Algorithms for Traversal-Based Generic Programming

Northeastern University

Bryan Chadwick & Karl Lieberherr

WGP'10
Sept. 26th 2010

Generic Functions for OO

Univeral datatype(s)

- Transform values to/from representation
- Write functions at the *universal* level

OO Types are different

- Abstract classes
- Concrete classes are types too
- OO programmers lift cases

What is TBGP?

An approach to generic programming

Applicable in non-OO contexts...

But for today:

- Generic functions for OO
- Based on Adaptive OO Programming
- *Adaptive, flexible, extensible folds*

Adaptive (OO) Programming

Visitors

- Visit objects along paths
- Collect information in instance variables

Strategies and Paths

- Object/Class *Graphs*
- Strategy selects paths

"Structure Shy"

- Visitors require little structural information
- A deep traversal *adapts* to structures

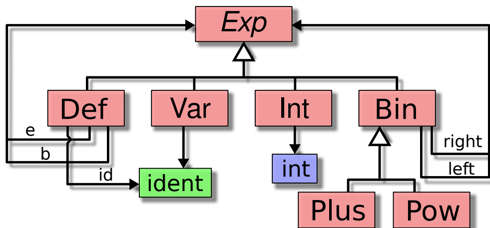
OO Interpretation of TBGP

Merge traversals and functional programming

- Function-objects instead of Visitors
- Adaptive traversal as *deep* fold
- Select functions with multiple dispatch

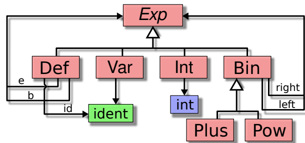
Example Structures: Exps

```
// Class Dictionary
Exp = (Int | Var | Def | Bin).
Int = <v> int.
Var = <id> ident.
Def = <id> ident " = " <e> Exp " ; " <b> Exp.
Bin = (Plus | Pow) <l> Exp <r> Exp.
Plus = "+" .
Pow = "^" .
```



Function Example: `size`

Via Methods in C#




```
// In Exp
abstract int size();
// In Int
override int size(){ return 1; }
// In Var
override int size(){ return 1; }
// In Def
override int size(){ return 1+e.size()+b.size(); }
// In Bin
override int size(){ return 1+l.size()+r.size(); }
```

TBGP Approach with *DemeterF*

Function-Classes

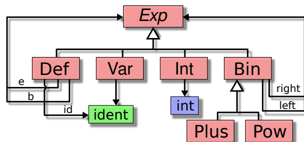
- Define *combine* methods
- Encapsulate computation
- Instances applied by Traversal (*multiple dispatch*)

Traversal

- A deep walk of a data structure instance
- Encapsulates structural recursion/`fold`

Size using DemeterF

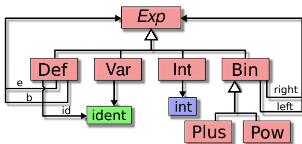
Via a function-class

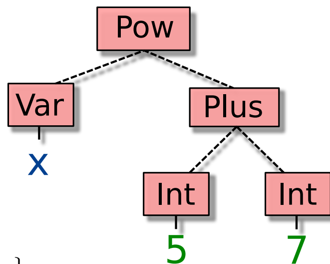



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
  int combine(Def d, ident id, int e, int b)
   { return 1+e+b; }
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e) //** Entry point
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

