Lecture 8

Pairs

- **Representing pairs:**

$$\texttt{LET}\ (E_1, E_2) = \lambda f.\ f\ E_1\ E_2$$

$$\texttt{LET fst} = \lambda p.\ p\ \texttt{true}$$

$$\texttt{LET snd} = \lambda p.\ p\ \texttt{false}$$

- (E_1, E_2) **represents an ordered pair**
  - **first component (i.e. E_1) is accessed with `fst`**
  - **second component (i.e. E_2) is accessed with `snd`**
- **The definitions work, e.g.:**

$$\begin{aligned}
\texttt{fst}\ (E_1, E_2) &= (\lambda p.\ p\ \texttt{true})\ (E_1, E_2)\\
&= (E_1, E_2)\ \texttt{true}\\
&= (\lambda f.\ f\ E_1\ E_2)\ \texttt{true}\\
&= \texttt{true}\ E_1\ E_2\\
&= (\lambda x\ y.\ x)\ E_1\ E_2\\
&= E_1
\end{aligned}$$

Tuples

- *n-tuples* **easily defined in terms of pairs:**

  $\texttt{LET}\ (E_1, E_2, \ldots, E_n) = (E_1, (E_2, (\cdots(E_{n-1}, E_n)\cdots)))$

- $(E_1, \ldots, E_n)$ **is an** n-*tuple*

  - **with** *components* E_1, $\ldots$, E_n **and** *length* n
  - **pairs are** 2**-tuples**

- **Extracting components of** n**-tuples:**

$$\texttt{LET}\ E \overset{n}{\downarrow} 1 = \texttt{fst}\ E$$

$$\texttt{LET}\ E \overset{n}{\downarrow} 2 = \texttt{fst}(\texttt{snd}\ E)$$

$$\vdots$$

$$\texttt{LET}\ E \overset{n}{\downarrow} i = \texttt{fst}(\underbrace{\texttt{snd}(\texttt{snd}(\cdots(\texttt{snd}}_{i-1\ \text{sndS}}\ E)\cdots))) \qquad \textbf{(if } i < n\textbf{)}$$

$$\vdots$$

$$\texttt{LET}\ E \overset{n}{\downarrow} n = \underbrace{\texttt{snd}(\texttt{snd}(\ldots(\texttt{snd}}_{n-1\ \text{sndS}}\ E)\ldots)))$$

Verifying tuple selection works

$$
\begin{aligned}
(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} 1 &= (E_1, (E_2, (\ldots))) \overset{n}{\downarrow} 1 \\
&= \texttt{fst}\ (E_1, (E_2, (\ldots))) \\
&= E_1
\end{aligned}
$$

$$
\begin{aligned}
(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} 2 &= (E_1, (E_2, (\ldots))) \overset{n}{\downarrow} 2 \\
&= \texttt{fst}\ (\texttt{snd}\ (E_1, (E_2, (\ldots)))) \\
&= \texttt{fst}\ (E_2, (\ldots)) \\
&= E_2
\end{aligned}
$$

- $(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} i = E_i$ **all** i **such that** $1 \leq i \leq n$

- **Usually write** $E \downarrow i$ **instead of** $E \overset{n}{\downarrow} i$
  - **when** n **clear from context**
  - **e.g.** $(E_1, \ldots, E_n) \downarrow i \ = \ E_i$ **(where** $1 \leq i \leq n$**)**

Representing numbers

- **Goal: define λ-expression $\underline{n}$ representing n**

- **Goal: represent arithmetical operations**

  - **e.g. need `suc`, `pre`, `add` and `iszero` representing:**
    the successor function ($n \mapsto n+1$),
    the predecessor function ($n \mapsto n-1$),
    addition,
    test for zero

- **Required properties:**

  $\texttt{suc}\ \underline{n} = \underline{n+1}$ **(for all numbers n)**

  $\texttt{pre}\ \underline{n} = \underline{n-1}$ **(for all numbers $n > 0$)**

  $\texttt{add}\ \underline{m}\ \underline{n} = \underline{m+n}$ **(for all numbers m and n)**

