Functional Adaptive Programming

Bryan Chadwick

Thesis Defense

August 18th 2010

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

Adaptive (OO) Programming

Problems

Limited to aggregation

Highly dependent on traversal implementation

Side effects make programs...

- Difficult to understand and extend
- Difficult to compose and reuse
- Difficult to verify or "*type check*"

Alternative/Functional Approaches

Polytypic/Generic Programming

- Write functions over a universal datatype
- Instantiated for specific types

Traversal abstractions

- Generalized folds
- Scrap Your Boilerplate

Alternative/Functional Approaches

Problems

Polytypic/Generic Programming

- Too low level
- Like programming with `Objects` and reflection

Traversal abstractions

- Inflexible traversal
- Restrictive types

A New Approach: DemeterF

Merge traversals and functional programming

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

Thesis Claim

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

Thesis Claim

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Structures, Traversals, and Function-Objects

Thesis Claim

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Structures, Traversals, and Function-Objects
* Usefulness and Flexibility

Thesis Claim

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Structures, Traversals, and Function-Objects
* Usefulness and Flexibility
* Safety and Types

Thesis Claim

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Structures, Traversals, and Function-Objects
* Usefulness and Flexibility
* Safety and Types
* Efficiency/Performance

Data Structures

OO : A Class hierarchy

```
abstract class Bar{ ...  }
class Foo extends Bar{ int i; }
```

FP : Structures/Unions or Type/Value Constructors

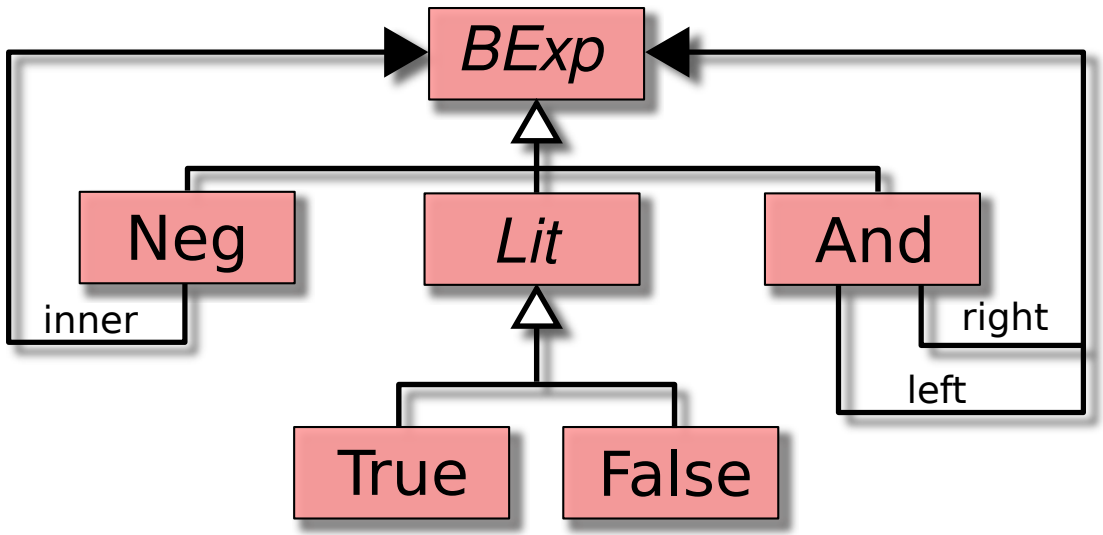
```
;; A Bar is one of Foo, ...
(define-struct Foo (i))

data Bar = Foo Int | ...
```

Java Example: Boolean Exprs

Example: $(true \wedge \neg false)$

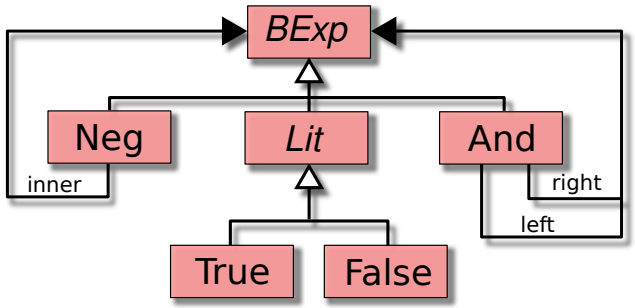
```
abstract class BExp{}
abstract class Lit extends BExp{}
class True extends Lit{}
class False extends Lit{}
```

```
class Neg extends BExp
 { BExp inner; }
class And extends BExp
 { BExp left, right; }
```



Function Example: `Size`

Via Methods




```
// In BExp
abstract int size();
// In Lit
int size(){ return 1; }
// In Neg
int size(){ return 1+inner.size(); }
// In And
int size(){ return 1+left.size()+right.size(); }
```

DemeterF Approach

Function-Classes

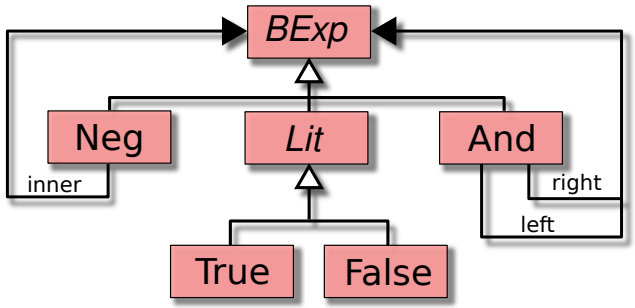
- Define *combine* methods
- Encapsulate computation
- Instances applied by Traversal (*multiple dispatch*)

