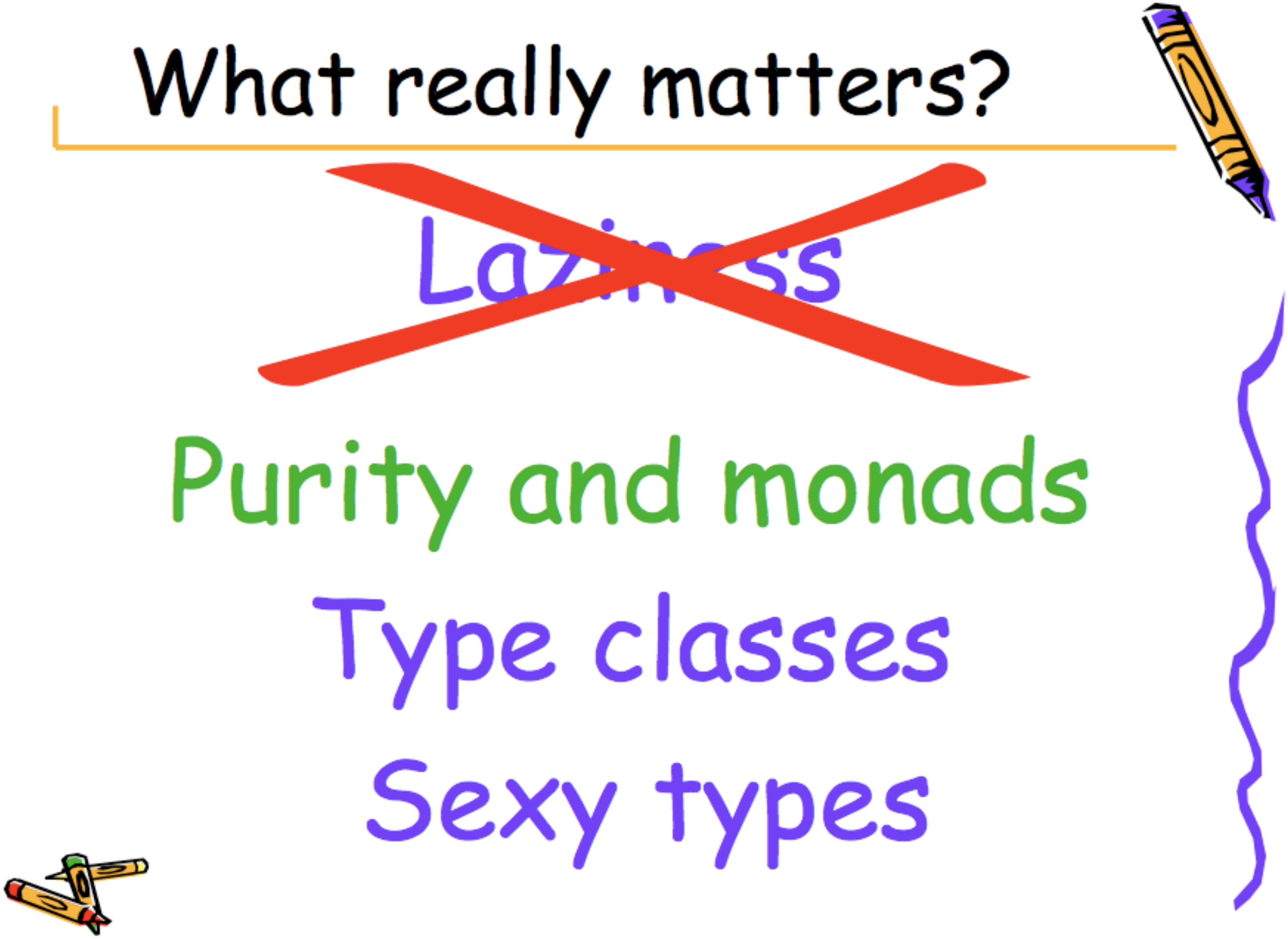
The Disciplined Disciple Compiler

Ben Lippmeier
Australian National University

SPJ 2003: Wearing the Hair Shirt



What really matters?

~~Laziness~~

Purity and monads

Type classes

Sexy types

What is purity, anyway?

