Programming in Omega
Part 3
Tim Sheard
Portland State University
Monotonic sorting functions

Sort :: [Nat’ n] -> Dss m

• Two problems
  – First, a list with type [Nat n] has elements all the same size, so sorting is unnecessary. What we want is
    • [ exists n . Nat’ n ]
  – Second we can’t know the size of the largest element of the sorted list in advance.

Sort :: [exists n . Nat’ n] -> (exists m . Dss m)
Covert Types

data Covert:: (Nat ~> *0) ~> *0 where
    Hide:: (t x) -> Covert t

inputList :: [Covert Nat']
inputList =
    [Hide #1, Hide #2, Hide #4, Hide #3]

msort :: [Covert Nat'] -> Covert Dss
Merge Sort

split [] pair = pair
split [x] (xs,ys) = (x:xs,ys)
split (x:y:zs) (xs,ys) = split zs (x:xs,y:ys)

msort :: [Covert Nat'] -> Covert Dss
msort [] = Hide Dnil
msort [Hide x] = Hide(Dcons x Base Dnil)
msort xs = let (y,z) = split xs ([],[]) 
            (Hide ys) = msort y
            (Hide zs) = msort z
            in case merge ys zs of
                Left z -> Hide z
                Right z -> Hide z
Test it!

inputList =
    [Hide #1, Hide #2, Hide #4, Hide #3]

ans = msort inputList

Hide (Dcons #4 (Step (Step (Step Base)))
    (Dcons #3 (Step (Step Base))
    (Dcons #2 (Step Base)
    (Dcons #1 Base Dnil))))
Problems

• Do we really need to store the \((\text{LE } a \ b)\) witness in the cons cell?
• It’s large, it’s costly to compute, and we must produce it at run-time
• Can we push these costs into compile-time activities
The prop declaration

```haskell
prop LE :: Nat ~> Nat ~> *0 where
  Base:: LE Z a
  Step:: LE a b -> LE (S a) (S b)
```

• Exactly like a data declaration. Introduces the type `LE` and the constructor functions `Base` and `Step`.

• `(LE a b)` is also introduced as a constraint like `(Eq Int)` or `(Show Bool)`.

• `prop` introduces the prolog like discharging rules:
  - `Base:: LE Z a`
  - `Step:: LE (S a) (S b) :- LE a b`

• These follow directly from the type of the constructor functions.
data Sss:: Nat ~> *0 where
  Snil:: Sss #0
  Scons:: LE a b => Nat' b -> Sss a -> Sss b

- We make the (LE a b) proof be a static obligation, that must be discharged at compile time
- Constrained type system just like Haskell

\[ \forall x y z \rightarrow \text{Scons} \ x \ \text{(Scons} \ y \ z) : : \ (\text{LE} \ a \ b, \text{LE} \ b \ c) \Rightarrow \text{Nat'} \ c \rightarrow \text{Nat'} \ b \rightarrow \text{Sss} \ a \rightarrow \text{Sss} \ c \]
Unit size witnesses

• Once we have static propositions we can build unit size witness objects.

```
data LE' :: Nat ~> Nat ~> *0
    where LE:: (LE m n) => LE' m n

le23 :: LE #2 #3
le23 = Step(Step Base)

Le23' :: LE' #2 #3
Le23' = LE
```
Unit size witness save space

compare :: Nat' a -> Nat' b -> Either (LE' a b) (LE' b a)

compare Z Z     = Left LE
compare Z (S x) =
    case compare Z x of
        Left LE -> Left LE
        Right LE -> Left LE
compare (S x) Z =
    case compare x Z of
        Right LE -> Right LE
        Left LE -> Right LE
compare (S x) (S y) =
    case compare x y of
        Right LE -> Right LE
        Left LE -> Left LE
How does it work?

```haskell
compare (a@(S x)) (b@(S y)) =
  case compare x y of
    Right (p@LE) -> Right LE
    Left LE    -> Left LE
```

- `a :: Nat' #(1+_c)`
- `b :: Nat' #(1+_d)`
- `x :: Nat' _c`
- `y :: Nat' _d`
- `compare x y :: Either(LE' _c _d)(LE' _d _c)`
- `p :: LE' _d _c`
Static Merging

merge2 :: Sss n -> Sss m -> Either(Sss n) (Sss m)
merge2 Snil ys = Right ys
merge2 xs Snil = Left xs
merge2 (a@(Scons x xs)) (b@(Scons y ys)) =
    case compare x y of
        Left LE -> case merge2 a ys of
                      Left ws -> R(Scons y ws)
                      Right ws -> R(Scons y ws)
        Right LE -> case merge2 b xs of
                      Left ws -> Left(Scons x ws)
                      Right ws -> Left(Scons x ws)
Static Sorting

