambda.\ \lambda.\ 1$
$\lambda x.\ \lambda y.\ \lambda s.\ \lambda z.\ x\, s\, (y\, s\, z)$	$\lambda.\ \lambda.\ \lambda.\ \lambda.\ 3\ 1\ (2\ 1\ 0)$
$(\lambda x.\ x\, x)\ (\lambda x.\ x\, x)$	$(\lambda.\ 0\ 0)\ (\lambda.\ 0\ 0)$
$(\lambda x.\ \lambda x.\ x)\ (\lambda y.\ y)$	$(\lambda.\ \lambda.\ 0)\ (\lambda.\ 0)$

Free variables

To represent a λ-expression that contains free variables in de Bruijn notation, we need a way to map the free variables to integers.

We will work with respect to a map Γ from variables to integers called a *context*.

Examples:

Suppose that Γ maps x to 0 and y to 1.

- Representation of $x\ y$ is 0 1
- Representation of $\lambda z.\ x\ y\ z$ $\lambda.$ 1 2 0

Shifting

To define substitution, we will need an operation that shifts by *i* the variables above a cutoff *c*:

$$\begin{array}{rcl} \uparrow_c^i (n) & = & \begin{cases} n & \text{if } n < c \\ n + i & \text{otherwise} \end{cases} \\ \uparrow_c^i (\lambda.e) & = & \lambda.(\uparrow_{c+1}^i e) \\ \uparrow_c^i (e_1\, e_2) & = & (\uparrow_c^i\ e_1)\,(\uparrow_c^i\ e_2) \end{array}$$

The cutoff *c* keeps track of the variables that were bound in the original expression and so should not be shifted.

The cutoff is 0 initially.

Substitution

Now we can define substitution:

$$\begin{array}{rcl} n\{e/m\} & = & \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\ (\lambda.e_1)\{e/m\} & = & \lambda.e_1\{(\uparrow^1_0 e)/m+1\} \\ (e_1\, e_2)\{e/m\} & = & (e_1\{e/m\})\,(e_2\{e/m\}) \end{array}$$

Substitution

Now we can define substitution:

$$\begin{array}{rcl} n\{e/m\} & = & \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\ (\lambda.e_1)\{e/m\} & = & \lambda.e_1\{(\uparrow^1_0 e)/m+1\} \\ (e_1\, e_2)\{e/m\} & = & (e_1\{e/m\})\,(e_2\{e/m\}) \end{array}$$

The β rule for terms in de Bruijn notation is just:

$$\frac{}{(\lambda.e_1)\, e_2 \;\rightarrow\; \uparrow^{-1}_0 (e_1\{\uparrow^1_0 e_2/0\})}\;\beta$$

Example

Consider the term $(\lambda u.\lambda v.u\ x)\ y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

Example

Consider the term $(\lambda u.\lambda v.u\, x)\, y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

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

Example

Consider the term $(\lambda u.\lambda v.u\,x)\,y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\;2)\;1 \\ \to & \uparrow_0^{-1}\;((\lambda.1\;2)\{(\uparrow_0^1\;1)/0\}) \end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\,y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\,2)\,1 \\ \to & \uparrow_0^{-1} ((\lambda.1\,2)\{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda.1\,2)\{2/0\}) \end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\,y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\,2)\,1 \\ \to & \uparrow_0^{-1} ((\lambda.1\,2)\{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda.1\,2)\{2/0\}) \\ = & \uparrow_0^{-1} \lambda.((1\,2)\{(\uparrow_0^1 2)/(0+1)\}) \end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\ x)\ y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\ 2)\ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda.1\ 2)\{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda.1\ 2)\{2/0\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{(\uparrow_0^1 2)/(0+1)\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{3/1\}) \end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\,y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{cl}
 & (\lambda.\lambda.1\,2)\,1 \\
\to & \uparrow_0^{-1} ((\lambda.1\,2)\{(\uparrow_0^1 1)/0\}) \\
= & \uparrow_0^{-1} ((\lambda.1\,2)\{2/0\}) \\
= & \uparrow_0^{-1} \lambda.((1\,2)\{(\uparrow_0^1 2)/(0+1)\}) \\
= & \uparrow_0^{-1} \lambda.((1\,2)\{3/1\}) \\
= & \uparrow_0^{-1} \lambda.(1\{3/1\})\,(2\{3/1\})
\end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\;y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl}
 & (\lambda.\lambda.1\,2)\;1 \\
\to & \uparrow_0^{-1}\;((\lambda.1\,2)\{(\uparrow_0^1\;1)/0\}) \\
= & \uparrow_0^{-1}\;((\lambda.1\,2)\{2/0\}) \\
= & \uparrow_0^{-1}\;\lambda.((1\,2)\{(\uparrow_0^1\;2)/(0+1)\}) \\
= & \uparrow_0^{-1}\;\lambda.((1\,2)\{3/1\}) \\
= & \uparrow_0^{-1}\;\lambda.(1\{3/1\})\;(2\{3/1\}) \\
= & \uparrow_0^{-1}\;\lambda.3\,2
\end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\;y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl}
 & (\lambda.\lambda.1\;2)\;1 \\
\to & \uparrow_0^{-1}\;((\lambda.1\;2)\{(\uparrow_0^1\;1)/0\}) \\
= & \uparrow_0^{-1}\;((\lambda.1\;2)\{2/0\}) \\
= & \uparrow_0^{-1}\;\lambda.((1\;2)\{(\uparrow_0^1\;2)/(0+1)\}) \\
= & \uparrow_0^{-1}\;\lambda.((1\;2)\{3/1\}) \\
= & \uparrow_0^{-1}\;\lambda.(1\{3/1\})\;(2\{3/1\}) \\
= & \uparrow_0^{-1}\;\lambda.3\;2 \\
= & \lambda.2\;1
\end{array}$$

