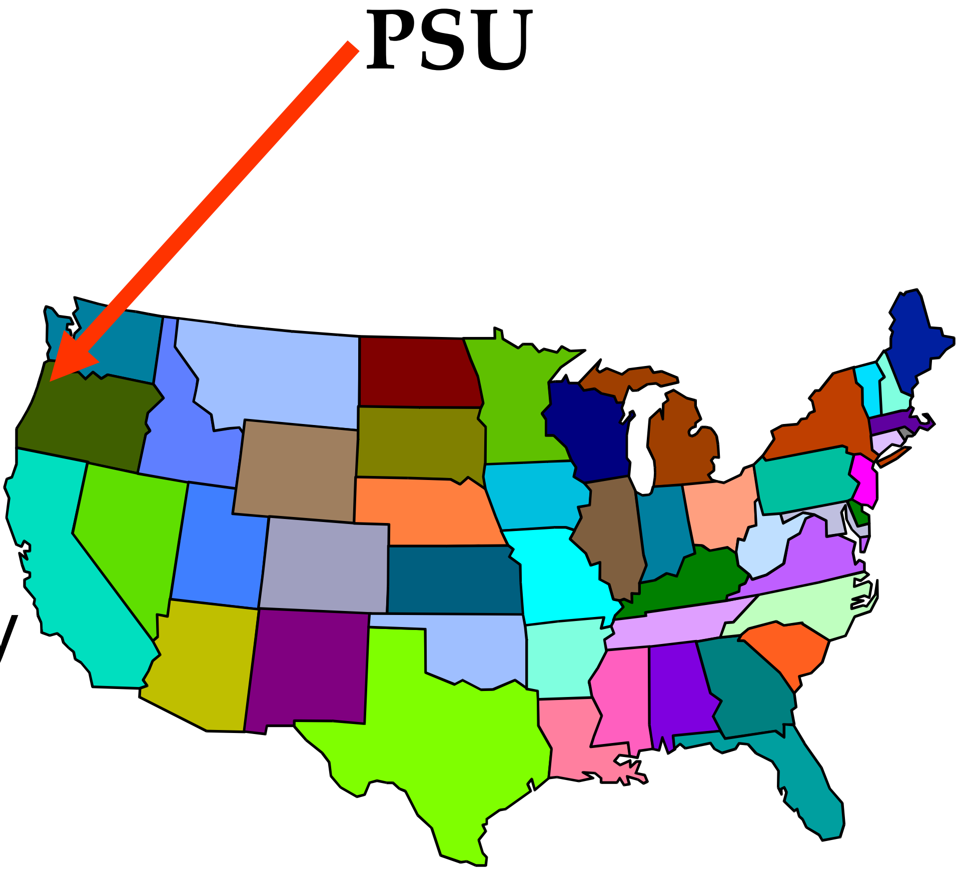
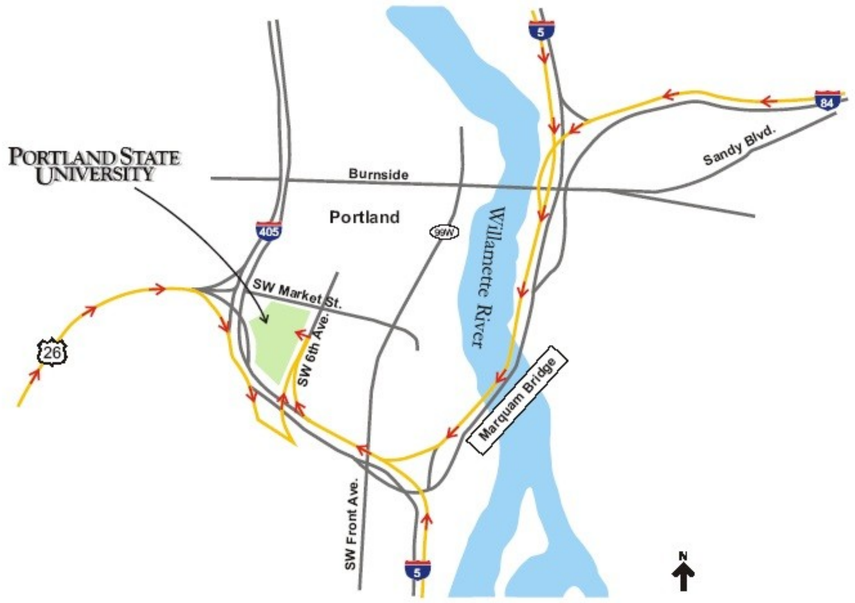
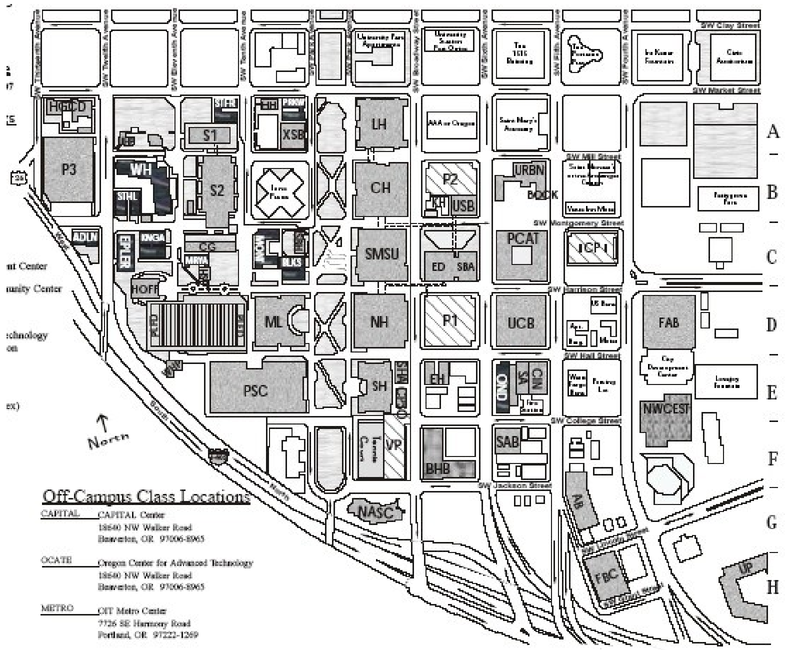
Programming in Omega Part 1