  $\texttt{iszero}\ \underline{0} = \texttt{true}$

  $\texttt{iszero}\ (\texttt{suc}\ \underline{n}) = \texttt{false}$

Preliminary notation

- **Define $f^n\ x$ to mean n applications of f to x**
  - **For example,** $f^5\ x\ =\ f(f(f(f(f\ x))))$
  - **By convention** $f^0\ x$ **defined to mean** x
- **More generally:**

$$\texttt{LET}\ E^0\ E'\ =\ E'$$

$$\texttt{LET}\ E^n\ E'\ =\ \underbrace{E(E(\cdots(E}_{n\ E\mathrm{s}}\ E')\cdots))$$

- **Note that** $E^n(E\ E')\ =\ E^{n+1}\ E'\ = E(E^n\ E')$
  - $f^4x\ =\ f(f(f(f\ x)))\ =\ f(f^3x)\ =\ f^3(f\ x)$

Church's numerals

- **Representation below due to Church**

- **Church's numerals:**

$$\texttt{LET } \underline{0} = \lambda f\ x.\ x$$

$$\texttt{LET } \underline{1} = \lambda f\ x.\ f\ x$$

$$\texttt{LET } \underline{2} = \lambda f\ x.\ f(f\ x)$$

$$\vdots$$

$$\texttt{LET } \underline{n} = \lambda f\ x.\ f^n\ x$$

$$\vdots$$

- **Arithmetical operations**

$$\texttt{LET suc} = \lambda n\ f\ x.\ n\ f(f\ x)$$

$$\texttt{LET add} = \lambda m\ n\ f\ x.\ m\ f\ (n\ f\ x)$$

$$\texttt{LET iszero} = \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true}$$

Properties (exercise)

- $\texttt{suc}\ \underline{0} = \underline{1}$
- $\texttt{suc}\ \underline{5} = \underline{6}$
- $\texttt{iszero}\ \underline{0} = \texttt{true}$
- $\texttt{iszero}\ \underline{5} = \texttt{false}$
- $\texttt{add}\ \underline{0}\ \underline{1} = \underline{1}$
- $\texttt{add}\ \underline{2}\ \underline{3} = \underline{5}$
- $\texttt{suc}\ \underline{n} = \underline{n{+}1}$
- $\texttt{iszero}\ (\texttt{suc}\ \underline{n}) = \texttt{false}$
- $\texttt{add}\ \underline{0}\ \underline{n} = \underline{n}$
- $\texttt{add}\ \underline{m}\ \underline{0} = \underline{m}$
- $\texttt{add}\ \underline{m}\ \underline{n} = \underline{m + n}$

Predecessor function pre

- $\texttt{pre}\ \underline{n}$ **defined using** $\lambda f\ x.\ f^n\ x$ **(i.e.** $\underline{n}$**)**
  - **goal: get a function that applies** f **only** $n{-}1$ **times**
  - **trick: 'throw away' the first application of** f **in** f^n

- **First define a function** $\texttt{prefn}$ **on pairs:**
  - $\texttt{prefn}\ f\ (\texttt{true}, x) = (\texttt{false}, x)$
  - $\texttt{prefn}\ f\ (\texttt{false}, x) = (\texttt{false}, f\ x)$

  From this it follows that:
  - $(\texttt{prefn}\ f)^n\ (\texttt{false}, x) = (\texttt{false}, f^n\ x)$
  - $(\texttt{prefn}\ f)^n\ (\texttt{true}, x) = (\texttt{false}, f^{n-1}\ x)$ **(if** $n > 0$**)**

- n **applications of** $\texttt{prefn}$ **to** $(\texttt{true}, x)$ **result in** $n{-}1$ **applications of** f **to** x

- **Definition of** $\texttt{prefn}$

$$\texttt{LET prefn} = \lambda f\ p.\ (\texttt{false},\ (\texttt{fst}\ p \to \texttt{snd}\ p \mid (f(\texttt{snd}\ p))))$$

Properties of `prefn`

- $\texttt{prefn}\ f\ (b, x) = (\texttt{false}, (b \to x \mid f\ x))$
- $\texttt{prefn}\ f\ (\texttt{true}, x) = (\texttt{false}, x)$
- $\texttt{prefn}\ f\ (\texttt{false}, x) = (\texttt{false}, f\ x)$
- $(\texttt{prefn}\ f)^n\ (\texttt{false}, x) = (\texttt{false}, f^n\ x)$
- $(\texttt{prefn}\ f)^n\ (\texttt{true}, x) = (\texttt{false}, f^{n-1}\ x)$

