Extending the Lambda Calculus: An Eager Functional Language

Syntax of the basic constructs:

canonical forms	z	cfm	::=	$intcfm \mid boolcfm \mid funcfm \mid tuplecfm \mid altcfm$
		$intcfm$	::=	$\mathtt{0} \mid \mathtt{1} \mid \mathtt{-1} \mid \ldots$
		$boolcfm$	::=	$boolconst$
		$funcfm$	::=	$\lambda var.\, exp$
		$tuplecfm$	::=	$\langle cfm, \ldots cfm \rangle$
		$altcfm$	::=	$@\ tag\ cfm$
		$boolconst$	::=	$\mathbf{true} \mid \mathbf{false}$
		$natconst$	::=	$\mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots$
expressions	e	exp	::=	$natconst \mid \mathtt{-}exp \mid exp + exp \mid \ldots$
			$\mid$	$boolconst \mid \neg exp \mid exp \wedge exp \mid \ldots$
			$\mid$	$\mathbf{if}\ exp\ \mathbf{then}\ exp\ \mathbf{else}\ exp$
			$\mid$	$funcfm \mid var \mid exp\ exp$
			$\mid$	$\langle exp, \ldots exp \rangle \mid exp\,.\,tag$
			$\mid$	$@\ tag\ exp \mid \mathbf{sumcase}\ exp\ \mathbf{of}\ (exp, \ldots exp)$
			$\mid$	$\mathbf{error} \mid \mathbf{typeerror}$
labels (tags)	k	tag	::=	$natconst$

Syntactic Sugar: Definitions and Patterns

Syntax of derived forms:

$$\begin{array}{rrcl} & \mathit{funcfm} & ::= & \ldots \mid \lambda \mathit{pat}.\, \mathit{exp} \\ & \mathit{exp} & ::= & \ldots \mid \textbf{let } \mathit{pat} \equiv \mathit{exp}, \ldots \mathit{pat} \equiv \mathit{exp} \textbf{ in } \mathit{exp} \\ \text{patterns } p & \mathit{pat} & ::= & \mathit{var} \mid \langle \mathit{pat}, \ldots \mathit{pat} \rangle \end{array}$$

Semantics of the derived forms:

$$\begin{array}{rcl} \textbf{let } p_1 \equiv e_1, \ldots p_n \equiv e_n \textbf{ in } e & \stackrel{\text{def}}{=} & (\lambda p_1. \ldots \lambda p_n.\, e)\, e_1 \ \ldots \ e_n \\ \lambda \langle p_0, \ldots p_{n-1} \rangle.\, e & \stackrel{\text{def}}{=} & \lambda v.\, \textbf{let } p_0 \equiv v\,.\,0, \ldots p_{n-1} \equiv v\,.\,\lfloor n{-}1 \rfloor \textbf{ in } e \\ & & \text{where } v \notin FV(e) \end{array}$$

Example:

$$\begin{array}{l} \textbf{let } \langle \mathsf{x}, \mathsf{y} \rangle \equiv \langle \mathsf{1}, \mathsf{2} \rangle \textbf{ in } \mathsf{x{+}y} \\ = \ (\lambda \langle \mathsf{x}, \mathsf{y} \rangle .\, \mathsf{x{+}y})\, \langle \mathsf{1}, \mathsf{2} \rangle \\ = \ (\lambda \mathsf{z}.\, \textbf{let } \mathsf{x} \equiv \mathsf{z\,.\,0}, \mathsf{y} \equiv \mathsf{z\,.\,1} \textbf{ in } \mathsf{x{+}y})\, \langle \mathsf{1}, \mathsf{2} \rangle \\ = \ (\lambda \mathsf{z}.\, (\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x{+}y})\, (\mathsf{z\,.\,0})\, (\mathsf{z\,.\,1}))\, \langle \mathsf{1}, \mathsf{2} \rangle \end{array}$$

