Programming Languages (I2)

Implementing PicoML

(A polymorphically-typed lazy functional language)

Bernard Sufrin

Version 1.20 26th February, 2002 09:46

PicoML

- ➩ Based on the Lambda Calculus (like Haskell)
- ➩ Lazy (like Haskell)
- ➩ Polymorphically typed (like OCAML, Haskell, SML)
- ➩ Numbers represented as arbitrary precision rationals (just for fun)
- ➩ Simple *Iswim*-style concrete syntax (like OCAML, SML)

  But

- ➩ No pattern-matching (yet)

➥

In these lectures we shall present

- ➩ an operational semantics
- ➩ an OCAML interpreter
- ➩ a type system and typechecker

Supplementary Reading:

```
The Implementation of Functional Programming Languages
S.L.Peyton-Jones
Prentice-HallInternational
```

➥

Abstract Syntax (Caml notation) of expressions ($e, e_1, e_2, ...$)

`expr =`

`| Id`	`of identifier`	written as $i,\ \ j,\ \ k$
`| Num`	`of string`	written as `1`, `2`, `1.6`, `0.016e2`, ...
`| ...`		... other kinds of constant ...
`| Pair`	`of expr*expr`	written as $(e_1,\ \ e_2)$
`| Apply`	`of expr*expr`	written as $e_1\ \ e_2$
`| Fun`	`of identifier*expr`	written as `{i -> e}` or $\lambda\ i\ \cdot\ e$
`| Let`	`of decl*expr`	written as $\textbf{let}\ \ \Delta\ \ \textbf{in}\ \ e\ \ \textbf{end}$
`| If`	`of expr*expr*expr`	written as $\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3$

Examples:

`f x y`	is represented by `Apply(Apply(Id "f", Id "x"), Id "y")`
`x+y`	is represented by `Apply(Apply(Id "+", Id "x"), Id "y")`
`(+)`	is represented by `Id "+"`
`(x, y)`	is represented by `Pair(Id "x", Id "y")`

Abstract syntax of declarations $(\Delta, \Delta_1, \Delta_2, ...)$

`decl =`

`| ValDec  of identifier*expr` written as $i \;=\; e$

`| AndDec  of decl*decl` written as Δ_1 **and** Δ_2

`| Seq     of decl*decl` written as $\Delta_1 \;;\; \Delta_2$

`| ...` ... other forms of declaration ...

`| RecDec  of decl` written as **rec** Δ

`| HasType of identifier*typexpr` written as $i : \tau$

Infixed operators ($\otimes$) are just a convenient notation for application

$$e_1 \otimes e_2 \text{ sugars } (\otimes)\ e_1\ e_2$$

Function definitions are also sugared, *e.g.*

```
rec  map f xs = if null xs then nil else f (hd xs) :: map f (tl xs)
```

sugars

$$\begin{aligned}&\textbf{rec}\ \ \texttt{map =}\\ &\quad \lambda\ \texttt{f}\ .\ \lambda\ \texttt{x}\ .\ \textbf{if}\ \texttt{null xs}\ \ \textbf{then}\ \texttt{nil}\ \textbf{else}\ \underbrace{\texttt{(::)}\ \ \texttt{(f (hd xs))}\ \ \texttt{(map f (tl xs))}}_{\text{application of ::}}\end{aligned}$$

and

```
{ xs x -> x::xs }
```

sugars

$$\lambda\ \texttt{xs}\ .\ \lambda\ \texttt{x}\ .\ \texttt{x::xs}$$

Abstract syntax of type expressions ($\tau, \tau_1, \tau_2, ...$)

`typexpr =`

`| UnitType`		written as `()`
`| NumType`		written as Num
`| BoolType`		written as Bool
`| CharType`		written as Char
`| PairType`	`of typexpr * typexpr`	written as (τ_1, τ_2)
`| ListType`	`of typexpr`	written as $[\tau]$
`| ConType`	`of identifier*typexpr list`	user-defined types
`| FunType`	`of typexpr * typexpr`	written as $\tau_1 \rightarrow \tau_2$
`| VarType`	`of identifier`	written as $a, \; b, \; c, \; ...$
`| PolyType`	`of identifier list * typexpr`	written as $\forall \; i_1, \; ... \; \cdot \; \tau$
		or $@i_1, \; ... \; \cdot \; \tau$

- ➩ Lists and pairs are the only inbuilt composite data types.
- ➩ Strings are character-lists.
- ➩ User-defined data types in the manner of Haskell.
- ➩ Explicit (but not nested) quantification of the type variables in polymorphic types.

Examples:

$\forall a, b \cdot (a \rightarrow b) \rightarrow ([a] \rightarrow [b])$	the type of map
$(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$	sugar for the type of map
$(\mathrm{Num}, \mathrm{Num}) \rightarrow (\mathrm{Num}, \mathrm{Num})$	the type of $\lambda x \cdot (fst\ x - snd\ x, snd\ x - fst\ x)$
$(\mathrm{Num}, \mathrm{Num}, \mathrm{Num})$	the type of $(1, 2, 3)$ and of $(1, (2, 3))$

Tuple notation sugars pair notation. For example:

(w, x, y, z)	sugars	$(w, (x, (y, z)))$
$(\tau_1, \tau_2, \tau_3, \tau_4)$	sugars	$(\tau_1, (\tau_2, (\tau_3, \tau_4)))$

An extract from the picoML library:

```
let
    map    :  (arg->result)->[arg]->[result];
    len    :  [a]->Num;
    ( ++ ) :  [a] -> [a] -> [a];
    ( >> ) :  (c->b)->(b->a)->c->a;
    ( << ) :  (c->b)->(a->c)->a->b;
    rev    :  [a]->[a];

    rec  map f xs = if null xs then nil else f (hd xs) :: map f (tl xs);
    rec  len xs   = if null xs then 0 else 1+len(tl xs);
    rec  xs ++ ys = if null xs then ys else hd xs :: (tl xs ++ ys);
    (f << g) x = f(g x);
    (f >> g) x = g(f x);
    rev = foldleft [] {xs x -> x::xs}
 end
```

- ⇨ **let** *declarations* **end** – extends the top-level environment.
- ⇨ *id*:*type* – type annotation (checked) of subsequently defined value.

An interactive session consists of a sequence of declarations and expressions.

```
% picoml.opt picoml.pml
zipWith : @a,b,c.(c->b->a)->[c]->[b]->[a]
(!!)    : @a.[a]->Num->a
map     : @result,arg.(arg->result)->[arg]->[result]
...
```

Start the (fast) interpeter
(loading the standard library)

```
: let rec nats = 0 :: map { x -> x+1 } nats;;
nats : [Num]
```

Define an infinite list

```
: let rec facs = 1 :: zipWith (*) facs (tl nats);;
facs : [Num]
```

Define another list

```
: facs!!100;;
9332621544394415268169923885626670049071596826438162146859296389521759999
3229915608941463976156518286253697920827223758251185210916864000000000000
0000000000 :  Num (0.02 secs, 1576/2873 shared, 3176 apps, 200 cons)
```

Calculate the 100th factorial

```
: facs!!100;;
9332621544394415268169923885626670049071596826438162146859296389521759999
3229915608941463976156518286253697920827223758251185210916864000000000000
0000000000 :  Num (0.01 secs, 400/600 shared, 798 apps, 0 cons)
```

Calculate it again

Constructors (tuples, lists) are lazy

```
: fst(3, 1/0);;
3 : Num (0/1 shared, 1 apps, 0 cons)

: fst(snd(3, (1/0, "silly")));;
Runtime error: create_ratio infinite or undefined rational number
```

Top-level constructions are evaluated incrementally

```
: facs;;
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800,
 87178291200,1307674368000,20922789888000,355687428096000,6402373705728000,
 121645100408832000,2432902008176640000, [Interrupted]

: map (100/) [10,5,0];;
[10,20,
Runtime error: create_ratio infinite or undefined rational number
```

Natural ("big-step") semantics defines "complete evaluations".

The judgement $E\surd\surd$ means:

"the expression E is in normal form"

The judgement $E\surd$ means:

"the expression E is in (weak) normal form"

The judgement $E \overset{\surd}{\Longrightarrow} E'$ means:

"the expression E evaluates to the (weak) normal form E'."

➥

The judgement $E \Longrightarrow E'$ means:

"the expression E has the same (weak) normal form as E'."

Definitions (normal forms)

- A *normal form* ($\surd\surd$) is a term which *cannot* be further evaluated
  - Built-in operators, numbers, *etc.* are in normal form.
  - Function abstractions are in normal form.
  - Partly-applied built-in binary arithmetic operators are in normal form.
- A *weak normal* form ($\surd$) is a term which (as a matter of language design policy) will not be further evaluated

➥

Built-in primitive functions (hd , tl , $(+), (/), ...$) are in normal form, as are numbers (n), truth-values, *etc.* Furthermore

$$\frac{}{(\lambda\ x{\cdot}E)\surd\surd}\ (\lambda\surd\surd)$$

$$\frac{\left(\Omega \text{ is one of } \left\{(+), (-), (/), (\times), ...\right\}\right)}{\Omega\ n\surd\surd}\ (\Omega\ n\surd\surd)$$

Judgements are related by the following rules

$$\frac{E\surd\surd}{E\surd}$$

$$\frac{E\surd}{E \Longrightarrow E}\;\text{Normal}$$

$$\frac{E\surd}{E \overset{\surd}{\Longrightarrow} E}\;\text{Normal}$$

$$\frac{E \Longrightarrow E_1 \qquad E_1 \overset{\surd}{\Longrightarrow} E'}{E \overset{\surd}{\Longrightarrow} E'}\;\text{Step}$$

Substitution

The notation $[\![\overline{v} := \overline{E}]\!]E$ stands for

“the expression E with expressions $\overline{E}$ substituted for free occurences of variables $\overline{v}$.”

Examples

$$[\![foo, bar := 3, foo + 4]\!](foo, bar) \text{ is } (3, foo + 4)$$

$$[\![foo, bar := K(bar), K'(foo)]\!](K(bar), K'(foo)) \text{ is } (K(K'(foo)), K'(K(bar)))$$

Scoping conventions for bound variables are the usual ones.

$$[\![x, y := 3, foo + 4]\!](\lambda x \cdot (x, y)) \text{ is } (\lambda x \cdot (x, foo + 4))$$

$$[\![x, y := 3, foo + 4]\!]((\lambda x \cdot (x, y))\ x) \text{ is } (\lambda x \cdot (x, foo + 4))(3)$$

Computation Rules

$$\frac{F \overset{\surd}{\Longrightarrow} (\lambda\ x \cdot B) \qquad [\![x := A]\!]B \Longrightarrow R}{F\ A \Longrightarrow R}\ \text{(apply)}$$

$$\frac{[\![x := A]\!]B \Longrightarrow R}{\textbf{let}\ x = A\ \textbf{in}\ B\ \textbf{end} \Longrightarrow R}\ \textbf{let}\ \text{(first approximation)}$$

$$\frac{E_1 \overset{\surd}{\Longrightarrow} \textbf{true} \qquad E_2 \Longrightarrow R}{\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3 \Longrightarrow R}\ \textbf{if}_{\ \textbf{true}}$$

$$\frac{E_1 \overset{\surd}{\Longrightarrow} \textbf{false} \qquad E_3 \Longrightarrow R}{\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3 \Longrightarrow R}\ \textbf{if}_{\ \textbf{false}}$$

➥

Arithmetic Rules (examples)

$$\frac{F \overset{\surd}{\Longrightarrow} (+) \qquad A \overset{\surd}{\Longrightarrow} n_1}{F\ A \Longrightarrow ((+)\ n_1)}\ +_1$$

$$\frac{F \overset{\surd}{\Longrightarrow} ((+)\ n_1) \qquad A \overset{\surd}{\Longrightarrow} n_2}{F\ A \Longrightarrow \overline{n_1 + n_2}}\ +_2$$

$$\frac{F \overset{\surd}{\Longrightarrow} (/) \qquad A \overset{\surd}{\Longrightarrow} n_1}{F\ A \Longrightarrow ((/)\ n_1)}\ /_1$$

$$\frac{F \overset{\surd}{\Longrightarrow} ((/)\ n_1) \qquad A \overset{\surd}{\Longrightarrow} n_2 \qquad (n_2 \neq 0)}{F\ A \Longrightarrow \overline{n_1 \div n_2}}\ /_2$$

Binary arithmetic operators are strict in both their arguments.

➥

“Pairing is lazy”

$$\frac{}{(E_1, E_2)\surd}$$

$$\frac{F \overset{\surd}{\Longrightarrow} \textsf{fst} \qquad A \overset{\surd}{\Longrightarrow} (E_1, E_2)}{F\ A \Longrightarrow E_1}\ \textsf{fst}$$

$$\frac{F \overset{\surd}{\Longrightarrow} \textsf{snd} \qquad A \overset{\surd}{\Longrightarrow} (E_1, E_2)}{F\ A \Longrightarrow E_2}\ \textsf{snd}$$

- ➪ Pairs are *already* in WNF.
- ➪ fst and snd force evaluation to WNF.
- ➪ but neither result in evaluation of the projected term.

“List Cons is lazy”

$$\frac{}{E_1{::}E_2\surd} \qquad \frac{}{(::)\ E_1\surd\surd} \qquad \frac{}{Nil\surd\surd}$$

$$\frac{F \overset{\surd}{\Longrightarrow} (::)}{F\ E \Longrightarrow (::)\ E}\ {::}_1$$

$$\frac{F \overset{\surd}{\Longrightarrow} (::)\ E_1}{F\ E_2 \Longrightarrow E_1{::}E_2}\ {::}_2$$

null , hd , and tl force evaluation to WNF.

$$\frac{F \overset{\surd}{\Longrightarrow} \mathsf{null} \qquad A \overset{\surd}{\Longrightarrow} E_1{::}E_2}{F\ A \Longrightarrow \mathbf{false}}\ \mathsf{null}_{F}$$

$$\frac{F \overset{\surd}{\Longrightarrow} \mathsf{null} \qquad A \overset{\surd}{\Longrightarrow} Nil}{F\ A \Longrightarrow \mathbf{true}}\ \mathsf{null}_{T}$$

$$\frac{F \overset{\surd}{\Longrightarrow} \mathsf{hd} \qquad A \overset{\surd}{\Longrightarrow} E_1{::}E_2}{F\ A \Longrightarrow E_1}\ \mathsf{hd}$$

$$\frac{F \overset{\surd}{\Longrightarrow} \mathsf{tl} \qquad A \overset{\surd}{\Longrightarrow} E_1{::}E_2}{F\ A \Longrightarrow E_2}\ \mathsf{tl}$$

Example computation #1

$$
\dfrac{
  \dfrac{
    \dfrac{\dfrac{}{\text{fst}\ \surd}\ \text{fst}\ \surd}{\text{fst}\ \xRightarrow{\surd}\ \text{fst}}\ \mathit{Normal}
    \qquad
    \dfrac{\dfrac{}{(3, 1/0)\surd}\ (E1, E2)\surd}{(3, 1/0)\ \xRightarrow{\surd}\ (3, 1/0)}\ \mathit{Normal}
  }{
    \dfrac{\text{fst}\ (3, 1/0) \Longrightarrow 3}{\mathbf{let}\ f = \text{fst}\ \ \mathbf{in}\ f(3, 1/0)\ \mathbf{end} \Longrightarrow 3}\ \mathit{Let}
  }\ \mathit{Fst}
}{
  \mathbf{let}\ x = (3, 1/0)\ \mathbf{in}\ \mathbf{let}\ f = \text{fst}\ \ \mathbf{in}\ f\ x\ \mathbf{end}\ \mathbf{end} \Longrightarrow 3
}\ \mathit{Let}
$$

Remarks

- The $\frac{E\surd}{E \xRightarrow{\surd} E}$ subcomputations are necessary – we have omitted $\frac{E\surd}{E\surd\surd}$ steps.
- The rules are there to specify computations, not facilitate human insight.
- The static presentation doesn’t show the order in which things happened.