Tim Sheard
Portland State University

- Tim Sheard
- Computer Science Department
- Portland State University
- Portland, Oregon







Off-Campus Class Locations

CAPITAL	CAPITAL Center 18640 NW Walker Road Beaverton, OR 97006-8965
OCATE	Oregon Center for Advanced Technology 18640 NW Walker Road Beaverton, OR 97006-8965
METRO	OIT Metro Center 7726 SE Harmony Road Portland, OR 97222-1269

PL Research at Portland State University

- The Programming Language Group at PSU.
  - Sergio Antoy (Curry)
  - Andrew Black (Emerald)
  - Mark Jones (Hugs, Constrained Types)
  - Jim Hook (PacSoft)
  - Tim Sheard (MetaML, Template Haskell, Omega)
  - Andrew Tolmach (House)
  - **Galois Connections** (John Launchbury) 3 train stops away
- We are looking for new students. Come Join us, or send your good graduates our way! www.cs.pdx.edu

Omega

- Omega is modeled after Haskell
- Additions
  - Unbounded number of computational levels
    - values (*0), types (*1), kind (*2), sorts (*3), …
  - Data-structures at all levels
  - Generalized Algebraic Datatypes
  - Functions at all levels
  - Staging
- Subtractions
  - The class system
  - Laziness

end-to-end example

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

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 are types

Type functions

We write functions by using 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 brackets

← value name space	*type name space →*		
value	*type*	*kind*	*sort*
	Tree	:: *0 ~> *0	:: *1
Fork	:: Tree a -> Tree a -> Tree a	:: *0	:: *1
Node	:: a -> Tree a	:: *0	:: *1
Tip	:: Tree a	:: *0	:: *1
	Z	:: Nat	:: *1
	S	:: Nat ~> Nat	:: *1
	plus	:: Nat ~> Nat ~> Nat	:: *1
	{plus #1 #3 }	:: Nat	:: *1
	Seq	:: *0 ~> Nat ~> *0	:: *1
Snil	:: Seq a Z	:: *0	:: *1
Scons	:: a -> Seq a b -> Seq a (S b)	:: *0	:: *1
app	:: Seq a n -> Seq a m -> Seq a {plus n m}	:: *0	:: *1
	Tp	:: Shape	:: *1
	Nd	:: Shape	:: *1
	Fk	:: Shape	:: *1
	Tree	:: Shape ~> *0 ~> *0	:: *1
Tip	:: Tree Tp a	:: *0	:: *1
Node	:: a -> Tree Nd a	:: *0	:: *1
Fork	:: Tree x a -> Tree y a -> Tree (Fk x y) a	:: *0	:: *1
find	:: (a -> a -> Bool) -> a -> Tree sh a -> [Path sh a]	:: *0	:: *1
	T	:: Boolean	:: *1
	F	:: Boolean	:: *1
	le	:: Nat ~> Nat > Boolean	:: *1
	{le #0 #2}	:: Boolean	:: *1
	LE	:: Nat ~> Nat > *0	:: *1
LeZ	:: LE Z a	:: *0	:: *1
LeS	:: LE n m -> LE (S n) (S m)	:: *0	:: *1
	Even	:: Nat ~> *0	:: *1
EvenZ	:: Even Z	:: *0	:: *1
EvenSS	:: Even n -> Even (S(S n))	:: *0	:: *1

