Programming in Omega
Part 2

Tim Sheard

Portland State University

Recall

- Introduce new kinds
- Use GADTs to build indexed data
- Indexed data can supports
  - singleton types (`Nat'`)
  - relationships between types (`LE`)
- Types can be used to witness true properties of types (`Proof`)
- Use type functions to relate invariants of inputs and outputs
  - `app2:: Seq a n -> Seq a m -> Seq a {plus n m}`
- Type checking is constraint solving
  - **usually solving equations between type functions**

Today

- Leibniz Equality
- Using witnesses
  - To describe properties of data
  - Computing witness objects at run time.
  - Exploring the structure of singletons to build witnesses to properties
- Staging
- A large example: Balanced trees – AVL trees
- Use Syntactic Extension to make things readable.

Leibniz Equality

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

A single polymorphic constructor

Equal is a GADT, even though its range is all type variables, because both the type variables are the same.

Examples

```
Eq :: Equal Int INt
Eq:: Equal (S Z) {plus Z (S z)}
```

interesting arguments to Equal usually involve type functions

Dynamically Computing Witnesses

```
comp:: Nat' a ->
       Nat' b ->
       Either (LE a b) (LE b a)

comp Z Z     = Left Base
comp Z (S x) = Left Base
comp (S x) Z = Right Base
comp (S x) (S y) = case comp x y of
                     Right p -> Right (Step p)
                     Left p -> Left (Step p)
```

Putting witness types to work

- By storing witness types in data structures we can enforce invariants on those structures. Consider Dynamic Sorted Sequences

```
data Dss:: Nat ~> *0 where
   Dnil:: Dss Z
   Dcons:: (Nat' n) -> (LE m n)
             -> (Dss m) -> Dss n
```

A sequence of type `(Dss n)` has largest (and first) element of size `n`

Making use of Comp

```
merge :: Dss n -> Dss m -> Either (Dss n) (Dss m)
merge Dnil ys = Right ys
merge xs Dnil = Left xs
merge a@(Dcons x px xs)
      b@(Dcons y py ys) =
  case comp x y of
    Left p -> case merge a ys of
                Left ws -> Right(Dcons y p ws)
                Right ws -> Right(Dcons y py ws)
    Right p -> case merge b xs of
                 Left ws -> Left(Dcons x p ws)
                 Right ws -> Left(Dcons x px ws)
```

Exercise 16

- Singleton types allow us to construct Equal objects at run time.Because of the one-to-one relationship between singleton values and their types, knowing the shape of a value determines its type. In a similar manner knowing the type of a singleton determines its shape. Write the function in Omega that exploits this fact: I have written the first clause. You can finish it.

```
sameNat:: Nat' a -> Nat' b -> Maybe(Equal a b)
sameNat Z Z = Just Eq
```

Computing programs simultaneously with their properties

```
app1:: Seq a n -> Seq a m ->
       exists p . (Seq a p,Plus n m p)
```

exists specifies an existential type.
p is a concrete but unknown type
of kind Nat

```
app1 Snil ys = Ex(ys,PlusZ)
app1 (Scons x xs) ys =
   case (app1 xs ys) of
     Ex(zs,p) ->  Ex(Scons x zs,PlusS p)
```

Ex is the pack operator of Cadelli, used in a
pattern match it is the unpack operator
It turns a normal type: (Seq a p,Plus n m p) into
an existential one: exists p.(Seq a p,Plus n m p)

Exercise 17

- The filter function drops some elements from a list. Thus, the length of the resulting list cannot be known statically. But, we can compute the length of the resulting list along with the list. Write the Omega function with type:

```
filter :: (a->Bool) -> Seq a n ->
          exists m . (Nat' m,Seq a m)
```

- Since filter never adds elements to the list, that weren't already in the list, the result-list is never shorter than the original list. We can compute a proof of this fact as well. Write the Omega function with type:

```
filter :: (a->Bool) -> Seq a n ->
          exists m . (LE m n,Nat' m,Seq a m)
```

- Hint: You may find the functions `predLE` and `trans` from Exercises 13 and 14 useful.