Definition of $\texttt{pre}$

- $\texttt{LET pre} = \lambda n\ f\ x.\ \texttt{snd}\ (n\ (\texttt{prefn}\ f)\ (\texttt{true}, x))$

- **If** $n > 0$

$$
\begin{array}{rcll}
\texttt{pre}\ \underline{n}\ f\ x & = & \texttt{snd}\ (\underline{n}\ (\texttt{prefn}\ f)\ (\texttt{true}, x)) & \textbf{(defn of } \texttt{pre}\textbf{)} \\
& = & \texttt{snd}\ ((\texttt{prefn}\ f)^n\ (\texttt{true}, x)) & \textbf{(defn of } \underline{n}\textbf{)} \\
& = & \texttt{snd}(\texttt{false}, f^{n-1}\ x) & \textbf{(as above)} \\
& = & f^{n-1}\ x &
\end{array}
$$

  - **hence by extensionality**

$$
\begin{array}{rcll}
\texttt{pre}\ \underline{n} & = & \lambda f\ x.\ f^{n-1}\ x & \\
& = & \underline{n{-}1} & \textbf{(definition of } \underline{n{-}1}\textbf{)}
\end{array}
$$

- **Properties of** $\texttt{pre}$

  - $\texttt{pre}\ (\texttt{suc}\ \underline{n}) = \underline{n}$

  - $\texttt{pre}\ \underline{0} = \underline{0}$

Another numeral systems

- **Numerals with simpler predecessor function**

$$\texttt{LET}\ \underline{\hat{0}} = \lambda x.x$$

$$\texttt{LET}\ \underline{\hat{1}} = (\texttt{false}, \underline{\hat{0}})$$

$$\texttt{LET}\ \underline{\hat{2}} = (\texttt{false}, \underline{\hat{1}})$$

$$\vdots$$

$$\texttt{LET}\ \underline{\widehat{n+1}} = (\texttt{false}, \underline{\widehat{n}})$$

$$\vdots$$

- **Can define** $\widehat{\texttt{suc}}$, $\widehat{\texttt{iszero}}$, $\widehat{\texttt{pre}}$ **such that**
  - $\widehat{\texttt{suc}}\ \underline{\widehat{n}} = \underline{\widehat{n+1}}$
  - $\widehat{\texttt{iszero}}\ \underline{\hat{0}} = \texttt{true}$
  - $\widehat{\texttt{iszero}}\ (\widehat{\texttt{suc}}\ \underline{\widehat{n}}) = \texttt{false}$
  - $\widehat{\texttt{pre}}\ (\widehat{\texttt{suc}}\ \underline{\widehat{n}}) = \underline{\widehat{n}}$

Definition by recursion

- **To represent multiplication would like to define** `mult` **such that:**

$$\texttt{mult}\ m\ n = \underbrace{\texttt{add}\ n\ (\texttt{add}\ n\ (\ \cdots\ (\texttt{add}\ n\ \underline{0})\ \cdots\ ))}_{m\ \texttt{add}s}$$

- **Achieved if** `mult` **satisfies:**

$$\texttt{mult}\ \ m\, n = (\texttt{iszero}\ \ m \to \underline{0} \mid \texttt{add}\ \ n\,(\texttt{mult}\ (\texttt{pre}\ \ m)\ n))$$

- **If this held then, for example,**

$$\begin{aligned}
\texttt{mult}\ \underline{2}\ \underline{3} &= (\texttt{iszero}\ \underline{2} \to \underline{0} \mid \texttt{add}\ \underline{3}\ (\texttt{mult}\ (\texttt{pre}\ \underline{2})\ \underline{3})) \\
&\qquad\qquad\qquad\qquad \text{(by the equation)} \\
&= \texttt{add}\ \underline{3}\ (\texttt{mult}\ \underline{1}\ \underline{3}) \\
&\qquad\qquad \text{(by properties of } \texttt{iszero}\text{, the conditional and } \texttt{pre}) \\
&= \texttt{add}\ \underline{3}\ (\texttt{iszero}\ \underline{1} \to \underline{0} \mid \texttt{add}\ \underline{3}\ (\texttt{mult}\ (\texttt{pre}\ \underline{1})\ \underline{3})) \\
&\qquad\qquad\qquad\qquad \text{(by the equation)} \\
&= \texttt{add}\ \underline{3}\ (\texttt{add}\ \underline{3}\ (\texttt{mult}\ \underline{0}\ \underline{3})) \\
&\qquad\qquad \text{(by properties of } \texttt{iszero}\text{, the conditional and } \texttt{pre}) \\
&= \texttt{add}\ \underline{3}\ (\texttt{add}\ \underline{3}\ (\texttt{iszero}\ \underline{0} \to \underline{0} \mid \texttt{add}\ \underline{3}\ (\texttt{mult}\ (\texttt{pre}\ \underline{0})\ \underline{3}))) \\
&\qquad\qquad\qquad\qquad \text{(by the equation)} \\
&= \texttt{add}\ \underline{3}\ (\texttt{add}\ \underline{3}\ \underline{0}) \\
&\qquad\qquad \text{(by properties of } \texttt{iszero} \text{ and the conditional)}
\end{aligned}$$