Fig. 1. The level hierarchy for some of the examples in the paper.

Using kinds to index types

*0 means Seq is a type, and Snil and Scons are values

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

We explicitly classify both Seq, and its constructor functions, Snil and Scons, with their full classification

The second argument to Seq is a natural number

Type indexed data

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

- Parameters of data types, that are not of kind *0, are type indexes.
- Indexes describe an invariant of the data.
- Consider a value of type

```
   (Seq Int (S Z))
```

This is a parameter, we expect things of type Int inside

This is an index, we don't expect things of type (S Z) inside, instead it tells us the list has length 1

A value-level function whose type mentions a type-level function

We write value-level functions by using pattern matching equations.

The plus function appears in the type of app

```
app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

Type Checking

- Type checking *is* compile-time computation.

$$\frac{\Gamma \vdash f : c \rightarrow d \qquad \Gamma \vdash x : b \qquad b \cong c}{\Gamma \vdash f\ x : d}$$

$b \cong c$ means b is mutually consistent

Mutually consistent

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

Type checking by constraint solving

- 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.

Computing Equations

```
app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

expected type	`Seq a n`	—	`Seq a m`	→	`Seq a {plus n m}`
equation	`app (Scons x xs)`		`ys`	=	`Scons x (app xs ys)`
computed type	`Seq a (S b)`		`Seq a m`		`Seq a (S{plus b m})`
equalities	`n = S b`			⇒	`{plus n m} = S{plus b m}`

Exercise 1

- Write an Omega function that defines the length function over sequences.

```
length:: Seq a n -> Int
```

- You will need to create a file, and paste the definition for **`Seq`** into the file, as well as write the length function.The **`Nat`** kind is predefined. You will need to include the function prototype, above, in your file (type inference is limited in Omega).

- How might we reflect the fact that the resulting **`Int`** should have size **`n`**?

Guide to the rest of Lecture 1

- New Features
  - Kinds
  - Functions at the type level
  - GADTs – Generalized algebraic datatypes
- New Patterns
  - witnesses
  - comparing type functions and witnesses
  - singleton types
  - Nat' (a pun)

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, …

$$\begin{tikzcd}
 & & & \ast 2 \arrow[d] & & \\
\text{sorts} & & & \ast 1 \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{kinds} & \ast 0 \arrow[d] & \ast 0 \arrow[d] & \ast 0 \sim> \ast 0 \arrow[d] & \text{Nat} \arrow[d] & \text{Nat} \sim> \text{Nat} \arrow[d] \\
\text{types} & \text{Int} \arrow[d] & [\ \text{Int}\ ] \arrow[d] & [\ ] & \text{Zero} & \text{Succ} \\
\text{values} & 5 & [5] & & &
\end{tikzcd}$$