Traversal

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

Size using DemeterF

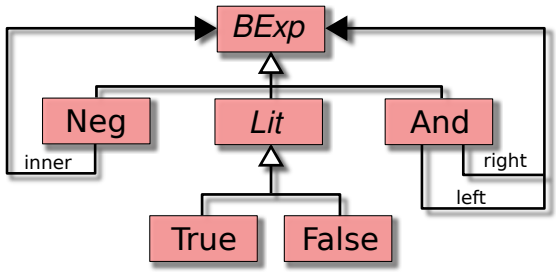
Via a function-class

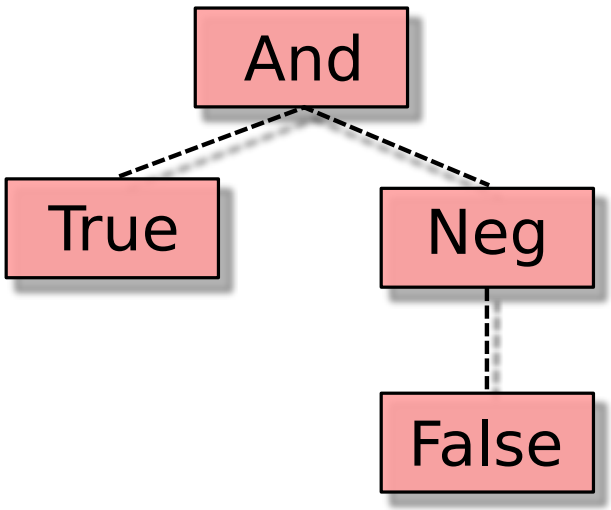



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

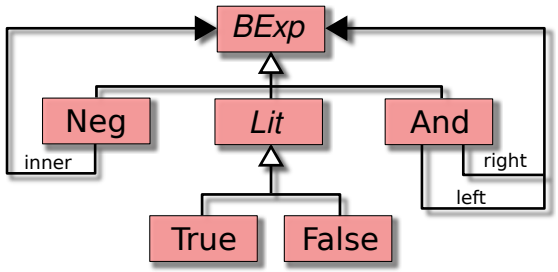


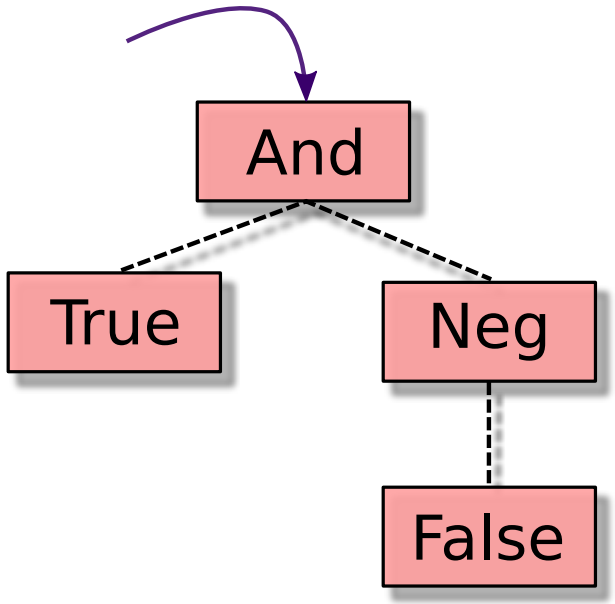



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

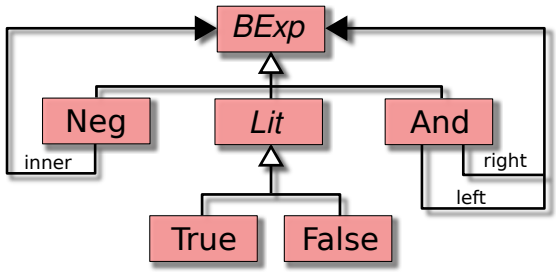


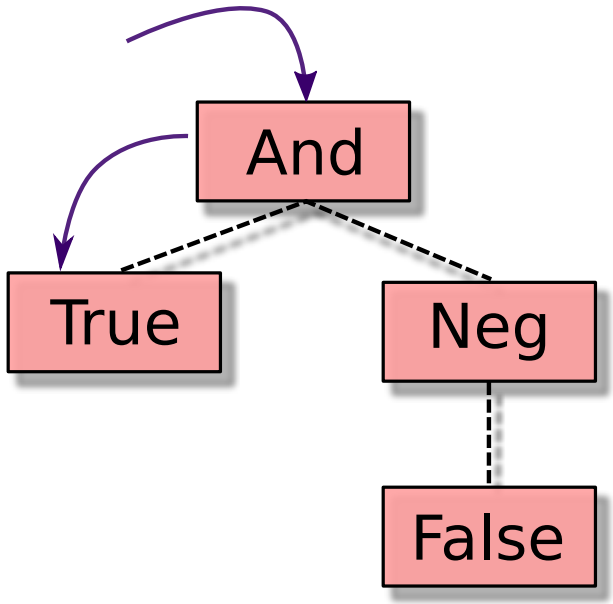



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

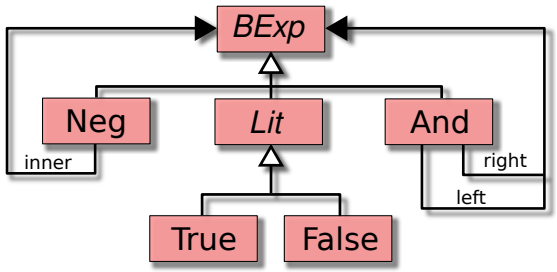


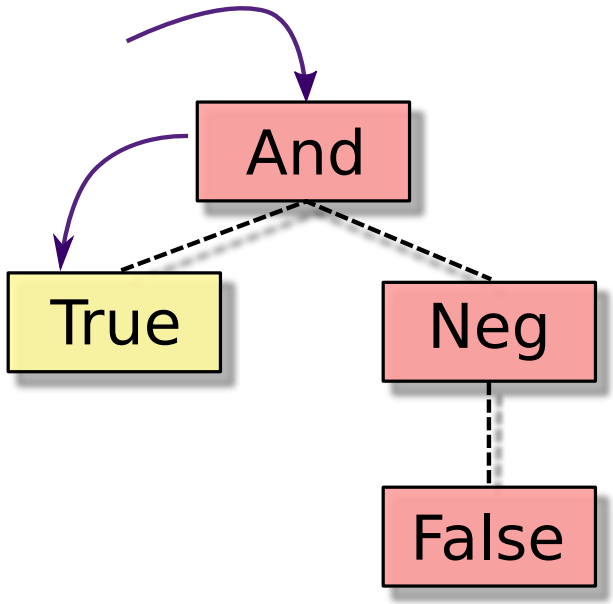



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

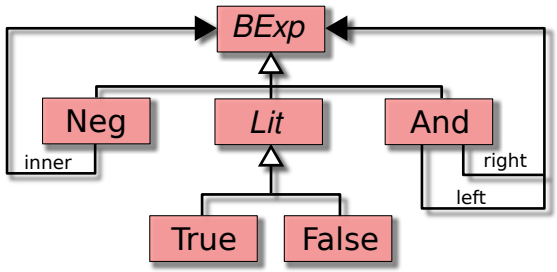


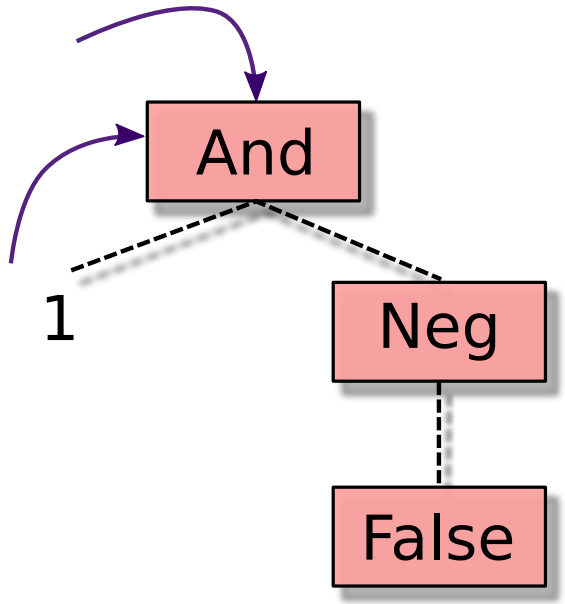



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

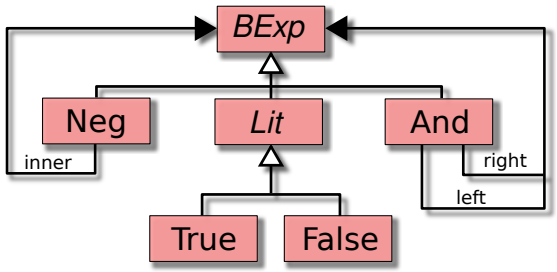


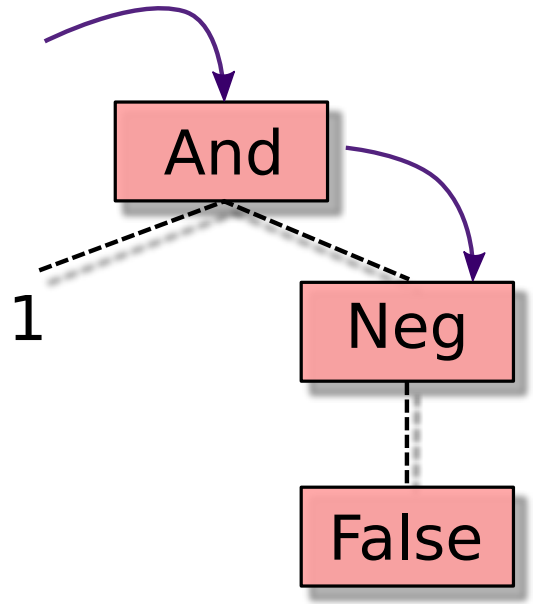



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

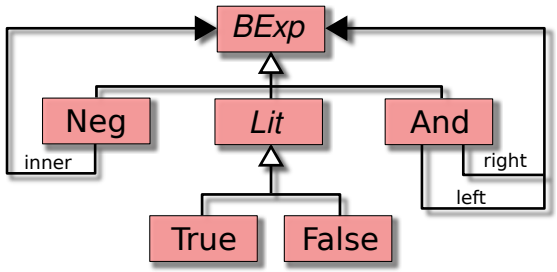


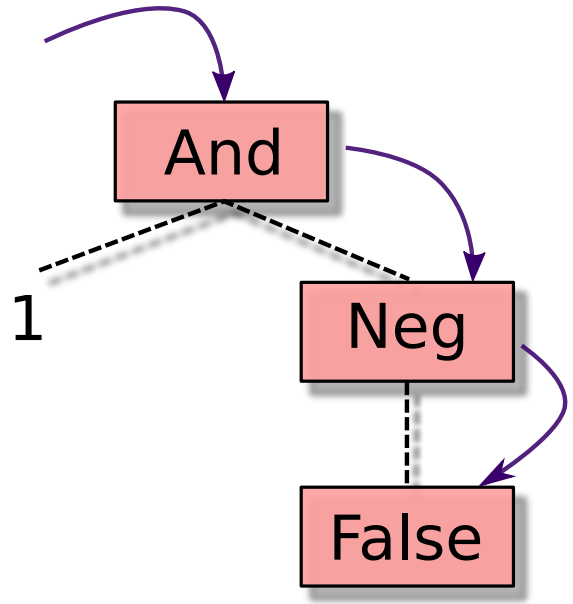



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

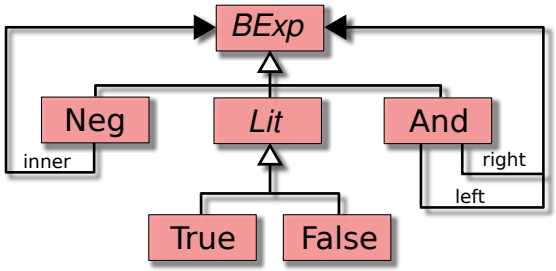