Recursion

- **Equation above suggests `mult` be defined by:**

$$\texttt{mult} = \lambda m\, n.\, (\texttt{iszero}\ m \to \underline{0} \mid \texttt{add}\ n\, (\underset{\uparrow \atop \text{N.B.}}{\texttt{mult}}\, (\texttt{pre}\ m)\, n))$$

- **This cannot be used to define `mult`**
  - `mult` **must** *already be defined* **for the λ-expression to the right of the equals to make sense**

- **There is a technique for constructing λ-expressions that satisfy arbitrary equations**
  - **applied to the equation above, this gives the desired definition of `mult`.**

Fixed points

- $\mathtt{Y}$ **is such that, for any expression** E**:**

$$\mathtt{Y}\ E = E\ (\mathtt{Y}\ E)$$

  - $\mathtt{Y}\ E$ **is unchanged when** E **is applied to it**
  - **if** $E\ E' = E'$ **then** E' **is called a** *fixed point* **of** E

- **A** λ**-expression** `Fix` **with the property**

$$\mathtt{Fix}\ E = E(\mathtt{Fix}\ E)$$

  (for any E**) is called a** *fixed-point operator*

  - **infinitely many different fixed-point operators**
  - $\mathtt{Y}$ **is the most famous one**

The fixed-point operator $\mathtt{Y}$

- **Definition of $\mathtt{Y}$:**

$$\mathtt{LET\ Y} = \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

- **$\mathtt{Y}$ is a fixed-point operator:**

$$\begin{aligned}
\mathtt{Y}\ E &= (\lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))\ E && \textbf{(defn of } \mathtt{Y}\textbf{)}\\
&= (\lambda x.\ E(x\ x))\ (\lambda x.\ E(x\ x)) && \textbf{(}\beta\textbf{-conversion)}\\
&= E\ ((\lambda x.\ E(x\ x))\ (\lambda x.\ E(x\ x))) && \textbf{(}\beta\textbf{-conversion)}\\
&= E\ (\mathtt{Y}\ E) && \textbf{(the line before last)}
\end{aligned}$$

- **Hence every E has a fixed point $\mathtt{Y}\ E$**

Defining `mult`

- **Define `multfn` by:**

$$\texttt{LET multfn} = \lambda f\ m\ n.\ (\texttt{iszero}\ \ m \to \underline{0} \mid \texttt{add}\ \ n\ (f\ (\texttt{pre}\ \ m)\ n))$$

- **Define `mult` by:**

$$\texttt{LET mult} = \texttt{Y multfn}$$

- **Then:**

$$\begin{aligned}
\texttt{mult}\ \ m\ n &= (\texttt{Y multfn})\ m\ n && \text{(defn of }\texttt{mult}\text{)}\\
&= \texttt{multfn}\ (\texttt{Y multfn})\ m\ n && \text{(property of }\texttt{Y}\text{)}\\
&= \texttt{multfn}\ \texttt{mult}\ m\ n && \text{(defn of }\texttt{mult}\text{)}\\
&= (\lambda f\ m\ n.\ (\texttt{iszero}\ \ m \to \underline{0} \mid \texttt{add}\ \ n\ (f\ (\texttt{pre}\ \ m)\ n)))\ \texttt{mult}\ \ m\ n &&\\
& && \text{(defn of }\texttt{multfn}\text{)}\\
&= (\texttt{iszero}\ \ m \to \underline{0} \mid \texttt{add}\ \ n\ (\texttt{mult}\ (\texttt{pre}\ \ m)\ n)) && (\beta\text{-conversion})
\end{aligned}$$