Unreachable Clauses

```
smaller :: Proof {le (S a) (S b)} -> Proof {le a b}
smaller Triv = Triv

diff:: Proof {le a b} -> Nat' a -> Nat' b ->
       exists c .(Nat' c,Equal {plus a c} b)
diff Triv Z m = Ex (m,Eq)
diff Triv (S m) Z = unreachable
diff (q@Triv) (S x) (S y) =
  case diff (smaller q) x y of
   Ex (m,Eq) -> Ex (m,Eq)
```

This clause requires
a = (s m)
b = Z
and (S m) ≤ Z
a contradiction

Staging

brackets build code

```
inc x = x + 1
c1a = [| 4 + 3 |]

c2a = [| \ x -> x + $c1a |]


c3 = [| let f x = y - 1
              where y = 3 * x
        in f 4 + 3 |]
c4 = [| inc 3 |]
c5 = [| [| 3 |] |]
c6 = [| \ x -> x |]
```

The escape $, splices previously existing code (c1a) into the hole in the brackets marked by $c1a

An example

- count 0 = []
- count n = n: count (n-1)

- count' 0 = [| [] |]
- count' n = [| n : $(count' (n-1)) |]

Exercise 18

- The traditional staged function is the power function. The term **`(power 3 x)`** returns x to the third power. The unstaged power function can be written as:

```
power:: Int -> Int -> Int
power 0 x = 1
power n x = x * power (n-1) x
```

Write a staged power function:

```
pow:: Int -> Code Int -> Code Int
```

such that **`(pow 3 [|99|])`** evaluates to

`[| 99 * 99 * 99 * 99 * 1 |]`.

This can be written simply by placing staging annotations in the unstaged version.

Balanced trees

- Binary search trees can provide (log n) performance for both search and insertion, provided the trees are balanced.

AVL (all caps) is the type that users see

```
data AVL:: *0 where
  AVL:: (Avl h) -> AVL
```

An Avl (first letter Capital) is indexed by its height. We define it later.

Empty trees and has element

```
empty :: AVL
empty = AVL Tip

element :: Int -> AVL -> Bool
element x (AVL t) = elem x t

elem :: Int -> Avl h -> Bool
elem x Tip = False
elem x (Node _ l y r)
  | x == y  = True
  | x < y   = elem x l
  | x > y   = elem x r
```

creating the empty tree takes constant time

Finding an element runs in (log n = height) time if the tree is balanced

Height indexed trees

```
data Avl:: Nat ~> *0 where
  Tip:: Avl Z
  Node:: Balance i j k ->
         Avl i -> Int -> Avl j -> Avl (S k)
```

The index is the height of the tree

Balance is a witness to the legal sub-heights that can make a tree balanced

```
data Balance:: Nat ~> Nat ~> Nat ~> *0 where
  Same :: Balance n n n
  Less :: Balance n (S n) (S n)
  More :: Balance (S n) n (S n)
```

Insertion may change the height, but maintains balance!

`ins :: Int -> Avl n -> (Avl n + Avl (S n))`

```
ins x Tip = R(Node Same Tip x Tip)
ins x (Node bal lc y rc)
  | x == y = L(Node bal lc y rc)
  | x < y  = case ins x lc of
               L lc -> L(Node bal lc y rc)
               R lc ->
                 case bal of
                   Same -> R(Node More lc y rc)
                   Less -> L(Node Same lc y rc)
                   More -> rotr lc y rc -- rebalance
  | x > y  = case ins x rc of
               L rc -> L(Node bal lc y rc)
               R rc ->
                 case bal of
                   Same -> R(Node Less lc y rc)
                   More -> L(Node Same lc y rc)
                   Less -> rotl lc y rc -- rebalance
```