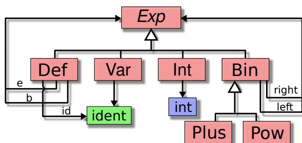


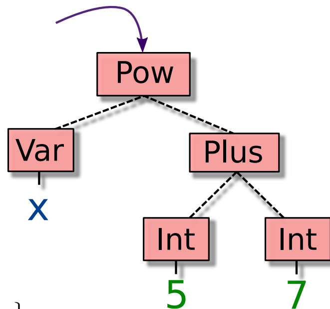



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

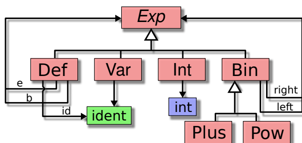


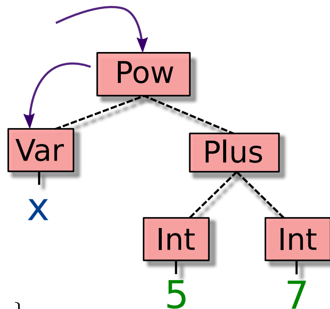



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

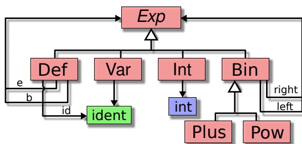


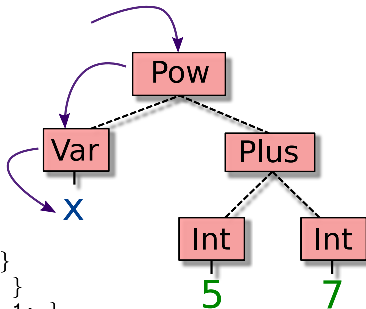



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

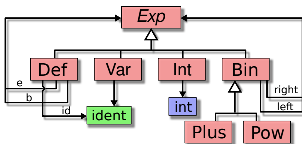


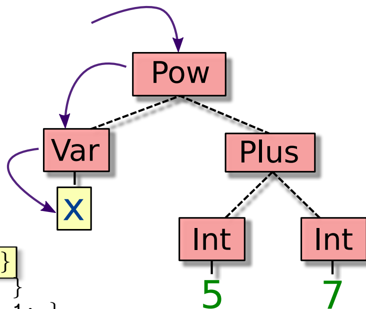



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

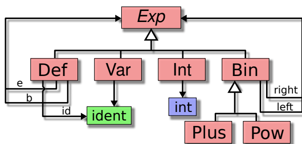


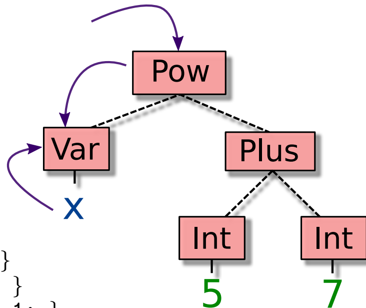



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

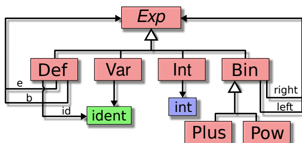


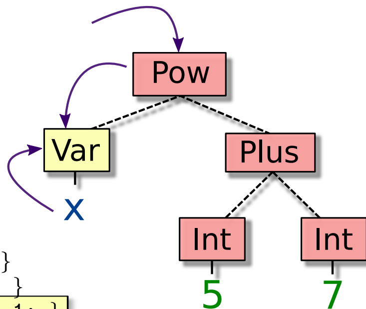



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

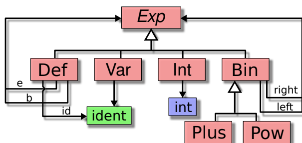


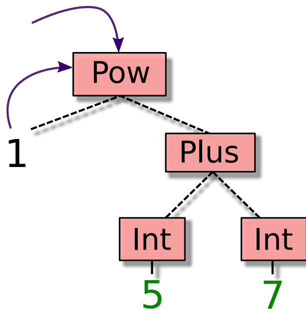



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

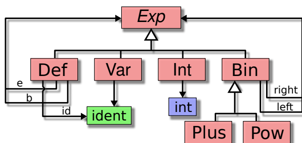


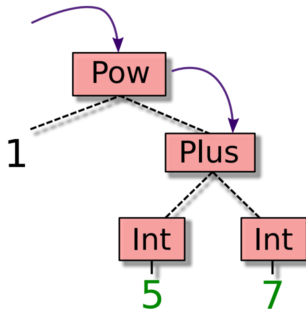



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

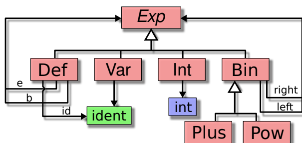


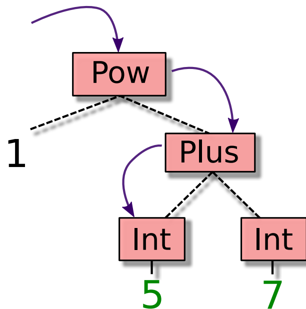



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

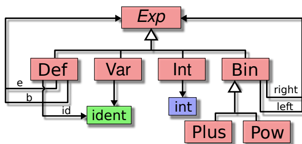