Semantics of the Basic Constructs

$$\frac{}{z \Rightarrow z}$$

$$\frac{e \Rightarrow \lfloor i \rfloor}{\text{-}e \Rightarrow \lfloor -i \rfloor}$$

$$\frac{e \Rightarrow \lfloor i \rfloor \qquad e' \Rightarrow \lfloor i' \rfloor}{e + e' \Rightarrow \lfloor i + i' \rfloor}$$

$$\frac{e \Rightarrow \textbf{true} \qquad e' \Rightarrow z}{\textbf{if } e \textbf{ then } e' \textbf{ else } e'' \Rightarrow z}$$

$$\frac{e_1 \Rightarrow z_1 \quad \ldots \quad e_n \Rightarrow z_n}{\langle e_1, \ldots e_n \rangle \Rightarrow \langle z_1, \ldots z_n \rangle}$$

$$\frac{e \Rightarrow z}{@\, k\, e \Rightarrow @\, k\, z}$$

$$\frac{e \Rightarrow \lambda v.\, e_0 \qquad e' \Rightarrow z' \qquad (e_0/v \rightarrow z') \Rightarrow z}{e\, e' \Rightarrow z}$$

$$\frac{e \Rightarrow \lfloor b \rfloor}{\neg e \Rightarrow \lfloor \neg b \rfloor}$$

$$\frac{e \Rightarrow \lfloor b \rfloor \qquad e' \Rightarrow \lfloor b' \rfloor}{e \wedge e' \Rightarrow \lfloor b \wedge b' \rfloor}$$

$$\frac{e \Rightarrow \textbf{false} \qquad e'' \Rightarrow z}{\textbf{if } e \textbf{ then } e' \textbf{ else } e'' \Rightarrow z}$$

$$\frac{e \Rightarrow \langle z_0, \ldots z_{n-1} \rangle}{e.k \Rightarrow z_k} \quad \text{if } k < n$$

$$\frac{e \Rightarrow @\, k\, z \qquad e_k\, z \Rightarrow z'}{\textbf{sumcase } e \textbf{ of } (e_0, \ldots e_{n-1}) \Rightarrow z'} \quad \text{if } k < n$$

Recursion

In the untyped language we can define recursive functions using the (eager) fixed-point combinator Y_v.

But in a simple typing discipline Y_v is not typable.

One solution: a special construct for recursive definitions:

$$exp \;::=\; \ldots \mid \mathbf{letrec}\; var \equiv \lambda pat.\, exp, \ldots var \equiv \lambda pat.\, exp\; \mathbf{in}\; exp$$

with the reduction rule

$$\frac{(\lambda v_0.\, \ldots \lambda v_{n-1}.\, e)\,(\lambda u_0.\, e_0^*)\;\ldots\;(\lambda u_{n-1}.\, e_{n-1}^*) \;\Rightarrow\; z}{\mathbf{letrec}\; v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\; \mathbf{in}\; e \;\Rightarrow\; z}$$

where $e_i^* \stackrel{\text{def}}{=} \mathbf{letrec}\; v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\; \mathbf{in}\; e_i$

and $\{v_0, \ldots\} \cap \{u_0, \ldots\} = \{\}$

Syntactic sugar:

$$\mathbf{letrec}\; v_0\, p_0 \equiv e_0, \ldots\; \mathbf{in}\; e \stackrel{\text{def}}{=} \mathbf{letrec}\; v_0 \equiv \lambda p_0.\, e_0, \ldots\; \mathbf{in}\; e$$

Programming in the Eager Functional Language

Recall the reduction rule for **letrec**:

$$\frac{(\lambda v_0.\ \ldots \lambda v_{n-1}.\, e)\,(\lambda u_0.\, e_0^*)\ \ldots\ (\lambda u_{n-1}.\, e_{n-1}^*)\ \Rightarrow\ z}{\mathbf{letrec}\ v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\ \mathbf{in}\ e\ \Rightarrow\ z}$$

where $e_i^* \stackrel{\text{def}}{=} \mathbf{letrec}\ v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\ \mathbf{in}\ e_i$

and $\{v_0, \ldots\} \cap \{u_0, \ldots\} = \{\}$

An Example: Generating a List

$$\textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ \underbrace{\textbf{if } \mathsf{n} = 0 \textbf{ then } @\ 0\ \langle\rangle \textbf{ else } @\ 1\ \langle \mathsf{n},\ \mathsf{f}\ (\mathsf{n}-1)\rangle}_{e} \textbf{ in } \mathsf{f}\ 3$$