`msort2 :: [Covert Nat'] -> Covert Sss`

`msort2 [] = Hide Snil`

`msort2 [Hide x] = Hide(Scons x Snil)`

`msort2 xs =
    let (y,z) = split xs ([],[])`  
    `(Hide ys) = msort2 y`  
    `(Hide zs) = msort2 z`  
    `in case merge2 ys zs of`  
    `    Left z -> Hide z`  
    `    Right z -> Hide z`

`ans2 = msort2 inputList`

`Hide (Scons #4 (Scons #3`

`(Scons #2 (Scons #1 Snil))))`
Logics and Languages

• Logical Languages
  – Logical part (quantifiers and connectives)
  – Extra-logical (constants, functions, predicates)
    • These are the domain of discourse in the logic

• Curry-Howard provides a good mechanism for the first part. But we often lack extra-logical operations that relate directly to the programs we are trying to reason about.

• GADT’s, Kinds, Witnesses, Singletons, are the extra-logical terms, and are semantically connected to the program.
Strategy

• Extend your favorite language (Haskell)
  – New constructs to encode propositions as types
    • GADTs (for example: \( \mathbb{E} (\mathbb{O} \mathbb{Z}) \) )
  – New constructs to build extra-logical operators that relate directly to the programs of interest
    • Extensible Kinds (for example: \( \text{odd} (1+1+1+0) \) )
  – New use of the constrained type system of Haskell to manage and solve constraints
    • Static propositions and constraint solving rules

• The logic and the language become 1 entity.
Benefits

• New constructs (GADTs and Kinds) provide a direct link between a program and its properties.
• Each of the new constructs has semantic meaning within the language.
  – The connection between the property and the program is not clouded by an imprecise encoding.
Benefits (continued)

- Management of constraints is performed inside the language, they cannot be lost, forgotten, mislaid, or forged.
- Constraint solving can be either dynamic (flexible) or static (efficient). The framework provides a mechanism for effortlessly sliding between the two mechanisms, even in the same program.
Pattern Review

- Indexed Datatypes (List a n)
- Witness types (LE n m)
- Singleton Types (Nat’ n)
- Dynamically Creating Witnesses (compare)
- One point types (LE’)
- Storing Proofs in Data (Dss)
- Using type functions to relate properties of inputs and outputs (app)
Other Examples we have done

- Typed, staged interpreters
  - For languages with binding, with patterns, algebraic datatypes

- Type preserving transformations
  - \texttt{Simplify} :: \texttt{Exp\ t} -> \texttt{Exp\ t}
  - \texttt{Cps} :: \texttt{Exp\ t} -> \texttt{Exp\ \{trans\ t\}}

- Proof carrying code

- Data Structures
  - Red-Black trees, Binomial Heaps, Static length lists

- Languages with security properties

- Typed self-describing databases, where meta data in the database describes the database schema

- Programs that slip easily between dynamic and statically typed sections. Type-case is easy to encode with no additional mechanism
Some other examples

• Typed Lambda Calculus
• A Language with Security Domains
• A Language which enforces an interaction protocol
Typed lambda Calculus
Exp with type $t$ in environment $s$

data $V:: *0 \rightarrow *0 \rightarrow *0$ where
  $Z:: V (t,m) t$
  $S:: (V m t) \rightarrow V (x,m) t$

data $Exp:: *0 \rightarrow *0 \rightarrow *0$ where
  $IntC:: Int \rightarrow Exp s Int$
  $BoolC:: Bool \rightarrow Exp s Bool$
  $Plus:: (Exp s Int) \rightarrow (Exp s Int) \rightarrow Exp s Int$
  $Lteq:: (Exp s Int) \rightarrow (Exp s Int) \rightarrow Exp s Bool$
  $Var:: (V s t) \rightarrow Exp s t$
Language with Security Domains

