# Adding Pattern Matching to Existing Object-Oriented Languages

Changhee Park
(Joint work with Sukyoung Ryu and Guy L. Steele Jr.)

PLRG @ KAIST

August 26, 2010

# Introduction

## Pattern Matching

- One of main features of functional programming languages
- Identifies an object and manipulates the object by deconstrucing it

## Object-Oriented languages

- Programming languages pursuing data encapsulation and information hiding

# Example : Binary Tree in Fortress

```
trait Tree
  getter item(): ℤ32
  getter depth(): ℤ32
end

object Node(left: Tree, item: ℤ32, right: Tree) extends Tree
  getter depth(): ℤ32 = 1 + (left.depth MAX right.depth)
end

object Leaf(item: ℤ32) extends Tree
  getter depth(): ℤ32 = 1
end
```

# Tree Manipulation with *instanceOf* and *cast*

```
sum(t: Tree) =
  if instanceOf[[Node]](t)
  then n = cast[[Node]](t)
       if instanceOf[[Leaf]](n.left)
       then  if instanceOf[[Leaf]](n.right)
             then n.left.item + t.item + n.right.item
             else n.left.item + t.item + sum(n.right)
             end
       else  if instanceOf[[Leaf]](n.right)
             then sum(n.left) + t.item + n.right.item
             else sum(n.left) + t.item + sum(n.right)
             end
       end
  else t.item
  end
```

# Tree Manipulation with Pattern Matching

$$
\begin{array}{l}
sum(t\colon \mathrm{Tree}) = \\
\quad \texttt{typecase } t \texttt{ of} \\
\qquad \mathrm{Leaf}(i) \Rightarrow i \\
\qquad \mathrm{Node}\big(\mathrm{Leaf}(l), i, \mathrm{Leaf}(r)\big) \Rightarrow l + i + r \\
\qquad \mathrm{Node}\big(\mathrm{Leaf}(l), i, r\big) \Rightarrow l + i + sum(r) \\
\qquad \mathrm{Node}\big(l, i, \mathrm{Leaf}(r)\big) \Rightarrow sum(l) + i + r \\
\qquad \mathrm{Node}(i, i, r) \Rightarrow sum(l) + i + sum(r) \\
\quad \texttt{end}
\end{array}
$$

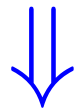
Concise, more readable, and writable!

# Steps to Adding Pattern Matching to Fortress

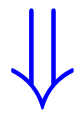
1. Change syntax
   - to define pattern-matchable fields in traits
   - to use patterns in various places such as variable declarations, function expressions, function or method declarations, and `typecase` expressions
2. Implement a pattern-matching desugarer
   - to desugar away the language constructs with patterns besides `typecase` expressions
3. Handle `typecase` expressions
   - to check completeness and redundancy

# Desugaring Patterns in Fortress

$$tree_1 : \mathrm{Tree} = \mathrm{Leaf}(1)$$
$$tree_2 : \mathrm{Node} = \big(\mathrm{Leaf}(3), 4, \mathrm{Leaf}(5)\big)$$
$$tree_3 : \mathrm{Node}\big(\mathrm{Node}(i, j, k), item, right : \mathrm{Leaf}\big) = \mathrm{Node}(tree_2, 6, tree_1)$$



$$tree_3 : \mathrm{Node} = \mathrm{Node}(tree_2, 6, tree_1)$$
$$\big(temp\$1 : \mathrm{Node}(i, j, k), item, right : \mathrm{Leaf}\big) = (tree_3.left, tree_3.item, tree_3.right)$$



$$tree_3 : \mathrm{Node} = \mathrm{Node}(tree_2, 6, tree_1)$$
$$(temp\$1 : \mathrm{Node}, item, right : \mathrm{Leaf}) = (tree_3.left, tree_3.item, tree_3.right)$$
$$(i, j, k) = (temp\$1.\texttt{left}, temp\$1.\texttt{item}, temp\$1.\texttt{right})$$

# `typecase` Expressions

```
trait Tree(item: ℤ32) comprises {Node, Leaf}
  getter depth(): ℤ32
end

sum(t: Tree) =
  typecase t of
    Leaf(i) ⇒ i
    Node(Leaf(l), i, Leaf(r)) ⇒ l + i + r
    Node(Leaf(l), i, r) ⇒ l + i + sum(r)
    Node(l, i, Leaf(r)) ⇒ sum(l) + i + r
    Node(i, i, r) ⇒ sum(l) + i + sum(r)
  end
```

- Matching process at run time
- Completeness and redundancy checks at compile time

# Adding Pattern Matching to Fortress

## Modest Cost to Add

- All patterns, except for in `typecase` expressions, are handled by desugaring.

## Succinct Definition/Use of Patterns

- Complex patterns such as nested patterns are described concisely.

## Static Checks

- Patterns can be checked for completeness and redundancy in `typecase` expressions.

# Future Work

- Static checking
- Patterns for generic types
- Formalizing the pattern matching mechanism
- Code generation of the pattern matching mechanism
- Performance evaluation