Study one case

```
ins x (Node bal lc y rc)
| x < y  =
    case ins x lc of
      L lc -> L(Node bal lc y rc)
      R lc ->
        case bal of
          Same -> R(Node More lc y rc)
          Less -> L(Node Same lc y rc)
          More -> rotr lc y rc
```

insert in left sub-tree

same size result

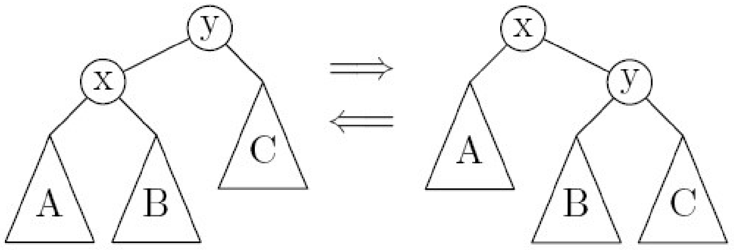
result height is one larger

it used to be the same, but one larger is ok, provided we mark it so.

rc is two deeper than lc, some rebalancing is required

tree rotation

- Tree rotation maintains the search invariant
- Tree rotation changes the height of the sub trees.



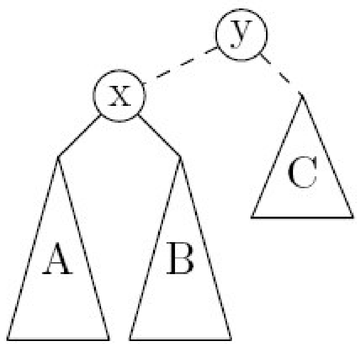

```
rotr :: Avl (2+n)t ->
        Int ->
        Avl n ->
       (Avl(2+n)t + Avl (3+n)t)
```

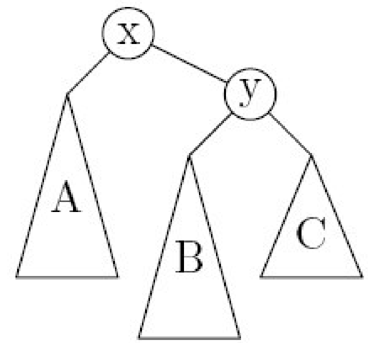
```
rotr :: Avl (2+n)t -> Int -> Avl n -> (Avl(2+n)t + Avl (3+n)t)

rotr Tip u a = unreachable
rotr (Node Same b v c) u a =
   R(Node Less b v (Node More c u a))
rotr (Node More b v c) u a =
   L(Node Same b v (Node Same c u a))
rotr (Node Less b v Tip) u a = unreachable
rotr (Node Less b v (Node Same x m y)) u a =
   L(Node Same (Node Same b v x) m (Node Same y u a))
rotr (Node Less b v (Node Less x m y)) u a =
   L(Node Same (Node More b v x) m (Node Same y u a))
rotr (Node Less b v (Node More x m y)) u a =
   L(Node Same (Node Same b v x) m (Node Less y u a))
```

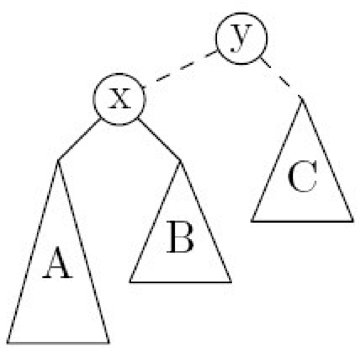
Rotation 1

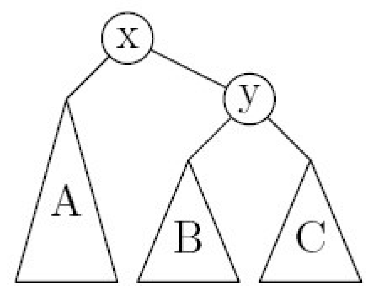
```
rotr (Node Same a x b) y c      = R(Node Less a x (Node More b y c))
```