Exp with type $\tau$ in env $s$ in domain $d$

kind Domain = High | Low

data D $\tau$
  = Lo where $\tau$ = Low
  | Hi where $\tau$ = High

data Dless $x$ $y$
  = LH where $x$ = Low, $y$ = High
  | LL where $x$ = Low, $y$ = Low
  | HH where $x$ = High, $y$ = High

data Exp $s$ $d$ $\tau$
  = Int Int where $\tau$ = Int
  | Bool Bool where $\tau$ = Bool
  | Plus (Exp $s$ $d$ Int) (Exp $s$ $d$ Int) where $\tau$ = Int
  | Lteq (Exp $s$ $d$ Int) (Exp $s$ $d$ Int) where $\tau$ = Bool
  | forall $d2$. Var (V $s$ $d2$ $\tau$) (Dless $d2$ $d$)
Language with interaction protocol

Command with store \( \text{St} \) starting in state \( x \),
ending in state \( y \)

kind State = Open | Closed

data V s t
  = forall st . Z where s = (t, st)
  | forall st t1 . S (V st t)
    where s = (t1, st)

data Com st x y
  = forall t . Set (V st t) (Exp st t) where x=y
  | forall a . Seq (Com st x a) (Com st a y)
  | If (Exp st Bool) (Com st x y) (Com st x y)
  | While (Exp st Bool) (Com st x y) where x = y
  | forall t . Declare (Exp st t) (Com (t, st) x y)
  | Open where x = Closed, y = Open
  | Close where x = Open, y = Closed
  | Write (Exp st Int) where x = Open, y = Open
Next time

• Building structures to parameterize over for generic programming
Generic Programming in Omega
Part 3

Tim Sheard
Portland State University
What is Generic Programming?

• Generic programming is writing one algorithm that can run on many different datatypes.
• Saves effort because a function need only be written once and maintained in only one place.
• Examples:
  
  - `equal :: a -> a -> Bool`
  - `display :: a -> String`
  - `marshall :: a -> [Int]`
  - `unmarshall :: [Int] -> a`
Flavors of Generic programming

• I know of several different ways to implement Generics all of which depend on representing types as data

• Universal type embedding
• Shape based type embeddings
  • With isomorphism based equality proofs
  • With leibniz based equality proofs (Ralf Hinze’s example)
  • With Omega style Eq proofs
• Cast enabling embeddings

• In this world
  • equal :: Rep a -> a -> a -> Bool
  • display :: Rep a -> a -> String
  • marshall :: Rep a -> a -> [ Int ]
  • unmarshall :: Rep a -> [Int] -> a
Getting started

• We’ll start with the explicit Rep based approach where the representation type is passed as an explicit argument to generic functions

• How do we represent types as data?

• That depends in what you want to do with the types.
Ralf Hinze showed a shape based approach

• Declare a type that represent the shape of values

```haskell
data Type :: *0 ~> *0 where
    Int :: Type Int
    Char :: Type Char
    Unit :: Type ()
    Pair :: Type a -> Type b -> Type (a, b)
    Sum :: Type a -> Type b -> Type (a + b)
    List :: Type a -> Type [a]
    Type :: Type a -> Type (Type a)
    Dynamic :: Type Dynamic
    Typed :: Type a -> Type (Typed a)
```
Pair values with shapes

data Typed :: *0 ~> *0 where
    With :: Type a -> a -> Typed a

data Dynamic :: *0 where
    Dyn :: Typed t -> Dynamic

val (Dyn (With t r)) = r
typedef (Dyn (With t r)) = t
Type is a singleton

• Note that Type is a singleton type.
  – Only one element inhabits \((\text{Type } a)\).
  – The shape of that value is in 1-to-1 correspondance with its type index \(a\)
    • \text{Pair Int Char :: Type(Int,Char)}
  – If you know the type of \((x :: \text{Type } a)\), you know its shape.
  – You can discover the type of a value \((x :: \text{Type } a)\) by exploring its shape.
Inspect Shape to write generic functions

equal :: Type a -> a -> a -> Bool

equal Int x y = x==y
equal Char x y = eqStr [x] [y]
equal Unit () () = True
equal (Pair a b) (w,x) (y,z) =
    equal a w y && equal b x z
equal (Sum a b) (L x) (L y) = equal a x y
equal (Sum a b) (R x) (R y) = equal b x y
equal (Sum a b) _ _ = False
equal (List a) x y = equalL (equal a) x y

equalL f [] [] = True
equalL f (x:xs) (y:ys) = f x y && equalL f xs ys
Are two reps equal?

```haskell
data Equal:: *0 ~> *0 ~> *0 where
    Eq :: Equal a a

test :: Type a -> Type b -> Maybe (Equal a b)
test Int Int = return Eq
test Char Char = return Eq
test Unit Unit = return Eq
test (Pair x y) (Pair a b) =
    do { Eq <- test x a; Eq <- test y b; return Eq }
test (Sum x y) (Sum a b) =
    do { Eq <- test x a; Eq <- test y b; return Eq }
test (List x) (List y) =
    do { Eq <- test x y; return Eq }
test _ _ = Nothing
```