  (the outer **let** doesn’t “know” the answer is going to be 3)

Example computation #2

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \dfrac{
     \dfrac{\dfrac{}{\textbf{true}\surd}\,\textbf{true}\surd}{\textbf{true} \overset{\surd}{\Longrightarrow} \textbf{true}}\,\textit{Normal}
     \quad
     \dfrac{\dfrac{}{\textsf{fst}\ \surd}\,\textsf{fst}\ \surd}{\textsf{fst} \Longrightarrow \textsf{fst}}\,\textit{Normal}
    }{\textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ \textsf{fst}\ \ \textbf{else}\ \ \textsf{snd} \Longrightarrow \textsf{fst}}\,\textit{IfT}
    \quad
    \dfrac{\dfrac{}{\textsf{fst}\ \surd}\,\textsf{fst}\ \surd}{\textsf{fst} \overset{\surd}{\Longrightarrow} \textsf{fst}}\,\textit{Normal}
   }{\textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ \textsf{fst}\ \ \textbf{else}\ \ \textsf{snd} \overset{\surd}{\Longrightarrow} \textsf{fst}}\,\textit{Step}
   \quad
   \dfrac{\dfrac{}{(3,1/0)\surd}\,(E1,E2)\surd}{(3,1/0) \overset{\surd}{\Longrightarrow} (3,1/0)}\,\textit{Normal}
  }{(\textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ \textsf{fst}\ \ \textbf{else}\ \ \textsf{snd}\ )(3,1/0) \Longrightarrow 3}\,\textit{Fst}
 }{\textbf{let}\ \ f = (\textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ \textsf{fst}\ \ \textbf{else}\ \ \textsf{snd}\ )\ \textbf{in}\ \ f(3,1/0)\ \textbf{end} \Longrightarrow 3}\,\textit{Let}
}{\textbf{let}\ \ x = (3,1/0)\ \textbf{in}\ \ \textbf{let}\ \ f = (\textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ \textsf{fst}\ \ \textbf{else}\ \ \textsf{snd}\ )\ \textbf{in}\ \ f\ x\ \ \textbf{end}\ \ \textbf{end} \Longrightarrow 3}\,\textit{Let}
$$

Example computation #3 and #4 ($\overset{\surd}{\Longrightarrow}$ written as $\Longrightarrow\ \surd$)

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\scriptstyle(+)\surd\\ ((+))\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ ((+))\Longrightarrow\surd((+))\end{array}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\scriptstyle\mathsf{hd}\surd\\ \mathsf{hd}\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ \mathsf{hd}\Longrightarrow\surd\mathsf{hd}\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle(\mathsf{E1}::\mathsf{E2})\surd\\ (3::4::(1/0)::\mathsf{Nil})\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ 3::4::(1/0)::\mathsf{Nil}\Longrightarrow\surd(3::4::(1/0)::\mathsf{Nil})\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{hd}\\ \mathsf{hd}(3::4::(1/0)::\mathsf{Nil})\Longrightarrow 3\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle\mathsf{num}\surd\\ 3\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ 3\Longrightarrow\surd 3\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Step}\\ \mathsf{hd}(3::4::(1/0)::\mathsf{Nil})\Longrightarrow\surd 3\end{array}}}{\begin{array}{c}\scriptstyle +1\\ (((+)))(\mathsf{hd}(3::4::(1/0)::\mathsf{Nil}))\Longrightarrow(((+)))3\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle((+)\mathsf{num})\surd\\ (((+)))3\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ (((+)))3\Longrightarrow\surd(((+)))3\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Step}\\ (((+)))(\mathsf{hd}(3::4::(1/0)::\mathsf{Nil}))\Longrightarrow\surd(((+)))3\end{array}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\scriptstyle\mathsf{hd}\surd\\ \mathsf{hd}\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ \mathsf{hd}\Longrightarrow\surd\mathsf{hd}\end{array}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\scriptstyle\mathsf{tl}\surd\\ \mathsf{tl}\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ \mathsf{tl}\Longrightarrow\surd\mathsf{tl}\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle(\mathsf{E1}::\mathsf{E2})\surd\\ (3::4::(1/0)::\mathsf{Nil})\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ 3::4::(1/0)::\mathsf{Nil}\Longrightarrow\surd(3::4::(1/0)::\mathsf{Nil})\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{tl}\\ \mathsf{tl}(3::4::(1/0)::\mathsf{Nil})\Longrightarrow 4::(1/0)::\mathsf{Nil}\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle(\mathsf{E1}::\mathsf{E2})\surd\\ (4::(1/0)::\mathsf{Nil})\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ 4::(1/0)::\mathsf{Nil}\Longrightarrow\surd 4::(1/0)::\mathsf{Nil}\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Step}\\ \mathsf{tl}(3::4::(1/0)::\mathsf{Nil})\Longrightarrow\surd(4::(1/0)::\mathsf{Nil})\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{hd}\\ \mathsf{hd}(\mathsf{tl}(3::4::(1/0)::\mathsf{Nil}))\Longrightarrow 4\end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c}\scriptstyle\mathsf{num}\surd\\ 4\surd\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Normal}\\ 4\Longrightarrow\surd 4\end{array}}}{\begin{array}{c}\scriptstyle\mathsf{Step}\\ (\mathsf{hd}(\mathsf{tl}(3::4::(1/0)::\mathsf{Nil})))\Longrightarrow\surd 4\end{array}}$$

Example computation #5 ($\overset{\surd}{\Longrightarrow}$ written as $\Longrightarrow\surd$)

$$
\dfrac{
\dfrac{
\dfrac{}{\overset{\lambda\surd}{(\lambda \mathsf{x}\bullet(\mathsf{hd\ x})+(\mathsf{hd(tl\ x)}))\surd}}
}{
\overset{\mathsf{Normal}}{\lambda \mathsf{x}\bullet(\mathsf{hd\ x})+(\mathsf{hd(tl\ x)})\Longrightarrow\surd\lambda \mathsf{x}\bullet(\mathsf{hd\ x})+(\mathsf{hd(tl\ x)})}
}
\qquad
\dfrac{
\dfrac{}{\overset{\mathsf{Computation\ \#3}}{((+))(\mathsf{hd}(3::4::(1/0)::\mathsf{Nil}))\Longrightarrow\surd((+))3}}
\quad
\dfrac{}{\overset{\mathsf{Computation\ \#4}}{\mathsf{hd}(\mathsf{tl}(3::4::(1/0)::\mathsf{Nil}))\Longrightarrow\surd 4}}
}{
\overset{+2}{(\mathsf{hd}(3::4::(1/0)::\mathsf{Nil}))+(\mathsf{hd}(\mathsf{tl}(3::4::(1/0)::\mathsf{Nil})))\Longrightarrow 7}
}
}{
\dfrac{
\overset{\mathsf{Beta}}{(\lambda \mathsf{x}\bullet(\mathsf{hd\ x})+(\mathsf{hd(tl\ x)}))(3::4::(1/0)::\mathsf{Nil})\Longrightarrow 7}
}{
\dfrac{
\overset{\mathsf{Let}}{\mathsf{let\ f}=(\lambda \mathsf{x}\bullet(\mathsf{hd\ x})+(\mathsf{hd(tl\ x)}))\mathsf{in\ f}(3::4::(1/0)::\mathsf{Nil})\mathsf{end}\Longrightarrow 7}
}{
\overset{\mathsf{Let}}{\mathsf{let\ l}=3::4::(1/0)::\mathsf{Nil\ in\ let\ f\ x}\ =\ (\mathsf{hd\ x})+(\mathsf{hd(tl\ x)}))\mathsf{in\ f\ l\ end\ end}\Longrightarrow 7}
}
}
}
$$

Programs can get stuck for a variety of reasons

$$\dfrac{\dfrac{\dfrac{}{\text{hd}\ \surd} \qquad \dfrac{\dfrac{}{\text{snd}\ \surd} \quad \dfrac{}{(3, Nil)\surd}}{\text{snd}\ (3, Nil) \Longrightarrow Nil}\,\text{snd}}{\text{hd}\ (\text{snd}\ (3, Nil)) \Longrightarrow \boxed{?}}\,\boxed{\text{Rule?}}}{(\mathbf{let}\ x = (3, Nil)\ \mathbf{in}\ \text{hd}\ (\text{snd}\ x)\ \mathbf{end}) \Longrightarrow \boxed{?}}\,\beta$$

- ➪ There is no rule for hd Nil
- ➪ So no progress can be made at $\boxed{?}$

$$\dfrac{\dfrac{\dfrac{}{\text{fst}\ \surd} \quad \dfrac{\dfrac{}{\text{fst}\ \surd} \quad \dfrac{}{(3, Nil)\surd}}{\text{fst}\ (3, Nil) \Longrightarrow 3}\,\text{fst}}{\text{fst}\ (\text{fst}\ (3, Nil)) \Longrightarrow \boxed{?}}\,\boxed{\text{Rule?}}}{(\mathbf{let}\ x = (3, Nil)\ \mathbf{in}\ \text{fst}\ (\text{fst}\ x)\ \mathbf{end}) \Longrightarrow \boxed{?}}\,\beta$$

- ➩ There is no rule for fst (3)
- ➩ So no progress can be made at $\boxed{?}$

Q: Why is there no rule for simple variables?

A: Because none is needed!

(Since variables are “substituted out of bodies” by application and declaration steps)

Corollary:

- ⇨ A well-formed program need have no free variables.
- ⇨ Substitution can be naive rather than safe.

  (Since variable capture is not possible in a well-formed program)

➥

Declarations can be composed and modified

For example, the declaration

```
    rec  f n    = if  n=0  then  1  else  g n (n-1)
    and  g n n' = n × f n'
```

has tree structure

```
RecDec
   AndDec
      ValDec ...
      ValDec ...
```

- ➩ Two simple declarations composed to be “simultaneous”
- ➩ Outcome is modified to be “recursive”
- ➩ **rec** and **and** are “declaration compositions”.
- ➩ other compositions are *e.g.* **where** ...

To capture this formally we need a new form of judgement

$$\Delta \overset{\Delta}{\Longrightarrow} \sigma$$

- Δ is a declaration
- σ is a “substitution” (mapping from identifiers to expressions)
  - Substitutions are (here) written in the form $[\![\overline{v} := \overline{E}]\!]$
  - Application of a substitution σ to an expression E is written σE

➥

- We use the usual operations to combine mappings, *e.g.*:
  - $\sigma_1 \cup \sigma_2$ is the union of mappings with disjoint domains

$$[\![x, y := U, V]\!] \cup [\![p, q := R, S]\!] = [\![x, y, p, q := U, V, R, S]\!]$$

  - $\sigma_1 \oplus \sigma_2$ is the overriding of σ_1 by σ_2

$$[\![x, y := U, V]\!] \oplus [\![x, q := R, S]\!] = [\![x, y, q := R, V, S]\!]$$

  - $\sigma_2 \cdot \sigma_1$ is the composition of σ_1 with σ_2

$$[\![x, y := U, V]\!] \cdot [\![a, b := R(x), S(y)]\!] = [\![a, b := R(U), S(V)]\!]$$

Local Variables

$$\frac{}{x = A \overset{\Delta}{\Longrightarrow} [\![x := A]\!]}\ \text{Simple}$$

$$\frac{\begin{array}{c}\Delta_1 \overset{\Delta}{\Longrightarrow} \sigma_1 \\ \ldots \\ \Delta_n \overset{\Delta}{\Longrightarrow} \sigma_n\end{array}}{\Delta_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ \Delta_n \overset{\Delta}{\Longrightarrow} \sigma_1 \cup \ldots \cup \sigma_n}\ \mathbf{and}$$

Revised **let**

$$\frac{\begin{array}{c}\Delta \overset{\Delta}{\Longrightarrow} \sigma \\ \sigma B \Longrightarrow R\end{array}}{\mathbf{let}\ \Delta\ \mathbf{in}\ B \Longrightarrow R}\ \mathbf{let}$$

?

Example computation #6

$$\dfrac{f = \mathsf{fst}\ \ \mathbf{and}\ s = \mathsf{snd}\ \overset{\Delta}{\Longrightarrow} [\![f, s := \mathsf{fst}\ , \mathsf{snd}\ ]\!] \qquad \dfrac{\nabla_6}{(\mathsf{fst}\ (\mathsf{snd}\ (1, 2, 3))) \Longrightarrow 2}\ \mathsf{fst}}{\Big(\mathbf{let}\ f = \mathsf{fst}\ \ \mathbf{and}\ s = \mathsf{snd}\ \ \mathbf{in}\ f(s(1, 2, 3))\ \mathbf{end}\Big) \Longrightarrow 2}\ \mathsf{let}$$

Example computation #7

$$\dfrac{f = \mathsf{fst}\ \ \mathbf{and}\ s = \mathsf{snd}\ \overset{\Delta}{\Longrightarrow} [\![f, s := \mathsf{fst}\ , \mathsf{snd}\ ]\!] \qquad \dfrac{\dfrac{\nabla_7}{\mathsf{snd}\ (\mathsf{snd}\ (1, (2, 3))) \Longrightarrow 3}\ \mathsf{snd}}{(\lambda\ f \cdot f(\mathsf{snd}\ (1, (2, 3))))(\mathsf{snd}\ ) \Longrightarrow 3}\ \beta}{\Big(\mathbf{let}\ f = \mathsf{fst}\ \ \mathbf{and}\ s = \mathsf{snd}\ \ \mathbf{in}\ (\lambda\ f \cdot f(s(1, (2, 3))))(\mathsf{snd}\ )\ \mathbf{end}\Big) \Longrightarrow 3}\ \mathsf{let}$$

We “unroll” the substitution that results from a recursive declaration.

$$\frac{\Delta \overset{\Delta}{\Longrightarrow} \sigma}{\mathbf{rec}\ \Delta \overset{\Delta}{\Longrightarrow} unroll(\sigma)}\ \textsf{rec}$$

where $unroll$ is *specified* by

$$unroll[\![\overline{v} := \overline{E}]\!] = [\![\overline{v} := (unroll[\![\overline{v} := \overline{E}]\!])\overline{E}]\!]$$

For example, if σ is $[\![f, g := F(f, g), G(f, g)]\!]$

$$\begin{array}{cl} & (unroll(\sigma)) \\ = & [\![f, g := unroll(\sigma)(F(f, g), G(f, g))]\!] \\ = & [\![f, g := (unroll(\sigma)(F(f, g)), unroll(\sigma)(G(f, g)))]\!] \\ = & [\![f, g := \left(\begin{array}{l} (F(unroll(\sigma)(F(f, g)), unroll(\sigma)(G(f, g)))), \\ (G(unroll(\sigma)(F(f, g)), unroll(\sigma)(G(f, g)))) \end{array} \right)]\!] \\ = & [\![f, g := (F(F(...), G(...)), G(F(...), G(...)))]\!] \\ = & ... \end{array}$$

In practice we can represent an unrolled substitition finitely – despite it containing infinite expressions.

