Programming with Intent

Summer School
November 12, 2008

Christopher Dutchyn

University of Saskatchewan

Quicksort

```
-- qsort ::  int List -> int List
```

Quicksort

```
-- qsort ::  int List -> int List

qsort lst = lst
```

Quicksort

```
-- qsort ::  int List -> int List

-- tests!
-- qsort [1,2,3] --> [1,2,3]
-- qsort [3,2,1] --> [1,2,3]
```

Quicksort

```
-- qsort ::  int List -> int List

-- tests!
-- qsort [1,2,3] --> [1,2,3]
-- qsort [3,2,1] --> [1,2,3]

qsort [1,2,3] = [1,2,3]
qsort [3,2,1] = [1,2,3]
qsort     lst = lst
```

Quicksort

```
-- qsort ::  int List -> int List

-- tests!
-- qsort [1,2,3] --> [1,2,3]
-- qsort [3,2,1] --> [1,2,3]

qsort [1,2,3] = [1,2,3]
qsort [3,2,1] = [1,2,3]
qsort _   lst = lst
```

- testing proves correctness at point level
  - powerful but limited range

Can we do better?

Idea: lets use types to express programmer intent

Omega ≈ Haskell

- Additions
  - Unbounded number of computational levels
    - values (*0), types (*1), kind (*2), sorts (*3), …
  - Data structures at all levels
  - Generalized Algebraic Data Types (GADTs)
  - Functions at all levels
  - *Staging*

- Subtractions
  - Type classes
  - Laziness

Programming with Types†

An object with structure at the type level

```
data Nat:: *1 where
  Z:: Nat
  S:: Nat ~> Nat
```

the **`*1`** means **`Nat`** is a *kind*, and **`S`** and **`Z`** yield *types*

†with kudos to Stephanie Weirich

Kinds

Objects with Structure at the type Level

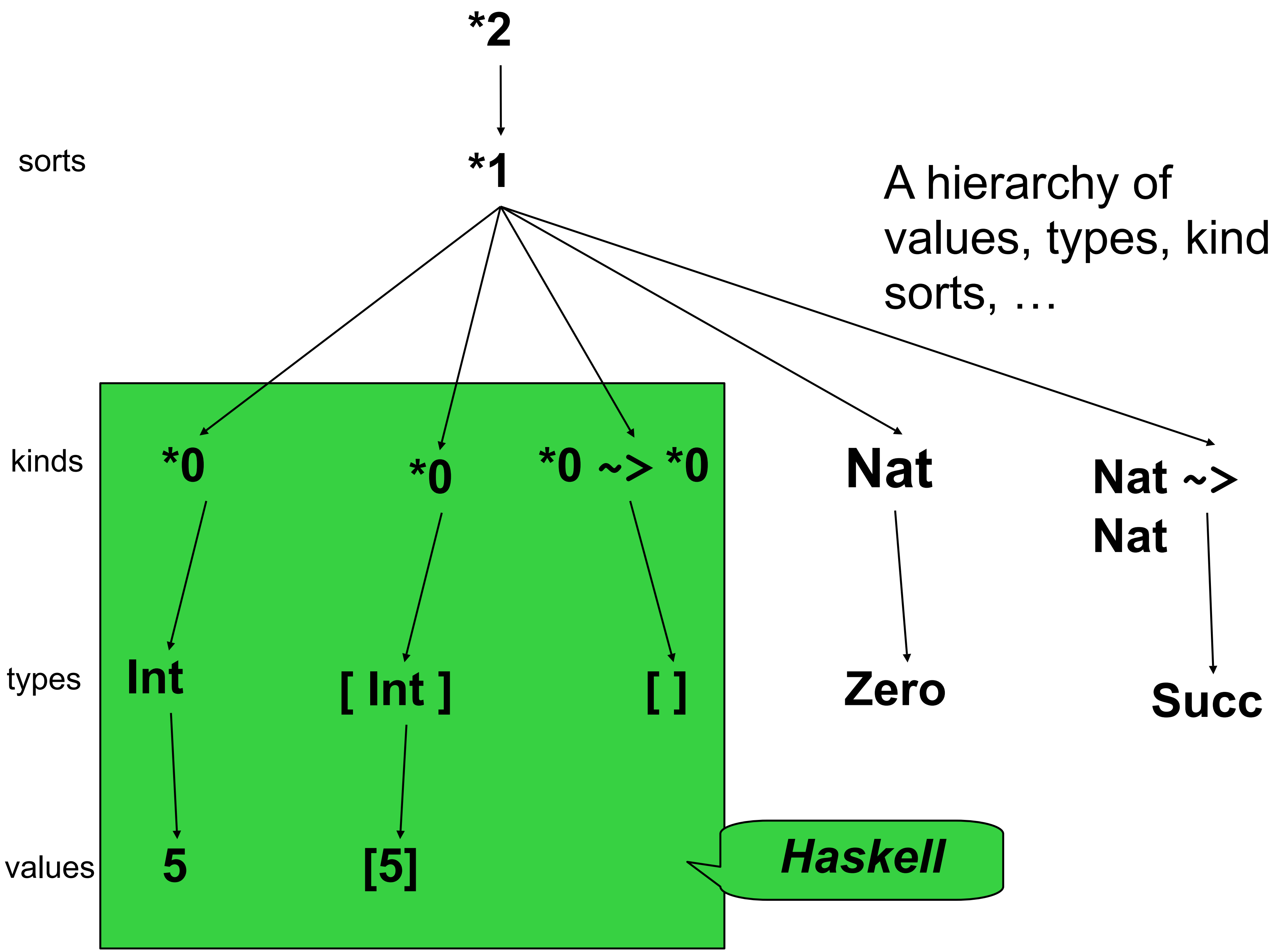
*1 means a kind

```
data Nat:: *1 where
   Z:: Nat
   S:: Nat ~> Nat
```