Really exploiting the singleton properties here!
Explore the type checking

test (Pair x y) (Pair a b) =
  check
  do { Eq <- test x a
       ; Eq <- test y b
       ; check (return Eq) }
*** Check: do { (p1 @ Eq) <- test x a  
    ; (p2 @ Eq) <- test y b  
    ; Check (return Eq)}
*** expected type: Maybe (Equal (c+d) (e+f) )
check> x
x :: Type c
check> y
y :: Type d
check> a
a :: Type e
check> b
b :: Type f
check> :q

*** Check: return Eq
*** expected type: Maybe (Equal (c+d) (c+d) )
check> x
x :: Type c
check> y
y :: Type d
check> a
a :: Type c
check> b
b :: Type d
check> p1
p1 :: Equal c c
check> p2
p2 :: Equal d d
Universal Embedding Approach

- Let there be a universal type that can encode all values of the language (a datatype that uses dynamic tags (constructor functions) to distinguish different kinds of values. Embedding Lisp into Omega.

```haskell
data Val
  = Vint Int             -- basic types
    | Vchar Char
    | Vunit
    | Vfun (Val -> Val)     -- functions
    | Vdata String [Val]    -- data types
    | Vtuple [Val ]         -- tuples
    | Vpar Int Val
```
Interesting functions

• Note there are several interesting functions on the universal domain Value

• Equality
• Mapping
• Showing
Show

```haskell
plist sh start xs sep end = start++ f xs ++ end
  where f [] = ""
    f [x] = sh x
    f (x:xs) = sh x ++ sep ++ f xs

showV :: Val -> String
showV (Vint n) = show n
showV (Vchar c) = show c
showV Vunit = "()"
showV (Vfun f) = "fn"
showV (Vdata s []) = s
showV (Vdata s xs) =
  "(" ++ s ++ plist showV " " xs " " ")"
showV (Vtuple xs) = plist showV "(" xs "," ")"
showV (Vpar n x) = showV x
```
Equality

equal (Vint n) (Vint m) = n==m
equal (Vchar n) (Vchar m) = eqStr [n] [m]
equal Vunit Vunit = True
equal (Vdata s xs) (Vdata t ys)
    = eqStr s t && equalL xs ys
equal (Vtuple xs) (Vtuple ys)
    = equalL xs ys
equal (Vpar n x) (Vpar m y) = equal x y
equal _ _ = False

equalL [] [] = True
equalL (x:xs) (y:ys) =
    equal x y && equalL xs ys
equalL _ _ = False
Mapping

mapVal :: (Val -> Val) -> Val -> Val
mapVal f (Vpar n a) = Vpar n (f a)
mapVal f (Vint n) = Vint n
mapVal f (Vchar c) = Vchar c
mapVal f Vunit = Vunit
mapVal f (Vfun h) =
  error "can't mapVal Vfun"
mapVal f (Vdata s xs) =
  Vdata s (map (mapVal f) xs)
mapVal f (Vtuple xs) =
  Vtuple(map (mapVal f) xs)
Generic Functions

• Strategy:

3) Push (or pull) values into (out of) the universal domain.

5) Then manipulate the “typeless” data

7) Pull the result (if necessary) out of the universal domain.
A Rep is a pair of functions

```haskell
data Rep t = Univ (t -> Val) (Val -> t)
```

• We represent a type \( t \) by a pair of functions that inject and project from the universal type.