- Order of evaluation does not matter when reducing a term.

$$\begin{tikzcd}
& (\lambda x.\ \text{double } x)\ (\text{succ } 3) \arrow[dl, "\text{call-by-name}"'] \arrow[dr, "\text{call-by-value}"] & \\
\text{double } (\text{succ } 3) \arrow[dr] & & (\lambda x.\ \text{double } x)\ 4 \arrow[dl] \\
& \text{double } 4 \arrow[d] & \\
& 8 &
\end{tikzcd}$$

Order matters for functions with “side-effects”

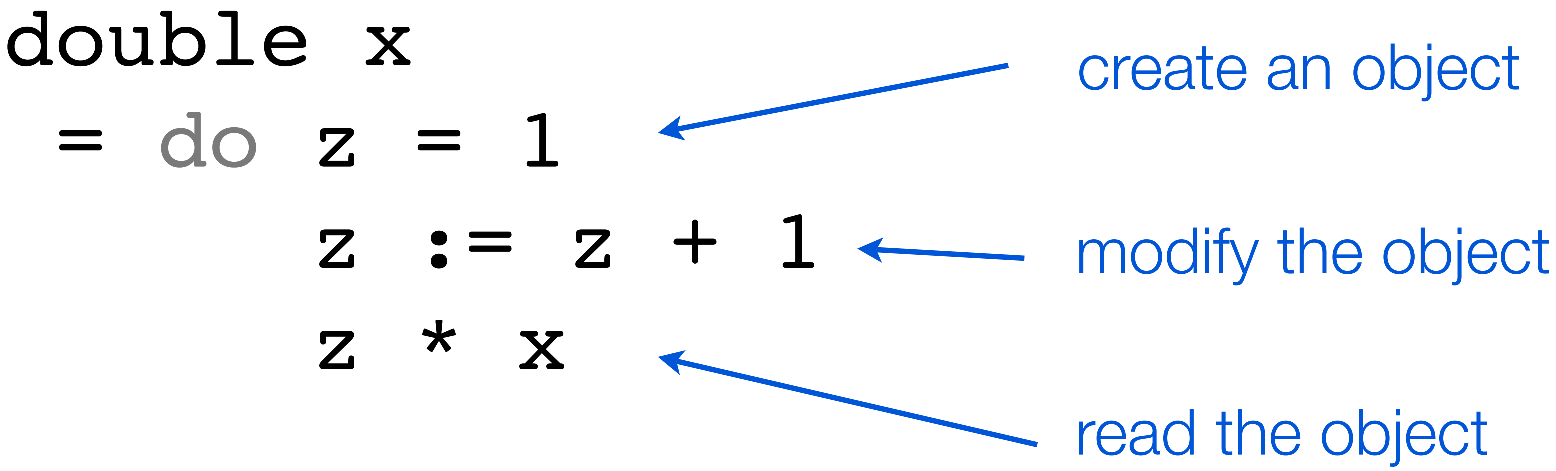
- IO actions affect the outside world.

```
greet name
 = do putStr (“hello ” ++ name)
      putStr “have a nice day”
```

```
checkStatus ()
 = if inTrouble ()
       then launchMissles ()
       else eatCake ()
```

Order matters when accessing mutable data

- Sometimes the desired sequence is explicit in the source program.



```
double x
 = do z = 1
      z := z + 1
      z * x
```

create an object

modify the object

read the object

- But sometimes not...