Example

Consider the term $(\lambda u.\lambda v.u\,x)\,y$ with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl}
 & (\lambda.\lambda.1\,2)\,1 \\
\to & \uparrow_0^{-1} ((\lambda.1\,2)\{(\uparrow_0^1 1)/0\}) \\
= & \uparrow_0^{-1} ((\lambda.1\,2)\{2/0\}) \\
= & \uparrow_0^{-1} \lambda.((1\,2)\{(\uparrow_0^1 2)/(0+1)\}) \\
= & \uparrow_0^{-1} \lambda.((1\,2)\{3/1\}) \\
= & \uparrow_0^{-1} \lambda.(1\{3/1\})\,(2\{3/1\}) \\
= & \uparrow_0^{-1} \lambda.3\,2 \\
= & \lambda.2\,1
\end{array}$$

which, in standard notation (with respect to Γ), is the same as $\lambda v.y\,x$.

Combinators

Another way to avoid the issues having to do with free and bound variable names in the λ-calculus is to work with closed expressions or *combinators*.

With just three combinators, we can encode the entire λ-calculus.

Combinators

Another way to avoid the issues having to do with free and bound variable names in the λ-calculus is to work with closed expressions or *combinators*.

With just three combinators, we can encode the entire λ-calculus.

$$\begin{aligned} \mathsf{K} &= \lambda x.\lambda y.\, x \\ \mathsf{S} &= \lambda x.\lambda y.\lambda z.\, x\, z\, (y\, z) \\ \mathsf{I} &= \lambda x.\, x \end{aligned}$$

Combinators

We can even define independent evaluation rules that don't depend on the λ-calculus at all.

Behold the "SKI-calculus":

$$\begin{aligned}
&\mathsf{K}\ e_1\ e_2 \to e_1\\
&\mathsf{S}\ e_1\ e_2\ e_3 \to e_1\ e_3\ (e_2\ e_3)\\
&\mathsf{I}\ e \to e
\end{aligned}$$

You would never want to program in this language—it doesn't even have variables!—but it's exactly as powerful as the λ-calculus.

Bracket Abstraction

The function $[x]$ that takes a combinator term M and builds another term that behaves like $\lambda x.M$:

$$\begin{array}{rcll} [x]\,x & = & \mathsf{I} & \\ [x]\,N & = & \mathsf{K}\,N & \text{where } x \notin fv(N) \\ [x]\,N_1\,N_2 & = & \mathsf{S}\,([x]\,N_1)\,([x]\,N_2) & \end{array}$$

The idea is that $([x]\,M)\,N \to M\{N/x\}$ for every term N.

Bracket Abstraction

We then define a function $(e)*$ that maps a λ-calculus expression to a combinator term:

$$\begin{array}{rcl} (x)* & = & x \\ (e_1\ e_2)* & = & (e_1)*\ \ (e_2)* \\ (\lambda x.e)* & = & [x]\ (e)* \end{array}$$

Example

As an example, the expression $\lambda x.\lambda y.\, x$ is translated as follows:

$$\begin{aligned} & (\lambda x.\lambda y.\, x)* \\ = \; & [x]\,(\lambda y.\, x)* \\ = \; & [x]\,([y]\,x) \\ = \; & [x]\,(\mathsf{K}\,x) \\ = \; & (\mathsf{S}\,([x]\,\mathsf{K})\,([x]\,x)) \\ = \; & \mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I} \end{aligned}$$

No variables in the final combinator term!