Haskell portion of the hierarchy

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
```

Exercise 3

- Write a data declaration introducing a new kind called **`Color`** with types **`Red`** and **`Black`**. Are there any values with type **`Red`**? Now write a data declaration introducing a new type **`Tree`** which is indexed by **`Color`** (this will be similar to the use of **`Nat`** in the declaration of **`Seq`**).
- There should be some values classified by the type **`(Tree Red)`** ,and others classified by the type **`(Tree Black)`**.

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 (polymorhphic)
  - `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

ADT's provide an abstract interface to heap data.

We can define parametric polymorphic data

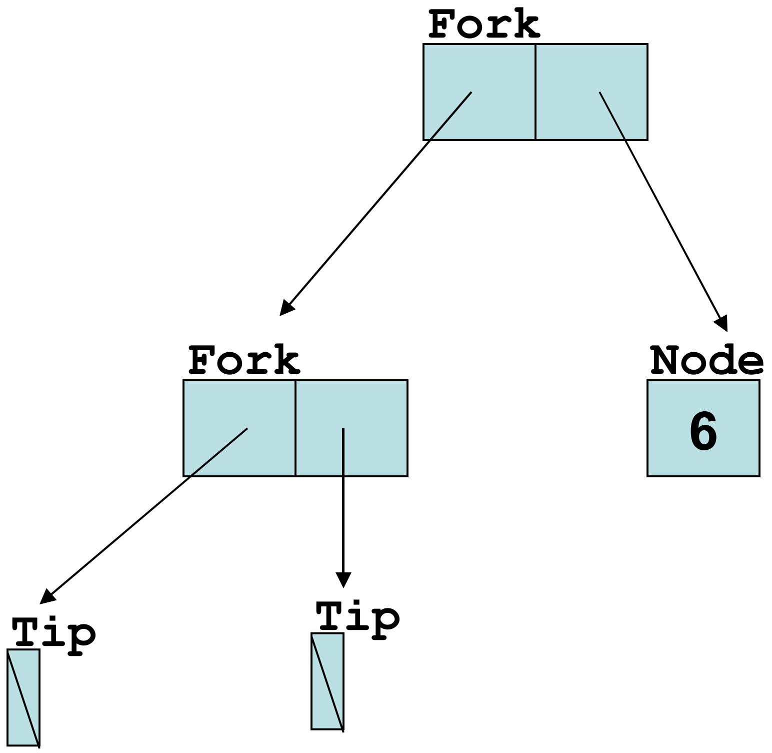
- `Data Tree a`
  ```
   = Fork (Tree a) (Tree a)
   | Node a
   | Tip
  ```

Inductivley 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 matching



Constructors are tags on data

We observe the tags by using pattern matching

```
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
```

Recall he kind 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

Note the different range types!

```
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
```

- **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
```

We can write functions whose types tells us important 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)
```

Exercises 7-8

- Write an Omega function with type
  - `extract:: Path sh a -> Tree sh a -> a`

  which extracts the value of type **`a`**, stored in the tree at the location pointed to by the path. This function will pattern match over two arguments simultaneously. Some combinations of patterns are not necessary. Why? See section 3.10 for how you can document this fact.

- Replicate the shape index pattern for lists. Write two Omega GADTs. One at the kind level which encodes the shape of lists, and one at the type level for lists indexed by their shape. Also, write a find function for your new types.

```
find:: (a -> a -> Bool) -> a ->
             List sh a -> Maybe(ListPath sh a)
```

  which returns the first path, if one exists.

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

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

Exercise 4-6

- Write the function **`mult`**, which is the multiplication function at the type level over natural numbers. It should be classified by the kind
  - **`mult:: Nat ~> Nat ~> Nat`**
- Write the **`odd`** function classified by
  - **`Nat ~> Boolean`**
- Write the or and not functions, that are classified by the kinds
  - **`or:: Boolean ~> Boolean ~> Boolean`**
  - **`not:: Boolean ~> Boolean`**
- Which arguments of **`or`** should you pattern match over? Does it matter? Experiment, Omega won't allow some combinations. See Appendix 2 on inductively sequential definitions and narrowing for the reason why.

Employing type functions