Z and S are types

- A kind of natural numbers
  - Classifies types `Z`, `S Z`, `S (S Z)`…
  - Such types don't classify values

A hierarchy of values, types, kinds, sorts, …



Example Kinds

```
data State:: *1   where
  Locked:: State
  Unlocked:: State
  Error:: State

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

More Examples

```
data Boolean:: *1 where
   T:: Boolean
   F:: Boolean

data Shape :: *1 where
   Tp:: Shape
   Nd:: Shape
   Fk:: Shape ~> Shape ~> Shape
```

Type functions

Functions use pattern-matching equations. Every type function must have a prototype.

```
plus :: Nat ~> Nat ~> Nat

{plus Z m} = m

{plus (S n) m} = S {plus n m}
```

At the type level and above, type constructor application uses juxatposition.

At the type level and above we surround function application with braces.

Functions over types

```
even :: Nat ~> Boolean
{even Z} = T
{even (S Z)} = F
{even (S (S n))} = {even n}
```

More examples

```
and:: Boolean ~> Boolean ~> Boolean
{and T x} = x
{and F x} = F
```

Type level data structures

```
data LE :: Nat ~> Nat ~> *0 where
    Base_LE :: LE Z a
    Step_LE :: LE a b -> LE (S a) (S b)
```

`Base_LE` witnesses that `Z` (zero as a type-level natural number) is known to be less than any other type-level natural number.

`Step_LE` extends this inductively to cover all larger successive cases

Relationships between types

```
le23 :: LE #2 #3

le23 = Step_LE (Step_LE Base_LE)


le2x :: LE #2 #(2+a)

le2x = Step_LE (Step_LE Base_LE)
```

Type Functions v.s. Witnesses

```
even:: Nat ~> Boolean

{even Z} = T

{even (S Z)} = F

{even (S (S n))}={even n}

le:: Nat ~> Nat

      ~> Boolean

{le Z n} = T

{le (S n) Z} = F

{le (S n) (S m)} =

   {le n m}
```

```
data Even:: Nat ~> *0
  where
   EvenZ:: Even Z
   EvenSS:: Even n ->
            Even (S (S n))


data LE:: Nat ~> Nat ~> *0
  where
   LeZ:: LE Z n
   LeS:: LE n m ->
         LE (S n) (S m)
```

Type indexed data

*0 means **BSeq** is a *type*, and **Nil** and **Cons** are *values*

```
data BSeq :: Nat ~> Nat ~> *0 where
     Nil  :: LE min max => BSeq min max
     Cons :: (LE min m, LE m max) =>
                   Nat' m -> BSeq min max
                          -> BSeq min max
```

Explicitly classify both **BSeq**, and its constructor functions, **Nil** and **Cons**, with their full classification

LE automatically ensures the type-level constants **min** and **max** satisfying **LE min m** and **LE m max** exist

Helper function

The + is the disjoint union

```
mapP :: (a -> b) -> (c -> d) -> (a+c) -> (b+d)
mapP f g (L x) = L (f x)
mapP f g (R x) = R (g x)
```

Value-level fn typed by type-level fn

Nat' will be our value-level data -- an actual natural number.

Every natural number is either <= or > another natural number