• Property \((\text{Univ } f \ g) :: \text{ Rep } t\)

• For all \( x :: t \) . \( g(f \ x) == x \)

• Functions

\[
\text{into } (\text{Univ } f \ g) = f \\
\text{out } (\text{Univ } f \ g) = g
\]
Example Reps

\[
\begin{align*}
\text{int} & = \text{Univ Vint} (\lambda (\text{Vint } n) \to n) \\
\text{char} & = \text{Univ Vchar} (\lambda (\text{Vchar } c) \to c) \\
\text{unit} & = \text{Univ (const Vunit) (const ())}
\end{align*}
\]

\[
\begin{align*}
\text{pair} :: (\text{Rep } a) \to (\text{Rep } b) \to \text{Rep } (a,b) \\
\text{pair } (\text{Univ } \text{to1 from1}) (\text{Univ } \text{to2 from2}) & = \text{Univ } f \ g \\
\text{where } f & (x,y) = \text{Vtuple}[\text{to1 } x, \text{to2 } y] \\
\text{g } (\text{Vtuple}[x,y]) & = (\text{from1 } x, \text{from2 } y)
\end{align*}
\]

\[
\begin{align*}
\text{arrow } r1 \ r2 & = \text{Univ } f \ g \\
\text{where } f \ h & = \text{Vfun}(\text{into } r2 \ . \ h \ . \ \text{out } r1) \\
\text{g } (\text{Vfun } h) & = \text{out } r2 \ . \ h \ . \ \text{into } r1
\end{align*}
\]
Datatype Reps - List

list (Univ to from) = Univ h k

where

\[ h \;[] = \text{Vdata} \; "[]" \; [] \]
\[ h \;(x:xs) = \text{Vdata} \; ";" \; [ \;\text{Vpar} \;1 \; (\text{to} \; x)\; , \; h \; xs \;] \]
\[ k \; (\text{Vdata} \; "[]" \; []) = [] \]
\[ k \; (\text{Vdata} \; ";" \; [\; \text{Vpar} \;1 \; x, \; xs \;]) = (\text{from} \; x) \; : \; k \; xs \]
Datatype Reps - Either

either (Univ to1 from1) (Univ to2 from2) = Univ h k
where
  h (Left x) = Vdata "Left" [Vpar 1 (to1 x)]
  h (Right x) = Vdata "Right" [Vpar 2 (to2 x)]
  k (Vdata "Left" [Vpar 1 x]) = Left (from1 x)
  k (Vdata "Right" [Vpar 2 x]) = Right (from2 x)
marshall (Univ to from) x =
    reverse (flat (to x) [])

flat :: Val -> [Int] -> [Int]
flat (Vint n) xs = n : 1 : xs
flat (Vchar c) xs = ord c : 2 : xs
flat Vunit xs = 3 : xs
flat (Vfun f) xs = error "no Vfun in marshall"
flat (Vdata s zs) xs =
    flatList zs (length zs : (flatString s (5 : xs)))
flat (Vtuple zs) xs =
    flatList zs (length zs : 6 : xs)
flat (Vpar n x) xs = flat x (7 : xs)
Helper functions

```haskell
flatList [] xs = xs
flatList (z:zs) xs =
    flatList zs (flat z xs)

flatString s xs =
    (reverse (map ord s)) ++
    ((length s) : xs)

unflatList 0 xs = ([],xs)
unflatList n xs = (v:vs,zs)
    where (v,ys)= unflat xs
        (vs,zs) = unflatList (n-1) ys
```
unmarshall (Univ to from) xs = from j
   where (j,ks) = (unflat xs)

unflat :: [Int] -> (Val,[Int])
unflat (1: x : xs) = (Vint x,xs)
unflat (2: x : xs) = (Vchar (chr x),xs)
unflat (3: xs) = (Vunit,xs)
unflat (5: xs) = (Vdata s ws,zs)
   where (s,n : ys) = unflatString xs
   (ws,zs) = unflatList n ys
unflat (6: n : xs) =
   (Vtuple ws,zs) where (ws,zs) = unflatList n xs
unflat (7:n: xs) = (Vpar n x,ys)
   where (x,ys) = unflat xs
unflat zs =    error ("Bad unMarshal Case"++ show zs)
gmap :: Rep b -> Rep c ->
(foreall a . Rep a -> Rep(t a)) ->
(b -> c) -> t b -> t c

gmap repB repC t f x =
  out repLC (help (into repLB x))
where repLB = t repB
  repLC = t repC
help xs = mapVal trans xs
trans x =
  into repC (f(out repB x))
But is this safe?