Example

We can check that this behaves the same as our original λ-expression by seeing how it evaluates when applied to arbitrary expressions e_1 and e_2.

$$\begin{array}{rl} & (\lambda x.\lambda y.\, x)\; e_1\; e_2 \\ \rightarrow & (\lambda y.\, e_1)\; e_2 \\ \rightarrow & e_1 \end{array}$$

Example

We can check that this behaves the same as our original λ-expression by seeing how it evaluates when applied to arbitrary expressions e_1 and e_2.

$$\begin{array}{ll} & (\lambda x.\lambda y.\, x)\; e_1\; e_2 \\ \to & (\lambda y.\, e_1)\; e_2 \\ \to & e_1 \end{array}$$

and

$$\begin{array}{ll} & (\mathsf{S}\;(\mathsf{K}\;\mathsf{K})\;\mathsf{I})\; e_1\; e_2 \\ \to & (\mathsf{K}\;\mathsf{K}\; e_1)\;(\mathsf{I}\; e_1)\; e_2 \\ \to & \mathsf{K}\; e_1\; e_2 \\ \to & e_1 \end{array}$$

SKI Without I

Looking back at our definitions...

$$\begin{aligned}&\mathsf{K}\ e_1\ e_2 \rightarrow e_1\\&\mathsf{S}\ e_1\ e_2\ e_3 \rightarrow e_1\ e_3\ (e_2\ e_3)\\&\mathsf{I}\ e \rightarrow e\end{aligned}$$

...*I* isn't strictly necessary. It behaves the same as S K K.

SKI Without I

Looking back at our definitions...

$$\begin{aligned}&\mathsf{K}\ e_1\ e_2 \to e_1\\&\mathsf{S}\ e_1\ e_2\ e_3 \to e_1\ e_3\ (e_2\ e_3)\\&\mathsf{I}\ e \to e\end{aligned}$$

...*I* isn't strictly necessary. It behaves the same as S K K.

Our example becomes:

$$\mathsf{S}\ (\mathsf{K}\ \mathsf{K})\ (\mathsf{S}\ \mathsf{K}\ \mathsf{K})$$

Encodings

The pure λ-calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure λ-calculus. We can however encode objects, such as booleans, and integers.

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned}
\text{AND TRUE FALSE} &= \text{FALSE} \\
\text{NOT FALSE} &= \text{TRUE} \\
\text{IF TRUE } e_1\, e_2 &= e_1 \\
\text{IF FALSE } e_1\, e_2 &= e_2
\end{aligned}$$

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned}
\text{AND TRUE FALSE} &= \text{FALSE} \\
\text{NOT FALSE} &= \text{TRUE} \\
\text{IF TRUE } e_1\ e_2 &= e_1 \\
\text{IF FALSE } e_1\ e_2 &= e_2
\end{aligned}$$

Let’s start by defining TRUE and FALSE:

$$\begin{aligned}
\text{TRUE} &\triangleq \\
\text{FALSE} &\triangleq
\end{aligned}$$

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

Let’s start by defining TRUE and FALSE:

$$\begin{aligned} \text{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \text{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

Booleans

We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

Booleans

We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

We can also write the standard Boolean operators.

$$\begin{aligned} \text{NOT} &\triangleq \\ \text{AND} &\triangleq \\ \text{OR} &\triangleq \end{aligned}$$

Booleans

We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq \lambda b.\, b\ \text{FALSE TRUE}$$

$$\text{AND} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\ b_2\ \text{FALSE}$$

$$\text{OR} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\ \text{TRUE}\ b_2$$

Church Numerals

Let's encode the natural numbers!

We'll write $\overline{n}$ for the encoding of the number n. The central function we'll need is a *successor* operation:

$$\text{SUCC}\ \overline{n} = \overline{n+1}$$

Church Numerals

Church numerals encode a number *n* as a function that takes *f* and *x*, and applies *f* to *x* *n* times.

$$\begin{aligned}
\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\
\overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\
\overline{2} &\triangleq \lambda f.\, \lambda x.\, f\, (f\, x)
\end{aligned}$$

Church Numerals

Church numerals encode a number *n* as a function that takes *f* and *x*, and applies *f* to *x* *n* times.

$$\begin{aligned}\overline{0} &\triangleq \lambda f.\, \lambda x.\, x\\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x\\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\,(f\, x)\end{aligned}$$

We can write a successor function that “inserts” another application of *f*:

$$\text{SUCC} \triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x)$$

Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function n_1 times to n_2.

$$\text{PLUS} \triangleq$$

Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function n_1 times to n_2.

$$\mathsf{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \mathsf{SUCC}\ n_2$$