$$\begin{array}{l}
\quad (\lambda \mathsf{f}.\ \mathsf{f}\ 3)\ (\lambda \mathsf{n}.\ \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in } e) \\
\qquad (\lambda \mathsf{n}.\ \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in } e)\ 3 \\
\qquad\quad \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in if } 3 = 0 \textbf{ then } @\ 0\ \langle\rangle \textbf{ else } @\ 1\ \langle 3,\ \mathsf{f}\ (3-1)\rangle \\
\qquad\qquad (\lambda \mathsf{f}.\ \textbf{if } 3 = 0 \textbf{ then } @\ 0\ \langle\rangle \textbf{ else } @\ 1\ \langle 3,\ \mathsf{f}\ (3-1)\rangle)\ (\lambda \mathsf{n}.\ \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in } e) \\
\qquad\qquad\quad \textbf{if } 3 = 0 \textbf{ then } @\ 0\ \langle\rangle \textbf{ else } @\ 1\ \langle 3,\ (\lambda \mathsf{n}.\ \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in } e)\ (3-1)\rangle \\
\qquad\qquad\qquad @\ 1\ \langle 3,\ (\lambda \mathsf{n}.\ \textbf{letrec } \mathsf{f} \equiv \lambda \mathsf{n}.\ e \textbf{ in } e)\ (3-1)\rangle \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad @\ 1\ \langle 3,\ @\ 1\ \langle 2,\ @\ 1\ \langle 1,\ @\ 0\ \langle\rangle\rangle\rangle\rangle \\
\qquad\qquad\quad \ldots \\
\qquad\qquad \ldots \\
\qquad\quad \ldots \\
\quad \ldots \\
@\ 1\ \langle 3,\ @\ 1\ \langle 2,\ @\ 1\ \langle 1,\ @\ 0\ \langle\rangle\rangle\rangle\rangle
\end{array}$$

Derived Data Types: Lists

Following Standard ML's syntax for lists

$$\ldots\ e_1\texttt{::}e_2\texttt{::}\ldots\texttt{::}e_n\texttt{::}\mathbf{nil}\ \ldots$$

$$\ldots\ \texttt{case}\ e\ \texttt{of nil =>}\ e'\ \texttt{|}\ \texttt{x::xs =>}\ e''\ \ldots$$

we can extend the language with

$$\begin{array}{rcll} exp & ::= & \ldots \mid \mathbf{nil} \mid exp\texttt{::}exp & \text{constructors} \\ & \mid & \mathbf{listcase}\ exp\ \mathbf{of}\ (exp,\ exp) & \text{deconstructor} \\ cfm & ::= & \mathbf{nil} \mid cfm\texttt{::}cfm & \end{array}$$

Semantics and Encoding of Lists

Evaluation semantics:

$$\frac{}{\mathbf{nil} \Rightarrow \mathbf{nil}} \qquad \frac{e_1 \Rightarrow z_1 \quad e_2 \Rightarrow z_2}{e_1 :: e_2 \Rightarrow z_1 :: z_2}$$

$$\frac{e \Rightarrow \mathbf{nil} \quad e' \Rightarrow z}{\mathbf{listcase}\ e\ \mathbf{of}\ (e', e'') \Rightarrow z} \qquad \frac{e \Rightarrow z_1 :: z_2 \quad e''\,\langle z_1, z_2\rangle \Rightarrow z}{\mathbf{listcase}\ e\ \mathbf{of}\ (e', e'') \Rightarrow z}$$

Encoding of lists in terms of products and sums:

$$\mathbf{nil} \stackrel{\text{def}}{=} @\ 0\ \langle\rangle$$

$$e_1 :: e_2 \stackrel{\text{def}}{=} @\ 1\ \langle e_1, e_2\rangle$$

$$\mathbf{listcase}\ e\ \mathbf{of}\ (e', e'') \stackrel{\text{def}}{=} \mathbf{sumcase}\ e\ \mathbf{of}\ (\lambda\_.\, e', e'')$$

Note: LISP-style languages define a tester **null?** and deconstructors **car** and **cdr** but **car** and **cdr** are not well-defined on **nil**.