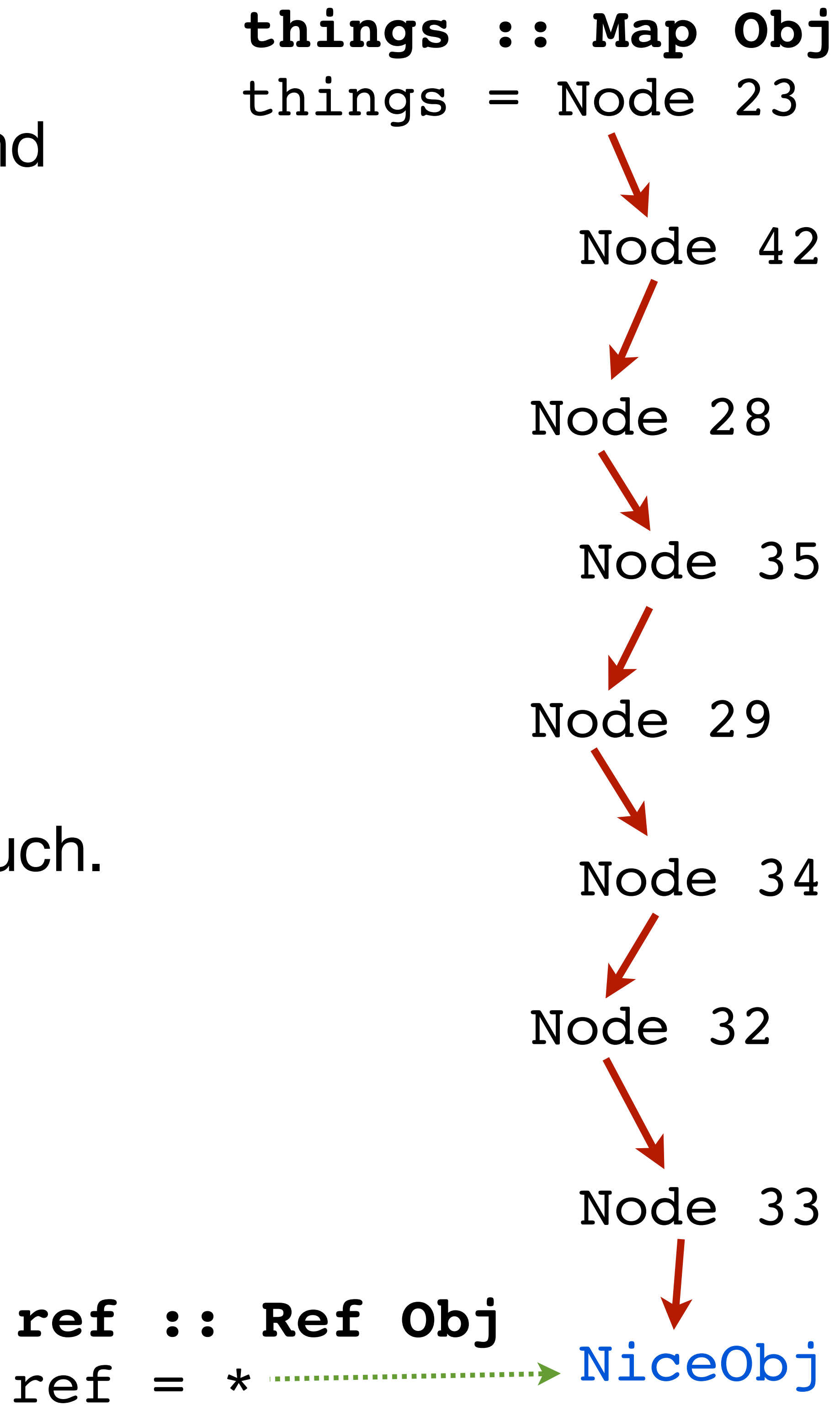
```
f x = g (do { x++; x }) x
```

Why destructive update matters

- Update plays a critical role in the abstraction and performance of code.

- To modify `NiceObj` purely we must:
  - know how the container works.
  - traverse the tree down to the desired node.
  - reallocate all nodes back to the root.

- Advanced data structures will only get us so much. `Data.Map` is a binary tree. For n = 1000, the tree is 10 levels deep.



```
things :: Map Obj
things = Node 23
```

Node 42

Node 28

Node 35

Node 29

Node 34

Node 32

Node 33

`NiceObj`

```
ref :: Ref Obj
ref = *
```

Haskell: pure(-ly) functional programming

- Deep in the heart of the GHC type inferencer...

```
data TcTyVarDetails
     = SkolemTv SkolemInfo
     | MetaTv   BoxInfo (IORef MetaDetails)
```

Let's not pretend that effects aren't needed to write real programs!