- Recall that anything can be made into a value
- Not all values actually represent real things
- There is “junk” in values
- Use GADTs to get rid of the junk
Type indexed Values

```haskell
data Constr :: *0 ~> *0 where
  Con :: a -> String -> Constr a
  A:: Constr(a -> b) -> Value a -> Constr b

data Value:: *0 ~> *0 where
  IntV:: Int -> Value Int
  CharV:: Char -> Value Char
  UnitV:: Value ()
  PairV :: Value a -> Value b -> Value (a,b)
  ParV:: Int -> Value a -> Value a
  ConV:: Constr a -> Value a
  FunV:: Rep2 a -> (Value a -> Value b)
       -> Value(a -> b)
```
to and from Values

data Rep2 t = Inject (t -> Value t)

to:: Rep2 t -> t -> Value t
    to (Inject f) x = f x

from:: Value a -> a
    from (IntV n) = n
    from (CharV c) = c
    from UnitV = ()
    from (PairV x y) = (from x, from y)
    from (ConV (Con x s)) = x
    from (ConV (A c v)) = from (ConV c) (from v)
    from (FunV (Inject inj) f) = \ v -> from (f(inj v))
A rep is just an injection

• Before, a (Rep a) was an injection and a projection.
• Now only an injection is needed as projection come for free.
Sample Reps

\[
\begin{align*}
\text{int2} &= \text{Inject} \ \text{IntV} \\
\text{char2} &= \text{Inject} \ \text{CharV} \\
\text{unit2} &= \text{Inject} \ (\text{const} \ \text{UnitV}) \\
\text{list2} \ (\text{Inject} \ f) &= \text{Inject} \ g \\
&\text{where} \\
&\quad g \ [] = \text{ConV} \ (\text{Con} \ [] \ "[]") \\
&\quad g \ (x:xs) = \\
&\quad \quad \text{ConV} \ ((\text{Con} \ (:) \ "::") \ `\text{A}` \ (f \ x) \ `\text{A}` \ (g \ xs))
\end{align*}
\]
Generic equality

equal2 :: Rep2 a -> a -> a -> Bool
equal2 (Inject f) x y = eq (f x) (f y)

eq:: Value a -> Value a -> Bool
eq (IntV n) (IntV m) = n==m
eq (CharV n) (CharV m) = eqStr [n] [m]
eq UnitV UnitV = True
eq (ConV x) (ConV y) = eqCon x y
eq (PairV w x) (PairV y z) = eq w y && eq x z
eq (ParV n x) (ParV m y) = eq x y
eq _ _ = False

eqCon:: Constr a -> Constr a -> Bool
eqCon (Con _ x) (Con _ y) = eqStr x y
eqCon (A w x) (A y z) = eqCon w y && eq x z
Overhead

• Note that both the Shape based approach and the Universal embedding approach involve a level of interpretation
  – The universal value approach has more overhead than the shape based approach

• Can we remove the interpretation?
• Stage the generic program

• \texttt{equal :: Type } a \rightarrow \texttt{Code(a } \rightarrow \texttt{a } \rightarrow \texttt{Bool)}
Code in Omega

- Omega supports a notion of code
- Programmers can generate code at runtime
- They can also execute the code they generate
- Programmers annotate their program to express when they are generating code and when they are directing the generator.
Staging Annotations

<table>
<thead>
<tr>
<th>notation</th>
<th>pronounced</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>[l _ l]</td>
<td>brackets</td>
<td>(build code)</td>
</tr>
<tr>
<td>$ _</td>
<td>escape</td>
<td>(splice in code)</td>
</tr>
<tr>
<td>lift _</td>
<td>lift</td>
<td>(turn values into code)</td>
</tr>
<tr>
<td>run _</td>
<td>run</td>
<td>(execute runtime code)</td>
</tr>
</tbody>
</table>
Simple example

trip :: (Int,(Code Int,Code Int))
trip = (3+4,[| 3+4 |], lift (3+4))

f :: (a,(Code Int,b)) -> Code Int
f (x,y,z) = [| 8 - $y |]

code :: Code Int
code = f trip

ans = run code

```cpp
prompt> trip
(7,([| 3 + 4 |],[| 7 |])) :: (Int,(Code Int,Code Int))
prompt> f
<fn> :: (forall a b . (a,(Code Int,b)) -> Code Int)
prompt> code
[| 8 - 3 + 4 |] :: Code Int
prompt> ans
1 :: Int
```
Larger Example

\[
mult :: \text{Code Int} \rightarrow \text{Int} \rightarrow \text{Code Int}
mult x n =
    \begin{cases}
    & \text{if } n == 0 \text{ then } ||1|| \\
    & \text{else } || x \times (\text{mult } x (n-1)) ||
    \end{cases}
\]