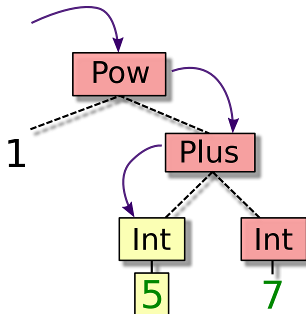



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

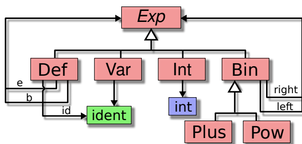


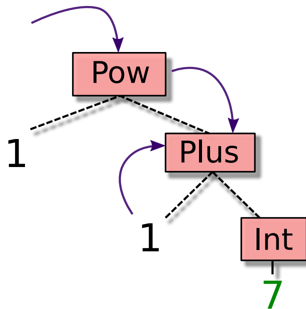



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

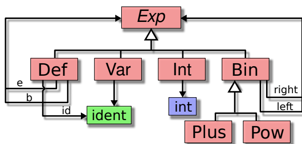


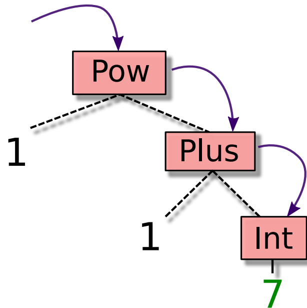



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

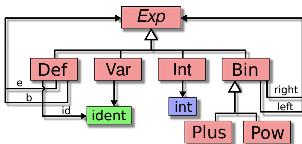


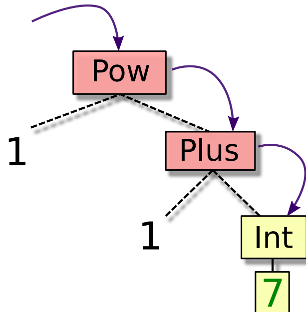



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

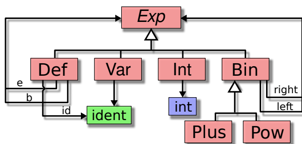


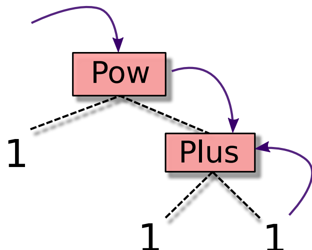



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

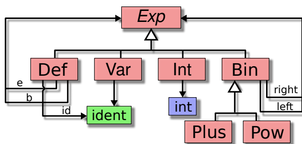


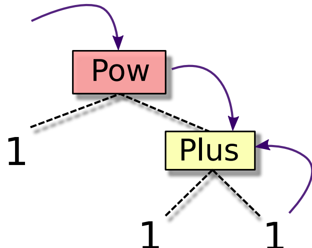



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

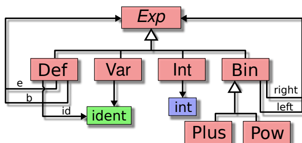


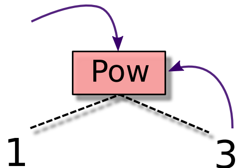



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
   /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

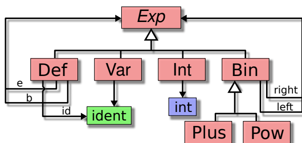


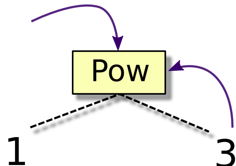



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```



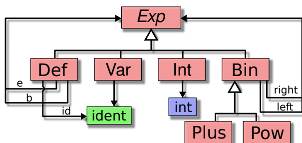



```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
   /* ... Def ... */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

Size Step-through