Consider evaluating

$$\textbf{let } f\ n = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times f(n-1) \textbf{ in } f\ 2 \textbf{ end}$$

The **let** , and **rec** rules make this equivalent to evaluating $(unroll(\sigma))(f\ 2)$ with

$$\sigma = [\![f := \left\{ n \to \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times f(n-1) \right\}]\!]$$

Writing the essence of the subsequent computation in the order in which it is done we have

$$\begin{array}{lll}
 & (unroll(\sigma))(f\ 2) & \Longrightarrow \boxed{} \\
\text{(unrolling)} & [\![f := \left\{ n \to \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (unroll(\sigma) f)(n-1) \right\}]\!](f\ 2) & \Longrightarrow \boxed{} \\
\text{(substituting)} & \left\{ n \to \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (unroll(\sigma) f)(n-1) \right\}(2) & \Longrightarrow \boxed{} \\
\text{(by } \beta\text{, etc.)} & \textbf{if } 2 = 0 \textbf{ then } 1 \textbf{ else } 2 \times (unroll(\sigma) f)(2-1) & \Longrightarrow \boxed{} \\
\text{(by arithmetic } \textbf{if}_F\text{, etc.)} & 2 \times (unroll(\sigma) f)(2-1) & \Longrightarrow \boxed{} \\
\text{(...)} & 2 \times 1 \times (unroll(\sigma) f)(2-1-1) & \Longrightarrow \boxed{} \\
\text{(...)} & 2 \times 1 \times (\textbf{if } 2-1-1 = 0 \textbf{ then } 1 \textbf{ else } 2 \times (unroll(\sigma) f)(2-1-1)) & \Longrightarrow \boxed{} \\
\text{(by arithmetic, } \textbf{if}_T\text{, etc.)} & 2 \times 1 \times 1 & \Longrightarrow \boxed{}
\end{array}$$

The substitution gets unrolled a layer at a time – when identifiers in its domain are needed.

?

Interactive evaluation

- ➪ We need to explain what a top-level declaration does in the PicoML environment.
- ➪ We need to explain the “read-evaluate-print” behaviour of the PicoML environment.
- ➪ If an expression yields a constant (say a number) then we just want to print it.
- ➪ What happens to structures (cons-cells, pairs, ...)?

Our approach:

- ➪ The state of the top level environment is defined by a substitution.
- ➪ The substitution changes when interactive declarations are made.
- ➪ The substitution is applied to each expression before evaluation.
- ➪ The substitution is applied to each declaration after evaluation (to give meaning to its free variables).

The judgements

$$\begin{array}{lcl} \sigma, E & \overset{\square}{\Longrightarrow} & \sigma', T \\ \sigma, \textbf{let}\ \Delta\ \textbf{end} & \overset{\square}{\Longrightarrow} & \sigma', T \end{array}$$

mean “Evaluation of expression E (declaration Δ) in state σ prints the text T, and changes the state to σ'”.

Interactive evaluation of a declaration extends the current substitution state and prints nothing.

$$\frac{\Delta \quad \Longrightarrow \quad \sigma'}{\sigma, \mathbf{let}\ \Delta\ \mathbf{end} \overset{\square}{\Longrightarrow} \sigma \oplus (\sigma \cdot \sigma'), ''''} \text{Interactive declaration}$$

Judgement $E \overset{\smile}{\Longrightarrow} T$ means “Expression E has printable form T”

Interactive evaluation of an expression shows its printable form

$$\frac{\sigma E \overset{\smile}{\Longrightarrow} T}{\sigma, E \overset{\square}{\Longrightarrow} \sigma, T} \text{Interactive expression}$$

$$\frac{E \overset{\surd}{\Longrightarrow} V \qquad V\surd\surd}{E \overset{\smile}{\Longrightarrow} \text{The printable form of } V} \text{Print}\surd\surd$$

How to print a cons-cell?

- ➪ Print its head and then print its tail.
- ➪ Don't start to *evaluate* the tail until after the head has been printed (in case of infinite lists).
- ➪ "Printing a list causes its elements to be evaluated."

- ➪ This is formalizeable
- ➪ Details are technical and won't add to our understanding – indeed *au contraire*!
- ➪ Informally, we print an expression E as follows

```
Print($E$) =
Reduce $E$ to (weak) normal form $V$
if   $V$ is a constant or abstraction   then
     Output its printed representation
 else
if   $V$ is of the form $E_0 :: E_1$   then
     Print($E_0$);
     Print($E_1$)
 else
     ...
```

Most implementations will "put the punctuation in", *i.e.* print lists in the form $[v_0, v_1, ...]$.

Environment Semantics

- ⇨ Substitution is not necessarily a good *implementation* method.

  (It is computationally expensive, because terms get (re)constructed, so the cost of substitution for a single variable in an expression is proportional to the size of the expression)

- ⇨ In this section we shall develop a style of semantics (environment semantics) that supports a much more efficient implementation.

- ⇨ The semantics depends on having a representation for substitutions which reduces the cost of applying a substitution to an expression.

➥

Representing “expressions-with-substitutions-applied”

In OCAML we can define the recursive types

```
type value       = Val of environment * expr
and  environment = (identifier, value) mapping
```

➩ A *value* is an *expr*ession paired with an *environment*

➩ The *environment* provides enough information for the free variables of the *expr* to be evaluated.

➩ An *environment* is a mapping from identifiers to *value*s

The big ideas (omitting `Val` for conciseness):

➩ If E is represented by (env, F) and E' by (env', F')
then $[\![x := E']\!]E$ is represented by $(env \oplus \left\{ x \mapsto (env', F') \right\}, F)$
(so an environment is analogous to a substitution)

➩ The environment $env_1 \oplus env_2$ is cheap to construct/compute.

➩ (env, E) represents the expression E if E has no free variables mapped by env.

➩ (env, E) represents the expression $\texttt{subst}(env, E)$ otherwise.

➥

The main judgement is

$$env, E \;\rightsquigarrow\; env', E'$$

"If E' in environment env' has a normal form, then E in environment env, has *the same normal form*."

We also define $\overset{\surd}{\rightsquigarrow}$ ("has normal form") by

$$\frac{\text{(E is a normal form.)}}{env, E \;\overset{\surd}{\rightsquigarrow}\; env, E}$$

$$\frac{env, E \;\rightsquigarrow\; env_1, E_1 \qquad env_1, E_1 \;\overset{\surd}{\rightsquigarrow}\; env', E'}{env, E \;\overset{\surd}{\rightsquigarrow}\; env', E'}$$

Start of Theoretical Interlude: Consistency

Environment semantics is consistent with substitution semantics so long as

$$\text{if } \; env, E \overset{\surd}{\rightsquigarrow} \; env', E' \; \text{ then } \; subst(env, E) \overset{\surd}{\Longrightarrow} subst(env', E')$$

For this to be so

- We need to ensure that environments arising during a computation are “hereditarily closed”, *viz:*

$$hclosed(env) \Leftrightarrow \forall i \in \mathbf{dom}\; env \cdot closed(env \;\; i)$$
$$closed(env, E) \Leftrightarrow hclosed(env) \wedge freevariables(E) \subseteq \mathbf{dom}\; env$$

- Informal meaning: “you can’t reach an identifier with no specified value”
- *Should be* true for programs in which no free variable appears.
- We have used this as an informal check on the rules where it matters, but not tried to prove it rigorously, because

we will not attempt to prove consistency

$\texttt{subst} : value \rightarrow expr$ shows what expression a value represents.

$$
\begin{array}{lll}
\texttt{subst}(env,\ \text{“}n\text{”}) & = & \text{“}n\text{”} \\
\texttt{subst}(env,\ \text{“}\textbf{true}\text{”}) & = & \text{“}\textbf{true}\text{”} \\
\texttt{subst}(env,\ \text{“}\textbf{false}\text{”}) & = & \text{“}\textbf{false}\text{”}
\end{array}
$$

... likewise for the other constants and operators ...

$$
\begin{array}{lll}
\texttt{subst}(env,\ \text{“}i\text{”}) & = & \begin{cases} \texttt{subst}(env\ i) & \textbf{if } i \in \textbf{dom } env \\ \text{“}i\text{”} & \textbf{otherwise} \end{cases} \\
\texttt{subst}(env,\ \text{“}(E_1, E_2)\text{”}) & = & \text{“}(\texttt{subst}(env, E_1), \texttt{subst}(env, E_2))\text{”} \\
\texttt{subst}(env,\ \text{“}\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3\text{”}) & = & \begin{array}{ll} \text{“}\textbf{if } (\texttt{subst}(env, E_1)) & \textbf{then } (\texttt{subst}(env, E_2)) \\ & \textbf{else } (\texttt{subst}(env, E_3))\text{”} \end{array} \\
\texttt{subst}(env,\ \text{“}(E_1\ E_2)\text{”}) & = & \text{“}(\texttt{subst}(env, E_1))\ (\texttt{subst}(env, E_2))\text{”} \\
\texttt{subst}(env,\ \text{“}\lambda\, i \cdot E\text{”}) & = & \text{“}\lambda\, i \cdot (\texttt{subst}(env \ominus \left\{\ i\ \right\}, E))\text{”} \\
\texttt{subst}(env,\ \text{“}\textbf{let } \Delta \textbf{ in } E\text{”}) & = & \text{“}\textbf{let } (\texttt{dsubst}(env, \Delta)) \textbf{ in } (\texttt{subst}(env \ominus(\texttt{defined}\ \Delta), E))\text{”})
\end{array}
$$

Example: suppose $env = \left\{\ x \mapsto \left(\ \left\{\ i \mapsto \left(\ \left\{\ \ \right\}, \text{“}y \times y\text{”}\ \right)\ \right\}, \text{“}i \times j\text{”}\ \right)\ \right\}$, then

$$
\begin{array}{ll}
 & \texttt{subst}(env, \text{“}x + 1\text{”}) \\
= & \text{“}\ \texttt{subst}(env, \text{“}x\text{”}) + 1\text{”} \\
= & \text{“}\ \texttt{subst}\left(\ \left\{\ i \mapsto \left(\ \left\{\ \ \right\}, \text{“}y \times y\text{”}\ \right)\ \right\}, \text{“}i \times j\text{”}\ \right) + 1\text{”} \\
= & \text{“}\ \texttt{subst}\left(\ \left\{\ i \mapsto \left(\ \left\{\ \ \right\}, \text{“}y \times y\text{”}\ \right)\ \right\}, \text{“}i\text{”}\ \right) \times \texttt{subst}\left(\ \left\{\ i \mapsto \left(\ \left\{\ \ \right\}, \text{“}y \times y\text{”}\ \right)\ \right\}, \text{“}j\text{”}\ \right) + 1\text{”} \\
= & \text{“}(y \times y) \times j + 1\text{”}
\end{array}
$$

End of Theoretical Interlude

Identifier rule: “look it up in the environment”

$$\frac{(i \in \mathbf{dom}\ env)}{env, i \ \leadsto \ env\ i}\ \textsf{Id}$$

Interpret an identifier i in env by interpreting the expression bound to i in the environment it carries with it.

Example:

$$\left\{i \mapsto \left(\left\{j \mapsto (\left\{\right\}, 3)\right\}, j + j\right)\right\}, i \ \leadsto \ \left\{j \mapsto (\left\{\right\}, 3)\right\}, j + j \ \leadsto \ ... \ \leadsto \ \left\{\right\}, 3 + 3$$

Local variables evaluated only when needed

$$\frac{(env \oplus \left\{x \mapsto (env, A)\right\}), B \ \leadsto \ env'', R}{env, \textbf{let } x = A \textbf{ in } B \textbf{ end} \ \leadsto \ env'', R}\ \textbf{let}\ \text{(first approximation)}$$

Compare this with:

$$\frac{[\![x := A]\!]B \Longrightarrow R}{\textbf{let } x = A \textbf{ in } B \Longrightarrow R}\ \textbf{let}\ \text{(first approximation)}$$

- For both the **let** and the Apply rules, the environment in which the body is evaluated is extended. This “simulates” the substitution.
- Observation: if all free variables of B other than x are bound in env, then all free variables in B are bound in $(env \oplus \left\{\ x \mapsto (env, A)\ \right\})$
- Corollary: if env is “hereditarily” closed, then so is $(env \oplus \left\{\ x \mapsto (env, A)\ \right\})$

Parameters are evaluated only when needed.

$$\frac{env, F \overset{\surd}{\leadsto} env', (\lambda\ x{\cdot}B) \qquad (env' \oplus \left\{x \mapsto (env, A)\right\}), B \leadsto env'', R}{env, (F\ A) \leadsto env'', R}\ \text{(apply)}$$

- ⇨ Instead of substituting the Arg for the bound variable in the body, we extend the environment in which the body is evaluated.
- ⇨ In the extended environment, A carries with it the environment in which its variables are to be interpreted.

Compare this with:

$$\frac{F \Longrightarrow (\lambda\ x{\cdot}B) \qquad [\![x := A]\!]B \Longrightarrow R}{F\ A \Longrightarrow R}\ \text{(apply)}$$

Constants, arithmetic, abstractions ...

$$\dfrac{(n\ \text{is a number})}{env, n \rightsquigarrow \{\}, n}\ n\surd$$

$$\dfrac{}{env, (+) \rightsquigarrow \{\}, (+)}\ (+)\surd$$

$$\dfrac{env, E_1 \overset{\surd}{\rightsquigarrow} env', n_1}{env, (+)E_1 \rightsquigarrow \{\}, (+)n_1}\ +_1$$

$$\dfrac{env, E_2 \overset{\surd}{\rightsquigarrow} env', n_2}{env, (+)n_1E_2 \rightsquigarrow \{\}, \overline{n_1 + n_2}}\ +_2$$

$$\dfrac{}{env, \lambda\ x \cdot B \rightsquigarrow env, \lambda\ x \cdot B}$$

Conditional expressions

$$\frac{env, E_1 \overset{\surd}{\rightsquigarrow} env', \mathbf{true}}{env, \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \rightsquigarrow env, E_2}\ \mathbf{if}_{\ \mathbf{true}}$$

$$\frac{env, E_1 \overset{\surd}{\rightsquigarrow} env', \mathbf{false}}{env, \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \rightsquigarrow env, E_3}\ \mathbf{if}_{\ \mathbf{false}}$$

- ➪ env' is *irrelevant* in these rules
- ➪ env' is *irrelevant* in rules $+_1, +_2$
- ➪ This suggests an optimisation in the implementation
  (represent constants without an environment component)

Lexical (static) binding illustrated

$$\{\ \} \quad , \quad \left\langle \begin{array}{l}\textbf{let}\ y = 20\ \textbf{in}\\ \textbf{let}\ axx = \lambda\ x \cdot x + y\ \textbf{in}\\ \textbf{let}\ y = 2\ \textbf{in}\ axx\ y\ \textbf{end end end}\end{array}\right\rangle$$

(by let) $\leadsto$

$$\{\ y \mapsto (\{\ \}, 20)\ \} \quad , \quad \left\langle \begin{array}{l}\textbf{let}\ axx = \lambda\ x \cdot x + y\ \textbf{in}\\ \textbf{let}\ y = 2\ \textbf{in}\ axx\ y\ \textbf{end end}\end{array}\right\rangle$$

(by let) $\leadsto$

$$\left\{ \begin{array}{ll} y & \mapsto \left(\{\ \}, 20 \right),\\ axx & \mapsto \left(\{\ y \mapsto (\{\ \}, 20)\ \}, \lambda\ x \cdot x + y \right) \end{array} \right\} \quad , \quad \left\langle\ \textbf{let}\ y = 2\ \textbf{in}\ axx\ y\ \textbf{end}\ \right\rangle$$

(by let) $\leadsto$

$$\begin{array}{l} \left\{ \begin{array}{ll} y & \mapsto \left(\{\ \}, 20 \right),\\ axx & \mapsto \left(\{\ y \mapsto (\{\ \}, 20)\ \}, \lambda\ x \cdot x + y \right) \end{array} \right\}\\ \oplus\\ \left\{\ y \ \mapsto \left(\{\ \}, 2 \right) \right\} \end{array} \quad , \quad \left\langle\ axx\ y\ \right\rangle$$

(simplifying $\oplus$)

$$\left\{ \begin{array}{ll} y & \mapsto \left(\{\ \}, 2 \right),\\ axx & \mapsto \left(\{\ y \mapsto (\{\ \}, 20)\ \}, \lambda\ x \cdot x + y \right) \end{array} \right\} \quad , \quad \left\langle\ axx\ y\ \right\rangle$$

(by apply) $\leadsto$

$$\underbrace{\{\ y \mapsto (\{\ \}, 20)\ \}}_{env'} \oplus \left\{\ x \mapsto \left(\{\ \}, 2 \right) \right\} \quad , \quad \left\langle\ x + y\ \right\rangle$$

(by $+_1, +_2$) $\leadsto$

$$\{\ \} \quad , \quad \left\langle\ 22\ \right\rangle$$

?