\[
cube :: \text{Code (Int} \rightarrow \text{Int})
cube = || \ \ y \rightarrow (\text{mult } ||y|| 3) ||
\]

\[
exponent :: \text{Int} \rightarrow \text{Code (Int} \rightarrow \text{Int})
exponent n = || \ \ y \rightarrow (\text{mult } ||y|| n) ||
\]
Generic Programming?

```
prompt> exponent 4
[| \ y -> y * y * y * y * 1 |
  Code (Int -> Int)
```

prompt>
Lets combine generics and staging

• First we’ll write an unstaged generic program
• Then in a second pass we’ll add staging annotations
Add up all the Ints

```haskell
sumR :: Type a -> a -> Int
sumR Int n = n
sumR Char c = 0
sumR Unit () = 0
sumR (Pair r s) (x,y) = sumR r x + sumR s y
sumR (List a) [] = 0
sumR (List a) (x:xs) = sumR a x + sumR (List a) xs
sumR _ x = 0
```
Testing

t1 = List(Pair Int Char)
x1 = [(5,'z'),(2,'z'),(3,'w')]

prompt> sumR t1 x1
10 : Int
sum2 :: Type a -> Code a -> Code Int
sum2 Int n = n
sum2 Char c = [|| 0 ||]
sum2 Unit _ = [|| 0 ||]
sum2 (Pair r s) x =
    [|| $(sum2 r [|| fst $x ||]) +
     $(sum2 s [|| snd $x ||]) ||]
sum2 (List a) xs =
    [|| if null $xs
        then 0
        else $(sum2 a [|| hd $xs ||]) +
              $(sum2 (List a) [|| tl $xs ||]) ||]
sum2 _ x = [|| 0 ||]
Test it

t2 = Pair Int (Pair Char Int)

prompt> [\ x -> $(sum2 t2 [\ x |]) ]

[\ x -> %fst x + 0 + %snd (%snd x) ]
: Code ((Int,(Char,Int)) -> Int)

BUT

t1 = List(Pair Int Char)
prompt> [\ x -> $(sum2 t1 [\ x |]) ]

Never returns !!
Oops!

\[
\text{sum2} \ (\text{List} \ a) \ \text{xs} = \\
[\ | \ \text{if null} \ \text{xs} \\
\text{then} \ 0 \\
\text{else} \ (\text{sum2} \ a [\ | \ \text{hd} \ \text{xs} \ |]) + \\
(\text{sum2} \ (\text{List} \ a) [\ | \ \text{tl} \ \text{xs} \ |])]
\]
Second try

\[
\text{sum2 (List a) xs =}
\begin{array}{l}
[| \text{let } f [] = 0 \\
\quad f (z:zs) = $(\text{sum2 a [| z |]) +} \\
\quad \quad f \ zs \\
\quad \text{in } f \ $xs |]
\end{array}
\]
Test it

prompt> [| \ x -> $(sum2 t1 [| x |]) |]

[| \ x34 ->
    let f38 [] = 0
    f38 (z39 : zs40) =
        %fst z39 + 0 + f38 zs40
    in f38 x34 |]

: Code ([(Int,Char)] -> Int)
Next time

• Richer examples
• We’ll try and write something on the fly
Generic Programming in Omega
Part 4

Tim Sheard
Portland State University
Let's abstract over something new

- So far all our generic programs are abstractions over types. Let's abstract over something else

- Build an indexed specification type
  - `Spec t`

- Write a type function that relates the specification index to a type
  - `{result t} = ...

- Generate code with that type given a spec as input.
  - `gen:: Spec t -> {result t}`
N-ary objects

• Consider the family of functions

\ x  \rightarrow  x
\ x  \rightarrow  \ y  \rightarrow  x+y
\ x  \rightarrow  \ y  \rightarrow  \ z  \rightarrow  x+y+z
\ x  \rightarrow  \ y  \rightarrow  \ z  \rightarrow  \ w  \rightarrow  x+y+z+w
type function

• Define a type function

sumTy :: Nat ~> *0
{sumTy Z} = Int
{sumTy (S n)} = Int -> {sumTy n}

{sumTy #4}
Int -> Int -> Int -> Int -> Int
Write a function

\[
\text{nsum} :: \text{Nat'} \ n \to \text{Int} \to \{\text{sumTy} \ n\}
\]
\[
\text{nsum} \ Z \ x = x
\]
\[
\text{nsum} \ (S \ m) \ x = \\lambda \ y \to \text{nsum} \ m \ (x+y)
\]