List Processing

Some functions from the list libraries of ML and Haskell
expressed in our eager functional language:

$$\mathbf{letrec}\ \mathsf{append} \equiv \lambda \mathsf{xs}.\ \lambda \mathsf{ys}.\ \mathbf{listcase}\ \mathsf{xs}\ \mathbf{of}\ (\mathsf{ys},\ \lambda\langle \mathsf{x},\ \mathsf{xs}'\rangle.\ \mathsf{x} :: \mathsf{append}\ \mathsf{xs}'\ \mathsf{ys})\ \mathbf{in}$$

$$\mathbf{letrec}\ \mathsf{map} \equiv \lambda \mathsf{f}.\ \lambda \mathsf{xs}.\ \mathbf{listcase}\ \mathsf{xs}\ \mathbf{of}\ (\mathbf{nil},\ \lambda\langle \mathsf{x},\ \mathsf{xs}'\rangle.\ \mathsf{f}\ \mathsf{x} :: \mathsf{map}\ \mathsf{f}\ \mathsf{xs}')\ \mathbf{in}$$

$$\mathbf{letrec}\ \mathsf{foldr} \equiv \lambda \mathsf{f}.\ \lambda \mathsf{z}.\ \lambda \mathsf{xs}.\ \mathbf{listcase}\ \mathsf{xs}\ \mathbf{of}\ (\mathsf{z},\ \lambda\langle \mathsf{x},\ \mathsf{xs}'\rangle.\ \mathsf{f}\ \mathsf{x}\ (\mathsf{foldr}\ \mathsf{f}\ \mathsf{z}\ \mathsf{xs}'))\ \mathbf{in}$$

$$\mathbf{letrec}\ \mathsf{revappend} \equiv \lambda \mathsf{xs}.\ \lambda \mathsf{ys}.\ \mathbf{listcase}\ \mathsf{xs}\ \mathbf{of}\ (\mathsf{ys},\ \lambda\langle \mathsf{x},\ \mathsf{xs}'\rangle.\ \mathsf{revappend}\ \mathsf{xs}'\ (\mathsf{x} :: \mathsf{ys}))\ \mathbf{in}$$

$$\mathbf{let}$$

$$\mathsf{map}' \equiv \lambda \mathsf{f}.\ \mathsf{foldr}\ (\lambda \mathsf{x}.\ \lambda \mathsf{ys}.\ \mathsf{f}\ \mathsf{x} :: \mathsf{ys})\ \mathbf{nil},$$

$$\mathsf{append}' \equiv \lambda \mathsf{xs}.\ \lambda \mathsf{ys}.\ \mathsf{foldr}\ (\lambda \mathsf{z}.\ \lambda \mathsf{zs}.\ \mathsf{z} :: \mathsf{zs})\ \mathsf{ys}\ \mathsf{xs},$$

$$\mathsf{rev} \equiv \lambda \mathsf{xs}.\ \mathsf{revappend}\ \mathsf{xs}\ \mathbf{nil},$$

$$\mathsf{append}'' \equiv \lambda \mathsf{xs}.\ \mathsf{revappend}\ (\mathsf{rev}\ \mathsf{xs})$$

$$\mathbf{in}\ \ldots$$

Direct Denotational Semantics

Let the predomain of values be $V \underset{\psi}{\overset{\phi}{\rightleftarrows}} V_{int} + V_{bool} + V_{fun} + V_{tup} + V_{alt}$, where

$$\begin{array}{rcl} V_{int} & = & \mathbf{Z} \\ V_{bool} & = & \mathbf{B} \\ V_{fun} & = & [V \to V_*] \\ V_{tup} & = & \bigcup_{n=0}^{\infty} V^n \\ V_{alt} & = & \mathbf{N} \times V \end{array}$$

with injections

$$\begin{array}{rcl} \iota_{int} & = & \psi \cdot \iota_0 \in [V_{int} \to V] \\ \iota_{bool} & = & \psi \cdot \iota_1 \in [V_{bool} \to V] \\ \iota_{fun} & = & \psi \cdot \iota_2 \in [V_{fun} \to V] \\ \iota_{tup} & = & \psi \cdot \iota_3 \in [V_{tup} \to V] \\ \iota_{alt} & = & \psi \cdot \iota_4 \in [V_{alt} \to V] \end{array}$$

and the domain of results is $V_* = (V + \{\mathbf{error}, \mathbf{typeerror}\})_\perp$:

$$\begin{array}{rcll} \iota_{norm} & = & \iota_\uparrow \cdot \iota_0 & \in [V \to V_*] \\ \text{err} & = & \iota_\uparrow(\iota_1\, \mathbf{error}) & \in V_* \\ \text{tyerr} & = & \iota_\uparrow(\iota_1\, \mathbf{typeerror}) & \in V_* \end{array}$$