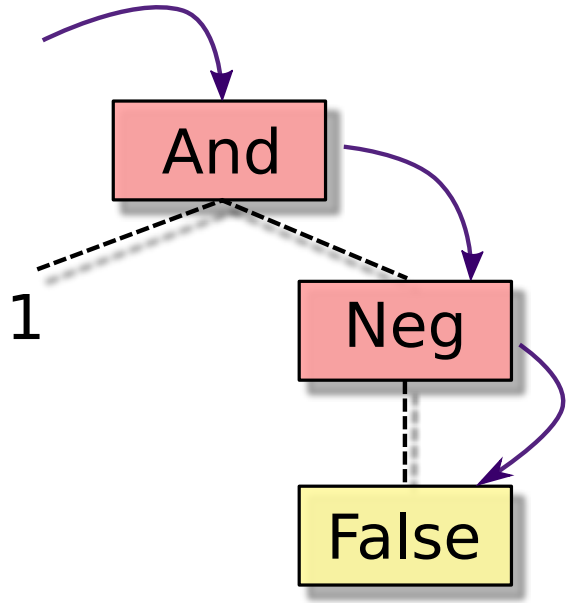



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

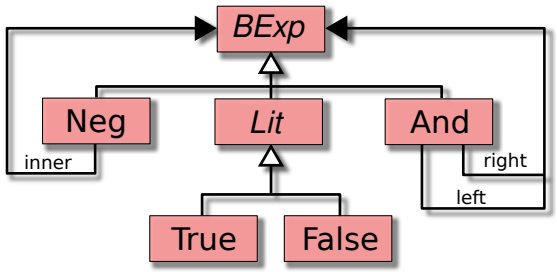


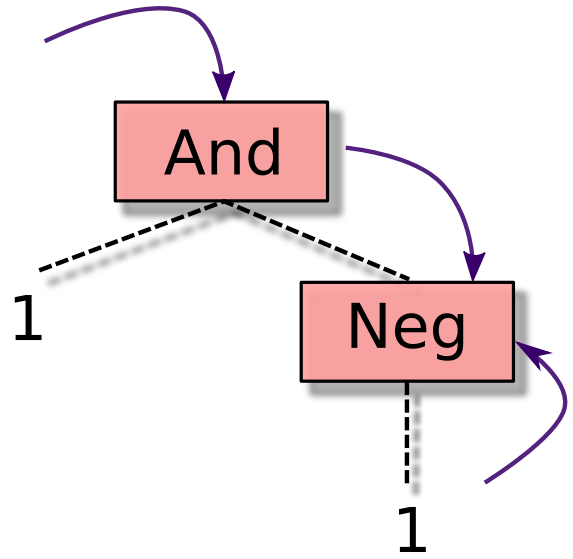



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

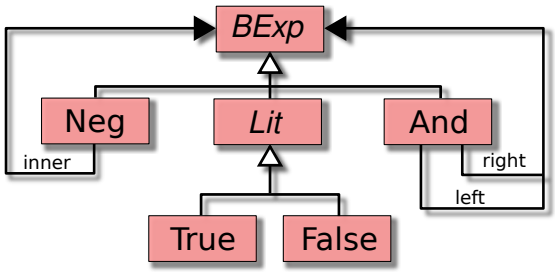


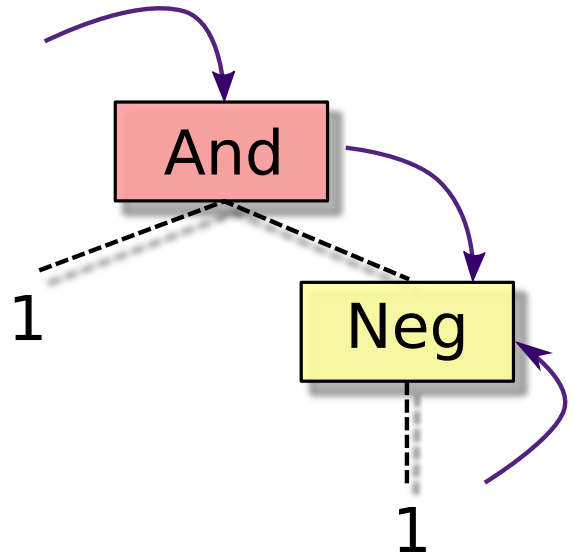



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

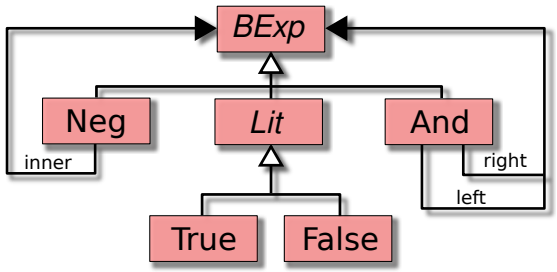


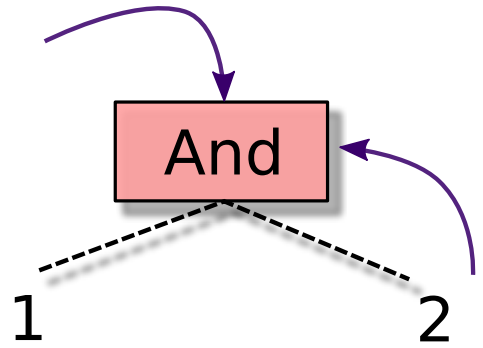



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

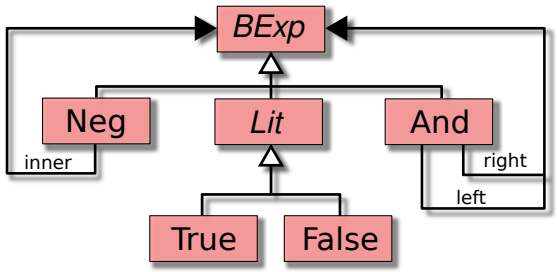


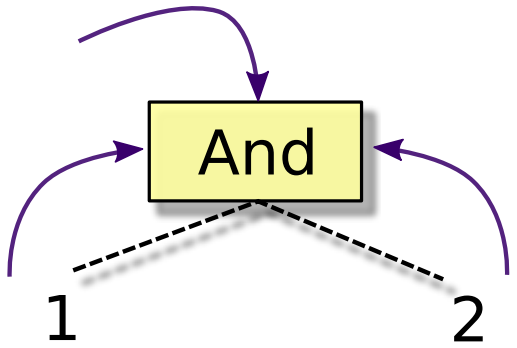



```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```

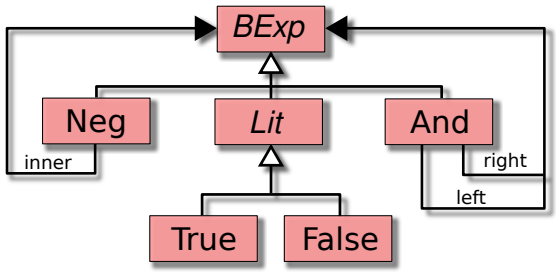





```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Size Step-through

```
BExp e = new And(new True(), new Neg(new False()));
int s = new Size().size(e);
```




```
class Size extends FC{
  int combine(Lit l){ return 1; }
  int combine(Neg n, int inr)
   { return 1+inr; }
  int combine(And a, int lft, int rht)
   { return 1+lft+rht; }

  int size(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

DemeterF: Key Points

combine methods = interesting functionality