```
compare :: Nat' a -> Nat' b
                  -> (LE a b + LE b a)
compare Z _ = L Base_LE
compare (S x) Z = R Base_LE
compare (S x) (S y) = mapP Step_LE
                           Step_LE
                           (compare x y)
```

Another Operation

```
qsplit :: (LE min piv, LE piv max) =>
             Nat' piv -> BSeq min max
                      -> (BSeq min piv,
                          BSeq piv max)
qsplit piv Nil = (Nil,Nil)
qsplit piv (Cons x xs) =
    case compare x piv of
      L p1 -> (Cons x small, large)
      R p1 -> (small, Cons x large)
    where (small,large) = qsplit piv xs
```

small and **large** are only bounded, not sorted.

A useful definition -- of a sorted list

```
data SL :: Nat ~> Nat ~> *0 where
   SNil :: SL x x
   SCons :: LE min min' =>
                 Nat' min -> SL min' max
                          -> SL min max
```

a sorted list between **min'** and **max**

a value (*0) less than **min'**

a sorted list between **min** and **max**

Another utility -- appending sorted lists

```
app :: SL min piv -> SL piv max -> SL min max
app SNil ys = ys
app (SCons min xs) ys = SCons min (app xs ys)
```

sorting property is preserved

Using these Operations ...

```
qsort :: BSeq min max ->
              exists t . LE min t => SL t max

qsort Nil = Ex (SNil)
qsort (x @ (Cons pivot tail)) =
              (Ex (app smaller'
                       (SCons pivot larger')))
    where (smaller,larger) = qsplit pivot tail
          (Ex (smaller'))  = (qsort smaller)
          (Ex (larger'))   = (qsort larger)
```

Ex is an anonymous existential type

Ensuring Static Checking

```
prop LE :: Nat ~> Nat ~> *0 where
    Base_LE :: LE Z a
    Step_LE :: LE a b -> LE (S a) (S b)
```

Why Not Use C or Haskell?

- Most traditional languages like C don't have strong type systems that enforce the discipline necessary,
- Even in Haskell, we can't create data structures whose types can capture the types of **`Z`**, **`E`**, and **`O`**.
  - GHC is adding this capability
- We can't parameterize types (like **`Even`** and **`Odd`**) with objects like **`Z`** and **`(S Z)`** since these are values not types.
  - GHC Type families are growing this capability

Summary

- Techniques exist for writing verified programs
  - not just tested ... verified
  - including compilers [Leroy, 2007]
- This is one approach
  - extracting program from proof is another
- The future of programming is visible!
  - proven programs

Acknowledgements

- thanks to Tim Sheard for gracious permission to use parts of his Omega material

- Omega
  - web.cecs.pdx.edu/~sheard/Omega/index.html

Yes! We Can!

And, it's not that hard!

What Makes This Work

- type checking as computation
  - closely related to typing as abstract interpretation
  - cf. Cousot and Cousot
- guarded algebraic data types
- types as propositions / programs as proofs
  - Curry Howard isomorphism
-

Type Checking

- Type checking *is* compile-time computation.

$$\frac{\Gamma \mid\!- f : c \to d \qquad \Gamma \mid\!- x : b \qquad b \cong c}{\Gamma \mid\!- f\ x : d}$$

$b \cong c$ means b is mutually consistent

Mutually consistent

- Pascal
  - b ≅ c means b and c are structurally equal
- Haskell
  - b ≅ c means b and c unify
- Java
  - b ≅ c means b is a subtype of c
- Dependent typing
  - b ≅ c means b and c "mean the same thing"

Type Checking = CSP

- Every function leads to a set of constraints
- If the constraints have a solution, the function is well typed.
- In Omega (as in dependent typing),
  - constraints are all about the semantic equality of type expressions.

GADTS

- How do GADTs generalize ADTS?
  - at every level (instead of just at level *0)
  - ranges are not restricted to distinct variables
- How are they declared?
- What kind of expressive power do they add?

ADT Declaration

- Structures
  - `data Person = P Name Age Address`
- Unions
  - `data Color = Red | Blue | Yellow`
- Recursive
  - `data IntList = None`
  - `               | Add Int IntList`
- Parameterized (polymorphic)
  - `data List a = Nil | Cons a (List a)`

Algebraic Datatypes

- Inductively formed structured data
  - Generalizes enumerations, records & tagged variants
- Well typed *constructor functions* are used to prevent the construction of ill-formed data.
- Pattern matching allows abstract high level (yet still efficient) access

ADTs provide abstract interface to data