Pair construction is lazy

$$\frac{env, F \overset{\surd}{\leadsto} env'_F, \mathsf{fst} \qquad env, E \overset{\surd}{\leadsto} env'_E, (E_1, E_2)}{env, F\ E \leadsto env'_E, E_1}\ \mathsf{fst}$$

$$\frac{env, F \overset{\surd}{\leadsto} env'_F, \mathsf{snd} \qquad env, E \overset{\surd}{\leadsto} env'_E, (E_1, E_2)}{env, F\ E \leadsto env'_E, E_2}\ \mathsf{snd}$$

So is list construction (see exercise 6).

?

Declaration judgement form: “declaration Δ defines *extension* env' to environment env”

$$env, \Delta \overset{\Delta}{\leadsto} env'$$

How is the extension used?

$$\frac{env, \Delta \overset{\Delta}{\leadsto} env' \qquad (env \oplus env'), B \leadsto env'', R}{env, \textbf{let}\ \Delta\ \textbf{in}\ B\ \textbf{end} \leadsto env'', R}\ \textbf{let}$$

Declaration rules straightforward!

$$\frac{}{env, x = A \overset{\Delta}{\leadsto} \left\{x \mapsto (env, A)\right\}}\ \text{Simple}$$

$$\frac{\begin{array}{c} env, \Delta_1 \overset{\Delta}{\leadsto} env_1 \\ \ldots \\ env, \Delta_n \overset{\Delta}{\leadsto} env_n \end{array}}{env, \Delta_1\ \textbf{and}\ \ldots\ \textbf{and}\ \Delta_n \overset{\Delta}{\leadsto} env_1 \cup \ldots \cup env_n}\ \textbf{and}\ \ (env_1 \ldots env_n \text{ have disjoint domains})$$

?

Note that actual parameters are evaluated once per (strict) use of the formal parameter

$$\begin{array}{ll}
\{\ \} & ,\ \textbf{let}\ \mathit{double}(x) = x + x\ \textbf{in}\ \mathit{double}(3 \times 4)\ \textbf{end} \\
\{ \mathit{double} \mapsto (\{\ \}, \lambda\, x{\cdot}x + x) \} & ,\ \mathit{double}(3 \times 4) \\
\{ x \mapsto (\{\ \}, 3 \times 4) \} & ,\ x + x
\end{array}$$

$$\begin{array}{ll}
\hline
\{ x \mapsto (\{\ \}, 3 \times 4) \} & ,\ x \\
\{\ \} & ,\ 3 \times 4 \\
\{\ \} & ,\ 12 \\
\hline
\{ x \mapsto (\{\ \}, 3 \times 4) \} & ,\ x \\
\{\ \} & ,\ 3 \times 4 \\
\{\ \} & ,\ 12 \\
\hline
\end{array}$$

$$\begin{array}{ll}
\{\ \} & ,\ 24
\end{array}$$

➩ This inefficiency can be avoided (by sharing the results of evaluations)

➩ Our implementation will do this very efficiently

Recursion and top-level evaluation are treated much as before.

From Environment Semantics to Efficient Interpreter

Key design issues

- ➪ Representation of expression-environment pairs
- ➪ Avoidance of repeated parameter evaluation
- ➪ Implementing *unroll* finitely

Our approach

- ➪ Use a new type *value* with concise representations of normal forms
- ➪ Overwrite expression-environment pairs when the expression has been evaluated
- ➪ Represent recursive environments by cyclic data structures

For a conventional (strict) functional language implementation we might define a space of values like this

```
type value =    NumVal     of num
|               BoolVal    of bool
|               PairVal    of value * value
|               FunVal     of environment * identifier * expr
|                ... other kinds of  value ...
and  environment = ... a mapping from identifiers to values ...
```

and an evaluation function recursively over syntax

```
let rec eval: environment -> expr -> value = fun env -> function
|   Id   i           ->  ... look up i in env ...
|   Num  n           -> NumVal(num_of_string n)
|   Pair (e1, e2)    -> PairVal(eval env e1, eval env e2)
|   Fun  (bv, body) -> FunVal(env, bv, body)
|    ... other kinds of expression ...
|   Apply (f, e)     ->
    match eval env f with
    | FunVal    (env', bv, body) -> eval (env' ⊕{ bv ↦ eval env e }) body
    | ...
```

We will take this as our starting point, and progressively reform it

Reform #1: delaying the evaluation of parameters and sharing evaluations between occurences

```
type value =           ... as before ...
|                      Delay      of expr_or_value ref

and  expr_or_value  = Expression of environment * expr
|                      Value      of value
...
```

- A *Delay* expression corresponds to an $(env, expr)$ pair and is in one of two states:
  - $Expression(expr, value)$ – as-yet unevaluated
  - $Value(v)$ – already evaluated
- *Delay*ed expressions are evaluated (“forced”) when their value is needed
- A *Delay* in the *Value*-state never returns to the *Expression*-state

eval implements $\leadsto$

```
let rec eval: environment -> expr -> value = fun env -> function
...
|   Pair (e1, e2) -> PairVal(delay env e1, delay env e2)
...
|   Apply (f, a)  ->
    match stricteval env f with
    | FunVal    (env', bv, body) -> eval (env' ⊕{ bv ↦ delay env e }) body
    | ...
```

delay constructs a delayed expression

```
and delay: environment -> expr -> value = fun env -> fun expr ->
    Delay(ref(Expression(env, expr)))
```

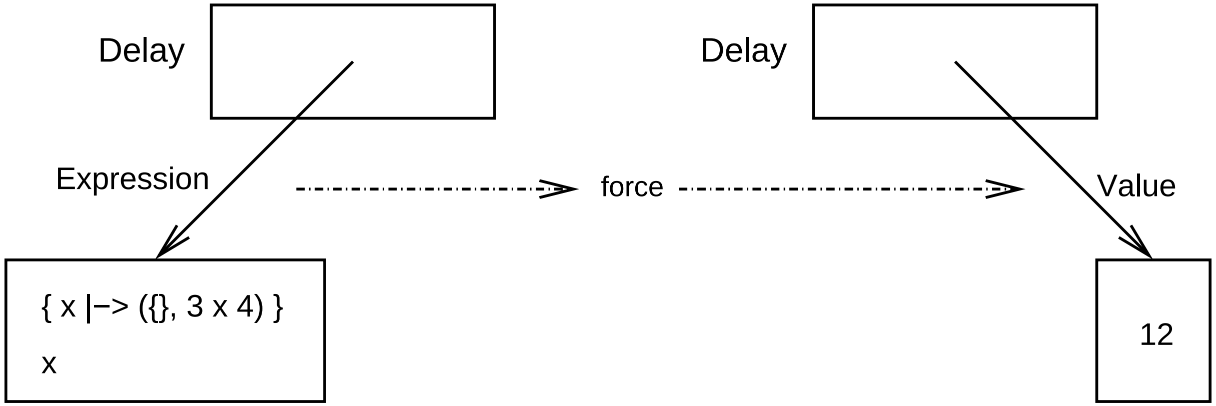
...(*delay env expr*)... in the interpreter corresponds to ...$(env, expr)$... in the semantics

stricteval implements $\overset{\surd}{\leadsto}$

```
and stricteval: environment -> expr -> value =  fun env expr ->
    force(eval env expr)
```

force forces evaluation of a delayed expression

```
and force: value -> value =  function
| Delay (expr) -> (match !expr with
                  | Value v -> v
                  | Expression (env’, e) ->
                    let v = stricteval env’ e in  expr := Value v; v)
| other -> other
```



Primitives can be lazy or strict.

```
type value =          ... as before ...
|                     Strict of (value -> value)
|                     Lazy   of (value -> value)
|                     Cons   of (value * value)
|                     Nil
```

The application case of *eval* deals with parameter evaluation

```
let rec eval: environment  -> expr -> value = fun env -> function
...
|   Apply (f, e) ->
( match stricteval env f with
  | FunVal(env', bv, body) -> $eval\ (env' \oplus \left\{ bv \mapsto delay\ env\ e \right\})\ body$
  | Strict     ocamlfun    ->  ocamlfun (stricteval env e)
  | Lazy       ocamlfun    ->  ocamlfun (delay env e)
)
```

Design decision: primitives and constants are introduced via the initial environment.

Some primitives

```
let trueVal  = BoolVal true
let falseVal = BoolVal false
let cons     = Lazy   (fun x -> Lazy (fun xs -> Cons(x, xs)))
let null     = Strict (function Nil-> trueVal | _ -> falseVal)
let fst      = Strict (function PairVal(a,  b) -> a)
```

Part of the initial environment

```
{ "::"    ↦ cons,
  "null"  ↦ null,
  "fst"   ↦ fst,
  "True"  ↦ trueVal,
  "False" ↦ falseVal,
  ...
}
```

dval implements $\overset{\Delta}{\rightsquigarrow}$

```
and dval: environment  -> decl -> environment = fun env -> function
|   ValDec   (id, e)  -> { id  ↦  delay env e }
|   AndDec   (d1, d2) -> dval env d1 ⊕ dval env d2
|   ...
|   RecDec d ->  to be discussed
```

```
let rec eval: environment  -> expr -> value = fun env -> function
...
|   Let (dec, body) ->
    let env' = dval env dec in eval (env ⊕ env') body
```

Recall the specification of *unroll*

$$unroll[\![\overline{v} := \overline{E}]\!] = [\![\overline{v} := (unroll[\![\overline{v} := \overline{E}]\!])\overline{E}]\!]$$

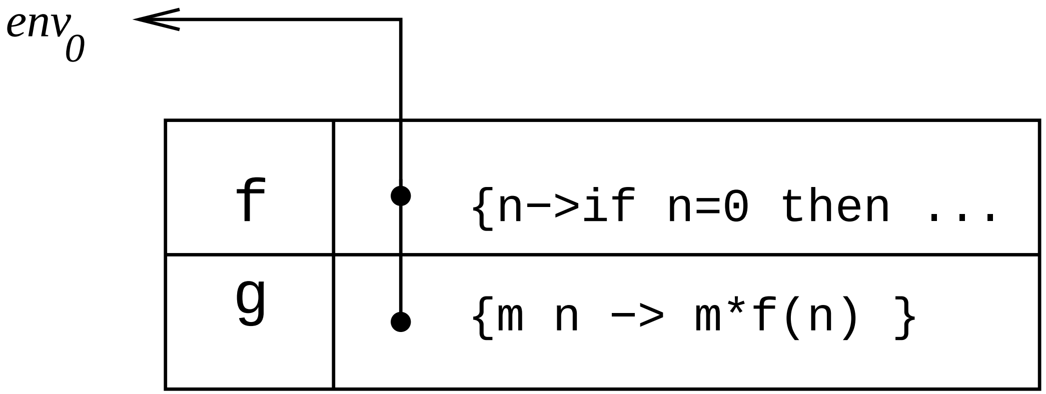
Rephrased in terms of environments, we require

$$\begin{array}{l} \forall i \in \mathbf{dom}\ env \cdot \\ \quad \exists E, env_0 \cdot \\ \quad \left(\begin{array}{l} (unroll\ \ env)i = (unroll\ \ env, E) \\ \wedge \\ env\ \ i = (env_0, E) \end{array} \right) \end{array}$$

The unrolled environment

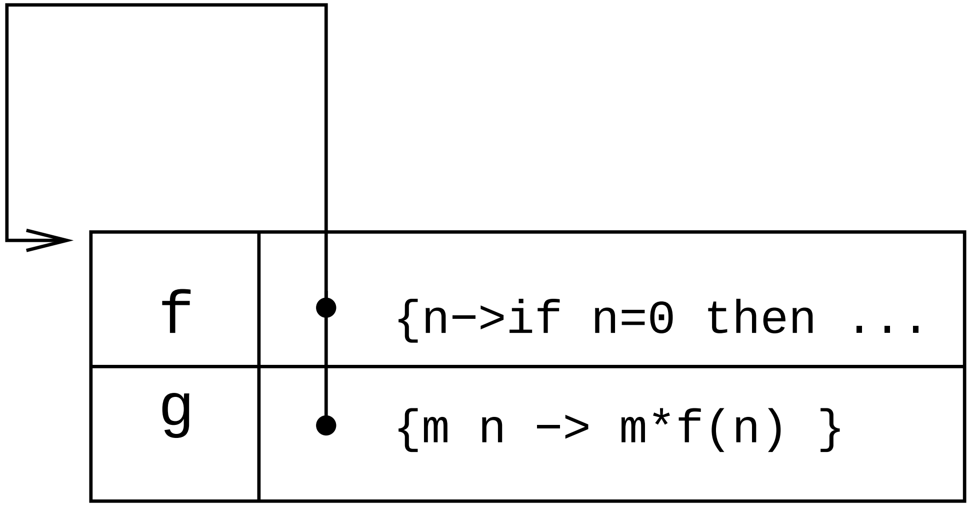
- ➩ Maps the same identifiers
- ➩ ... to the same expressions
- ➩ ... and to the unrolled environment itself

Unrolling this environment extension



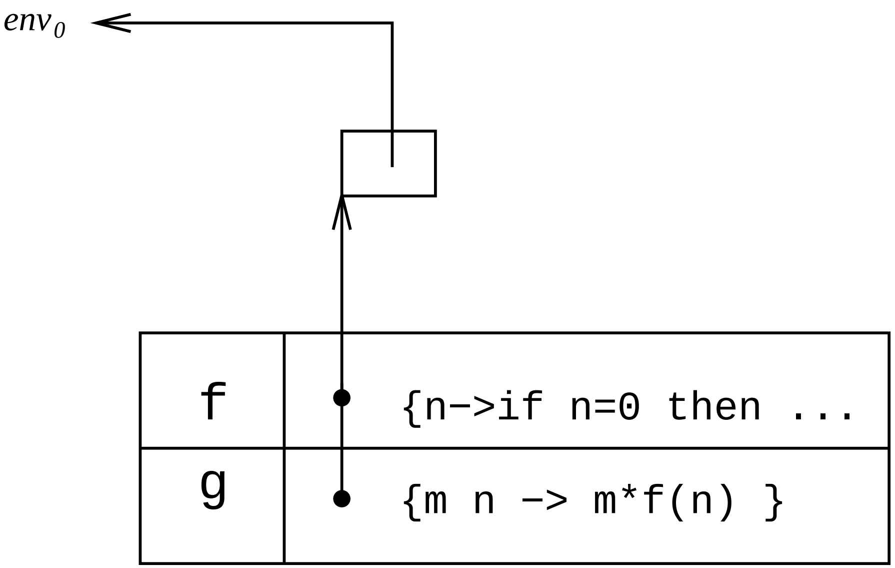
f	{n->if n=0 then ...
g	{m n -> m*f(n) }