Uncontrolled side effects are bad news

- Bad for optimisation...

```
map f (map g xs) == map (f . g) xs
```

a rewrite to save constructing and collecting an entire list
but it only works when ‘f’ and/or ‘g’ are pure

Uncontrolled side effects are bad news

- ...and bad for code quality.

```
somethingHarmless :: String -> ()
```

does this write to the screen?
access the file system?
modify a global variable?
create shared state?
can I run it in parallel with X?
can it throw an exception?
allocate memory?
kill my dog?

Solution 1: Thread the world

- A phantom state token is passed around explicitly, providing the required data dependencies.

```
greet name s
 = let s2 = putStr ... s
       s3 = putStr ... s2
   in  s3
```

- Simple, easy to implement. Used in Clean.
- Tedious. Error prone. Not fun to program with.

Solution 1.1: State monads

- Hide the state threading behind a data type

```
type State s a = (s -> (s, a))

return :: a -> State s a
return x = λs. (s, x)


bind :: State s a -> (a -> State s b)
     -> State s b

bind m f = λs. let (s', x) = m s
               in  (f x) s'
```

Haskell programmers love monads...

- Syntactic sugar allows us to express uses of return/bind with do{..} notation.

```
f x
 = do y <- g x
      z <- h y
      return z
```

```
f x
 = g x `bind` λy.
   h y `bind` λz.
   return z
```

- We can use same structure to define exactly what sequencing means for other types too: Maybe, Lists, Parsers, Exceptions ...

.. but they leak into everything we do.

- A monadic computation as a different type from a “pure” one...

```
map  ::              (a ->   b) -> [a] ->   [b]
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

`filter` *vs* `filterM`   `lookup` *vs* `lookupM`

`zipWith` *vs* `zipWithM`

- They can have a substantial overhead at runtime.

  In C parlance: every semi-colon is now a function call.

- State monads over sequence non-interfering computations.

Full Circle: Make the state monad implicit

- Effect typing is used to determine what operations must be sequenced.

- Monad style:

```
putStr :: String -> IO ()
```

- Effect style:

```
putStr :: String -(!Console)> ()

putStr :: String -(!e1)> ()
          :- !e1 = !Console
```

Full Circle: Allow arbitrary destructive update

- Allow, but *track it carefully*.

```
updateInt
    :: forall %r1 %r2
    .  Int %r1 -> Int %r2 -(!e1)> ()
    :- !e1 = !{ !Read %r2; Write %r1 }
    ,  Mutable %r1
```

- Region constraints track what data is `Mutable` and what is `Const`

Higher order functions

- Effect variables reveal when function arguments might be called.

```
map :: forall a b %r1 %r2 !e1
    .  (a -(!e1)> b) -> List %r1 a -(!e2)> List %r2 b
    :- !e2 = { !Read %r1; !e1 }

map f []     = []
map f (x:xs) = f x : map f xs
```

Type elaboration

- The types contain lots of low level detail...
  ... but we usually don't have to bother with it.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- The extra effect and region information is orthogonal to the shape of the type. The compiler can fill this in behind the scenes.

- We need to specify it when importing foreign functions.

- We need to be aware of it when mixing laziness and side effects.

Even higher order functions

- The types of functions of order ≥ 3 have extra constraints on effect variables.

```
succ :: Int -> Int
succ x  = x + 1
third f = succ (f succ)

third
  :: forall %r0 %r1 %r2 %r3 !e0 !e1
  .  ((Int %r2 -(!e1)> Int %r3) -(!e0)> Int %r0) -(!e2)> Int %r1
  :- !e2  =  !{!e0; !Read %r0}
  ,  !e1  :> !Read %r2
```

Effect `!e1` is ‘at least’ `!Read %r2`

- When was the last time you used a 3rd order function?

- Not much test code around...

Fuzz testing for completeness issues...

- Lack of H.O test code necessitates automatic generation.

