Type Instances: Declaration

- == = False
  (Branch of True) = (True
  If True = True
  Else = False

Pull into class Bg

Data Tag = Less | Branch the Tree

E - Recall Tree:

You pull into the appropriate type class.
Pull into another way: in order to overload an operator for your data type, you can pull any data type into a type class, but you have to implement the operators.

Type Classes

Can we extend this overloading to my data types and my operations?

How is this overloading implemented?

What is the actual type of the x in polymorphism?

Recall the question:

raise the question:

branch Tree a)

data Tree = Less | Branch Tree

The operator overload.

Objectively, a type class declarations a new operator and functions for

A type variable representing the set of all types, unrestricted.

Conceivably, a type class represents a restricted set of types. (Contrast

All of the above problems and questions are resolved in Haskell by type

The Equality: The Problem

1. We would like to overload == and not use the name treeEq.

2. Tree Eq.

There are two problems:

treeEq a = False

branch (Branch x z) = treeEq x == treeEq z

Suppose we want to write a function that determines if two trees are equal:

data Tree a = Less | Branch (Tree a) (Tree a)

Recall the type data type we defined:

Operator Overloading: The Questions

Recall our previous story about numbers: we said
True Equality: Solution 2

```
instance Bag a => Eq (List a) where
  bag == [] => bag == []
  bag == [x] => bag == [x]
  bag == [x, y] => bag == [y, x]

the type is
true == true  = True
true == false = False
false == true  = False
false == false = True
```

Note that the only implemented == but not /=. This is because Bag

Types of Overloaded Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>True equality</td>
</tr>
<tr>
<td>/=</td>
<td>False equality</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>/=</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

In addition to the standard operators, Bag also overloads the equality operator. This allows for more concise and readable code when working with Bag instances.
Go into internal nodes, and "null pointers" are modeled by NIL.

We require the key type to come from the data class. For simplicity, keys
data are a = BST a

Now we can define our polymorphic binary search tree:

In defines the comparison operators. It also requires the type to belong to

BST

We wish to allow any type of keys, but the type must come with the < and

the < operator. These come from the data class (for ordered):

We can now define polymorphic but restricted data types.

Hash keys can automatically generate such name definitions:
probably 99% of your data types will have == defined in a similar fashion

The Equality Solution 3