```
Exp e = new Pow(new Var(new ident("x")),
                new Plus(new Int(5), new Int(7)));
int s = new Size().size(e);
```

5




```
class Size : FC{
  ident combine(ident id){ return id; }
  int combine(Int i, int v){ return 1; }
  int combine(Var v, ident id){ return 1; }
    /* ...  Def ...  */
  int combine(Bin b, int lft, int rht)
   { return 1+lft+rht; }

  int size(Exp e)
   { return new Traversal(this).traverse<int>(e); }
}
```

TBGP: Key Points

combine methods = interesting functionality

- Separate structural recursion
- Implicit invocation
- Encapsulation

Adaptive traversal

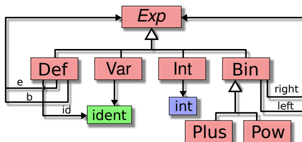
- Like *deep fold*, but more flexible
- Different implementations, same semantics

Generic functions

- *combine* methods are nearsighted

Generic Functions

Type-unifying functions



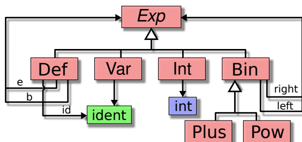

```
class UsedVars : TU<Set<Var>>{
   override Set<Var> combine()
    { return Set.create<Var>(); }
   override Set<Var> fold(Set<Var> a, Set<Var> b)
    { return a.union(b); }

   Set<Var> combine(Var v)
    { return Set.create(v); }
}
```

TU uses `fold` for other compound cases

Like object-oriented SYB *queries*

Generic Functions

Type-preserving functions




```
class Simpler : TP{
   Exp combine(Pow p, Int l, Int r)
    { return new Int((int)Math.Pow(l.v, r.v)); }
   Exp combine(Plus p, Int l, Int r)
    { return new Int(l.v+r.v); }
   Exp combine(Def d, ident id, Exp e, Int b)
    { return b; }
}
```

TP reconstructs other cases

Like object-oriented SYB *transformations*

Algorithms to Support TBGP

Implementation: DemeterF (Java, C#)

Mix of dynamic libraries and static tools

Interesting Algorithms

- Generating useful/abstract function-classes
- Implementing multiple dispatch
- Type checking traversals
- Function/method signature coverage

Function Generation

Model OO datatypes as Class Dictionaries

- Abstract Classes

  $A = T_1 \mid \cdots \mid T_n$
- Concrete Classes

  $C = \langle f_1 \rangle \; T_1 \; \cdots \; \langle f_n \rangle \; T_n$

Functions over CDs that generate functions

- Type-unifying/preserving function-classes
- Haskell-like `Show`
- Different implementations of `Traversal`

Generation Example: TP

Type-preserving function-class template:

```
class TP : FC{
   ∀C ∈ CD . GenTP(C)
}
```

Method Generation:

$\textsc{GenTP}(C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\ ) \rightsquigarrow$

```
   C combine(C _h, T1 f1, ···, Tn fn)
   { return new C(f1, ···, fn); }
```

Generated TP for Exps

```
Exp = (Int | Var | Def | Bin).
Int = <v> int.
    /* ...  */
Bin = (Pow | Plus) <l> Exp <r> Exp.
Pow = .
Plus = .
```

```
class TP : FC{
    Int combine(Int i, int v)
     { return new Int(v); }
           /* ...  */
    Pow combine(Pow p, Exp l, Exp r)
     { return new Pow(l, r); }
    Plus combine(Plus p, Exp l, Exp r)
     { return new Plus(l, r); }
}
```

Generation Example: Show

String conversion function-class template:

```
class Show : FC{
   string combine(int p){ return ""+p; }
        /* ...  */
   ∀C ∈ CD . GenShow(C)
}
```

Method Generation:

$\text{GenShow}(C = \langle f_1 \rangle \; T_1 \; \cdots \; \langle f_n \rangle \; T_n) \; \rightsquigarrow$