```
v4 = \v5 -> v5 23 (\v6 -> v6 (\v7 -> ()) ())
```

```
FREAKOUT in Core.Reconstruct
    applyTypeT: error in type application.
      in application: (\/ !eTC4 :> !{!eTC7; !eTC5} :: ! -> ...) (!PURE)
                type: !PURE
            is not :> !{!eTC7; !eTC5}

Inferred type for v4 was:

  v4 ::  forall tTC393 tTC399 v7 %rTC0
                !eTC1 !eTC3 !eTC4 !eTC5 !eTC7 !eTS0
                $cTC3 $cTC6 $cTS0 $cTS1 $cTS2 $cTS3
     .  (Int %rTC0 -(!eTC3 $cTS3)> (((v7 -(!eTS0 $cTC6)> Unit)
                   -(!eTC7 $cTS1)> Unit -(!eTC5 $cTS0)> tTC399)
                   -(!eTC4 $cTC3)> tTC399) -(!eTC1 $cTS2)> tTC393)
          -(!eTC0 $cTC0)> tTC393
     :- !eTC0   = !{!eTC3; !eTC1}
     ,  !eTC4   :> !{!eTC7; !eTC5}
     ,  $cTC0   = $cTC3 \ v5
```

mmmm... k?

Shape constraints

- If the type of `(==)` required its arguments to have the same type
  … then we couldn't compare `Mutable` with `Const` data.

```
(==) :: a -> a -> Bool
x    :: Int %r1 :- Mutable %r1
y    :: Int %r2 :- Const %r2

if x == y then ...
```

unification makes `%r1` == `%r2`
But the result can't be both
`Mutable` and `Const`

- The Shape constraint forces its arguments to have the same overall shape, but allows their regions to vary.

```
(==) :: a -> b -> Bool
     :- Shape2 a b
```

Explicit Laziness

- Disciple uses strict/call-by-value evaluation order by default
  Laziness in introduced explicitly. Thunks are forced implicitly.

```
mapLS :: (a -> b) -> [a] -> [b]
mapLS f []      = []
mapLS f (x:xs) = (:) (f x) (mapLS @ f xs)
```

(LS == lazy spine)

```
mapLE :: (a -> b) -> [a] -> [b]
mapLE f []      = []
mapLE f (x:xs) = (:) (f @ x) (mapLE f xs)
```

(LE == lazy elements)

- Lazy and Direct objects are interchangeable.
  Knowing that an object will *never* be a thunk is a big win for optimisation.

Purification of effects

- Suspending a function application purifies its visible effects.

```
suspend1 :: forall a b !e1
          .  (a -(!e1)> b) -> a -> b
          :- Pure !e1, LazyH b

succ     :: forall %r1 %r2
          .  Int %r1 -(!e1)> Int %r2
          :- !e1 = !Read %r1

lazySucc :: forall %r1 %r2
          .  Int %r1 -> Int %r2
          :- Const %r1, Lazy %r2

lazySucc x = suspend1 succ x
```

Closures track data sharing

```
fun :: forall %r1 %r2
    .  () -> () -($c1)> (Int %r1, Int %r2)
    :- $c1 = x : %r2

fun ()
 = let x        = 5
       inner () = (23, x)
   in  inner
```

‘`x`’ is shared between calls to inner

```
fun2 :: forall %r1
     .  () -($c1)> (Int %r1, Int %rS)
     :- $c1 = x : %rS