Recursion in general

- **An equation of the form**

$$f\ x_1\ \cdots\ x_n = E$$

is called *recursive* **if** f **occurs free in** E

- `Y` **provides a way of solving such equations**

- **start with an equation:**

$$\texttt{f}\ x_1 \ldots x_n = \smallfrown\!\texttt{f}\!\smallfrown$$

where $\smallfrown\!\texttt{f}\!\smallfrown$ **is some** λ**-expression containing** `f`

- **The following defines** `f` **so this holds:**

$$\texttt{LET f} = \texttt{Y}\ (\lambda f\ x_1 \ldots x_n .\smallfrown f \smallfrown)$$

- **Then:**

$$\begin{array}{rll}
\texttt{f}\ x_1 \ldots x_n & = \texttt{Y}\ (\lambda f\ x_1 \ldots x_n.\smallfrown f \smallfrown)\ x_1 \ldots x_n & \text{(defn of } \texttt{f}\text{)} \\
& = (\lambda f\ x_1 \ldots x_n.\smallfrown f \smallfrown)\ (\texttt{Y}\ (\lambda f\ x_1 \ldots x_n.\smallfrown f \smallfrown))\ x_1 \ldots x_n & \\
& & \text{(fixed-point property)} \\
& = (\lambda f\ x_1 \ldots x_n.\smallfrown f \smallfrown)\ \texttt{f}\ x_1 \ldots x_n & \text{(defn of } \texttt{f}\text{)} \\
& = \smallfrown\texttt{f}\smallfrown & (\beta\text{-conversion})
\end{array}$$

Functions with several arguments

- **λ-expressions can only be applied to a single argument**

- **However, this argument can be a** *tuple*

- **Thus can write:**

  $$E(E_1, \ldots, E_n)$$

  which actually abbreviates:

  $$E(E_1, (E_2, (\cdots (E_{n-1}, E_n) \cdots)))$$

- **Example:** $E(E_1, E_2)$ **abbreviates** $E(\lambda f.\ f\ E_1\ E_2)$

Currying

- **Can encode multi-argument functions as:**

  (i) $(f\ x_1\ \ldots\ x_n)$, **or**

  (ii) the application of f **to** n**-tuple** $(x_1, \ldots, x_n)$

- **In (i),** f **expects its arguments 'one at a time'**
  - **said to be** *curried*
  - **after a logician called Curry**
  - **actually invented by Schönfinkel**

- `and`, `or` **and** `add` **are curried**

- **Curried functions can be 'partially applied'**
  - **for example,** `add` $\underline{1}$
  - **the result of partially applying** `add` **to** $\underline{1}$
  - **denotes the function** $n \mapsto n+1$

`curry` and `uncurry`

- **Consider:**

$$\texttt{LET curry} = \lambda f\ x_1\ x_2.\ f\ (x_1, x_2)$$

$$\texttt{LET uncurry} = \lambda f\ p.\ f\ (\texttt{fst}\ p)\ (\texttt{snd}\ p)$$

- **If `sum` and `prod` are defined by:**

$$\begin{aligned} \texttt{sum} &= \ \texttt{uncurry add} \\ \texttt{prod} &= \ \texttt{uncurry mult} \end{aligned}$$

- `sum`, `prod` **are 'uncurried' versions of** `add`, `mult`

- **For example:**

$$\begin{aligned} \texttt{sum}\ (\underline{m}, \underline{n}) &= \texttt{uncurry add}\ (\underline{m}, \underline{n}) \\ &= (\lambda f\ p.\ f\ (\texttt{fst}\ p)\ (\texttt{snd}\ p))\texttt{add}\ (\underline{m}, \underline{n}) \\ &= \texttt{add}\ (\texttt{fst}\ (\underline{m}, \underline{n}))\ (\texttt{snd}\ (\underline{m}, \underline{n})) \\ &= \texttt{add}\ \underline{m}\ \underline{n} \\ &= \underline{m{+}n} \end{aligned}$$