```
   string combine(C _h, string f_1, ···, string f_n)
   { return "C("+f_1+","+···+","+f_n+")"; }
```

Generated Show for Exps

```
Exp = (Int | Var | Def | Bin).
Int = <v> int.
    /* ...  */
Bin = (Pow | Plus) <l> Exp <r> Exp.
Pow = .
Plus = .
```

```
class Show : FC{
    string combine(Int i, int v)
     { return "Int("+v+")"; }
           /* ...  */
    string combine(Pow p, string l, string r)
     { return "Pow("+l+","+r+")"; }
    string combine(Plus p, string l, string r)
     { return "Plus("+l+","+r+")"; }
}
```

Generation Example: Traversal

Static traversal template:

```
class Traversal{
   FC fobj;
   Traversal(FC f){ fobj = f; }
   ∀A ∈ CD . GenTrav(A)
   ∀C ∈ CD . GenTrav(C)
}
```

Method Generation:

```
GenTrav( A = T1 | ··· | Tn ) ⇝
   R traverse<R>(A _h){
      if (_h is T1) return traverse<R>((T1)_h);
            ...
      if (_h is Tn) return traverse<R>((Tn)_h);
      throw new Exception("Unknown A Subtype");
   }
```

Generation Example: Traversal

Static traversal template:

```
class Traversal{
   FC fobj;
   Traversal(FC f){ fobj = f; }
   ∀A ∈ CD . GenTrav(A)
   ∀C ∈ CD . GenTrav(C)
}
```

Method Generation:

```
GenTrav( C = ⟨f1⟩ T1 ··· ⟨fn⟩ Tn ) ⇝
   R traverse<R>(C _h){
      object f1 = traverse<object>(_h.f1);
         ...
      object fn = traverse<object>(_h.fn);
      return apply(fobj, new object[]{_h, f1, ..., fn});
   }
```

Generated Traversal for Exps

```
Exp = Int | Var | Def | Bin
Int = <v> int
    /* ...  */
Bin = Pow | Plus
Pow = <l> Exp <r> Exp
Plus = <l> Exp <r> Exp
```

```
class Traversal : FC{
      /* ...  */
   R traverse<R>(Exp _h){
    if (_h is Int) return traverse<R>((Int)_h);
              ...
    if (_h is Bin) return traverse<R>((Bin)_h);
    throw new Exception("Unknown Exp Subtype");
   }
}
```

Generated Traversal for Exps

```
Exp = Int | Var | Def | Bin
Int = <v> int
    /* ...  */
Bin = Pow | Plus
Pow = <l> Exp <r> Exp
Plus = <l> Exp <r> Exp
```

```
class Traversal : FC{
      /* ...  */
   R traverse<R>(Pow _h){
    object l = traverse<object>(_h.l);
    object r = traverse<object>(_h.r);
    return apply(fobj, new object[]{_h, l, r});
  }
}
```

Generated Traversal for Exps

```
Exp = Int | Var | Def | Bin
Int = <v> int
    /* ...  */
Bin = Pow | Plus
Pow = <l> Exp <r> Exp
Plus = <l> Exp <r> Exp
```

```
class Traversal : FC{
      /* ...  */
  R traverse<R>(Pow _h){
    object l = traverse<object>(_h.l);
    object r = traverse<object>(_h.r);
    return apply(fobj, new object[]{_h, l, r});
  }
}
```

Multiple Dispatch

Find the most specific method...

How is DemeterF dispatch implemented?

Two forms:

- Dynamic/reflective *combine* selection
- Statically built dispatch decision trees

Multiple Dispatch

Find the most specific method...

How is DemeterF dispatch implemented?

Two forms:

- Dynamic/reflective *combine* selection
- Statically built dispatch decision trees

Rapid Development

Efficient Execution

Reflective Selection Example

Which method is *most specific* for `(A, D, C)`?

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

Reflective Selection Example

Which method is *most specific* for `(A, D, C)`?

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

Applicable Methods

Reflective Selection Example

Which method is *most specific* for `(A, D, C)`?

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

Most Specific

Applicable Methods

Dispatch Decision Example

Calculate dispatch *residue* at A given (A, B, B)

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .
```

```
class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