- Note we can apply test sum to a different number of arguments.
  \[
  \text{testsum} = (\text{nsum} \ #2 \ 0 \ 4 \ 5) == (\text{nsum} \ #1 \ 0 \ 9)
  \]

- But
  \[
  \text{prompt}> \text{nsum} \ #2 \ 2 \ 3 \ 4 \ 5
  \]
  The equations:
  \[
  \{\{\text{sumTy} \ #2\} == (\text{Int} \to \text{Int} \to \text{Int} \to a)\}
  \]
  have no solution
nsumG :: Nat' n -> Code Int -> Code {sumTy n}
nsumG Z x = x
nsumG (S n) x =
    [ | \ y -> $(nsumG n [ | $x + y | ]) | ]
Test it

testsumG = [\ y -> \ (nsumG #2 [\ y \]) ]

prompt> testsumG
[\ y4 -> \ y6 -> \ y8 -> y4 + y6 + y8 ]
: Code (Int -> Int -> Int -> Int)
Now for something completely different

- Goal – Use Omega types to write array package where the types prevent out of bounds access errors.

```haskell
data Vector:: *0 ~> Nat ~> *0 where
  Snil :: Vector a Z
  Scons:: a -> Vector a n -> Vector a (S n)

access :: Indx i n -> Vector a n -> a
loop:: Indx start n -> (Indx i n -> a) -> a
   -- for i = start, n do f
```
What is a bounded index?

prop LT :: Nat ~> Nat ~> *0 where
  LtZ :: LT Z (S x)
  LtS :: LT n m -> LT (S n) (S m)

data Indx i n =
  In (Nat' i) (LT i n) (Nat' n)

i0 = In Z LtZ #4
i1 = In (S Z) (LtS LtZ) #4
i2 = In (S (S Z)) (LtS (LtS LtZ)) #4
Not the types

prompt> i0
(In #0 LtZ #4) : Indx #0 #4

prompt> i1
(In #1 (LtS LtZ) #4) : Indx #1 #4

prompt> i2
(In #2 (LtS (LtS LtZ)) #4) : Indx #2 #4
Access function

access :: Indx i n -> Vector a n -> a
access (In Z LtZ _) (Scons x xs) = x
access (In (S n) (LtS p) (S q)) (Scons x xs)
    = access (In n p q) xs

All other cases are unreachable!
Something different again

- Representing kind indexed Type representations

- Idea

- Build a GADT in Omega which has two indexes, one a kind, and another a type indexed by that kind.
data Rep :: forall (k:: *2)(t::k) .
          (k ~> Row HasKind ~> t ~> *0) where
  Int  :: Rep *0 env Int
  Char :: Rep *0 env Char
  Unit :: Rep *0 env ()
  Pair :: Rep (*0 ~> *0 ~> *0) env (,)
  Sum  :: Rep (*0 ~> *0 ~> *0) env (+)
  Arr  :: Rep (*0 ~> *0 ~> *0) env (->)
  Ap   :: forall (a:: *1) (env:: Row HasKind) f x .
           Rep (*0 ~> a) env f ->
           Rep *0 env x -> Rep a env (f x)
Now for Abstractions

A list type at the type level

kind Row a = RCons a (Row a) | RNil

An environment type

kind HasKind:: *1 where
  HK:: forall (k:: *2)(t:: k) .
      Tag ~> k ~> t ~> HasKind

data Rep :: forall (k:: *2)(t::k) .
  (k ~> Row HasKind ~> t ~> *0) where
  Var:: forall(k2:: k)(t2:: k2)(l:: Tag)(env:: Row HasKind).
      Label l -> Rep k2 (RCons (HK l k2 t2) env) t2
prompt> Var `z
(Var `z) : Rep b {HK `z b c; d} c

prompt> Ap (Ap Pair Int) (Var `z))
(Ap (Ap Pair Int) (Var `z)) :
  Rep * {HK `z * a; b} (Int,a)
Now Abstractions

A:: forall (k2:: k) (t3:: k2) (t2:: k2)(l:: Tag) (env:: Row HasKind) .
Rep *0 (RCons (HK l k2 t2) env) t3 ->
Rep *0 env t3