Yields this environment extension

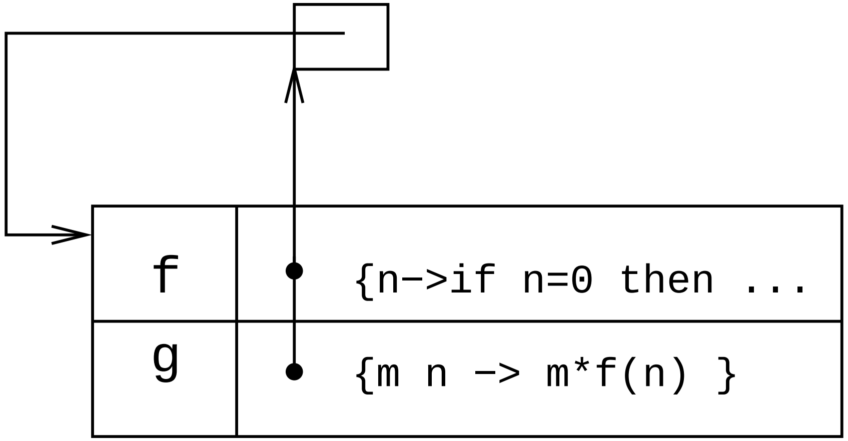


f	{n->if n=0 then ...
g	{m n -> m*f(n) }

This can be implemented in stages if environments are references to mappings



"Tying the knot"



We now reveal the definition of `environment`

```
type value       = ...
and  environment = value map ref
```

and the implementation of knot-tying

```
and dval: value map ref -> decl -> value map =
fun env -> function
|   RecDec d ->
    let env' = ref(!env)                     (* Make a new reference to the current value map
    in  let ext  = dval env' d               (* Build the extension using this reference in closures&thunks
        in  env' := !env |+| ext;            (* Incorporate the extension and tie the knot
            ext
|   ValDec   (id, e)  -> id |-> delay env e
|   AndDec   (d1, d2) -> dval env d1 |+| dval env d2
...
```

(extract from the signature of the `Mapping` module)

```
type 'a map                              (* m: 'a map  represents an identifier to 'a mapping  m
emptymap : 'a map                        (*            implements  { }
(|?|)  : 'a map -> string -> 'a          (* m |?|s     implements  m s
(|+|)  : 'a map -> 'a map -> 'a map      (* m1|+|m2    implements  m1 ⊕ m2
(|++|) : 'a map -> 'a map -> 'a map      (* m1|++|m2   implements  m1 ∪ m2
(|->)  : string -> 'a -> 'a map          (* s |-> a    implements  { s ↦ a }
```

Taking all this into account

```
let rec eval: environment -> expr -> value = fun env -> function
|   Id     i           -> !env |?| i
|   Num    n           -> NumVal(num_of_string n)
|   Char   s           -> CharVal s
|   Pair   (e1, e2)    -> PairVal(delay env e1, delay env e2)
|   Fun    (bv, body)  -> FunVal(env, bv, body)
|   Let    (dec, body) -> let env' = dval env dec in eval (env<+>env') body
|   If     (b, e1, e2) ->
    let e = match stricteval env b with
    | BoolVal true  -> e1
    | BoolVal false -> e2
    | other         -> failwith ("type error in condition: "^estring b)
    in
      eval env e
|   Apply (f, e) ->
    ( match stricteval env f with
      | FunVal(env', bv, body) -> eval (env' <+> (bv|->delay env e)) body
      | Strict     ocamlfun    -> ocamlfun (stricteval env e)
      | Lazy       ocamlfun    -> ocamlfun (delay env e)
      | other -> failwith ("type error in application: "^estring f)
    )
|   ...

and env <+> map = ref(!env |+| map)
```

- Printing from the interactive environment forces evaluation
- Cons cells, *etc.* are printed “lazily”
- Guarantee output of the head (first, ...) before the tail is evaluated

First approximation

```
let rec print_val: expr -> unit = function
| NumVal  i     -> print_string (string_of_num i)
| CharVal c     -> print_string ("'"^c^"'")
| BoolVal true  -> print_string "True"
| BoolVal false -> print_string "False"
| Nil           -> print_string "Nil"
| Cons (h, t)   -> print_string "[";
                   print_flush();                (* Show now that we know it’s a list
                   print_val(force h);           (* Evaluate and start printing the head
                   print_flush();                (* Ensure the user sees what has been printed so far
                   print_list_tail (force t);    (* Evaluate and start printing the tail
                   print_string "]";
                   print_flush();

| ...
```

Types and Typechecking

- Pragmatic Description – the primitive operations provided by most programming languages are *partial* with respect to the whole set of values that can be manipulated.

  This universe can usually be divided into types in a systematic way that can be useful in helping detect whether the program has failed or will fail.

Examples

- In $Pascal/Oberon/ML/Haskell$ the + operator requires both its operands to be numbers *of the same kind*.
- In C the + operator requires both its operands to be numbers.
- In Java a field selection operator $.foo$ may only be applied to an object of a type which has a field of name foo.
- In $BCPL$ – the “apply” operator has a left operand which must be a procedure of some kind, and a right operand which must be a parameter list.

Approaches to typing

- ➩ Conceptual typing: in assembly languages, whether low level like the ones we're used to, or "high level" like BCPL and Forth, every value is represented by a machine word. The abstract value such a word represents is kept in mind by readers and writers of the program. The language and its processors provide (little or) no support for doing so, and programmers are assumed to know what they're doing when invoking an operation.

- ➩ Dynamic typing: in many interpreted languages (Lisp, Scheme, Snobol, Perl, Python, OWHY) every value is represented by a composite structure which contains an encoding of its type. Primitive operations check that the type(s) of their operand(s) are appropriate before doing their work – usually aborting if not.

  Some others (Shell scripting languages, Tcl) represent every value by a string, and operators attempt to make sense of the strings which are provided as operands. ➥

- ➩ Static typing: set up the language so that *without running the program* the type of every expression in a program can be determined. When this is so, there's less (or no) need to check operand types for validity *at run time*. ➥

  The *leitmotif* of static typing is: "a type-correct program gives no nasty surprises".

PicoML is statically typed

- We give every expressible *value* in the language a type.
- We give every well-formed expression in the language a type
- The type-classification of expressions is explained by inference rules.
- The classification, and rules are such that (in any given context)
  - If we can deduce a type for an expression using the rules, then *the execution of that expression will not cause a run-time type error.*

    ```
    % picoml -x                     # turn off typechecking
    : "foo"-33;;
    Runtime error: wrong operand(s) for -
    : (3,4) 5;;
    Runtime error: type error in application: (3,4)
    ```

  - If we can deduce a type for an expression using the rules, then *the execution of that expression, if it terminates, leads to a value of the deduced type.*

Types for Values

The types of primitive values are obvious (as they are for some composites, and primitive functions)

Value		Type		OCAML Value	OCAML Type
$()$	:	$()$		$UnitVal$	$UnitType$
n	:	Num		$NumVal\ n$	$NumType$
true	:	Bool		$BoolVal\ True$	$BoolType$
false	:	Bool		$BoolVal\ True$	$BoolType$
$'c'$	:	Char		$CharVal$	$CharType$
(v_1, v_2)	:	(τ_1, τ_2)	**if** $v_i : \tau_i$	$PairVal(v_1, v_2)$	$PairType(\tau_1, \tau_2)$
$\left[\ v_1, ..., v_n\ \right]$	:	$[\tau]$	**if** $v_i : \tau$	$Cons(v_1, ...)$	$ListType$
$(+)$	:	Num → Num → Num			

But what about (for example)

$$\lambda\, x \cdot \lambda\, y \cdot (x, y)$$
[]
hd
fst

Intuition about a type for fst (for example) is based on the fact that (for any expressions E_1, E_2)

$$\text{fst } (E_1, E_2) \Longrightarrow E_1$$

One way of thinking about fst is that it is actually a family of function values indexed by argument type, and that

$$\begin{array}{lll} \text{fst } \left(3, 5 \right) & \text{means} & \text{fst }_{\text{(Num,Num)}}\left(3, 5 \right) \\ \text{fst } \left('\text{c}', 5 \right) & \text{means} & \text{fst }_{\text{(Char,Num)}}\left('\text{c}', 5 \right) \end{array}$$

Given this way of thinking we might attribute a type to each member of the family

$$\begin{array}{l} \forall \tau_1, \tau_2 \cdot \\ \quad \text{fst }_{(\tau_1,\tau_2)} : \left(\tau_1, \tau_2 \right) \rightarrow \tau_1 \end{array}$$

Of course having to write explicit type-subscripts in PicoML itself would be very inconvenient, but we shall shortly show how this is avoided.

Type rules use the following judgements:

$$env \ \vdash \ e : T$$

"Expression e has type T in type-environment env."

$$env \ \vdash \ d \ \overset{\Delta}{:} \ \ env'$$

"Declaration d yields environment extension env' in environment env."

where
env, env' are finite mappings from $identifier$ to $typexpr$
T is a $typexpr$
d is a $decl$
e is a $expr$

Some type inference rules

$$\frac{(id \in \mathbf{dom}\ env)}{env \vdash\ id\ : env\ id}\ \text{Id (\textit{first approximation})}$$

$$\frac{}{\begin{array}{lll} env \vdash & \mathbf{true} & : \text{Bool} \\ env \vdash & \mathbf{false} & : \text{Bool} \\ env \vdash & n & : \text{Num} \\ env \vdash & () & : () \end{array}}\ \text{Constant}$$

$$\frac{\begin{array}{lll} env \vdash & e_1 & : T_1 \\ env \vdash & e_2 & : T_2 \end{array}}{env \vdash\ (e_1, e_2) : (T_1, T_2)}\ \text{Pair}$$

$$\frac{\begin{array}{lll} env \vdash & f & : T_a \rightarrow T_r \\ env \vdash & a & : T_a \end{array}}{env \vdash\ f\ a\ : T_r}\ \text{Apply}$$

$$\frac{env \oplus \left\{x \mapsto T_a\right\} \vdash\ Body\ : T_r}{env\ \vdash\ \lambda\ x{\cdot}Body : T_a \rightarrow T_r}\ \text{Fun}$$

?

Declaration rules are straightforward at first glance

$$\frac{\begin{array}{ll} \mathit{env} & \vdash\ \Delta \overset{\Delta}{:}\ \mathit{env}' \\ \mathit{env} \oplus \mathit{env}' & \vdash\ \mathit{Body}\ :\ T \end{array}}{\begin{array}{ll} \mathit{env} & \vdash\ \textbf{let}\ \Delta\ \textbf{in}\ \mathit{Body}\ \textbf{end}\ :\ T \end{array}}\ \text{Let (\textit{first approximation})}$$

$$\frac{\begin{array}{l} \mathit{env} \vdash\ \Delta_1 \overset{\Delta}{:}\ \mathit{env}_1 \\ \quad \cdots \\ \mathit{env} \vdash\ \Delta_n \overset{\Delta}{:}\ \mathit{env}_n \\ \quad (\mathit{env}_1 ... \mathit{env}_n \text{have disjoint domains}) \end{array}}{\mathit{env} \vdash\ \Delta_1\ \textbf{and}\ \cdots\ \textbf{and}\ \Delta_n \overset{\Delta}{:}\ \mathit{env}_1 \cup \cdots \cup \mathit{env}_n}\ \textbf{and}$$

$$\frac{\mathit{env} \vdash\ i\ :\ T}{\mathit{env} \vdash\ i = E \overset{\Delta}{:} \left\{ i \mapsto T \right\}}\ \text{Simple}$$

?

Recursive declaration rule is deceptively straightforward

$$\frac{\mathit{env} \oplus \mathit{env}' \vdash \Delta \overset{\Delta}{:} \mathit{env}'}{\mathit{env} \vdash \mathbf{rec}\ \Delta \overset{\Delta}{:} \mathit{env}'}\ \mathbf{rec}$$

- ➪ This is a specification, not an algorithm
- ➪ The algorithm must find an *env*′ extension to *env* in which the Δ can be consistently typed
- ➪ The eventual algorithm can deal with the circularity in several ways.

➥

Start of Theoretical Spasm: Soundness

Q: What does it mean for the type rules to be sound?

A: We want them to predict the type of the value of a (terminating) program

More precisely, if

$$\{\ \} \vdash E : T$$

and

$$E \overset{\surd}{\Longrightarrow} E'$$

then

$$\{\ \} \vdash E' : T$$

Such a proof opens a realm of fascinating theory but is beyond the scope of this course.

End of Theoretical Spasm

Completed type inference tree

$$\dfrac{\dfrac{}{\text{Env , Fst|->(Num,Bool)->Num}\quad\text{|-Fst:(Num,Bool)->Num}}\,\text{Id}\qquad\dfrac{\dfrac{}{\text{Env , Fst|->(Num,Bool)->Num}\quad\text{|-3:Num}}\,\text{Constant}\quad\dfrac{}{\text{Env , Fst|->(Num,Bool)->Num}\quad\text{|-True:Bool}}\,\text{Constant}}{\text{Env , Fst|->(Num,Bool)->Num}\quad\text{|-(3,True):(Num,Bool)}}\,\text{Pair}}{\text{Env , Fst|->(Num,Bool)->Num}\quad\text{|-Fst(3,True):Num}}\,\text{Apply}$$

Concise presentation of the same tree

1:	Env , Fst|->(Num,Bool)->Num	Environment
2:	Fst:(Num,Bool)->Num	Id 1.2
3:	3:Num	Constant
4:	True:Bool	Constant
5:	(3,True):(Num,Bool)	Pair 3,4
6:	Fst(3,True):Num	Apply 2,5

$$\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\boxed{\mathrm{Fst}(3,\mathrm{True}):\_T}$$

$$\dfrac{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\boxed{\mathrm{Fst}:\_Ta-\!>\_T}\qquad \mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-(3,\mathrm{True}):\_Ta}{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\mathrm{Fst}(3,\mathrm{True}):\_T}\;\mathrm{Apply}$$

$$\dfrac{\dfrac{}{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\mathrm{Fst}:(\_T1,\_T2)-\!>\_T1}\;\mathrm{Id}\qquad \dfrac{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\boxed{3:\_T1}\qquad \mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\mathrm{True}:\_T2}{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-(3,\mathrm{True}):(\_T1,\_T2)}\;\mathrm{Pair}}{\mathrm{Env, Fst}\,|\!-\!>(\_T1,\_T2)-\!>\_T1\;\;|\!-\mathrm{Fst}(3,\mathrm{True}):\_T1}\;\mathrm{Apply}$$

$$\dfrac{\dfrac{}{\mathrm{Env, Fst}\,|\!-\!>(\mathrm{Num},\_T2)-\!>\mathrm{Num}\;\;|\!-\mathrm{Fst}:(\mathrm{Num},\_T2)-\!>\mathrm{Num}}\;\mathrm{Id}\qquad \dfrac{\dfrac{}{\mathrm{Env, Fst}\,|\!-\!>(\mathrm{Num},\_T2)-\!>\mathrm{Num}\;\;|\!-3:\mathrm{Num}}\;\mathrm{Constant}\qquad \mathrm{Env, Fst}\,|\!-\!>(\mathrm{Num},\_T2)-\!>\mathrm{Num}\;\;|\!-\boxed{\mathrm{True}:\_T2}}{\mathrm{Env, Fst}\,|\!-\!>(\mathrm{Num},\_T2)-\!>\mathrm{Num}\;\;|\!-(3,\mathrm{True}):(\mathrm{Num},\_T2)}\;\mathrm{Pair}}{\mathrm{Env, Fst}\,|\!-\!>(\mathrm{Num},\_T2)-\!>\mathrm{Num}\;\;|\!-\mathrm{Fst}(3,\mathrm{True}):\mathrm{Num}}\;\mathrm{Apply}$$

- Repeatedly: apply the rule whose consequent matches the selected goal then select the top-left remaining goal.
- A derivation containing unknowns stands for a family of possible derivations.
- Some rule applications constrain the unknowns.
- Unsatisfiable constraints mean an untypeable program

Checking with polymorphic functions is not completely straightforward

$$\frac{\text{Env} \;\vdash \boxed{\text{Fst}:\_\text{Ta}->\_\text{T}} \qquad \text{Env} \;\vdash (\text{Fst}(3,()),\text{True}):\_\text{Ta}}{\text{Env} \;\vdash \text{Fst}(\text{Fst}(3,()),\text{True}):\_\text{T}}\;\text{Apply}$$

1:	**Env, Fst|->(_T1,_T2)->_T1**	**Environment**
2:	**Fst:(_T1,_T2)->_T1**	**Id 1.2**
3:	**Fst:(_T1,_T2)->_T1**	**Id 1.2**
	. . .	
4:	**(3,()):(_T1,_T2)**	
5:	**Fst(3,()):_T1**	**Apply 3,4**
	. . .	
6:	**True:_T2**	
7:	**(Fst(3,()),True):(_T1,_T2)**	**Pair 5,6**
8:	**Fst(Fst(3,()),True):_T1**	**Apply 2,7**

- Start with type unknowns in the environment for fst .
- Rule applications constrain the unknowns.
- Eventually the constraints are insatisfiable.
- The problem? An unknown type is not the same as a “for any” type.