- Separate structural recursion
- Implicit invocation

Adaptive traversal

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

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Structures, Traversals, and Function-Objects
* Usefulness and Flexibility
* Safety and Types
* Efficiency/Performance

DemeterF Method Selection

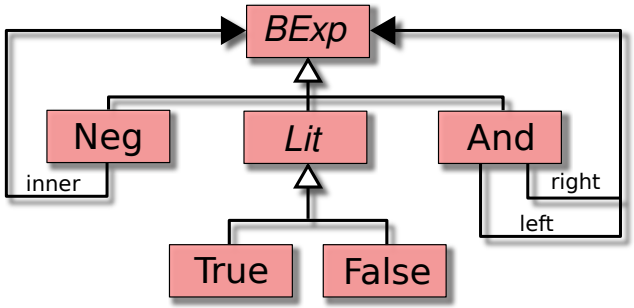
Multiple dispatch

Asymmetric, left-to-right precedence

Supports:

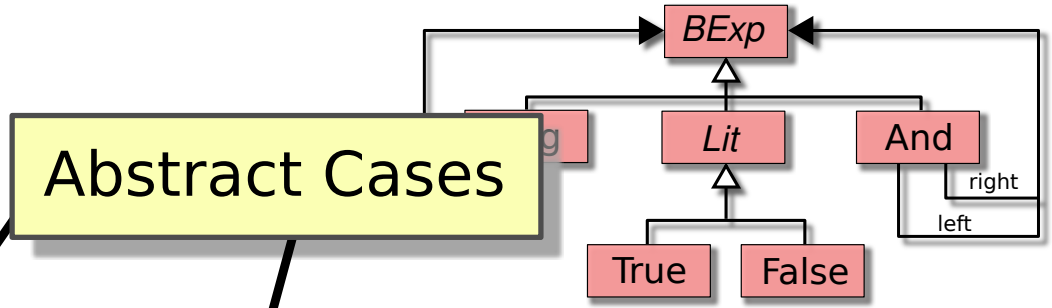
- Case abstraction/overloading
- Function extension
- Mutually recursive structures

Example: Strict Evaluation



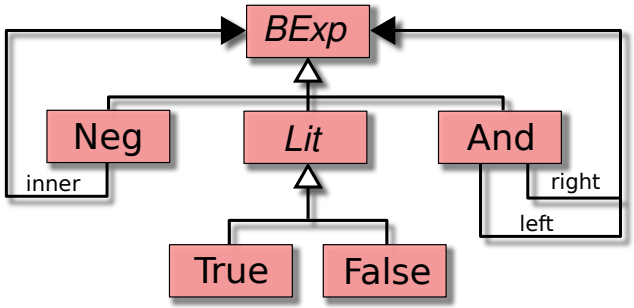

```
class StrictEval extends FC{
  Lit combine(Lit l){ return l; }

  Lit combine(Neg n, True inr){ return new False(); }
  Lit combine(Neg n, False inr){ return new True(); }

  Lit combine(And a, True lft, True rht){ return lft; }
  Lit combine(And a, Lit lft, Lit rht){ return new False(); }
}
```

Example: Strict Evaluation




```
class StrictEval extends FC{
  Lit combine(Lit l){ return l; }

  Lit combine(Neg n, True inr){ return new False(); }
  Lit combine(Neg n, False inr){ return new True(); }