fun2 = fun ()
```

`%rS` is not quantified in the type for fun2.
it has global lifetime

Some type rules...

$$\boxed{\Gamma \vdash_{\mathrm{K}} T \ :: \ K}$$

$$\frac{\Gamma \vdash_{\mathrm{K}} a :: K \in \Gamma}{\Gamma \vdash_{\mathrm{K}} a \ :: \ K} \ (\text{TyVar})$$

$$\Gamma \vdash_{\mathrm{K}} D_1 \xrightarrow{E} D_2 \ :: \ * \quad (\text{TyFun}) \qquad \Gamma \vdash_{\mathrm{K}} \mathrm{C}_k \ r \ :: \ * \quad (\text{TyData})$$

$$\Gamma \vdash_{\mathrm{K}} (\forall (e \sqsupseteq E) :: !.\ D) \ :: \ * \quad (\text{TyAllB}) \qquad \Gamma \vdash_{\mathrm{K}} (\forall a :: K.\ D) \ :: \ * \quad (\text{TyAll})$$

$$\Gamma \vdash_{\mathrm{K}} \mathrm{Read}\ r \ :: \ ! \quad (\text{Read}) \qquad \Gamma \vdash_{\mathrm{K}} \mathrm{const}\ R \ :: \ \mathrm{Const}\ r \quad (\text{Const})$$

$$\Gamma \vdash_{\mathrm{K}} \mathrm{Write}\ r \ :: \ ! \quad (\text{Write}) \qquad \Gamma \vdash_{\mathrm{K}} \mathrm{mutable}\ r \ :: \ \mathrm{Mutable}\ r \quad (\text{Mutable})$$

$$\Gamma \vdash_{\mathrm{K}} \mathrm{pure}\ \bot \ :: \ \mathrm{Pure}\ \bot \quad (\text{Pure})$$

$$\frac{\Gamma \vdash_{\mathrm{K}} W \ :: \ \mathrm{Const}\ r}{\Gamma \vdash_{\mathrm{K}} \mathrm{purify}\ (\mathrm{Read}\ R)\ W \ :: \ \mathrm{Pure}\ (\mathrm{Read}\ R)} \ (\text{Purify})$$

$$\frac{\Gamma \vdash_{\mathrm{K}} W_1 \ :: \ \mathrm{Pure}\ E_1 \qquad \Gamma \vdash_{\mathrm{K}} W_2 \ :: \ \mathrm{Pure}\ E_2}{\Gamma \vdash_{\mathrm{K}} \mathrm{pjoin}\ W_1\ W_2 \ :: \ \mathrm{Pure}\ (E_1 \vee E_2)} \ (\text{PureJoin})$$

$$\boxed{\Gamma \vdash S \sqsubseteq T}$$

$$\Gamma \vdash T \sqsubseteq T \quad (\text{SubRefl}) \qquad \frac{\Gamma \vdash T_1 \sqsubseteq T_2 \qquad \Gamma \vdash T_2 \sqsubseteq T_3}{\Gamma \vdash T_1 \sqsubseteq T_3} \ (\text{SubTrans})$$

$$\Gamma \vdash S \sqsubseteq \top \quad (\text{SubTop}) \qquad \Gamma \vdash \bot \sqsubseteq T \quad (\text{SubBot})$$

$$\frac{\Gamma \vdash E_1 \sqsubseteq F \qquad \Gamma \vdash E_2 \sqsubseteq F}{\Gamma \vdash E_1 \vee E_2 \sqsubseteq F} \ (\text{Join1}) \qquad \frac{\Gamma \vdash E \sqsubseteq F_1}{\Gamma \vdash E \sqsubseteq F_1 \vee F_2} \ (\text{Join2})$$

$$\frac{\begin{array}{r} \Gamma \vdash T_1 \sqsubseteq S_1 \qquad \Gamma \vdash S_2 \sqsubseteq T_2 \\ \Gamma \vdash E_1 \sqsubseteq E_2 \end{array}}{\Gamma \vdash S_1 \xrightarrow{E_1} S_2 \ \sqsubseteq \ T_1 \xrightarrow{E_2} T_2} \ (\text{SubFun}) \qquad \frac{a \sqsupseteq E \in \Gamma}{\Gamma \vdash E \sqsubseteq a} \ (\text{SubVar})$$

$$\frac{\Gamma \vdash D_1 \sqsubseteq D_2}{\Gamma \vdash (\forall a :: K.\ D_1) \ \sqsubseteq \ \forall a :: K.\ D_2} \ (\text{SubAll})$$

$$\frac{\Gamma,\ a \sqsupseteq E \ \vdash D_1 \sqsubseteq D_2}{\Gamma \vdash \forall (a \sqsupseteq E) :: K.\ D_1 \ \sqsubseteq \ \forall (a \sqsupseteq E) :: K.\ D_2} \ (\text{SubAllB})$$

$$\boxed{\Gamma \vdash t \ :: \ T}$$