Polymorphic functions need their types instantiated independently at each occurence

One near-solution (for built-ins) adds rule schemes such as

$$\dfrac{(\tau_1, \tau_2 \text{ can be any types})}{env \vdash \ \text{fst} \ : (\tau_1, \tau_2) \rightarrow \tau_1}\ \text{Built In}$$

“at any point where fst is used it has a type *of the form* $(\tau_1, \tau_2) \rightarrow \tau_1$”

$$\dfrac{\text{Env |–}\boxed{\text{Fst:\_Ta–>\_T}}\qquad \text{Env |–(Fst(3,()),True):\_Ta}}{\text{Env |–Fst(Fst(3,()),True):\_T}}\ \text{Apply}$$

$$\dfrac{\dfrac{}{\text{Env |–Fst:(\_T,\_T2)–>\_T}}\ \text{Built In}\qquad \dfrac{\dfrac{\text{Env |–}\boxed{\text{Fst:\_Ta–>\_T}}\quad \text{Env |–(3,()):\_Ta}}{\text{Env |–Fst(3,()):\_T}}\ \text{Apply}\qquad \text{Env |–True:\_T2}}{\text{Env |–(Fst(3,()),True):(\_T,\_T2)}}\ \text{Pair}}{\text{Env |–Fst(Fst(3,()),True):\_T}}\ \text{Apply}$$

$$\dfrac{\dfrac{}{\text{Env |–Fst:(\_T,\_T2)–>\_T}}\ \text{Built In}\qquad \dfrac{\dfrac{\dfrac{}{\text{Env |–Fst:(\_T,\_T3)–>\_T}}\ \text{Built In}\quad \text{Env |–}\boxed{\text{(3,()):(\_T,\_T3)}}}{\text{Env |–Fst(3,()):\_T}}\ \text{Apply}\qquad \text{Env |–True:\_T2}}{\text{Env |–(Fst(3,()),True):(\_T,\_T2)}}\ \text{Pair}}{\text{Env |–Fst(Fst(3,()),True):\_T}}\ \text{Apply}$$

But this does not solve the problem for *user-defined* functions.

Type rules are inadequate for programmer-defined polymorphic functions

$$\dfrac{\dfrac{\dfrac{}{\text{Env , x|->\_Ta |-x:\_Ta}}\,\text{Id}}{\text{Env |-\{x-->x\}:\_Ta->\_Ta}}\,\text{Fun} \qquad \dfrac{\dfrac{\dfrac{}{\text{Env , f|->\_Ta->\_Ta |-f:\_Ta->\_Ta}}\,\text{Id} \quad \text{Env , f|->\_Ta->\_Ta |-}\boxed{\text{3:\_Ta}}}{\text{Env , f|->\_Ta->\_Ta |-f 3:\_Ta}}\,\text{Apply} \quad \text{Env , f|->\_Ta->\_Ta |-f():\_T2}}{\text{Env , f|->\_Ta->\_Ta |-(f 3,f()):(\_Ta,\_T2)}}\,\text{Pair}}{\text{Env |-let f=\{x-->x\}in(f 3,f())end:(\_Ta,\_T2)}}\,\text{Let'0}$$

$$\dfrac{\dfrac{\dfrac{}{\text{Env , x|->Num |-x:Num}}\,\text{Id}}{\text{Env |-\{x-->x\}:Num->Num}}\,\text{Fun} \qquad \dfrac{\dfrac{\dfrac{}{\text{Env , f|->Num->Num |-f:Num->Num}}\,\text{Id} \quad \dfrac{}{\text{Env , f|->Num->Num |-3:Num}}\,\text{Constant}}{\text{Env , f|->Num->Num |-f 3:Num}}\,\text{Apply} \quad \text{Env , f|->Num->Num |-}\boxed{\text{f():\_T2}}}{\text{Env , f|->Num->Num |-(f 3,f()):(Num,\_T2)}}\,\text{Pair}}{\text{Env |-let f=\{x-->x\}in(f 3,f())end:(Num,\_T2)}}\,\text{Let'0}$$

Generalising from a derivation

Evidently if there is a derivation leading to a type containing unknowns, such as

$$\dfrac{\vdots}{env \vdash\; E : \_T_a \to \_T_r}\;\text{Fun}$$

then (providing $\_T_a, \_T_r$ do not appear in env) for *any* (monomorphic) types, τ_a, τ_r, there is a derivation

$$\dfrac{\vdots}{env \vdash\; E : \tau_a \to \tau_r}\;\text{Fun}$$

When this is so we make an analogy with proofs in logic, and say (informally) that

$$E \text{ has type } \forall a, r \cdot a \to r \text{ in environment } env$$

we call ∀... types *polymorphic*; the others are called *monomorphic*.

Generalization

When the type variables in T but not in env are $\_T_1, ...\_T_n$ we write (for arbitrary type variables $i_1, ..., i_n$)

$$env \vdash \; T \ll \forall i_1, ..., i_n \cdot T[\_T_1, ...\_T_n := i_1, ...i_n]$$

wmeaning “T generalizes to $\forall i_1, ..., i_n \cdot T[\_T_1, ...\_T_n := i_1, ...i_n]$ in env.”

Examples

$$\begin{array}{lll}
env & \vdash \; \_T_a \to \_T_r & \ll \; \forall a, r \cdot a \to r \\
env & \vdash \; (\_T_a \to \_T_r) \to [\_T_a \to \_T_r] & \ll \; \forall a, r \cdot (a \to r) \to [a] \to [r] \\
env \oplus \left\{ \; x \mapsto \_T_a \; \right\} & \vdash \; \_T_a \to (\_T_a, T_y) & \ll \; \forall y \cdot \_T_x \to (\_T_x, y)
\end{array}$$

Situations like the last can arise while typing a nested polymorphic function

```
let pr x =
    let pw y = (x, y)
    in  (pw 3, pw 'c', pw "foo")
end;;
```

By definition, when there are no type variables in T but not in env, T generalizes to itself. *i.e.*

$$env \vdash \; T {\ll} T$$

Revisiting the Id and Let rules

$$\frac{(env\ \ i = T)}{env \vdash\ i\ :\ T}\ \text{Id } (T \text{ a monotype})$$

$$\frac{(env\ \ i = \forall i_1, ...i_n \cdot T)}{env \vdash\ i\ :\ T[i_1, ..., i_n := T_1, ..., T_n]}\ \text{Polytype Id}$$

“Use a polymorphically typed name at any of its instances”

$$\frac{\begin{array}{ll} env & \vdash\ \Delta \overset{\Delta}{:} \left\{i_1 \mapsto T_1, ..., 1_n \mapsto T_n\right\} \\ env & \vdash\ T_1 \ll T'_1 \\ & ... \\ env & \vdash\ T_n \ll T'_n \\ env \oplus \left\{i_1 \mapsto T'_1, ..., i_n \mapsto T'_n\right\} & \vdash\ Body : T \end{array}}{\begin{array}{ll} env & \vdash\ \textbf{let}\ \Delta\ \textbf{in}\ Body\ \textbf{end} : T \end{array}}\ \textbf{let}$$

“Generalize types before adding them to the environment”

➥

The Type Inference Algorithm

- The rules are now deterministic – *except* for “Polytype Id”
- The typechecker is a special-purpose algorithm that emulates application of the rules
- How do we discover the appropriate types to use in “Polytype Id”
  - Invent type unknowns as rules are applied
  - Equate type expressions that the rules require to be equivalent, thereby constraining the unknowns

Type judgements are implemented by:

```
etype:     env -> expr -> typexpr
declTypes: env -> decl -> env
```

such that

$$\begin{array}{llllll} env \vdash E : T & \text{exactly when} & etype & env & E & = T \\ env \vdash \Delta \overset{\Delta}{:} env' & \text{exactly when} & declTypes & env & \Delta & = env' \end{array}$$

```
let rec etype: env -> expr -> typexpr = fun env -> function
|   ...
```

$$\frac{env \oplus \left\{x \mapsto T_a\right\} \vdash \ Body \quad : T_r}{env \qquad\qquad\quad \vdash \ \lambda\ x{\cdot}Body\colon T_a \to T_r}\ \text{Fun}$$

```
|   Fun(x, b) ->
    let ta = newUnknown() in
    let tr = etype (env |+| (x |-> ta)) b in
        FunType(ta, tr)
```

$$\frac{\begin{array}{l} env \vdash \ f \quad : T_a \to T_r \\ env \vdash \ a \quad : T_a \end{array}}{env \vdash \ f\ a\colon T_r}\ \text{Apply}$$

```
|   Apply(f, a) ->
    let tr, tf, ta = newUnknown(), etype env f, etype env a
        ... unify tf (FunType(ta, tr)) ... ;
        tr
```

`unify:  typexpr -> typexpr -> unit`

- equates two type expressions,
- solves constraints on type unknowns as a side effect
- raises an exception if the type expressions cannot be equated

For conditional expressions

- ➪ Guard must be a boolean
- ➪ True and false subexpressions must have the same type
- ➪ Type of the whole is the type of the true and false subexpressions

```
...
|   If (g, e1, e2) ->
    let tg, t1, t2  = etype env g, etype env e1, etype env e2 in
        ... unify t1 t2       ... ;
        ... unify tg BoolType ... ;
        t1
```

If unifications fail, then ap't error reports can be given

```
|   If (g, e1, e2) ->
    let tg, t1, t2  = etype env g, etype env e1, etype env e2 in
       (try unify t1 t2 with
        Failure message  ->
            failwith (estring e1^": "^tstring t1^" and "^
                      estring e2^": "^tstring t2^" should have the same type."));
       (try unify tg BoolType with
          Failure message  ->
            failwith (estring g^" should be a bool but has type: "^tstring tg));
       t1
```

Polytypes need to be given monotyped instances (with unknowns replacing variables)

```
    ...
    |   Id    i -> instantiate (env |?| i)
```

`instantiate` does the necessary work

$$\begin{array}{lcl} \mathit{instantiate}\ \ (\forall v_1, ...v_n \cdot T) & = & T[v_1, ..., v_n := \texttt{newUnknown}(), ..., \texttt{newUnknown}()] \\ \mathit{instantiate}\ \ T & = & T\ \ (\textbf{if}\ T\ \textbf{isn't}\ \forall ...) \end{array}$$

```
    ... instantiate : typexpr -> typexpr = function
    |   PolyType (vs, t) -> substitute (substitution vs) t
    |   t                -> t
```

`...` `substitution` $[v_1,\ \ ...\ \ v_n]\ \ =\ \ \left\{\ \ v_1 \mapsto \texttt{newUnknown}(),\ \ ...,\ \ v_n \mapsto \texttt{newUnknown}()\ \ \right\}$

`... substitute : env -> typexpr -> typexpr` implements substitution in type expressions

Unification implemented very efficiently (and subtly) in the PicoML system

Unification succeeds for identical types

```
let rec unify: typexpr -> typexpr -> unit =
    fun ty1 ty2 ->
    let ty1, ty2 = typeclass ty1, typeclass ty2
    in
    match ty1, ty2 with
    |   NumType,           NumType             -> ()
    |   BoolType,          BoolType            -> ()
    |   CharType,          CharType            -> ()
    |   ListType t,        ListType t’         -> unify t t’
    |   PairType(t1, t2),  PairType(t1’, t2’)  -> unify t1 t1’; unify t2 t2’
    |   FunType (t1, t2),  FunType (t1’, t2’)  -> unify t1 t1’; unify t2 t2’
```

Unification constrains unknown types

```
    ...
    |   UnknownType _,    _                    -> equate ty1 ty2
    |   _,                UnknownType _        -> equate ty2 ty1
    |   _  -> failwith ("types "^tstring ty1^" and "^ tstring ty2^" do not match.")
```

Representation of type unknowns

```
type typexpr     = ... as before
|                  UnknownType  of typeunknown
and typeunknown = typexpr pointer
```

Type Unknowns can be created, constrained (by setting them to be equivalent to a type expression), compared for equality, etc.

```
newUnknown  : unit -> typexpr
sameUnknown : typexpr -> typexpr -> bool
setUnknown  : typexpr -> typexpr -> unit
```

`equate typeunknown typexpr` sets the given type unknown to the given type expression – provided doing so wouldn't generate an infinite type.

```
let equate : typexpr -> typexpr -> unit = fun ty1 ty2 ->
match ty2 with
| UnknownType _ when sameUnknown ty1 ty2  -> ()
| _             when not(occurs ty1 ty2)  -> setUnknown ty1 ty2
| _  -> failwith ("type "^tstring ty1^" cannot be the same as "^ tstring ty2)
```

- A type unknown is either
  - unset – in which case there are no constraints on it, or
  - set – in which case it points to a type expression (possibly also an unknown) to which it has been constrained to be equivalent by a unification step.
- The function `typeclass`
  - maps a (set) type unknown to the type to which it is constrained to be equivalent
  - maps a known type (or unset type unknown) to itself

```
let rec typeclass : typexpr -> typexpr =
function UnknownType p when isSet p ->
         let t' = typeclass (value p) in
             set p t';
             t'
| t -> t
```

  - (as a side effect) it shortens the “constraint chains” that link type unknowns.
    - This ensures that all type unknowns that get constrained to be equivalent end up set to a single type unknown
    - it is that single unknown (returned from `typeclass`) that is used when equating unknowns during unification.

Note 1 (from page 1)

The *Lambda Calculus* was introduced in the 1930s by Alonzo Church as part of his work on the foundations of mathematics. Its *pure* form has an astonishingly simple characterisation, yet it can express all computable functions.[1] Although we have insufficient time to pursue its study in detail here, it is worthwhile discussing it briefly for it provides the theoretical underpinning for many of the functional languages in use today. Our treatment here is very superficial, and intended only to provide the background to the discussion of the semantics of PicoML.

Abstract syntax

The *terms* of the Lambda calculus are

$$\begin{array}{l} \mathit{variable} \\ \lambda \mathit{variable} \cdot \mathit{term} \\ \mathit{term}\ \mathit{term} \end{array}$$

Informally, a term of the form $\lambda v \cdot t$ means “that function of v whose value is t,” and a term of the form $t_1\ t_2$ means “apply the function that is the value of t_1 to the argument t_2.”

Examples:

x	the variable x
$\lambda x \cdot x$	the identity function
$\lambda v \cdot v$	the identity function (with a different bound variable)
$(\lambda x \cdot x)\ y$	an application of the identity function
$\lambda f \cdot \lambda g \cdot \lambda x \cdot f(g\ x)$	the composition function

Application is syntactically left-associative: in other words, the terms written

$$t_1\ t_2\ t_3 \quad \text{and} \quad (t_1\ t_2)\ t_3$$

have identical structure.

