DemeterF

Theory and Implementation

Bryan Chadwick

Csg111
12/4/2007

Overview

- Objectives
- Traversal Semantics
- Traversal Example
- Default Functions

Implementation

- Traversal Code
- Dispatch Algorithm
- Dispatch Example
- Future Additions

DemeterF Objectives

- **Functional** (*no assignments*)
- **Limit Traversal Code**
- **Static Types**
- **+ Parallelizable**
- **+ Easier to Analyze**

Traversal Semantics

Traversal Function: $T_{f,\beta,\alpha}$

$$
\begin{array}{lll}
T_{f,\beta,\alpha}(D,\ d_a) & \Rightarrow & \textbf{if}\ D\ \text{ is of } \textit{BuiltIn type}\ \ \textbf{then}\ \ f(D,\ d_a) \\[1em]
 & \Rightarrow & \textbf{if}\ D \equiv (d_0, \cdots, d_n)\ \ \textbf{then} \\
 & & \quad \begin{array}{lll}
 \textbf{let} & d'_a \leftarrow \alpha(D,\ d_a) & - \ \ \text{argument } \textit{update} \\
 & d'_i \leftarrow T_{f,\beta,\alpha}(d_i,\ d'_a) & - \ \ \text{traverse } \textit{fields} \\
 & \hat{D} \leftarrow \beta(D,\ d'_0, \cdots, d'_n,\ d_a) & - \ \ \textit{combine} \\
 \textbf{in} & f(\hat{D},\ d_a) &
 \end{array}
\end{array}
$$

Default Functions

$$id_f(d,\ d_a) \Rightarrow d$$

$$id_\beta(d,\ ...) \Rightarrow error$$

$$\beta_c(D\ , d'_0, \cdots\ , d'_n,\ d_a) \Rightarrow \textbf{new}\ C(d'_0, \cdots\ , d'_n)$$

$$id_\alpha(d,\ d_a) \Rightarrow d_a$$

Example: BinaryTree Height

program.cd:

```
import edu.neu.ccs.demeterf.*;
Main = <t> Tree.

Tree: Node | Leaf.
Node = "(" <d> Integer <l> Tree <r> Tree ")".
Leaf = .

Height = extends ID.
```

program.beh:

```
Height{ {{
    Integer update(Node n, Integer i){ return i+1; }
    Integer combine(Leaf l, Integer i){ return i; }
    Integer combine(Node n, Integer d, Integer l, Integer r){
       return Math.max(l,r);
    }
}} }

Integer h = new Traversal(new Height()).traverse(m.t, 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]            new Node(9,
[2]                      new Node(8,
[3]                                new Leaf(),
[4]                                new Leaf()),
[5]                      new Leaf())
```

Expanded Traversal:

```
traverse([1], 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]            new Node(9,
[2]                      new Node(8,
[3]                                new Leaf(),
[4]                                new Leaf()),
[5]                      new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)

combine([1], traverse([2], a1),
             traverse([5], a1), 0);
```

Example: BinaryTree Height

Input: (9 (8 _ _) _)

Instance:

```
[1]           new Node(9,
[2]                    new Node(8,
[3]                             new Leaf(),
[4]                             new Leaf()),
[5]                    new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
combine([1], combine([2], traverse([3], a2),
                          traverse([4], a2), a1),
             combine([5], a1), 0);
```

Example: BinaryTree Height

Input: (9 (8 _ _) _)

Instance:

```
[1]          new Node(9,
[2]                    new Node(8,
[3]                              new Leaf(),
[4]                              new Leaf()),
[5]                    new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
combine([1], combine([2], combine([3], a2),
                          combine([4], a2), a1),
             a1, 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]            new Node(9,
[2]                      new Node(8,
[3]                                new Leaf(),
[4]                                new Leaf()),
[5]                      new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
combine([1], combine([2], a2,
                          a2, a1),
             a1, 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]          new Node(9,
[2]                  new Node(8,
[3]                           new Leaf(),
[4]                           new Leaf()),
[5]                  new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
combine([1], Math.max(a2,
                      a2),
             a1, 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]          new Node(9,
[2]                  new Node(8,
[3]                          new Leaf(),
[4]                          new Leaf()),
[5]                  new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
combine([1], a2,
             a1, 0);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]            new Node(9,
[2]                      new Node(8,
[3]                                new Leaf(),
[4]                                new Leaf()),
[5]                      new Leaf())
```

Expanded Traversal:

```
a1 = update([1], 0)
a2 = update([2], a1)
Math.max(a2, a1);
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
[1]            new Node(9,
[2]                      new Node(8,
[3]                                new Leaf(),
[4]                                new Leaf()),
[5]                      new Leaf())
```