```
• Data Tree a
    = Fork (Tree a) (Tree a)
    | Node a
    | Tip
```

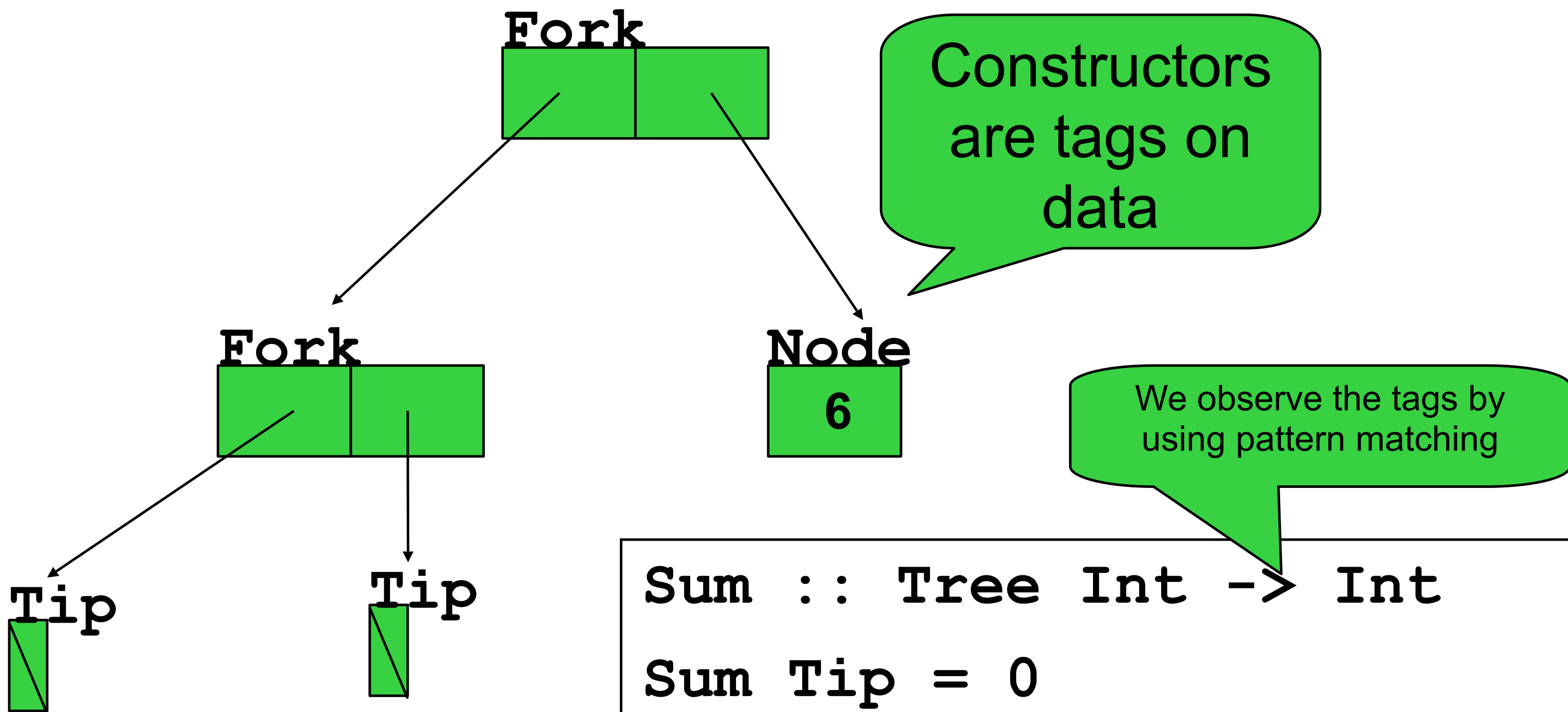
We can define parametric polymorphic data

Inductively defined data allows structures of unbounded size

```
• Fork :: Tree a -> Tree a -> Tree a
• Node :: a -> Tree a
• Tip :: Tree a
```

Note the “data” declaration introduces values and functions that construct instances of the new type.

Deconstruction by pattern




```
Sum :: Tree Int -> Int
Sum Tip = 0
Sum (Node x) = x
Sum (Fork m n) = sum m
               + sum n
```

ADT Type Restrictions

- ```
  Data Tree a
    = Fork (Tree a) (Tree a)
    | Node a
    | Tip
  ```

- `Fork :: Tree a -> Tree a -> Tree a`
- `Node :: a -> Tree a`
- `Tip :: Tree a`

Restriction: the range of every constructor matches exactly the type being defined
  ```

GADTS at every level

```
data Shape :: *1 where
   Tp:: Shape
   Nd:: Shape
   Fk:: Shape ~> Shape ~> Shape
```

The range of the introduced type selects the levels that the GADT introduces its constructors.