  Lit combine(And a, True lft, True rht){ return lft; }
  Lit combine(And a, Lit lft, Lit rht){ return new False(); }
}
```

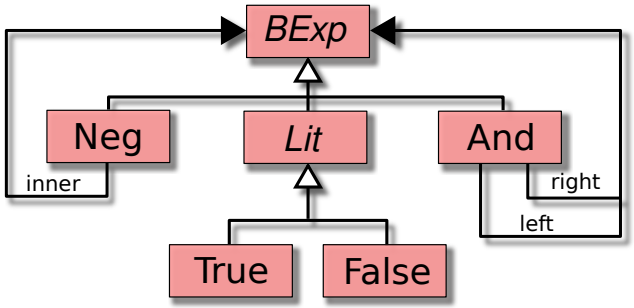
Example: Strict Evaluation




```
class StrictEval extends FC{
  Lit combine(Lit l){ return l; }

  Lit combine(Neg n, True inr){ return new False(); }
  Lit combine(Neg n, False inr){ return new True(); }

  Lit combine(And a, True lft, True rht){ return lft; }
  Lit combine(And a, Lit lft, Lit rht){ return new False(); }
}
```

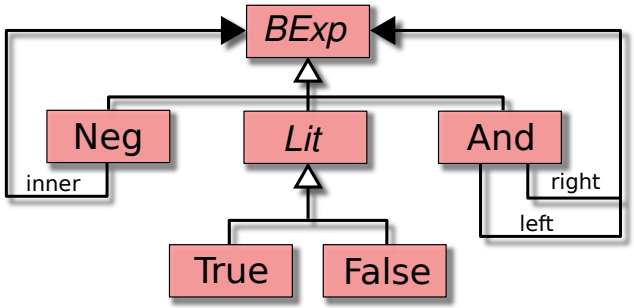
Specialized Cases

Example: Strict Eval. Alt.




```
class StrictEvalAlt extends FC{
  Lit combine(Lit l){ return l; }

  Lit combine(Neg n, True inr){ return new False(); }
  Lit combine(Neg n, False inr){ return new True(); }

  Lit combine(And a, True lft, Lit rht){ return rht; }
  Lit combine(And a, False lft, Lit rht){ return lft; }
}
```

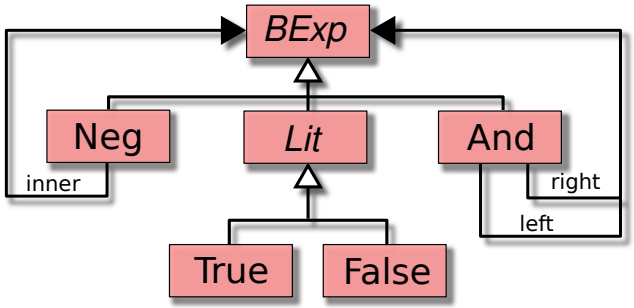
Example: Strict Eval. Alt.




```
class StrictEvalAlt extends FC{
  Lit combine(Lit l){ return l; }

  Lit combine(Neg n, True inr){ return new False(); }
  Lit combine(Neg n, False inr){ return new True(); }

  Lit combine(And a, True lft, Lit rht){ return rht; }
  Lit combine(And a, False lft, Lit rht){ return lft; }
}
```

Function Extension

Use *inheritance* to build functions



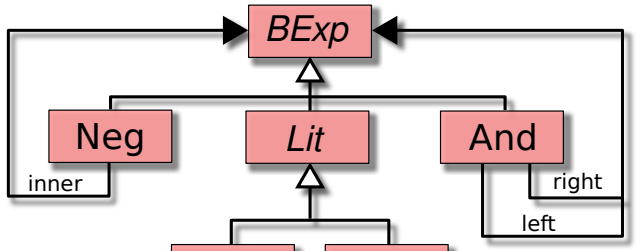

```
class Copy extends FC{
    Lit combine(Lit l){ return l; }
    Neg combine(Neg n, BExp inr)
     { return new Neg(inr); }
    And combine(And a, BExp lft, BExp rht)
     { return new And(lft, rht); }
}
```

```
class Simplify extends Copy{
    Lit combine(Neg n, True inr)
     { return new False(); }
    Lit combine(Neg n, False inr)
     { return new True(); }
    BExp combine(Neg n, Neg inr)
     { return inr.inner; }
}
```

Function Extension

Use *inheritance* to build functions

```
class Copy extends FC{
    Lit combine(Lit l){ return l; }
    Neg combine(Neg n, BExp inr)
     { return new Neg(inr); }
    And combine(And a, BExp lft, BExp rht)
     { return new And(lft, rht); }
}
```



Common Cases

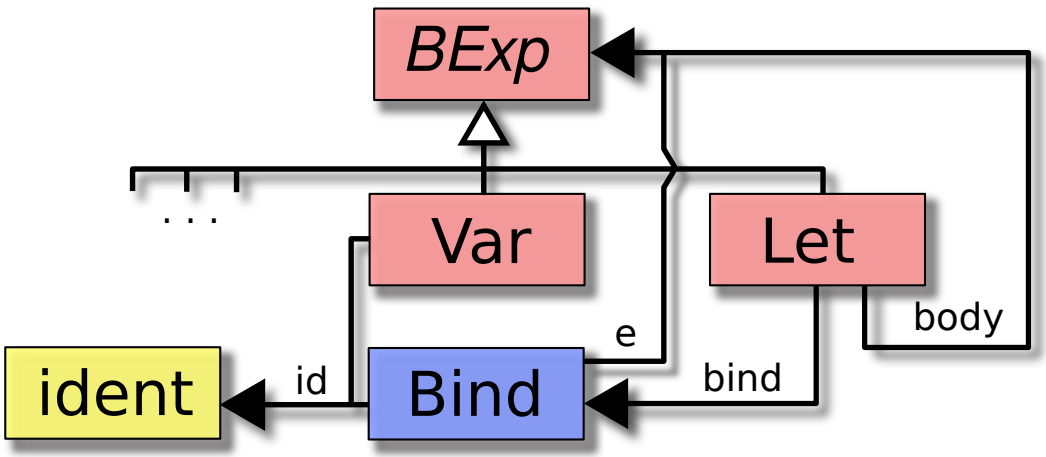
Special Cases