Expanded Traversal:

```
a2 = update([2], update([1], 0))
```

Example: BinaryTree Height

Input: `(9 (8 _ _ ) _ )`

Instance:

```
new Node(9,
          new Node(8,
                    new Leaf(),
                    new Leaf()),
          new Leaf())
```

Expanded Traversal:

```
a2 = 1+1+0 = 2
```

Traversal Impl.

```
<Ret> Ret traverse(Object o, Option arg){
   List<Field> fl = Util.getFields(c);
   Object ret[] = new Object[fl.size()+1],
          narg = applyA(new Object[]{o, arg});

   ret[0] = o;
   for(int i = 0; i < fl.size(); i++){
       try{
           Field f = fl.get(i);
           Object tret = f.get(o);
           if(!Util.isBuiltIn(tret.getClass())
              /* Traversal Control Goes Here... */)
               tret = traverse(tret, narg);
           else
               tret = applyF(tret, narg);
           ret[i+1] = tret;
       }catch(Exception e){ throw (RuntimeException)e; }
   }
   return (Ret)applyF(applyB(ret, arg), arg);
}
```

Dispatch Impl.

- **Function Object = A set of Methods**
- **Create a List of Method Argument Types**
- **Filter to find only applicable Methods**
- **Sort to find the** *Best* **one**

Dispatch Impl.

Filter:

```
class TypePred implements Pred<MethodEntry>{
    Type subType;
    int argNum;

    TypePred(Type c, int i){ argNum = i; subType = c; }

    boolean huh(MethodEntry e){
        return ((e.numArgs() <= argNum) ||
                e.arg(argNum).isAssignableFrom(subType));
    }
}
```

Dispatch Impl.

Sort:

```
class TypeSort implements Compare<MethodEntry>{

    // Is a MethodEntry *better* than another
    boolean less(MethodEntry e1, MethodEntry e2){

        if(e1.numArgs() > e2.numArgs())return true;
        if(e1.numArgs() < e2.numArgs())return false;

        for(int i = 0; i < e1.numArgs(); i++){
            // SuperTypeOf and NotEqual
            if(e1.arg(i).isAssignableFrom(e2.arg(i)) &&
               !e1.arg(i).equals(e2.arg(i)))
            return false;
        }
        return true;
    }
}
```

Dispatch Impl.

Actual Selection:

```
Method select(Type args[]){
    for(int i = 0; i < args.length; i++){
        left = left.filter(new TypePred(args[i], i));

    return left.sort(new TypeSort()).top();
}
```

Dispatch Example

```
A = <b1> B <b2> B
B: C | D.
C = <i> Integer.
D = <s> String.
```

```
class Build extends IDb{
  String combine(C c, Integer i){ return ""+i; }
  String combine(B b, String s){ return s; }

  String combine(A a, String s){ return s; }
  String combine(A a, String s1, String s2){ return s1+s2; }
}

String s = new Traversal(new Build())
                     .traverse(new A(new C(5), new D("D")));
```

Dispatch Example

```
A = <b1> B <b2> B
B: C | D.
C = <i> Integer.
D = <s> String.

   MethodList M = [ {C, Integer} -> String,
                    {B, String} -> String,
                    {A, String} -> String,
                    {A, String, String} -> String ]

   X = M.select( {C, Integer} ) : Exact Match is the Only Match
   Y = M.select( {D, String} )  : {B, String} is the Only Match
   Z = M.select( {A, X, Y} )    : ?
```

Both {`A, String`} and {`A, String, String`} Match

- Which is more Specific?

Dispatch Example

```
A = <b1> B <b2> B
B: C | D.
C = <i> Integer.
D = <s> String.

   MethodList M = [ {C, Integer} -> String,
                    {B, String} -> String,
                    {A, String} -> String,
                    {A, String, String} -> String ]

   X = M.select( {C, Integer} ) : Exact Match is the Only Match
   Y = M.select( {D, String} )  : {B, String} is the Only Match
   Z = M.select( {A, X, Y} )    : ?
```

Both {`A, String`} and {`A, String, String`} Match

- **Which is more Specific?** {`A, String, String`}

Future

- **Clean up Static Checking (no more runtime 'errors')**
- **Combining Traversals**
- **Paralellize Subtraversals**
- **A few more helpers? (E.g., to make search easier)**

Future

- Clean up Static Checking (no more runtime ’errors’)
- Combining Traversals
- Paralellize Subtraversals
- A few more helpers? (E.g., to make search easier)

Questions?

Future

- Clean up Static Checking (no more runtime 'errors')
- Combining Traversals
- Paralellize Subtraversals
- A few more helpers? (E.g., to make search easier)

Questions?

Thanks!