```
if(l is C){
   if(r is C)
      return fobj.combine(a, l, (C)r);
   return fobj.combine(a, (C)l, r);
}else{
   if(r is C)
      return fobj.combine(a, l, (C)r);
   return fobj.combine(a, l, r);
}
```

Dispatch Decision Example

Calculate dispatch *residue* at A given `(A, B, B)`

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .
```

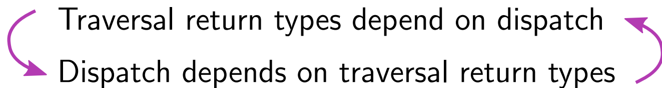
```
class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

Approx. Traversal Types

```
if(l is C){
   if(r is C)
      return fobj.combine(a, l, (C)r);
   return fobj.combine(a, (C)l, r);
}else{
   if(r is C)
      return fobj.combine(a, l, (C)r);
   return fobj.combine(a, l, r);
}
```

TBGP Type Checking

Approximating traversal return types...



Recursive types make this tricky (*e.g.*, `Exp`)

Require unification and dispatch approximation

Return types unify with *combine* arguments

Type Checking Example

Calculate traversal return types starting at `A`

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

Type Checking Example

Calculate traversal return types starting at A

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

$\mathtt{C} \longrightarrow \mathtt{B}$

Type Checking Example

Calculate traversal return types starting at `A`

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

$$C \longrightarrow B$$
$$D \longrightarrow B$$

Type Checking Example

Calculate traversal return types starting at A

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

$C \longrightarrow B$
$D \longrightarrow B$

$B \longrightarrow B$

Type Checking Example

Calculate traversal return types starting at A

```
// CD definitions
A = <l> B <r> B.
B = (C | D).
C = .
D = .

class Test : FC{
  B combine(B b){ return b; }
  C combine(A a, C l, B r){ return l; }
  C combine(A a, B l, C r){ return r; }
  C combine(A a, B l, B r){ return new C(); }
}
```

$C \longrightarrow B$

$D \longrightarrow B$

$B \longrightarrow B$

$A \longrightarrow C$

Type checking and Coverage

Efficient Traversal/Dispatch

Require completeness for safety/soundness

Type checking calculates traversal returns

Coverage checking rules out dispatch errors

Function-Class Coverage

Do all dispatch cases have a `combine` method?

Important for static dispatch correctness

Accuracy important for usability

> *Why handle `List` if you already handle `Cons` and `Empty`*

Function-Class Coverage

Represent type/class hierarchies as trees

$$covers(FC, T_0 \ldots T_n) \equiv$$
$$\forall \vec{a} \in (\texttt{leaves}(T_0) \times \cdots \times \texttt{leaves}(T_n)) \,.\, \exists \vec{m} \in FC \,.\, \vec{a} \leq \vec{m}$$

Simple Solution: force the *top* method signature

$\texttt{combine}(T_0 \; a_1, \ldots, T_n \; a_n) \in FC$

We can do better with a graph algorithm: *Leaf-Covering*

Leaf-Covering

```
class PowFunc : FC{
     /* ...  */
  Bin combine(Pow a, Pow lft, Bin rht){...}
  Bin combine(Pow a, Plus lft, Bin rht){...}
}
```

$covers(\texttt{PowFunc}, (\texttt{Pow}, \texttt{Bin}, \texttt{Bin}))$

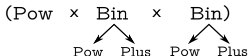
Graph Cartesian Product

Leaf-Covering