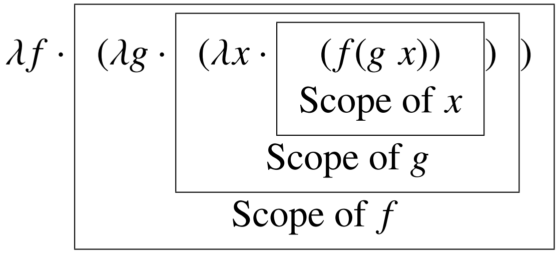
The variable v in $\lambda v \cdot t$ is called the *bound variable*: the term t is called the *body*. The term t_1 in $t_1\ t_2$ is called the *operator*, whilst t_2 is called the *operand*.

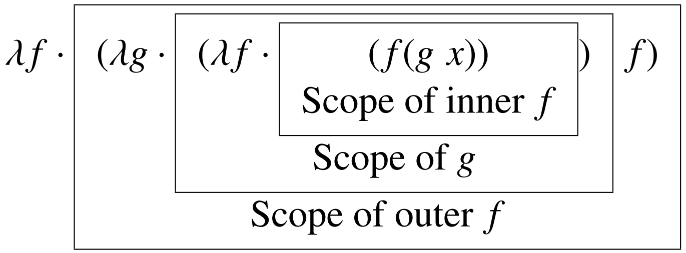
[1] This is the essence of the *Church-Turing thesis*.

The second and third examples above are (from the point of view of their intended meaning) essentially identical, since they differ only in the name of their bound variable. We say that they are α-equivalent.

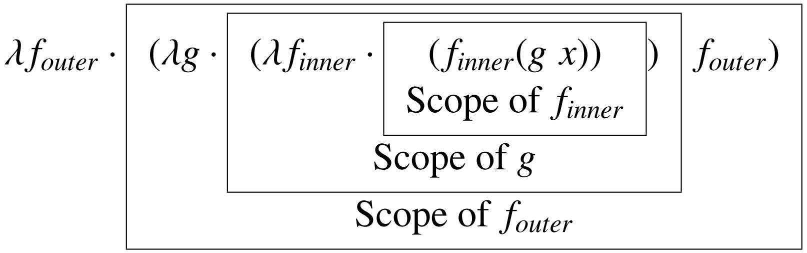
The occurence of the variable v next to λ is called is *binding occurence* (or just its *binding*), and the region of the body in which occurences of v mean the same variable is called the *scope* of that binding. Non-binding occurences of a variable within its scope are called *applied* occurences.

The scope of the binding of v in $\lambda v \cdot t$ is the whole of the term t except for subterms within which the same variable is bound again, thereby causing a "hole" in the scope of the outer binding





It is always possible to eliminate such holes by suitably renaming bound variables, for example:



An applied occurence of a variable in a term that is not bound in that term is called *free* in that term. Formally, for distinct variables v, and w, and terms t, t_1, t_2

v	occurs free in	v	
v	does not occur free in	w	
v	occurs free in	$\lambda w \cdot t$	if it occurs free in t
v	does not occur free	$\lambda v \cdot t$	
v	occurs free in	$t_1\ t_2$	if it occurs free in t_1 or in t_2

Substitution

Substitution of one term for the free occurences of a variable within another term plays a crucial role in defining the calculus. Here the notation $[\![variable := term_2]\!]term_1$ stands for the systematic *safe* substitution of $term_2$ for all *non-bound* occurences of $variable$ in $term_1$.

Examples:

The substitution	$[\![x := y]\!](x)$	yields the term	y
The substitution	$[\![x := y]\!](\lambda x \cdot x)$	yields the term	$(\lambda x \cdot x)$
The substitution	$[\![x := y]\!](\lambda v \cdot x)$	yields the term	$(\lambda v \cdot y)$

Structured Operational Semantics

The main relation defined by the semantics is the “reduces to” relation (which we write here as an arrow). The judgement

$$term \longrightarrow term'$$

means (informally) a single step in the simplification of $term$ yields $term'$.

The principal *computational* rule of the calculus is the (so-called) β-reduction rule, which captures the computational intuition that the result obtained by applying a function to an argument can be obtained by substitution of the argument for all occurences of the bound variable of the function in the body of the function.

$$\frac{}{(\lambda v \cdot t)\ t' \longrightarrow [\![v := t']\!]t} \quad \beta$$

The *structural* rules (often left implicit in treatments of the calculus) are as follows:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{Operator}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad \text{Operand}$$

$$\frac{t \longrightarrow t'}{\lambda v \cdot t \longrightarrow \lambda v \cdot t'} \qquad \text{Body}$$

Arithmetic

In order to motivate our exposition a little, we will extend the calculus by “building in” arithmetic.[2]

To do this we need add only add two new kinds of constant term: – the numerals n_i, and the term $'+'$. We write terms of the form $('+'\ t_1)\ t_2$ more conventionally as $t_1 + t_2$.

The main computational rule for arithmetic is (for numerals n_1, n_2, and n, with n representing the sum of the numbers represented by n_1 and n_2)

$$\frac{}{'+'\ \ n_1\ n_2 \longrightarrow n} \qquad +$$

Computation

If there is a finite sequence of terms $t_0, t_1, ...t_n$ with $t_0 \longrightarrow t_1$, and $t_1 \longrightarrow t_2$, ..., and $t_{n-1} \longrightarrow t_n$ then we write $t_0 \overset{+}{\longrightarrow} t_n$. We write $t_0 \overset{*}{\longrightarrow} t_n$ when t_0 and t_n are identical terms, or when $t_0 \overset{+}{\longrightarrow} t_n$.

For example, $((\lambda i \cdot (\lambda j \cdot i + 3))\ 4)\ 300 \overset{*}{\longrightarrow} 7$, by the following sequence of reduction steps.

[2] This is not strictly necessary, for (as Church showed) there are ways of representing numbers and arithmetic in the “pure” calculus. The details are a little complicated, and would obscure the present discussion, so they are omitted here.

$$\begin{array}{lll}
((\lambda i \cdot (\lambda j \cdot i + 3))\ 4)\ 300 & \longrightarrow & \text{by (Operator, }\beta) \\
(\lambda j \cdot 4 + 3)\ 300) & \longrightarrow & \text{by (}\beta) \\
4 + 3 & \longrightarrow & \text{by (+)} \\
7 & &
\end{array}$$

Note that the first “outermost” reduction step presented above is actually justified by 2 rule applications

$$\cfrac{\cfrac{}{(\lambda i \cdot (\lambda j \cdot i + 3))\ 4) \longrightarrow (\lambda j \cdot 4 + 3)}\ \beta}{((\lambda i \cdot (\lambda j \cdot i + 3))\ 4)\ 300 \longrightarrow (\lambda j \cdot 4 + 3)\ 300)}\ \text{Operator}$$

The β rule does the work, while the application of the operator rule is justified by the fact that the beta rule was applicable.

Normal Forms

A term is in *normal form* when no further β (or arithmetic) steps can be made. It is in *head normal form* if it is of the form $\lambda variable \cdot term$ (this form is called an *abstraction*), or ‘ + ‘ n (a “partially applied” arithmetic operator).

A normal form corresponds intuitively to a fully-evaluated term. Most functional language implementations dispense with the *Body* rule, and are content to stop evaluating complete program terms once they reach normal, or head normal form.[3] There’s a little more to the discussion of normal forms in a real functional language, since the question of the circumstances in which list- and tuple- constructors evaluate their arguments must be dealt with.[4]

We say that a term t has a normal form t' if t' is in normal form and $t \overset{*}{\longrightarrow} t'$.

Non-termination

Some terms have no normal form. For example, in the following computation the β rule is always applicable, but its use gives rise to the same expression again.

[3] For what is the point of evaluating “inside” the body of an abstraction?

[4] In a strict language (like OCAML) constructors evaluate their arguments; in a lazy language (like Haskell) constructors don’t evaluate their arguments – except when the resulting construction needs to be output.

$$
\begin{array}{l}
(\lambda f \cdot f\ f)(\lambda f \cdot f\ f) \longrightarrow \text{(by } \beta\text{)} \\
(\lambda f \cdot f\ f)(\lambda f \cdot f\ f) \longrightarrow \text{(by } \beta\text{)} \\
(\lambda f \cdot f\ f)(\lambda f \cdot f\ f) \longrightarrow \text{(by } \beta\text{)} \\
\ldots
\end{array}
$$

Evaluation strategies

The inference system that describes the semantics of the calculus is nondeterministic, in the sense that it does not prescribe the order in which rules are applied when more than one is applicable.

This has important consequences, for there are expressions for which the order of application can affect termination. For example, in the following term using the β rule on the outermost application yields a normal form immediately

$$
\begin{array}{l}
(\lambda x \cdot 3)((\lambda f \cdot f\ f)(\lambda f \cdot f\ f)) \longrightarrow \text{(by } \beta\text{)} \\
3
\end{array}
$$

whereas an attempt to reach a normal form by reducing the operand first will not terminate

$$
\begin{array}{l}
(\lambda x \cdot 3)((\lambda f \cdot f\ f)(\lambda f \cdot f\ f)) \longrightarrow \text{(by Operand, } \beta\text{)} \\
(\lambda x \cdot 3)((\lambda f \cdot f\ f)(\lambda f \cdot f\ f)) \longrightarrow \text{(by Operand, } \beta\text{)} \\
(\lambda x \cdot 3)((\lambda f \cdot f\ f)(\lambda f \cdot f\ f)) \longrightarrow \text{(by Operand, } \beta\text{)} \\
\ldots
\end{array}
$$

A mechanical evaluator must have a consistent (and simple) strategy for selecting the next position at which the β (or $+$) rule is to be applied. The two “extreme” strategies are known as *normal order* and *applicative order*.

The normal order strategy always performs the “leftmost-outermost” reduction that is possible: this corresponds to substituting operands into function bodies before the operands are in normal form.

The applicative order strategy reduces both operator and operand of an application to (head) normal form before applying the β rule. The second example above used this evaluation strategy, and it was the attempt to reduce the operand to normal form before performing the substitution that caused the nontermination.

The normal order strategy would have behaved as in the first example above, and since x didn't occur in the body of the operator, the (potentially nonterminating) operand disappeared as a consequence of the substitution. The applicative order strategy would have behaved as in our second example, and failed to terminate.

There are three key facts to know about these strategies.

1. If a term has a normal form, then the normal order strategy will always terminate with that normal form.
2. The applicative order strategy does not always terminate – even for a term with a normal form.
3. If both strategies terminate, then the normal form they produce will be the same (up to systematic renaming of bound variables).

Definitions

It is important to make formalisms readable: particularly those that are to be manipulated and understood by humans as well as machines. With this in mind we shall use the notation $\textbf{let } v = t' \textbf{ in } t$ to stand for the term $(\lambda v \cdot t)(t')$.

Example: normal order reduction of a term with two definitions

$$\begin{array}{lll}
\textbf{let } c = \lambda\, f \cdot \lambda\, g \cdot \lambda\, x \cdot (f(g\ x) \textbf{ in let } s = ('+'\ 1) \textbf{ in } (c\ s\ s\ 1) & \longrightarrow & \text{(desugar)} \\
(\lambda\, c{\cdot}((\lambda\, s{\cdot}(c\ s\ s\ 1))('+'\ 1)))(\lambda\, f{\cdot}\lambda\, g{\cdot}\lambda\, x{\cdot}(f(g\ x))) & \longrightarrow & (\text{by } \beta) \\
(\lambda\, s{\cdot}((\lambda\, f{\cdot}\lambda\, g{\cdot}\lambda\, x{\cdot}(f(g\ x)))\ s\ s\ 1))('+'\ 1) & \longrightarrow & (\text{by } \beta) \\
(\lambda\, f{\cdot}\lambda\, g{\cdot}\lambda\, x{\cdot}(f(g\ x)))\ ('+'\ 1)\ ('+'\ 1)\ 1 & \longrightarrow & (\text{by } operator, \beta) \\
(\lambda\, g{\cdot}\lambda\, x{\cdot}(('+'\ 1)(g\ x)))\ ('+'\ 1)\ 1 & \longrightarrow & (\text{by } operator, \beta) \\
(\lambda\, x{\cdot}(('+'\ 1)(('+'\ 1)\ x)))\ 1 & \longrightarrow & (\text{by } \beta) \\
('+'\ 1)(('+'\ 1)\ 1) & \longrightarrow & (\text{by } operand, +) \\
('+'\ 1)2 & \longrightarrow & (\text{by } +) \\
3 & &
\end{array}$$

Safe *vs.* Unsafe Substitution

The substitution

$$[\![f := g(h\ y)]\!](\lambda g \cdot \lambda x \cdot f(g\ x))$$

is *unsafe*, because the occurence of f that will be substituted for is in the scope of the binding of a variable that appears free in the term $(g(h\ y))$ that is being substituted.

An unsafe substitution can always be made safe by the expedient of inventing a new name for the bound variable that is causing the problem, (in this case g), and systematically renaming all free occurences of the bound variable in its scope. The new name should be distinct from that of any variable in the term term that is being substituted. For example, to make the above substitution safe again we can rename g to g':

$$[\![f := g(h\ y)]\!](\lambda g' \cdot \lambda x \cdot f(g'\ x)) \quad \text{safely yields the term} \quad (\lambda g' \cdot \lambda x \cdot g(h\ y)(g'\ x))$$

From Lambda Calculi to Functional Languages

It is because the normal order strategy always terminates when applied to terms with normal forms that it has been made the basis of pure functional programming languages.

As we shall see when we discuss the semantics of PicoML, it is possible to implement normal order evaluation very efficiently by *sharing* the cost of evaluating all the occurences of a substituted expression within the body of a function. This sharing strategy is called *lazy evaluation*.

Note 2 (from page 1)
This book is excellent supplementary reading for someone interested in implementingFunctional Languages . It covers the topic from a substantially different point of view, and much more thoroughly than we have space for here.

Note 3 (from page 5)
The abstract syntax doesn't show the non-nesting constraint, but when we come to discuss typechecking we'll see why it has to be imposed.

Note 4 (from page 8)
Expressions are typechecked and evaluated in the environment established by the declarations. The value and type of the expression are printed, along with statistics that show execution time, proportion of shared substitutions, *etc*.

Note 5 (from page 8)
Numbers are represented in the interpeter as arbitrary precision rationals

Note 6 (from page 10)
As we shall see (slide 11), there are two grades of "normal form". For the moment we shan't distinguish between them.

Note 7 (from page 11)
What we are trying to capture is the difference between (say) an expression (such as a constant) which cannot sensibly ever be further evaluated, and an expression (such as a "lazy" list cons or or a pair) that the semantics decrees need not be completely evaluated in a particular context.

Think of a normal form as "evaluated as far as conceivably possible" and a weak normal form as "evaluated as far as necessary to discover its superstructure."

Note 8 (from page 14)

Rules are given in the form

$$\frac{antecedent_1 \quad ... \quad antecedent_n}{consequent} \quad \text{rulename}$$

where the $antecedent_i$ and $consequent$ are judgements. The interpretation of a rule such as this is that the $consequent$ judgement holds provided all of the antecedent judgements hold. If there are no antecedents, then the consequent judgement holds unconditionally. People used to logic programming will recognise the form of rules as analogous to the form taken by a clause, namely $consequent :- antecedent_1, ..., antecedent_n$.

The first two rules capture the essence of the idea that picoML is a "normal order" language. Notice that the substitution of the formal parameter for the bound variable in the body of a function takes place *before* it is evaluated. The apply rule is usually called β – for historical reasons connected with Church's presentation of the λ calculus.

Note 9 (from page 15)
Notice that the first arithmetic rule shows that a "partly applied" arithmetic operator is a "normal form" of sorts. Clearly nothing more can be done with such a form until its second argument appears: what happens at that point is the subject of the second rule.