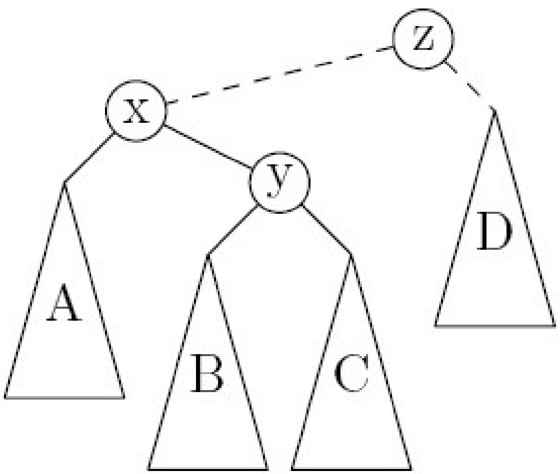

```
rotr (Node More a x b) y c      = L(Node Same a x (Node Same b y c))
```

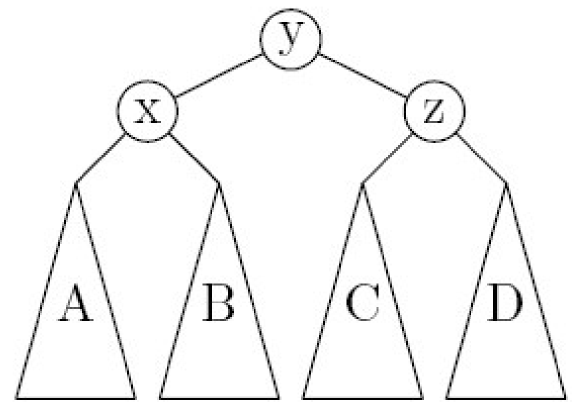




Rotation 2

```
rotr (Node Less a x
          (Node Same b y c)) z d  =  L(Node Same (Node Same a x b) y
                                                 (Node Same c z d) )
```





Exercise 22

- A red-black tree is a binary search tree with the following additional invariants:
  - Each node is colored either red or black
  - The root is black
  - The leaves are black
  - Each Red node has Black children
  - For all internal nodes, each path from that node to a descendant leaf contains the same number of black nodes.
- We can encode these invariants by thinking of each internal node as having two attributes: a color and a black-height. We will use a GADT, we call **`SubTree`**, with two indexes, one of them a **`Nat`** (for the black-height) and the other a **`Color`**.

```
data Color:: *1 where
  Red:: Color
  Black:: Color

data SubTree:: Color ~> Nat ~> *0 where
 Leaf:: SubTree Black Z
 RNode:: SubTree Black n -> Int -> SubTree Black n -> SubTree Red n
 BNode:: SubTree cL m  -> Int -> SubTree cR m -> SubTree Black (S m)

data RBTree:: *0 where
 Root:: SubTree Black n -> RBTree
```

- Note how the black height increases only on black nodes. The type **`RBTree`** encodes a ``full" Red-Black tree, forcing the root to be black, but placing no restriction on the black-height. Write an insertion function for Red-Black trees.

Writing interpreters

- Interpreters are important tools for the study of programming languages
- They have many parts
  - An object language
  - A value domain
  - A semantic mapping from object language to value domain

A simple object-language

```
data Exp:: *0 where
  Variable:: String  -> Exp
  Constant:: Int -> Exp
  Plus:: Exp
  Less:: Exp
  Apply:: Exp -> Exp -> Exp
  Tuple:: [Exp] -> Exp

-- exp1  represents “x+y”
exp1 = Apply Plus
            (Tuple [Variable "x"
                   ,Variable "y"])
```

A simple value domain

```
data Value :: *0 where
   IntV:: Int -> Value
   BoolV:: Bool -> Value
   FunV:: (Value -> Value) -> Value
   TupleV :: [Value] -> Value
```

Values are a disjoint sum of many different semantic things, so they will all have the same type. We say the values are tagged.