```
class PowFunc : FC{
     /* ...  */
  Bin combine(Pow a, Pow lft, Bin rht){...}
  Bin combine(Pow a, Plus lft, Bin rht){...}
}
```

$covers$(PowFunc, (Pow, Bin, Bin))

Graph Cartesian Product

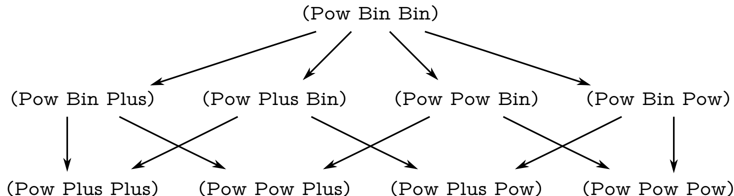


Leaf-Covering

```
class PowFunc : FC{
     /* ...  */
  Bin combine(Pow a, Pow lft, Bin rht){...}
  Bin combine(Pow a, Plus lft, Bin rht){...}
}
```

$covers($ PowFunc, (Pow, Bin, Bin) $)$

Graph Cartesian Product

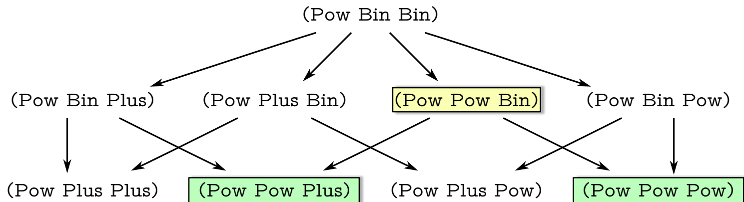


Leaf-Covering

```
class PowFunc : FC{
    /* ...  */
  Bin combine(Pow a, Pow lft, Bin rht){...}
  Bin combine(Pow a, Plus lft, Bin rht){...}
}
```

$$covers(\,\texttt{PowFunc},\,(\texttt{Pow},\texttt{Bin},\texttt{Bin})\,)$$

Graph Cartesian Product

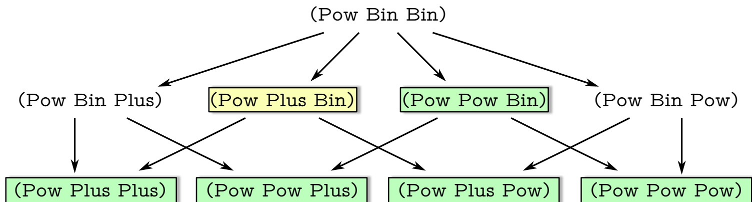


Leaf-Covering

```
class PowFunc : FC{
    /* ...  */
  Bin combine(Pow a, Pow lft, Bin rht){...}
  Bin combine(Pow a, Plus lft, Bin rht){...}
}
```

$covers$(PowFunc, (Pow, Bin, Bin))

Graph Cartesian Product



Leaf-Covering Attributes

Leaf-Covering is *coNP-Complete*

- DNF Validity polynomial-time reduces to Leaf-Covering
- A decision solution can be used to implement *search*

Leaf-Covering is *fixed-parameter tractable*

- Two solutions that dependent on different parameters
  - # of *fields*: brute-force
  - # of *methods*: counting/inclusion-exclusion
- Fixing the parameter makes each solution polynomial

Related Work

Generalized Folds

Meijer, Fokkinga, and Paterson, (1991)

Sheard and Fegaras, (1993)

Lämmel, Visser, and Kort, (2000)

Generic Functional Programming

Jansson and Jeuring, (1997)

Hinze, (1999)

Lämmel and Peyton Jones, (2003,2004)

Related Work

Generic Functional OO Programming

- Moors, Piessens, and Joosen, (2006)
- Oliveira and Gibbons, (2008)

Multiple Dispatch

- CLOS, MultiJava, JPred
- Chambers and Leavens, (1995)
- Chen, Turau, and Klas, (1994)

Conclusions

Traversal-Based Generic Programming

- Extensible generic functions for OO
- *Flexible, safe, adaptive folds*

Interesting Algorithms

- Generating useful function-classes
- Implementing multiple dispatch
- Type checking traversals
- Checking *combine* signature coverage

The End

Thank You

http://www.ccs.neu.edu/home/chadwick/demeterf/

chadwick@ccs.neu.edu