By $\overline{n_1 \div n_2}$ we mean the numeral which represents the number that is the quotient of the numbers represented by thenumerals n_1 and n_2 – in short, the quotient of n_1 and n_2. Similarly $\overline{n_1 + n_2}$ represents the sum of the numbers represented by n_1 and n_2.

Note 10 (from page 25)
Note that named constants (like $(+), 3, \textbf{true}, ...$) are distinct from variables.

Note 11 (from page 27)
This is consistent with the notation we are already using

Note 12 (from page 35)
A formal proof of the consistency of the semantics we present here with the substitution semantics is somewhat technical, and anyway beyond the scope of these lectures. We shall rely heavily on the readers' intuitions in explaining what is going on.

Note 13 (from page 36)
Henceforth (for conciseness) we will omit the use of the Val constructor when discussing $(environment, value)$ pairs.

Note 14 (from page 37)
For this to make sense,

- ➩ E must be closed w.r.t. env
- ➩ E' must be closed w.r.t. env'

i.e. all free variables of E (respectively E') must be in the domain of env (respectively env').

Note 15 (from page 38)
This interlude is for background only. It's designed to explain the precise technical meaning of our throwaway remark about the analogy between environments and substitutions. If you don't (want to) understand it then you need not worry – think of the environment in an (environment, expression) pair as "providing enough information about the variables for the expression to be completely evaluated."

Note 16 (from page 38)
We will pay for this should the implementation prove to have any serious bugs, but we simply don't have time to pursuse such a proof in a course of this kind.

Note 17 (from page 39)
`subst` plays a role in the explanation of the connection between substitution and environment semantics. It is not used in the definition of either semantics.

Note 18 (from page 39)
The quoted material in the definition of `subst` is intended to be abstract syntax expressed in concrete form, not strings – the quotes are there simply to set off the material from its surroundings.

Note 19 (from page 39)
Here we use the (idiosyncratic) notation $env \ominus \backslash i$ to mean “the mapping identical to env except at i, at which it is undefined.”

$$\begin{array}{ll} \mathbf{dom}(env \ominus S) & = \mathbf{dom}\, env \backslash S \\ (env \ominus S)(x) & = env\ x \mathbf{if}\ x\ \mathbf{in}\ (\mathbf{dom}\, env \backslash S) \end{array}$$

Note 20 (from page 51)
During the construction of the efficient interpreter we will use the mathematical font in places where we are not in a position to write a CAML expression, but we know the value we need that expression to yield. For example, $\left(env' \oplus \left\{ bv \mapsto eval\ env\ e \right\} \right)$

Note 21 (from page 61)
The implementation of recursive environments is rather subtle. On the one hand, because it is necessary to evaluate the right hand sides of the nested recursive declarations in the current environment, we need to provide an environment that has the same bindings as the current environment. On the other hand, we want to be able to “tie the knot” in the newly-constructed recursive environment by simply updating a reference: but this reference cannot be the reference that holds the current environment – for that way we would be tying a very tangled knot indeed! The solution is to construct a new reference to the current value mapping; use it temporarily to construct the environment extension, then assign to it once the extension has been constructed.

Note 22 (from page 63)
The actual $print_val$ procedure does things slightly differently. It has a special case for lists of characters, which it prints without the punctuation.

Note 23 (from page 66)
Try this experiment with the Bourne Again Shell: `/usr/local/bin/bash`

```
let x=50
let y=60
let z="sixty-60"
let q1=x+y
let q2=x+z
echo $q1 $q2
```

Note 24 (from page 66)
This isn't the same as being able to determine that no operation is applied to values outside its domain, and it certainly doesn't absolve programmers from having to consider the soundness of their programs relative to their specifications, but it's a good first approximation.

If a program text is written in such a way that an attempt is made to add a number is added to a string, then the chances are that the program has been mis-written. The outcome of trying to run such a program, whether it be the program halting with an "invalid operand" error, or (worse) carrying on with wholly spurious data, is likely to cause somebody a nasty surprise.

Note 25 (from page 67)
Auxiliary inference rules are needed to deal with declarations.

Note 26 (from page 67)
An *interpreter* for a language that *is* normally typechecked, often still has to perform some kind of action that is equivalent to checking types at runtime: a *compiler* usually doesn't.

Here (in the PicoML interpreter) the *Apply* case of *eval* has to discover what kind of function it's applying: a primitive, or a user-defined function. If the value being applied isn't one of these then the interpeter needs to draw attention to that fact. Were the *other* case not present, then the OCAML implementation itself would raise an exception, but that exception would, because generic, not carry enough information for the programmer to deduce what had gone wrong. Of course the *other* case will never be reached by a properly-typed program, so it is actually redundant.

```
|   Apply (f, e) ->
    ( match stricteval env f with
      | FunVal(env', bv, body) -> eval (env' <+> (bv|->delay env e)) body
      | Strict     ocamlfun    -> ocamlfun (stricteval env e)
      | Lazy       ocamlfun    -> ocamlfun (delay env e)
      | other -> failwith ("type error in application: "^estring f)
    )
```

Note 27 (from page 73)
In a polymorphic typechecker the best way to deal with the circularity inherent in checking a recursive declaration is to use the same machinery of type variables and unification as the remainder of the language needs. This is also the cleanest (though not the only) way of dealing with type inference for recursive declarations in languages without polymorphism. On the other hand, such languages usually require types to be declared explicitly, so the issue of *deducing* rather than simply*checking* types is less problematic.

Note 28 (from page 74)
Intrepid readers with a theoretical cast of mind may want to reflect on how it might be done. It's worth noting that we are arguing about the relationahip between two "proof systems" here, and (for that very reason) it will not be appropriate to use the proof method that usually comes naturally when arguing about expressions in programming languages, namely structural induction over the phrases of the language.

We note, in passing, that we will need to prove a lot of "local correctness" lemmas that confirm that the type system respects single computation steps.

Roughly, that if

$$env \ \vdash \ E : T$$

then if

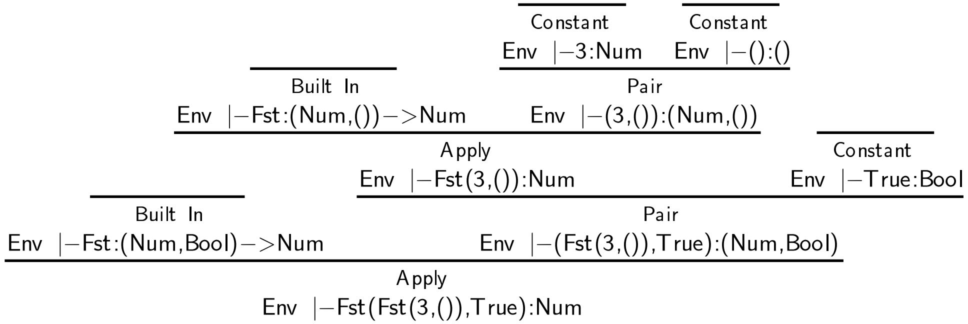
$$\overline{env}, E \ \leadsto \ \overline{env}', E'$$

(for a value-environment $\overline{env}$ consistent with env) then we can prove

$$E' : T$$

Roughly speaking we mean by “consistent value environment” that the value of an identifier in the value environment has the type to which the type environment maps it. This isn’t necessarily as simple as it seems, since there may still be unevaluated expressions within a value environment.

Note 29 (from page 78)
The completed derivation is



Note 30 (from page 82)
The Let rule outlined here is simple and understandable, and would be sensible for PicoML if it didn't have local or sequential forms of combining declarations. The presence of these forms requires a little modification to the generalization policy. Consider, for example, the following top-level declaration

```
let
    f₁ = ... l₁ ... l₂ ... ;
    f₂ = ... l₁ ... l₂ ...
 where
    l₁ = ... ;
    l₂ = ...
 end
```

We'd like the types of the locals l_1, l_2 to be generalized in the environment where they are seen by f_1, f_2. Moreover, we'd like the type of f_1 to be generalized in the environment in which f_2 sees it. We would, of course, expect to simultaneously generalize the types of variables defined simultaneously (by **and**).

The function *typeEnv* in the typechecker is built along the lines suggested in this note.

Note 31 (from page 87)
The typeclass of a known type is itself. That of an unknown class is the representative unknown of the equivalence class of all unknowns so far constrained to have the same type.

Note 32 (from page 88)
An `'a` pointer is either set to an `'a` or is unset. It has a human-readable name which is easily derivable from its identity.

Exercise 1 (from page 28)
PicoML has two additional forms of composite declaration – with concrete syntax $\Delta_1;\Delta_2$ and Δ_1 **where** Δ_2. Syntactically **where** binds least tightly of all declaration combinators, whilst **rec** binds less tightly than **and** and more tightly than ";".

In the composite declaration $\Delta_1;\Delta_2$, the scope of the declarations made in Δ_1 includes the declarations made in Δ_2, and the resulting declarations are effectively those of the first extended by thoseof the second.

In the composite declaration Δ_1 **where** Δ_2, the scope of the declarations made in Δ_2 includes the declarations made in Δ_1, but the resulting declarations are the declarations made in Δ_1. In other words, the declarations made in Δ_2 are "local" to the declarations made in Δ_1.

For example

```
let x=1; y=x+3 in x+y end ⟹ 9
let z=x+x; y=x+3 where x=1 in z+y end ⟹ 6
let x = 100 in let z=x+x; y=x+3 where x=1 in x+z+y end end ⟹ 106
```

Give rules which capture the intended meaning of these forms of composite declaration. (Answer)

Exercise 2 (from page 31)
Use the same style of presentation to show the computation of

```
let rec f n    = if n=0 then 1 else g n (n-1)
        and g n n' = n × f n'
 in f 2
 end
```

Exercise 3 (from page 31)
It can be argued that the present definition of *unroll* is problematic – in the sense that it yields substitutions containing infinite

expressions. An alternative definition (suggested by Mike Spivey) avoids this problem.

$$unroll[\![v_1, ..., v_n := E_1, ..., E_n]\!] = \left[\begin{array}{lcl} v_1, & & \textbf{let rec } v_1 = E_1 \textbf{ and } ... \textbf{ and } v_n = E_n \textbf{ in } E_1, \\ ..., & := & ..., \\ v_n & & \textbf{let rec } v_1 = E_1 \textbf{ and } ... \textbf{ and } v_n = E_n \textbf{ in } E_n \end{array}\right]$$

Show (in the style used on slide 31) the complete computation of the expression

$$\textbf{let } f\ n = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times f(n-1) \textbf{ in } f\ 2 \textbf{ end}$$

Exercise 4 (from page 39)
Complete the definition of *subst*. You will need to define the a function `dsubst` which substitutes in declarations, and a function `defined` which yields the variables defined by a declaration.

Exercise 5 (from page 45)
An alternative rule for application describes "dynamic binding" – the form used (by accident) in early versions of Lisp. It differs (at $\boxed{!}$) from the static binding rule, in that the environment used for evaluating the body is the environment of the *call*not of the definition of the function being called. Its effect is that the value of a free variable in the body of a function is determined not by its declaration (if any) in the context of the declaration of the procedure, but the declaration that is dynamically closest in the nested function-call heirarchy.

$$\frac{env, F \overset{\surd}{\leadsto} env', (\lambda\ x{\cdot}B) \qquad (\underbrace{env}_{\boxed{!}} \oplus \{x \mapsto (env, A)\}), B \leadsto env'', R}{env, (F\ A) \leadsto env'', R}\ \text{(apply)}$$

What difference would the use of this rule make to the computation outlined on slide 45?

What would happen (with this rule) if you systematically changed the name of the "inner" y to z, and evaluated the following expression instead?

$$\left\langle \begin{array}{l} \mathbf{let}\ y = 20\ \mathbf{in} \\ \mathbf{let}\ axx = \lambda\ x \cdot x + y\ \mathbf{in} \\ \mathbf{let}\ z = 2\ \mathbf{in}\ axx\ z\ \mathbf{end}\ \mathbf{end}\ \mathbf{end} \end{array} \right\rangle$$

How hard do you think the consistent use of this rule in a language would make it for programmers to document the meaning of functions that they define?

Do you think its use in languages such as Postscript and TeX is wise? If not, can you think of a way of achieving its effects when they are actually needed?

(Answer)

Exercise 6 (from page 46)
Write down the rules for hd , tl , null , and (::).

Exercise 7 (from page 47)
Give an answer to exercise 1 which is appropriate for the environment semantics. (Answer)

Exercise 8 (from page 62)
Consider your answers to exercise 1 and exercise 7 and complete the definition of `dval` given on slide 61.

Exercise 9 (from page 71)
Suggest a type inference rules for **if** . (Answer)

Exercise 10 (from page 72)
Suggest declaration rules for sequential ($\Delta_1 ; \Delta_2$) and local (Δ_1 **where** Δ_2) declaration combinations (as explained in exercise 1).

Exercise 11 (from page 74)

Explain why you believe that the rule you gave in answer to exercise 9 is sound.

Would execution of the following program give any nasty surprises in an untyped language? (*len* is the list-length function)

```
let f b = len(if b then [3] else ['c', 'd'])
in
   f True + f False
end;;
```

Would your type rule admit the following subexpression?

```
(if b then [3] else ['c', 'd'])
```

Any comments?

Exercise 1 (Page i)

$$\frac{\begin{array}{c}\Delta_1 \overset{\Delta}{\Longrightarrow} \sigma_1 \\ \Delta_2 \overset{\Delta}{\Longrightarrow} \sigma_2\end{array}}{\Delta_1 \ \textbf{where}\ \Delta_2 \overset{\Delta}{\Longrightarrow} \sigma_2 \cdot \sigma_1}\ \textbf{where}$$

"Interpret the free variables of the left in the context provided by the right."

$$\frac{\begin{array}{c}\Delta_1 \overset{\Delta}{\Longrightarrow} \sigma_1 \\ \Delta_2 \overset{\Delta}{\Longrightarrow} \sigma_2\end{array}}{\Delta_1 ; \Delta_2 \overset{\Delta}{\Longrightarrow} \sigma_1 \oplus (\sigma_1 \cdot \sigma_2)}\ ;$$

"Interpret the free variables of the right in the context provided by the left and override the left context." (*c.f.* the way in which the interactive interpreter accumulates the substitution that constitutes its state.

Exercise 5 (Page iii)
With dynamic binding the computation would proceed as before to the stage "by apply". The environment to the right of the arrow would simply be a duplicate of the "call-site" environment (which maps y to 2, not 20), so the result would be 4.

With z instead of y the answer would be 22 again, for nothing would have captured the outer y.

Here what I'm thinking of is the fact that a function is no longer really closed at the point of its definition: documentation has to tell you the names of any free variables that appear in it, and at the point of use of such a function you need to take care that you haven't inadvertently captured (or, I suppose "liberated") any of them. Higher order functions are particularly difficult to write. (Try writing a "map" function in such a way that its list parameter doesn't capture any free variables in its functional parameter!)

Exercise 7 (Page iii)

$$\frac{\begin{array}{c} env, \Delta_1 \overset{\Delta}{\rightsquigarrow} env_1 \\ env \oplus env_1, \Delta_2 \overset{\Delta}{\rightsquigarrow} env_2\end{array}}{env, \Delta_1 ; \Delta_2 \overset{\Delta}{\rightsquigarrow} env_1 \oplus env_2}\ ;$$

$$\frac{\begin{array}{c} env, \Delta_2 \overset{\Delta}{\rightsquigarrow} env_2 \\ env\oplus env_2, \Delta_1 \overset{\Delta}{\rightsquigarrow} env_1 \end{array}}{env, \Delta_1 \textbf{ where } \Delta_2 \overset{\Delta}{\rightsquigarrow} env_1}\ \textbf{where}$$

Exercise 9 (Page iii)

```
FROM  Env |- G:  Bool
AND   Env |- Et: T
AND   Env |- Ef: T
INFER Env |- if G then Et else Ef : T
```