A simple semantic mapping

```
eval:: (String -> Value) -> Exp -> Value
eval env (Variable s) = env s
eval env (Constant n) = IntV n
eval env Plus = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
eval env Less = FunV less
  where less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
eval env (Apply f x) =
  case eval env f of
    FunV g -> g (eval env x)
eval env (Tuple xs) = TupleV(map (eval env) xs)
```

Compared to a compiler, a mapping has two forms of overhead

- Interpretive overhead
- tagging overhead

Removing Interpretive overhead

- We can remove the interpretive overhead by the use of staging.
- I.e. for a given program, we generate a meta language program (here that is Omega) that when executed will produve the same result.
- Staged programs often run 2-10 times faster than un-staged ones.

A staged semantic mapping

```
stagedEval:: (String -> Code Value) -> Exp -> Code Value

stagedEval env (Variable s) = env s
stagedEval env (Constant n) = lift(IntV n)
stagedEval env Plus = [| FunV plus |]
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
stagedEval env Less = [| FunV less |]
  where less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
stagedEval env (Apply f x) =
   [| apply $(stagedEval env f) $(stagedEval env x) |]
  where apply (FunV g) x = g x
stagedEval env (Tuple xs) = [| TupleV $(mapLift
   (stagedEval env) xs) |]
 where mapLift f [] = lift []
       mapLift f (x:xs) = [| $(f x) : $(mapLift f xs) |]
```

Observe

```
ans = stagedEval f exp1
  where f "x" = lift(IntV 3)
        f "y" = lift(IntV 4)
```

```
[| %apply (%FunV %plus)
          (%TupleV [IntV 3,IntV 4])
|] : Code Value
```

Removing tagging

- Consider the residual program

```
[| %apply (%FunV %plus)
          (%TupleV [IntV 3,IntV 4])
|]
```

The **`FunV`**, **`TupleV`** and **`IntV`** are tags.

They make it possible for integers, tuples, and functions to have the same type (**`Value`**)

But, in a well typed object-language program they are superfluous.

Typed object languages

- We will create an indexed term of the object language.
- The index will state the type of the object-language term being represented.

```
data Term:: *0 ~> *0 where
  Const :: Int -> Term Int                    -- 5
  Add:: Term ((Int,Int) -> Int)               -- (+)
  LT:: Term ((Int,Int) -> Bool)               -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b       -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)        -- (x,y)
```

- Note there are no variables in this object language

The value domain

- The value domain is just a subset of Omega values.
- No tags are necessary.

A tag less interpreter

```
evalTerm :: Term a -> a
evalTerm (Const x) = x
evalTerm Add = \ (x,y) -> x+y
evalTerm LT = \ (x,y) -> x<y
evalTerm (Ap f x) =
    evalTerm f (evalTerm x)
evalTerm (Pair x y) =
    (evalTerm x,evalTerm y)
```

Exercise 23

- In the object-languages we have seen so far, there are no variables. One way to add variables to a typed object language is to add a variable constructor tagged by a name and a type. A singleton type representing all the possible types of a program term is necessary. For example, we may add a **`Var`** constructor as follows (where the **`Rep`** is similar to the **`Rep`** type from Exercise 9).

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t          -- x
  Const :: Int -> Term Int                 -- 5
..
```

- Write a GADT for **`Rep`**. Now the evaluation function for **`Term`** needs an environment that can store many different types. One possibility is use existentially quantified types in the environment as we did in Exercise 21. Something like:

```
type Env = [exists t . (String,Rep t,t)]

eval:: Term t -> Env -> t
```

- Write the evaluation function for the **`Term`** type extended with variables. You will need a function akin to **`sameNat`** from Exercise 13, except it will have type: **`SameRep:: Rep a -> Rep b -> Maybe(Equal a b).`**

Typed Representations for languages with binding.

- The type (`Term a`) tells us it represents an object-language term with type a
- If our language has variables, what type would (`Var “x”`) have?
- It depends upon the context.
- We need to reflect the type of the variables in a term, in an index of the term, as well as the type of the whole term itself.
- E.g. **`t :: Term {`a=Int,`b=Bool} Int`**

A side trip, Tags and labels

- Tags are symbols at the type level
- Labels are symbols at the value level
- Labels are singleton types reflecting Tags
- E.g. a finite example would be