$$\frac{x :: T \in \Gamma}{\Gamma \vdash x \ :: \ T \ ; \bot} \ (\text{Var})$$

$$\frac{\Gamma,\ x_1 :: T_1 \vdash t_2 \ :: \ T_2 \ ; E}{\Gamma \vdash (\lambda x_1 :: T_1.\ t_2) \ :: \ T_1 \xrightarrow{E} T_2 \ ; \bot} \ (\text{Abs})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_1 \ :: \ T_{11} \xrightarrow{E_a} T_{12} \ ; E_1 \\ \Gamma \vdash t_2 \ :: \ T_2 \ ; E_2 \\ \Gamma \vdash T_2 \sqsubseteq T_{11} \end{array}}{\begin{array}{l} \Gamma \vdash t_1\ t_2 \\ \qquad :: \ T_{12} \ ; E_1 \vee E_2 \vee E_a \end{array}} \ (\text{App})$$

$$\frac{\Gamma[a_1 :: K_1] \vdash t_2 \ :: \ T_2 \ ; E_2}{\begin{array}{l} \Gamma \vdash \Lambda (a_1 \sqsupseteq T_1) :: K_1.\ t_2 \\ \quad :: \forall (a_1 \sqsupseteq T_1) :: K_1.\ T_2 \ ; E_2 \end{array}} \ (\text{AbsT})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_1 \ :: \ (\forall a_1 :: K_{11}.\ T_{12}) \ ; E_1 \\ \Gamma \vdash_{\mathrm{K}} T_2 \ :: \ K_{11} \end{array}}{\Gamma \vdash (t_1\ T_2) \ :: \ [a_1 \mapsto T_2]\ T_{11} \ ; E_1} \ (\text{AppT})$$

$$\frac{\begin{array}{c} \Gamma[x_1 :: T_1] \vdash t_1 \ :: \ T_1 \ ; E_1 \\ \Gamma[x_1 :: T_1] \vdash t_2 \ :: \ T_2 \ ; E_2 \end{array}}{\Gamma \vdash (\texttt{let}\ x_1 \texttt{ = } t_1\ \texttt{in}\ t_2) \ :: \ T_2 \ ; E_1 \vee E_2} \ (\text{Let})$$

$$\frac{\begin{array}{r} \Gamma[\overline{w_i :: W_i},\ r_1 :: \%] \vdash t_1 \ :: \ T_1 \ ; E_1 \\ \overline{W_i}\ \mathit{wellfounded} \\ r \notin \mathit{free}(T_1) \end{array}}{\Gamma \vdash (\texttt{letregion}\ r_1\ \{\overline{w_i :: W_i}\}\ \texttt{in}\ t_i) \ :: \ T_1 \ ; E_1} \ (\text{LetRegion})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_2 \ :: \ T_1 \ ; E_2 \\ \Gamma \vdash t_3 \ :: \ T_2 \ ; E_3 \qquad \Gamma \vdash t_1 \ :: \ \texttt{Bool}\ R_1 \ ; E_1 \end{array}}{\begin{array}{l} \Gamma \vdash (\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) \ :: \ T_1 \\ \quad ; E_1 \vee E_2 \vee \texttt{Read}\ R_1 \end{array}} \ (\text{IfThenElse})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_2 \ :: \ \texttt{C}_k\ R_2 \ ; E_2 \\ \Gamma \vdash t_3 \ :: \ \texttt{C}_k\ R_3 \ ; E_3 \qquad \Gamma \vdash_{\mathrm{K}} W_1 \ :: \ \texttt{Mutable}\ r_2 \end{array}}{\begin{array}{l} \Gamma \vdash (\texttt{update}_c\ W_1\ t_2\ t_3) \ :: \ () \\ \quad ; E_2 \vee E_3 \vee \texttt{Read}\ R_3 \vee \texttt{Write}\ R_2 \end{array}} \ (\text{Update})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_2 \ :: \ T_{21} \xrightarrow{E_2} T_{22} \ ; E_1 \\ \Gamma \vdash t_3 \ :: \ T_{21} \ ; E_2 \qquad \Gamma \vdash_{\mathrm{K}} W_1 \ :: \ \texttt{Pure}\ E_2 \end{array}}{\Gamma \vdash (\texttt{suspend}\ W_1\ t_2\ t_3) \ :: \ T_{22} \ ; E_1 \vee E_2} \ (\text{Suspend})$$

$$\frac{\Gamma \vdash_{\mathrm{K}} R_1 \ :: \ \%}{\Gamma \vdash (c_{\#}\ R_1) \ :: \ \texttt{C}\ R_1 \ ; \bot} \ (\text{Constant})$$

Demos