Functional Traversal Control

Bryan Chadwick

Demeter Seminar
11/8/2007

What

- **Edge-based Traversal Control**
- **Evaluation of a Scheme-Like Language**
- **Optional Traversal Arguments**

Tools

- **Java**
- **DemeterJ**
- *DemeterF*

Traversal

Define: $t_{f,b}$

$$\begin{aligned}
t_{f,b}(d) &\Rightarrow\ d' \\
&\qquad \text{where}\ \ d' = f(d) \\
&\qquad\qquad\quad d\ \ \text{is}\ \ \text{atomic} \\
t_{f,b}(c(d_0, \cdots,\ d_n)) &\Rightarrow\ f(b(c(d_0, \cdots,\ d_n),\ d'_0, \cdots, d'_n)) \\
&\qquad \text{where}\ \ d'_i = t_{f,b}(d_i)
\end{aligned}$$

Traversal with Edge Control

Define: $t_{f,b,e}$

$e : (C \times L \rightarrow bool)$

$$
\begin{aligned}
t_{f,b,e}(d) \Rightarrow\ & d' \\
& \text{where } d' = f(d) \\
& \qquad\quad d \text{ is atomic} \\
t_{f,b,e}(c(l_0 : d_0, \cdots,\ l_n : d_n)) \Rightarrow\ & f(b(c(l_0 : d_0, \cdots,\ l_n : d_n),\ d'_0, \cdots, d'_n)) \\
& \text{where } d'_i = t_{f,b,e}(d_i) \quad \textbf{if} \ \ e(c, l_i) \\
& \qquad\quad d'_i = d_i \quad \textbf{otherwise}
\end{aligned}
$$

The Langauge

```
Exp: Var | Num | If | Lambda | Call | Op.
Var: Sym | Addr.
Sym = <name> Ident.
Addr = "{"<offset> Integer "}" extends Var.

Arg = "(" <type> Type <sym> Sym ")".

Num = <value> Integer.
If = "(if"  <cond> Exp <then> Exp  <otherwise> Exp ")".
Lambda = "(lambda" "(" <formals> ArgList ")" <body> Exp ")".
Call = "(" <proc> Exp <args> ExpList ")".

Op: Plus | Eq.

Plus = "+".
Eq = "=".

Type: BoolT | NumT | FuncT.
BoolT = "bool".
NumT = "int".
FuncT = "(" <args> TypeList "-> " <ret> Type ")".
```

Control

Why Control?

- `If` **: Don't want to evaluate both branches**
- `Lambda` **: Don't want to evaluate the body**
- **Leafs : No need to traverse** *into* **some leafs of our structures (***e.g.***,** `Num`**)**

Implementation

```
Val combine(Num n)
  { return new NumVal(n.value); }
Val combine(Addr a, Integer i, ValList s)
  { return s.lookup(i); }

Val combine(Op op){ return op; }

Val combine(Call c, Op op, ValList args)
  {  return args.fold(op); }

Val combine(Lambda l, ArgList f, Exp b, ValList s)
  { return new LambdaVal(l, s); }

Val combine(Call c, LambdaVal op)
  { return eval(op.proc.body, op.env.push(args)); }

Val combine(If i, NumVal c, Exp t, Exp e, ValList s)
  { return (c.value != 0)?eval(t,s):eval(e,s); }
```

Traversal Use

```
static Traversal trav =
    new Traversal(new EvalFunc(),
            EdgeBypass.create(new Edge(Lambda.class, "formals"),
                              new Edge(Lambda.class, "body"),
                              new Edge(If.class, "then"),
                              new Edge(If.class, "otherwise")));

static Val eval(Exp e, ValList stk)
  { return trav.<Val>traverse(e, stk); }
```

Next Steps

- **Type Checking: when can (dynamic) traversal go wrong?**
- **Formalization: does the traversal implementation match the definition?**
- **Features: Are there any other useful features that should be added?**

Questions?