```
class Simplify extends Copy{
    Lit combine(Neg n, True inr)
     { return new False(); }
    Lit combine(Neg n, False inr)
     { return new True(); }
    BExp combine(Neg n, Neg inr)
     { return inr.inner; }
}
```

Mutual Recursion

Example: `let` $a = false$ `in` $(true \wedge \neg a)$

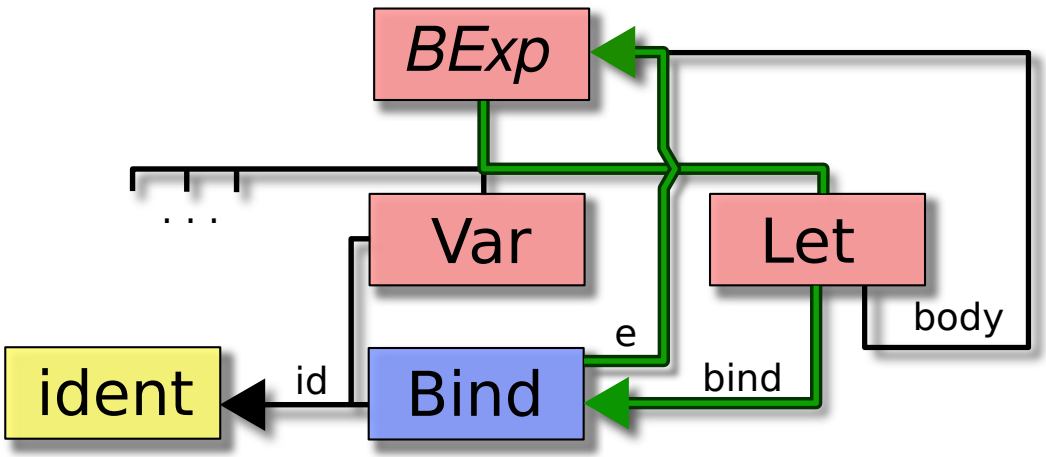
```
class Var extends BExp{ ident id; }
class Let extends BExp{ Bind bind; BExp body; }
class Bind{ ident id; BExp e; }
```



Mutual Recursion

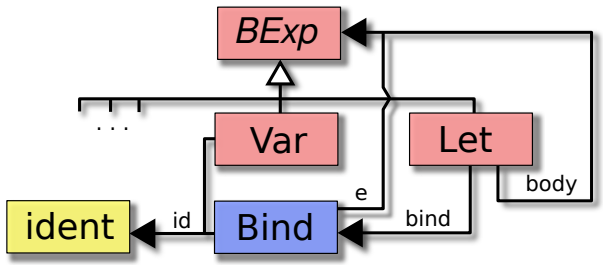
Example: `let` $a = false$ `in` $(true \wedge \neg a)$

```
class Var extends BExp{ ident id; }
class Let extends BExp{ Bind bind; BExp body; }
class Bind{ ident id; BExp e; }
```



Mutual Recursion: SizeWLet

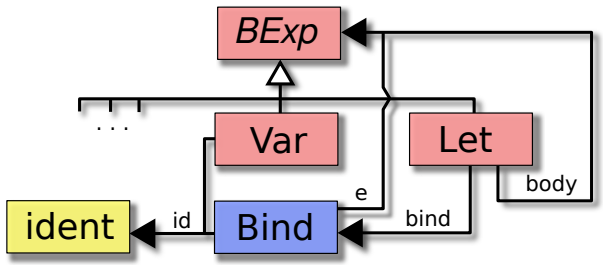
Self and *mutual* recursion are the same




```
class SizeWLet extends Size{
    int combine(ident id){ return 1; }
    int combine(Var v, int id)
     { return id; }
    int combine(Let l, int bnd, int bdy)
     { return 1+bnd+bdy; }
    int combine(Bind l, int id, int e)
     { return id+e; }
}
```

Mutual Recursion: CopyWLet

Self and *mutual* recursion are the same

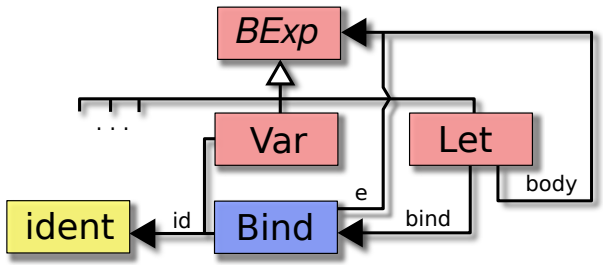



```
class CopyWLet extends Copy{
    ident combine(ident id){ return id; }
    Var combine(Var v, ident id)
     { return new Var(id); }
    Let combine(Let l, Bind bnd, BExp bdy)
     { return new Let(bnd, bdy); }
    Bind combine(Bind b, ident id, BExp e)
     { return new Bind(id, e); }
}
```

Mutual Recursion: CopyWLet

Self and *mutual* recursion are the same

Different Types




```
class CopyWLet extends Copy{
   ident combine(ident id){ return id; }
   Var combine(Var v, ident id)
    { return new Var(id); }
   Let combine(Let l, Bind bnd, BExp bdy)
    { return new Let(bnd, bdy); }
   Bind combine(Bind b, ident id, BExp e)
    { return new Bind(id, e); }
}
```

Polytypic Functions

Two common cases [Lämmel, PADL'02]:

- Type-Preserving: *Transformations* like `map`
- Type-Unifying: *Queries*, similar to `fold`

In DemeterF these are function-classes

- `TP`: like a generic `Copy`
- `TU`: like a parametrized `Size`
- Programmers extend with *`combine`* cases

Extra Features

Traversal Control

- Short-cutting recursion like in AP

Traversal Contexts

- Like inherited attributes
- For non-compositional functions

Generate function-classes from structures

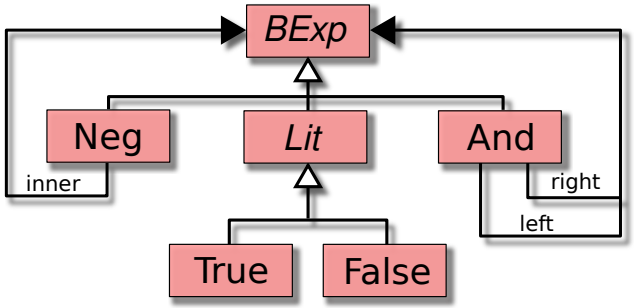
- `TP` and `TU`
- `Show` and `HashCode`

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Traversals and Function-Classes/Objects
* Usefulness and Flexibility
* Safety and Types
* Efficiency/Performance

What can go wrong?

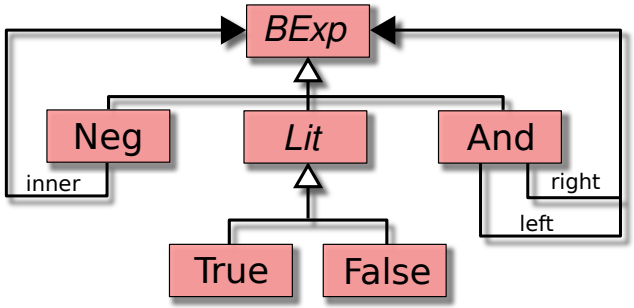
Missing method cases:




```
class Bad extends FC{
    Lit combine(Lit l){ return l; }
    Lit combine(And a, False lft, Lit rht)
     { return lft; }
}

new Traversal(new Bad())
  .traverse(new And(new True(), new False()))
```

What can go wrong?

Missing method cases:

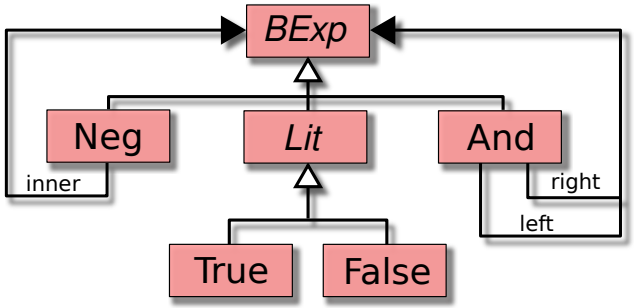