```
data Tag:: *1 where
  A:: Tag
  B:: Tag
  C:: Tag

data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
```

Primitive, infinite Tags & Labels

- All strings of alpha-numeric characters preceded by a back-tick
  - In a type context `x:: Tag
  - In a value context   `x:: Label `x
- Any string can be made into a Label with an existential type.

```
data HiddenLabel :: *0 where
  Hidden:: Label t -> HiddenLabel

newLabel:: String -> HiddenLabel
```

- A fresh (never before seen) seen label can be generated in the IO monad

```
freshLabel :: IO HiddenLabel
```

Witnessing Label Equality

- One can dynamically construct proofs of label equality at runtime.

```
labelEq :: forall (a:Tag) (b:Tag).
   Label a -> Label b -> Maybe (Equal a b)
```

Exercise 21

- A common use of labels is to name variables in a data structure used to represent some object language as data. Consider the GADT and an evaluation function over that object type.

```
data Expr:: *0 where
  VarExpr :: Label t -> Expr
  PlusExpr:: Expr -> Expr -> Expr

valueOf:: Expr -> [exists t .(Label t,Int)] -> Int
valueOf (VarExpr v) env = lookup v env
valueOf (PlusExpr x y) env =
    valueOf x env + valueOf y env
```

- Write the function:

```
lookup:: Label v -> [exists t .(Label t,Int)] -> Int
```

hint: don't forget the use of "**Ex**" .

Another side trip - Syntactic Extension

- We often prefer syntactic sugar

Lists - We prefer `[1,2,3]`

to `(Cons 1 (Cons 2 (Cons 3 Nil)))`

Nat - We prefer `3` to `S(S (S Z))`

Records - We prefer `{"a"=5, "b"=6}`

to `(RCons “a” 5 (RCons “b” 6 RNil))`

Syntactic Records

- Any data introduction with 2 Constructors

```
data T:: a1 -> … -> an -> *n where
   RN:: T x1 … xn
   RC:: a -> b -> T x1 … xn -> T y1 … yn
   deriving Record(i)
```

  - a ternary function (a record-cons function):
    - `a -> b -> T x1 … xn -> T y1 … yn`
  - a constant (a record-nil constant):
    - `T x1 … xn`

- `{}i              ---> RN`
- `{a=x,b=y}i       ---> RC a x (RC b y RN)`
- `{a=x;xs}i        ---> (RC a x xs)`
- `{a=x,b=y ; zs}i ---> RC a x (RC b y zs)`

Syntactic lists

- Any data introduction with 2 Constructors

  data T:: a_1 -> … -> a_n -> *n where

  N:: T x_1 … x_n

  C:: a -> T x_1 … x_n -> T y_1 … y_n

  deriving List(i)

  - a binary function (a cons function):
    - **a -> T x_1 … x_n -> T y_1 … y_n**
  - a constant (a nil constant):
    - **T x_1 … x_n**

- `[]i          ---> N`
- `[x,y,z]i     ---> C x(C y (C z N))`
- `[x;xs]i      ---> (C x xs)`
- `[x,y ; zs]i ---> C x (C y zs)`

Syntactic Nat

- Any data introduction with 2 Constructors

  data T:: a_1 -> … -> a_n -> *n where

    Z:: T x_1 … x_n

    S:: T x_1 … x_n -> T y_1 … y_n

    deriving Nat(i)

  – a unary function (a successor function):
    - **T x_1 … x_n -> T y_1 … y_n**
  – a constant (a zero constant):
    - **T x_1 … x_n**

- **4i   ---> S(S(S(S Z)))**
- **0i   ---> Z**
- **(2+x)i ---> S(S x)**

Exercise 20

- Consider the GADt with syntactic extension "i".