`curry` and `uncurry` are inverses

- **Can show:**

$$\texttt{curry}\ (\texttt{uncurry}\ E) = E$$

$$\texttt{uncurry}\ (\texttt{curry}\ E) = E$$

- **Hence:**

$$\texttt{add} = \texttt{curry sum}$$

$$\texttt{mult} = \texttt{curry prod}$$

n-ary currying and uncurrying

- **For $n > 0$ define:**

$$\texttt{LET curry}_n = \lambda f\ x_1 \cdots x_n.\ f\ (x_1, \ldots, x_n)$$
$$\texttt{LET uncurry}_n = \lambda f\ p.\ f\ (p \overset{n}{\downarrow} 1)\ \cdots\ (p \overset{n}{\downarrow} n)$$

- **If E represents a function expecting an n-tuple argument**
  - **then $\texttt{curry}_n\ E$ represents the curried function which takes its arguments one at a time**

- **If E represents a curried function of n arguments**
  - **then $\texttt{uncurry}_n\ E$ represents the uncurried version which expects a single n-tuple as argument**

- **Can show:**

$$\texttt{curry}_n\ (\texttt{uncurry}_n\ E) = E$$
$$\texttt{uncurry}_n\ (\texttt{curry}_n\ E) = E$$

Notation for uncurried functions

- **Generalized λ-abstractions:**

$$\texttt{LET}\ \lambda(V_1,\ldots,V_n).\ E = \texttt{uncurry}_n\ (\lambda V_1\ \ldots\ V_n.\ E)$$

- **Example:** $\lambda(x,y).\ \texttt{mult}\ x\ y$ **abbreviates:**

$$\begin{aligned}
&\texttt{uncurry}_2\ (\lambda x\ y.\ \texttt{mult}\ x\ y)\\
&\quad = (\lambda f\ p.\ f\ (p \overset{2}{\downarrow} 1)\ (p \overset{2}{\downarrow} 2))\ (\lambda x\ y.\ \texttt{mult}\ x\ y)\\
&\quad = (\lambda f\ p.\ f\ (\texttt{fst}\ p)\ (\texttt{snd}\ p))\ (\lambda x\ y.\ \texttt{mult}\ x\ y)\\
&\quad = \lambda p.\ \texttt{mult}\ (\texttt{fst}\ p)(\texttt{snd}\ p)
\end{aligned}$$

- **Thus:**

$$\begin{aligned}
&(\lambda(x,y).\ \texttt{mult}\ x\ y)\ (E_1,E_2)\\
&\quad = (\lambda p.\ \texttt{mult}\ (\texttt{fst}\ p)\ (\texttt{snd}\ p))\ (E_1,E_2)\\
&\quad = \texttt{mult}\ (\texttt{fst}(E_1,E_2))\ (\texttt{snd}(E_1,E_2))\\
&\quad = \texttt{mult}\ E_1\ E_2
\end{aligned}$$

Generalized β-conversion

- **Can derive:**

$$(\lambda(V_1, \ldots, V_n).\, E)\ (E_1, \ldots, E_n) = E\,[E_1, \ldots, E_n/V_1, \ldots, V_n]$$

  - $E\,[E_1, \ldots, E_n/V_1, \ldots, V_n]$ **is the** *simultaneous substitution* **of** $E_1, \ldots, E_n$ **for** $V_1, \ldots, V_n$**, respectively**
  - **none of these variables occur free in any of** $E_1, \ldots, E_n$

- **Can be derived from ordinary β-conversion**
  - **and the definitions of tuples**
  - **and generalized λ-abstractions**

- **A tuple of arguments is passed to each argument position in the body of the generalized abstraction**
  - **then each individual argument can be extracted from the tuple without affecting the others**

More syntactic sugar for abstractions

- **Convenient to extend notation $\lambda V_1\ V_2 \ldots V_n.\ E$**
  - **each V_i can either be an identifier**
  - **or a tuple of identifiers**

- $\lambda V_1\ V_2 \ldots V_n.\ E$ **still** $\lambda V_1.(\lambda V_2.(\cdots(\lambda V_n.\ E)\cdots))$
  - **if V_i is a tuple of identifiers**
  - **then the expression is a generalized abstraction**