```
class Bad extends FC{
    Lit combine(Lit l){ return l; }
    Lit combine(And a, False lft, Lit rht)
     { return lft; }
}

new Traversal(new Bad())
  .traverse(new And(new True(), new False()))
```

```
DemeterF: Did not find a match for:
    Bad.combine(And, True, False)
```

How can we prevent it?

Type Checking



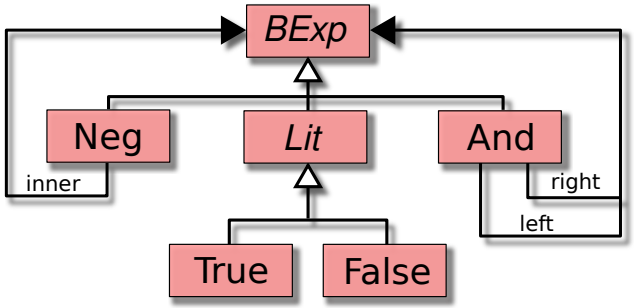

```
class Bad extends FC{
    Lit combine(Lit l){ return l; }
    Lit combine(And a, False lft, Lit rht)
     { return lft; }
}
```

Check return types vs. what is handled

Example: `Bad` traversing an `And`...

How can we prevent it?

Type Checking




```
class Bad extends FC{
    Lit combine(Lit l){ return l; }
    Lit combine(And a, False lft, Lit rht)
     { return lft; }
}
```

Check return types vs. what is handled

Example: `Bad` traversing an `And`...

```
!!Type Error
    No possible combines for:
        (And, True, Lit)
        (Neg, *)
```

Is it Sound? Prove it...

Build a Model

- Minimal Syntax
- Reduction Semantics

Impose a Type System

- Axioms and Typing Rules

Is it Sound? Prove it...

Build a Model

- Minimal Syntax
- Reduction Semantics

When can things go wrong??

Impose a Type System

- Axioms and Typing Rules

Prove that well-typed programs do not have these problems...

A DemeterF Model: Syntax 1

Structures/Types: T

$$\texttt{abstract}\ A\ \texttt{=}\ T_0\ \texttt{|}\ \ldots\ \texttt{|}\ T_n\,.$$
$$\texttt{concrete}\ C\ \texttt{=}\ T_1\ \texttt{*}\ \ldots\ \texttt{*}\ T_n\,.$$

Expressions: e

$$e ::= x \mid \texttt{new}\ C\texttt{(}e_1\texttt{, }\ldots\texttt{, }e_n\texttt{)} \mid \texttt{traverse(}\,e_0\texttt{, }F\,\texttt{)}$$

Functions/Sets

$$F ::= \texttt{funcset\{}\ f_1 \ldots f_n\ \texttt{\}}$$
$$f ::= \texttt{(}T_0\ x_0\texttt{, }\ldots\texttt{, }T_n\ x_n\texttt{)}\ \texttt{\{ return }\ e_0\texttt{; \}}$$

A DemeterF Model: Syntax 1

Structures/Types: T

$$\texttt{abstract}\ A\ \texttt{=}\ T_0\ \texttt{|}\ \ldots\ \texttt{|}\ T_n\texttt{.}$$
$$\texttt{concrete}\ C\ \texttt{=}\ T_1\ \texttt{*}\ \ldots\ \texttt{*}\ T_n\texttt{.}$$

Subtyping

Expressions: e

$$e ::= x \mid \texttt{new}\ C\texttt{(}e_1\texttt{, }\ldots\texttt{, }e_n\texttt{)} \mid \texttt{traverse(}e_0\texttt{, }F\texttt{)}$$

Functions/Sets

$$F ::= \texttt{funcset\{}\ f_1 \ldots f_n\ \texttt{\}}$$
$$f ::= \texttt{(}T_0\ x_0\texttt{, }\ldots\texttt{, }T_n\ x_n\texttt{)}\ \texttt{\{ return}\ e_0\texttt{; \}}$$

Function-Objects

A DemeterF Model: Syntax 2

Values: v

$$v ::= \texttt{new}\ C(v_1, \ldots, v_n)$$

Runtime Expressions

$$\begin{aligned} e ::= & \ldots \\ & \mid \texttt{dispatch}(F, v_0, e_1, \ldots, e_n) \\ & \mid \texttt{apply}(f, v_0, v_1, \ldots, v_n) \end{aligned}$$

Evaluation Contexts: E

$$\begin{aligned} E ::= & [\,] \mid \texttt{new}\ C(v \ldots, E, e \ldots) \\ & \mid \texttt{traverse}(E, F) \\ & \mid \texttt{dispatch}(F, v_0, v \ldots, E, e \ldots) \end{aligned}$$

A DemeterF Model: Semantics

Reduction Rules

[R-Trav]

$$\texttt{traverse}(\, v_0\,,\, F\,) \rightarrow \texttt{dispatch}(\, F\,,\, v_0\,,\texttt{traverse}(\, v_1\,,\, F\,)\,,\ldots,\texttt{traverse}(\, v_n\,,\, F\,))$$

where $v_0 = \texttt{new}\ C(v_1\,,\ldots,v_n)$

[R-Dispatch]

$$\texttt{dispatch}(\, F\,,\, v_0\,,\, v_1\,,\ldots,\, v_n) \rightarrow \texttt{apply}(\, f\,,\ v_0\,,\, v_1\,,\ldots,\, v_n\,) \quad \text{if } f \neq \texttt{error}$$

where $f = \mathit{choose}(F,\, \mathit{types}(\, v_0\ v_1 \ldots\ v_n))$

[R-Apply]

$$\texttt{apply}(\, f\,,\, v_0\,,\, v_1\,,\ldots,\, v_n\,) \rightarrow e[\overline{v_i/x_i}]$$

where $f = (T_0\ x_0\,,\ldots,\ T_n\ x_n)\,\{\,\texttt{return}\ e;\,\}$

A DemeterF Model: Semantics

Reduction Rules

[**R-Trav**]

$\texttt{traverse}(\, v_0 \,,\, F \,) \rightarrow$
$\quad \texttt{dispatch}(\, F \,,\, v_0 \,, \texttt{traverse}(\, v_1 \,,\, F \,) \,, \ldots \,, \texttt{traverse}(\, v_n \,,\, F \,))$
where $v_0 = \texttt{new}\ C(v_1 \,, \ldots \,, v_n)$

[**R-Dispatch**]

$\texttt{dispatch}(\, F \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n) \rightarrow \texttt{apply}(\, f \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n \,)$ if $f \neq \texttt{error}$
where $f = \mathit{choose}(F, \mathit{types}(\, v_0 \; v_1 \ldots v_n))$

[**R-Apply**]

$\texttt{apply}(\, f \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n \,) \rightarrow e[\overline{v_i / x_i}]$
where $f = (T_0 \; x_0 \,, \ldots \,, T_n \; x_n) \,\{\, \texttt{return}\ e; \,\}$

A DemeterF Model: Semantics

Reduction Rules

[R-Trav]

$\texttt{traverse}(\, v_0 \,,\, F \,) \rightarrow$
$\quad \texttt{dispatch}(\, F \,,\, v_0 \,, \texttt{traverse}(\, v_1 \,,\, F \,) \,, \ldots \,, \texttt{traverse}(\, v_n \,,\, F \,))$
where $v_0 = \texttt{new}\ C(v_1 \,, \ldots \,, v_n)$

[R-Dispatch]

$\texttt{dispatch}(\, F \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n) \rightarrow \texttt{apply}(\, f \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n \,)$ if $f \neq \texttt{error}$

where $f = \mathit{choose}(F, \mathit{types}(\, v_0 \; v_1 \ldots v_n))$

[R-Apply]

$\texttt{apply}(\, f \,,\, v_0 \,,\, v_1 \,, \ldots \,, v_n \,) \rightarrow e[\overline{v_i / x_i}]$

where $f = (T_0 \; x_0 \,, \ldots \,, T_n \; x_n) \,\{\, \texttt{return}\ e; \,\}$

Stuck?

Type System

Well Typed: three judgments

$$\Gamma \vdash_e e : T \quad \text{Expression Typing}$$

$$\Gamma \vdash_F f : T \quad \text{Function Return Typing}$$

$$\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle T, F \rangle : T' \quad \text{Traversal Results using } F$$

Important Typing Rules

T-Trav $\Gamma \vdash_e \texttt{traverse(}\, e_0\texttt{,}\, F\,\texttt{)} : T$

1. Find the type of e_0 (*e.g.*, T_0)
2. Check that traversal of T_0 with F returns T

Important Typing Rules

T-Trav $\Gamma \vdash_e \texttt{traverse(}\, e_0\texttt{,}\ F\,\texttt{)} : T$

1. Find the type of e_0 (*e.g.*, T_0)
2. Check that traversal of T_0 with F returns T

T-CTrav $\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F \rangle : T'$

1. Definition: $\texttt{concrete}\ C\ \texttt{=}\ T_1\ \texttt{*}\ \ldots\ \texttt{*}\ T_n\texttt{.}$
2. Check recursive traversals T_i (*e.g.*, T_i')
3. For $f \in possibleFs(F, (C\ T_1' \ldots T_n'))$
   $\Gamma \vdash_F f : T_f$ and $T_f \leq T$
4. Ensure complete coverage: $covers(F, (C\ T_1' \ldots T_n'))$

Important Typing Rules

T-Trav $\Gamma \vdash_e \texttt{traverse(}\ e_0\texttt{,}\ F\ \texttt{)} : T$

1. Find the type of e_0 (*e.g.*, T_0)
2. Check that traversal of T_0 with F returns T

T-CTrav $\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F \rangle : T'$

1. Definition: $\texttt{concrete}\ C\ \texttt{=}\ T_1\ \texttt{*}\ \ldots\ \texttt{*}$
2. Check recursive traversals T_i (*e.g.*, T_i')
3. For $f \in$ $possibleFs(F, (C\ T_1' \ldots T_n'))$
   $\Gamma \vdash_F f : T_f$ and $T_f \leq T$
4. Ensure complete coverage: $covers(F, (C\ T_1' \ldots T_n'))$

Subtypes or Supertypes of Function Signatures

Concrete Signature Coverage

Side Point: *Accurate Coverage*

LeafCovering: All concrete cases have a function

$$covers(F, (T_0 \ldots T_n)) \Leftrightarrow$$
$$\forall\, C_0, \ldots, C_n \text{ with } C_i \leq T_i \,.\, possibleFs(F, (C_0 \ldots C_n)) \neq \emptyset$$

Simple Solution: force the *top* method signature

$$(T_0\ x_0\texttt{, }\ldots\texttt{ , }T_n\ x_n)\ \{\ \texttt{return e;}\ \} \in F$$

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

Leaf-Covering