`Shape` is a kind, `Tp`, `Nd`, and `Fk` are types

GADTs remove the range restriction

```
data Tree :: Shape ~> *0 ~> *0 where
  Tip:: Tree Tp a
  Node:: a -> Tree Nd a
  Fork::  Tree x a -> Tree y a -> Tree (Fk x y) a
```

Note the different range types!

- Instead of indicating the arity of a type constructor by naming its parameters, give an explicit kind
- Give the explicit type for every constructor to remove the range restriction.

Trees are indexed by Shape

```
Tree :: Shape ~> *0 ~> *0 where
Tip:: Tree Tp a
  Node:: a -> Tree Nd a
  Fork::  Tree x a -> Tree y a -> Tree (Fk x y) a
```

- The kind index tells us about the shape of the tree. We can exploit this invariant

```
data Path:: Shape ~> *0 ~> *0 where
  None :: Path Tp a
  Here :: b -> Path Nd b
  Left :: Path x a -> Path (Fk x y) a
  Right:: Path y a -> Path (Fk x y) a
```

Function types tell us properties

```
find:: (a -> a -> Bool) -> a
                        -> Tree s a
                        -> [Path s a]

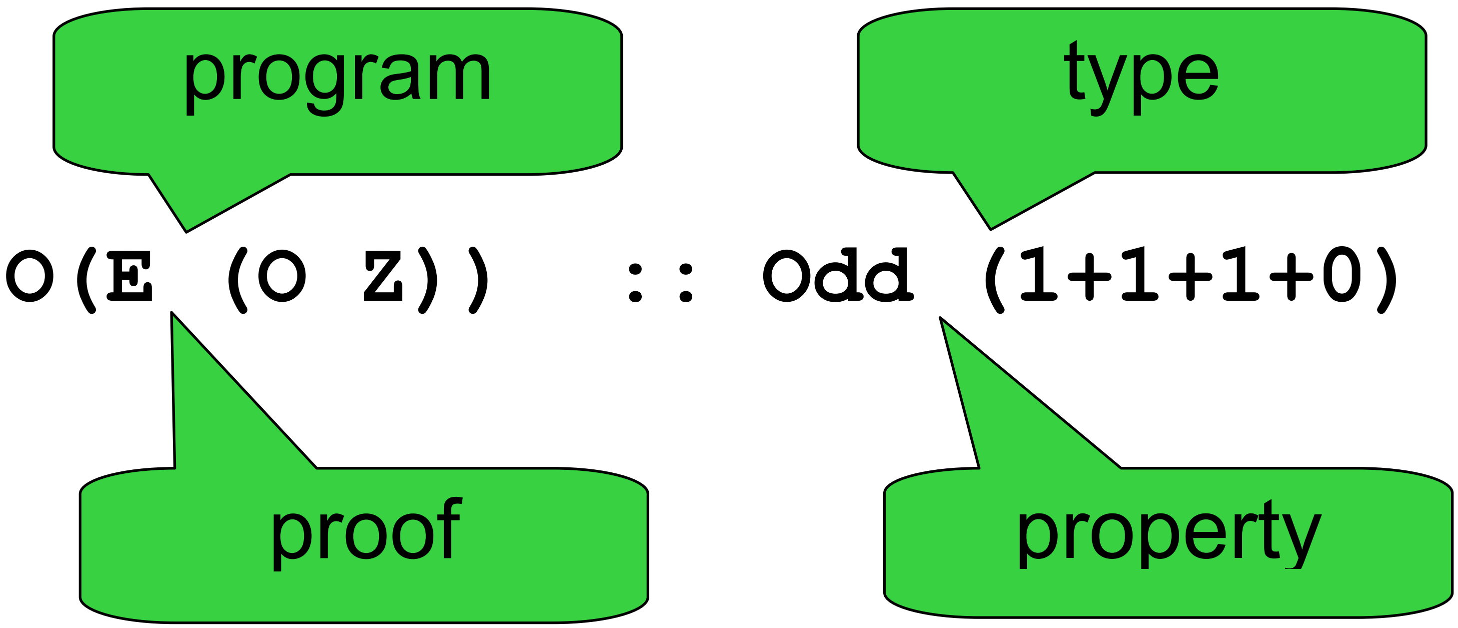
find eq n Tip = []
find eq n (Node m) =
  if eq n m then [Here n] else []
find eq n (Fork x y) =
  map Left (find eq n x) ++
  map Right (find eq n y)
```