Then $[\![-]\!] \in exp \to E \to V_*$, where $E = var \to V$ is the predomain of environments:

$$\begin{array}{rcl} [\![\mathbf{error}]\!]\eta & = & \text{err} \\ [\![\mathbf{typeerror}]\!]\eta & = & \text{tyerr} \\ [\![\lfloor n \rfloor]\!]\eta & = & \iota_{norm}\,(\iota_{int}\, n) \end{array}$$

Direct Denotational Semantics cont'd

$$[\![\text{-}e]\!]\eta = (\lambda n \in \mathbf{Z}.\, \iota_{norm}(\iota_{int}(-n)))_{int*}\,([\![e]\!]\eta)$$

$$f_{int*} \stackrel{\text{def}}{=} (f_{int})_*$$

where the lift $(-)_*$ from $[V \to V_*]$ to $[V_* \to V_*]$ is

$$\begin{aligned} f_*\,(\iota_{norm}\,z) &= f\,z \\ f_*\,\text{err} &= \text{err} \\ f_*\,\text{tyerr} &= \text{tyerr} \end{aligned}$$

and we define a type-checking lift $(-)_{int}$ of functions from $[V_{int} \to V_*]$ to $[V \to V_*]$:

$$\begin{aligned} f_{int}\,(\iota_{int}\,n) &= f\,n \\ f_{int}\,z &= \text{tyerr for other } z \in V \end{aligned}$$

$(-)_{bool}$, $(-)_{fun}$, $(-)_{tup}$, and $(-)_{alt}$ are defined similarly, and from them $(-)_{bool*}$ etc.

Denotational Semantics of Binary Operations

$$[\![e\text{+}e']\!]\eta = (\lambda n \in \mathbf{Z}.\,(\lambda n' \in \mathbf{Z}.\,\iota_{norm}(\iota_{int}(n+n')))_{int*}\,([\![e']\!]\eta))_{int*}\,([\![e]\!]\eta)$$

Note:

- this semantics evaluates e+e' from left to right
- the symmetry of + from the expression language is broken
- either of e and e' may fail (err or tyerr) or diverge

If $[\![e]\!]\eta = \perp$ and $[\![e]\!]'\eta =$ tyerr, then

$$\begin{aligned} [\![e\text{+}e']\!]\eta &= \perp \\ [\![e'\text{+}e]\!]\eta &= \text{tyerr} \end{aligned}$$

Runtime Errors

One approach to introducing types:
Consider sets of values closed under the semantics of certain language constructs,

e.g. integers are closed under the semantics of `+`, `-`, `*`.

But operations like the division $\div$ are not defined on all values in a type.

We could consider more fine-grained types, e.g. the set $\{\langle n, n' \rangle \mid n \in \mathbf{Z},\ n' \in \mathbf{Z} - \{0\}\}$, but a type system which infers them would be undecidable.

The prevailing approach is to

defer the check for definedness until run time, and signal an error if undefined:

$$[\![e \div e']\!]\eta = (\lambda n \in \mathbf{Z}.\, (\lambda n' \in \mathbf{Z}.\, \mathbf{if}\ n' = 0\ \mathbf{then}\ \mathrm{err} \\ \mathbf{else}\ \iota_{norm}(\iota_{int}(n \div n')))_{int*}\, ([\![e']\!]\eta))_{int*}\, ([\![e]\!]\eta)$$