```
app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys  = Scons x (app xs ys)
```

- Normal functions at the value level are given function prototypes by the programmer, that use functions at the type level.
- The type-functions relate (in a functional manner) the type indexes of the inputs and outputs. They relate the invariants, and hence say something about what the function does.

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

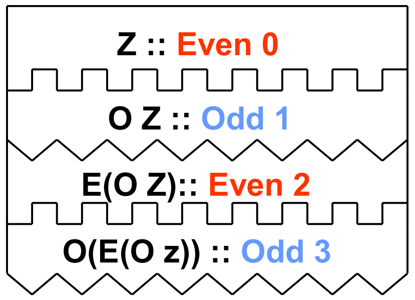
program

type

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

proof

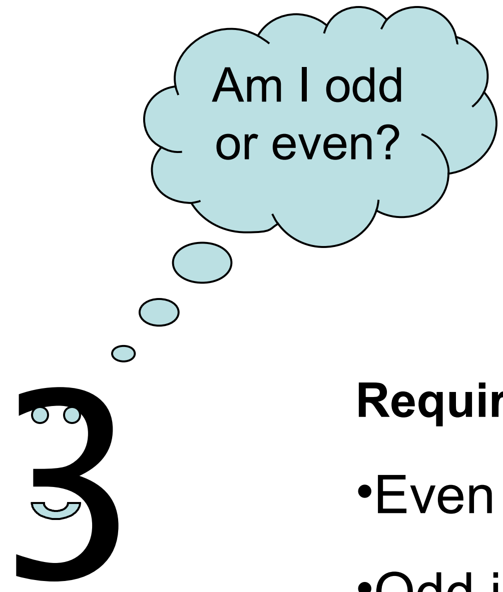
property



Odd 3

What is a proof?

Properties or Propositions



$$\frac{\dfrac{\dfrac{\text{0 is even}}{\text{1 is odd, if}}}{\text{2 is even, if}}}{\text{3 is odd, if}}$$

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 new 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

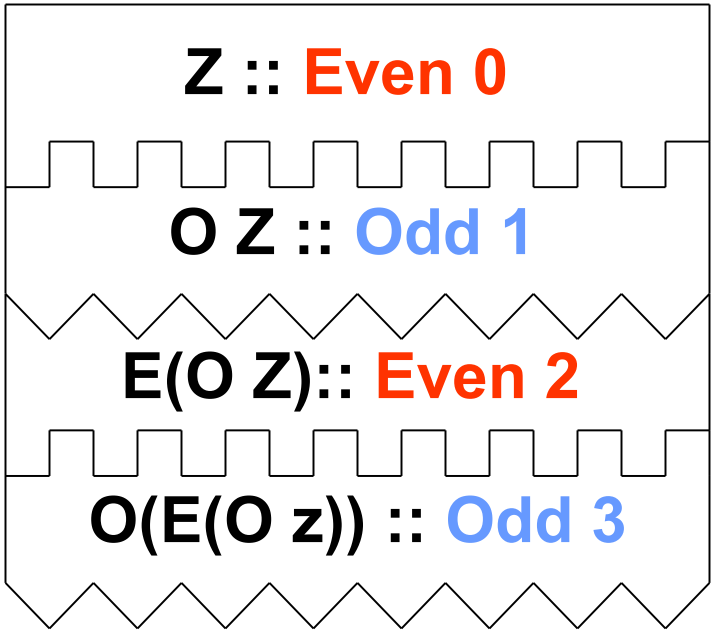
```
data Even m = …
Z:: Even 0
E:: Odd m -> Even (m+1)

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

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

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

Observation: Proofs are Data!



Relationships between types

```
data LE :: Nat ~> Nat ~> *0  where
  Base:: LE Z x
  Step:: LE x y -> LE (S x) (S y)

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

le2x :: LE #2 #(2+a)
le2x = Step(Step Base)
```

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)
```

Relating functions & witnesses

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

Exercises 10-11

Consider:

```
data Plus:: Nat ~> Nat ~> Nat ~> *0  where
  PlusZ:: Plus Z m m
  PlusS:: Plus n m z -> Plus (S n) m (S z)
```

- **Construct terms with the types** `(Plus 2t 3t 5t)`, `(Plus 2t 1t 3t)`, **and (**`Plus 2t 6t 8t)`. **What did you discover?**

- **Write an Omega function with the following type:**

  ```
  summandLessThanSum:: Plus a b c -> LE a c
  ```

  Hint: it is a recursive function. Can you write a similar function with type `(Plus a b c -> LE b c)?`

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)
```

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.

Kinds	`Nat`

Types	`(Nat′ Z)`
	`Z`
	`(S Z)`

Values	`Z`
	`(S Z)`

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.

Exercise 13-14

- Write the two Omega functions with types:

  ```
  same:: Nat' n -> LE n n
  ```

  and

  ```
  predLE:: Nat' n -> LE n (S n)
  ```

  Hint they are simple recursive functions.

- Write the Omega function which witnesses the transitivity of the less-than-or-equal to predicate.

  ```
  trans:: LE a b -> LE b c -> LE a c
  ```

  Hint: it is a recursive function with pattern matching over both arguments. One of the cases is not reachable.

Exercise 9

- Consider the GADT below.

```
 data Rep :: *0 ~> *0 where
     Int :: Rep Int
     Prod :: Rep a -> Rep b -> Rep (a,b)
     List :: Rep a -> Rep [a]
```

- Construct a few terms. Do you note any thing interesting about this type? Write a function with the following type:

```
 showR:: Rep a -> a -> String
```

- which given values of type (**`Rep a)`** and **`a`**, displays the second as a string. Extend this GADT with a few more constructors, then extend your **`showR`** function as well.

Why can’t we do this in traditional languages like C or even in more modern languages like 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.
- We can’t parameterize types (like Even and Odd) with objects like Z and (S Z) since these are values not types.

Next time

- We will discover how to use all these new tools.