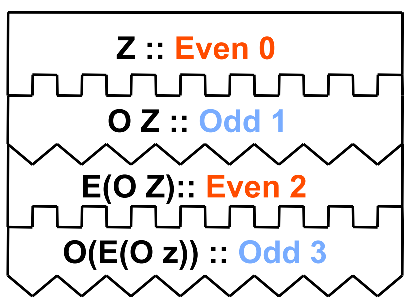
Curry-Howard isomorphism

- The Curry-Howard isomorphism states that there is an isomorphism between
    - programs/types
  - and
    - proofs/propositions

- What does this mean?
- How can we put this powerful idea to work in practical ways?

Curry-Howard





Odd 3

What is a proof?

Properties or Propositions

0 is even

1 is odd, if

2 is even, if

3 is odd, if

Requirements for a legal proof

- Even is always stacked above odd
- Odd is always stacked below even
- The numeral decreases by one in each stack
- Every stack ends with 0

Introduce data indexed by Nat

```
data Even:: Nat ~> *0 where …
Z:: Even 0
E:: Odd m -> Even (m+1)

data Odd:: Nat ~> *0 where
O:: Even m -> Odd (m+1)
```

Note the different range types! GADTS are essential here!

Properties as Functional Programs

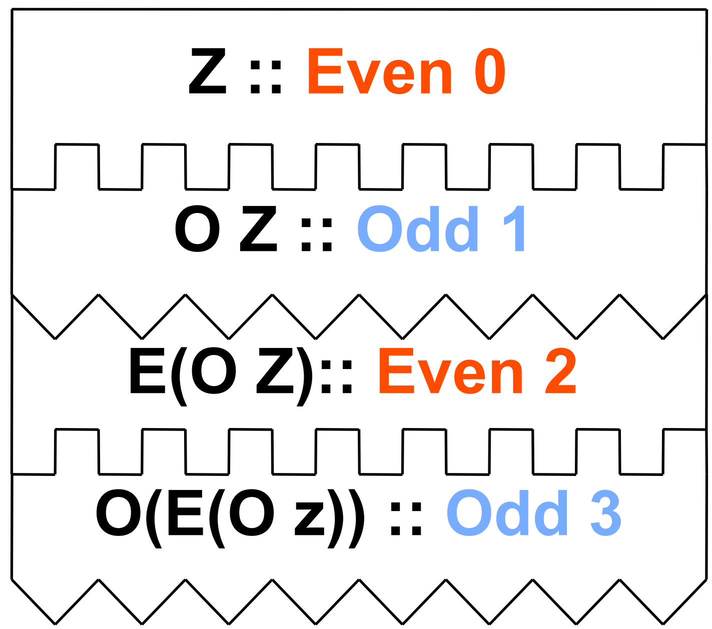
```
data Even m = …
Z:: Even 0
E:: Odd m -> Even (m+1)
```

Even and **Odd** type constructors,
Z,**E**, and **O** are data constructors

Observation: Proofs are Data!

```
data Odd m = …
O:: Even m -> Odd (m+1)
```

```
O(E (O Z))
   :: Odd (1+1+1+0)
```



Relating functions & witnesses

```
data Proof:: Boolean ~> *0 where
  Triv:: Proof T
```

Singleton Types

- GADTs allow us to reflect the structure of types as structure (data) at the value level

```
data Nat' :: Nat ~> *0 where
   Z :: Nat' Z
   S :: Nat' x -> Nat' (S x)
```

Kinds	`Nat`

Types	`(Nat' Z)`
	`Z`
	`(S Z)`

Values	`Z` `(S Z)`

Exploits the separation between the value name space and the type name space.
Because of this declaration **`Z`** and **`S`** are added to the value name space.

Properties of Singleton Types

- Only one element inhabits any singleton type.
- The shape of that value is in 1-to-1 correspondance with the type index of the type of that value
  - `S(S(S Z)) :: Nat' (S(S(S Z))`
- If you know the type of a singleton, you know its shape.
- You can discover the type of a singleton value by exploring its shape.