Functions, Tuples, Alternatives

$$
\begin{aligned}
[\![v]\!]\eta &= \iota_{norm}\,(\eta\, v) \\
[\![e\, e']\!]\eta &= (\lambda f \in [V_{fun} \to V_*].\, (\lambda z \in V.\, f\, z)_*\,([\![e']\!]\eta))_{fun*}\,([\![e]\!]\eta) \\
[\![\lambda v.\, e]\!]\eta &= \iota_{norm}\,(\iota_{fun}\,(\lambda z \in V.\, [\![e]\!][\eta \mid v : z])) \\
\\
[\![\langle e_1, \ldots\, e_n\rangle]\!]\eta &= (\lambda z_1 \in V.\, \ldots \quad (\lambda z_n \in V.\, \iota_{norm}(\iota_{tup}\langle z_1, \ldots\, z_n\rangle))_* \\
&\qquad\qquad ([\![e_n]\!]\eta)\ \ldots)_*\,([\![e_1]\!]\eta) \\
[\![e\,.\,k]\!]\eta &= (\lambda z \in V^*.\, \textbf{if } k \in \mathrm{dom}\, z \textbf{ then } \iota_{norm}\,(z\, k) \textbf{ else tyerr})_{tup*}\,([\![e]\!]\eta) \\
\\
[\![@\, k\, e]\!]\eta &= (\lambda z \in V.\, \iota_{norm}\,(\iota_{alt}\,\langle k,\, z\rangle))_*\,([\![e]\!]\eta) \\
[\![\textbf{sumcase } e \textbf{ of } (e_0, \ldots\, e_{n-1})]\!]\eta & \\
&= (\lambda\langle k, z\rangle \in \mathbf{N} \times V.\, \textbf{if } k < n \textbf{ then } (\lambda f \in [V \to V_*].\, f\, z)_{fun*}\,([\![e_k]\!]\eta)
\end{aligned}
$$

Semantics of letrec

The reduction rule for **letrec**

$$\frac{(\lambda v_0.\ \ldots \lambda v_{n-1}.\, e)\,(\lambda u_0.\, e_0^*)\ \ldots\ (\lambda u_{n-1}.\, e_{n-1}^*)\ \Rightarrow\ z}{\mathbf{letrec}\ v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\ \mathbf{in}\ e\ \Rightarrow\ z}$$

where $e_i^* \stackrel{\text{def}}{=} \mathbf{letrec}\ v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\ \mathbf{in}\ e_i$

and $\{v_0, \ldots\} \cap \{u_0, \ldots\} = \{\}$

suggests that the semantic equation is

$$[\![\mathbf{letrec}\ v_1 \equiv \lambda u_1.\, e_1, \ldots v_n \equiv \lambda u_n.\, e_n\ \mathbf{in}\ e]\!]\eta \quad = \quad [\![e]\!]\eta'$$

where $\eta' = [\eta \,|\, v_1 : [\![\lambda u_1.\, e_1]\!]\eta' \,|\, \ldots \,|\, v_n : [\![\lambda u_n.\, e_n]\!]\eta']$

The last equation is ill-formed: $\eta' \in E = var \to V$, while $[\![\lambda u_1.\, e_1]\!]\eta' \in V_*$,

however the r.h.sides are restricted syntactically to be values, and

$$[\![\lambda u_i.\, e_i]\!]\eta' = \iota_{norm}\,(\iota_{fun}\,(\lambda u_i \in V.\, [\![e_i]\!][\eta' \,|\, u_i : z]))$$

so we can use directly the corresponding element of V:

$$\eta' = [\eta \,|\, v_1 : \iota_{fun}\,(\lambda u_1 \in V.\, [\![e_1]\!][\eta' \,|\, u_1 : z]) \,|\, \ldots \,|\, v_n : \iota_{fun}\,(\lambda u_n \in V.\, [\![e_n]\!][\eta' \,|\, u_n : z])]$$

Semantics of letrec cont'd

One remaining problem: the equation

$$\eta' = [\eta \mid v_1 : \iota_{fun}\,(\lambda u_1 \in V.\, [\![e_1]\!][\eta' \mid u_1 : z]) \mid \ldots \mid v_n : \iota_{fun}\,(\lambda u_n \in V.\, [\![e_n]\!][\eta' \mid u_n : z])]$$

cannot be solved using the least fixed-point operator $\mathbf{Y}$

because $E = var \to V$ is a predomain, not a domain.

But we can break the loop at a different point, where the value is in a domain:

in the case of $n = 1$

$$\begin{aligned} \eta' &= [\eta \mid v : \iota_{fun}\, f] \\ \text{where } f &= \lambda u \in V.\, [\![e]\!][\eta' \mid u : z] \\ \text{hence } f &= \lambda u \in V.\, [\![e]\!][\eta \mid v : \iota_{fun}\, f \mid u : z] \end{aligned}$$

f is in V_{fun}, which is a domain, hence

$$f = \mathbf{Y}_{V_{fun}}\,(\lambda g \in V_{fun}.\, \lambda u \in V.\, [\![e]\!][\eta \mid v : \iota_{fun}\, g \mid u : z]).$$