```
data Nsum:: *0 ~> *0 where
  SumZ:: Nsum Int
  SumS:: Nsum x -> Nsum (Int -> x)
 deriving Nat(i)
```

- What is the type of the terms **`0i`**, **`1i`**, and **`2i`**. Can you write a function with type: **`add:: Nsum i -> i`**. Such a function sums *n* integers given *n* as input. For example:
- **`add 3i 1 2 3`**$\rightarrow$**`6`**

Languages with binding

```
data Lam:: Row Tag *0 ~> *0 ~> *0  where
  Var      :: Label s -> Lam (RCons s t env) t
  Shift    :: Lam env t -> Lam (RCons s q env) t
  Abstract :: Label a ->
              Lam (RCons a s env) t ->
              Lam env (s -> t)
  App      :: Lam env (s -> t) ->
              Lam env s ->
              Lam env t
```

A tag-less interpreter

```
data Record :: Row Tag *0 ~> *0 where
 RecNil :: Record RNil
 RecCons :: Label a -> b ->
            Record r -> Record (RCons a b r)

eval:: (Lam e t) -> Record e -> t
eval (Var s) (RecCons u x env) = x
eval (Shift exp) (RecCons u x env) =
   eval exp env
eval (Abstract s body) env =
   \ v -> eval body (RecCons s v env)
eval (App f x) env = eval f env (eval x env)
```

Exercise 24

- Another way to add variables to a typed object language is to reflect the name and type of variables in the meta-level types of the terms in which they occur. Consider the GADTs:

```
data VNum:: Tag ~> *0 ~> Row Tag *0 ~> *0 where
  Zv:: VNum l t (RCons l t row)
  Sv:: VNum l t (RCons a b row) ->
                VNum l t (RCons x y (RCons a b row))
 deriving Nat(u)

data Exp2:: Row Tag *0 ~> *0 ~> *0 where
  Var:: Label v -> VNum v t e -> Exp2 e t
  Less:: Exp2 e Int -> Exp2 e Int -> Exp2 e Bool
  Add:: Exp2 e Int -> Exp2 e Int -> Exp2 e Int
  If:: Exp2 e Bool -> Exp2 e t -> Exp2 e t -> Exp2 e t
```

- What are the types of the terms **`(Var `x 0u), (Var `x 1u)`**, and **`(Var `x 2u),`** Now the evaluation function for **`Exp2`** needs an environment that stores both integers and booleans. Write a datatype declaration for the environment, and then write the evaluation function. One way to approach this is to use existentially quantified types in the environment as we did in Exercise 21. Better mechanisms exist. Can you think of one?

A compiler = A staged, tag-less interpreter

```
data SymTab:: Row Tag *0 ~> *0 where
  Insert :: Label a -> Code b -> SymTab e ->
            SymTab (RCons a b e)
  Empty :: SymTab RNil

compile:: (Lam e t) -> SymTab e -> Code t
compile (Var s) (Insert u x env) = x
compile (Shift exp) (Insert u x env) =
   compile exp env
compile (Abstract s body) env =
   [| \ v -> $(compile body (Insert s [|v|] env)) |]
compile (App f x) env =
   [| $(compile f env) $(compile x env) |]
```

Exercise 25

- A staged evaluator is a simple compiler. Many compilers have an optimization phase. Consider the term language with variables from a previous Exercise.

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t
  Const :: Int -> Term Int                   -- 5
  Add:: Term ((Int,Int) -> Int)              -- (+)
  LT:: Term ((Int,Int) -> Bool)              -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b      -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)       -- (x,y)
```

- Can you write a well-typed staged evaluator the performs optimizations like constant folding, and applies laws like *(x+0) = x* before generating code?

Next-time

- Subject reduction proofs
- Defining and using theorems
- Hardware descriptions