- **Example:**

$$\lambda f\ (x, y).\ f\ x\ y$$

means

$$\lambda f.\ (\lambda(x, y).\ f\ x\ y)$$

which means

$$\lambda f.\ \texttt{uncurry}\ (\lambda x\ y.\ f\ x\ y)$$

which equals

$$\lambda f.\ (\lambda p.\ f\ (\texttt{fst}\ p)\ (\texttt{snd}\ p))$$

Mutual recursion

- **Consider the equations:**

$$\begin{aligned}\mathtt{f}_1 &= F_1\ \mathtt{f}_1 \cdots \mathtt{f}_n\\ \mathtt{f}_2 &= F_2\ \mathtt{f}_1 \cdots \mathtt{f}_n\\ &\vdots\\ \mathtt{f}_n &= F_n\ \mathtt{f}_1 \cdots \mathtt{f}_n\end{aligned}$$

- **Solution is:**

$\mathtt{f}_i = \mathtt{Y}\ (\lambda(f_1, \ldots f_n).\ (F_1\ f_1 \cdots f_n,\ \ldots\ , F_n\ f_1 \cdots f_n)) \downarrow i$

($1 \leq i \leq n$**)**

- **Works because if:**

$$\vec{\mathtt{f}} = \mathtt{Y}\ (\lambda(f_1, \ldots f_n).\ (F_1\ f_1 \cdots f_n,\ \ldots\ , F_n\ f_1 \cdots f_n))$$

- **Then** $\mathtt{f}_i = \vec{\mathtt{f}} \downarrow i$ **and hence:**

$$\begin{aligned}\vec{\mathtt{f}} &= (\lambda(f_1, \ldots, f_n).\ (F_1\ f_1 \cdots f_n,\ \ldots\ , F_n\ f_1 \cdots f_n))\vec{\mathtt{f}}\\ &= (F_1(\vec{\mathtt{f}} \downarrow 1) \cdots (\vec{\mathtt{f}} \downarrow n),\ \ldots\ , F_n(\vec{\mathtt{f}} \downarrow 1) \cdots (\vec{\mathtt{f}} \downarrow n))\\ &= (F_1\ \mathtt{f}_1 \cdots \mathtt{f}_n,\ \ldots\ , F_n\ \mathtt{f}_1 \cdots \mathtt{f}_n) \qquad \textbf{(as } \vec{\mathtt{f}} \downarrow i = \mathtt{f}_i\textbf{)}\end{aligned}$$

- **Hence:** $\mathtt{f}_i = F_i\ \mathtt{f}_1 \cdots \mathtt{f}_n$

Extending the λ-calculus

- **Can represent data-objects and data-structures with λ-expressions**
  - **often inefficient to do so**
  - **computers have hardware for arithmetic**
  - **why not use this, rather than λ-conversion**
- **Can 'interface' computation rules to λ-calculus**
- **Idea:**
  - **add a set of** *new constants*
  - **give rules for reducing applications involving these constants**
  - **such rules are called a** *δ-rules*

Example δ-rules

- **Add numerals and + as new constants**
- **Possible δ-rule:**

$$+ \; m \; n \xrightarrow[\delta]{} m{+}n$$

  - $E_1 \xrightarrow[\delta]{} E_2$ **means** E_2 **results by applying a** δ**-rule to some subexpression of** E_1

- **Must be careful to keep properties of λ-calculus**
  - **e.g. the Church-Rosser property**

Safe δ-rules

- **δ-rules are safe if they have the form:**

$$c_1 \; c_2 \; \cdots \; c_n \xrightarrow[\delta]{} e$$

  - **where c_1, $\ldots$, c_n are constants**
  - **and e is either a constant or a closed abstraction (such λ-expressions are sometimes called *values*)**

- **Example: add as constants `Suc`, `Pre`, `IsZero`, Δ_0, Δ_1, Δ_2, $\cdots$ with the δ-rules:**

$$\begin{aligned}
&\texttt{Suc}\ \Delta_n \xrightarrow[\delta]{} \Delta_{n+1}\\
&\texttt{Pre}\ \Delta_{n+1} \xrightarrow[\delta]{} \Delta_n\\
&\texttt{IsZero}\ \Delta_0 \xrightarrow[\delta]{} \texttt{true}\\
&\texttt{IsZero}\ \Delta_{n+1} \xrightarrow[\delta]{} \texttt{false}
\end{aligned}$$

  - **Δ_n represents the number n,**
  - **`Suc`, `Pre`, `IsZero` are new constants (i.e. not defined λ-expressions like `suc`, `pre`, `iszero`)**
  - **`true` and `false` defined above (both are values)**