```
class StrictEval extends FC{
        /* ...  */
  Lit combine(And a, True lft, True rht){...}
  Lit combine(And a, Lit lft, Lit rht){...}
}
```

$covers(\texttt{StrictEval}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$

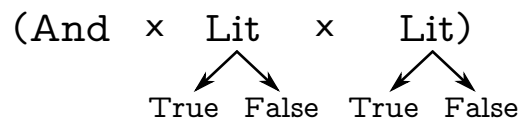
Graph Cartesian Product

Leaf-Covering

```
class StrictEval extends FC{
        /* ...  */
  Lit combine(And a, True lft, True rht){...}
  Lit combine(And a, Lit lft, Lit rht){...}
}
```

$$covers(\texttt{StrictEval}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$$

Graph Cartesian Product

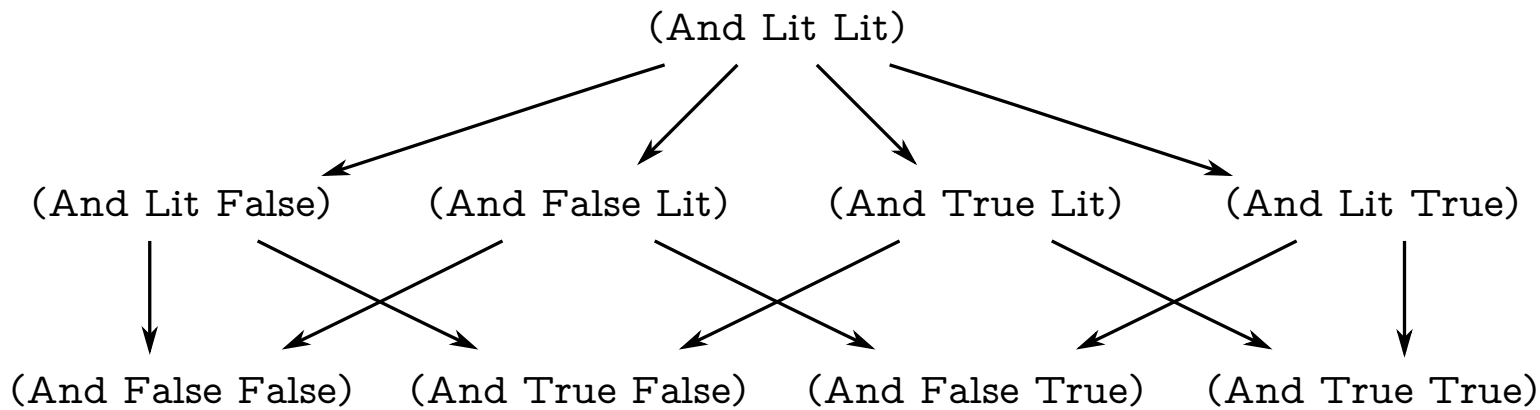


Leaf-Covering

```
class StrictEval extends FC{
        /* ...  */
  Lit combine(And a, True lft, True rht){...}
  Lit combine(And a, Lit lft, Lit rht){...}
}
```

$$covers(\texttt{StrictEval}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$$

Graph Cartesian Product

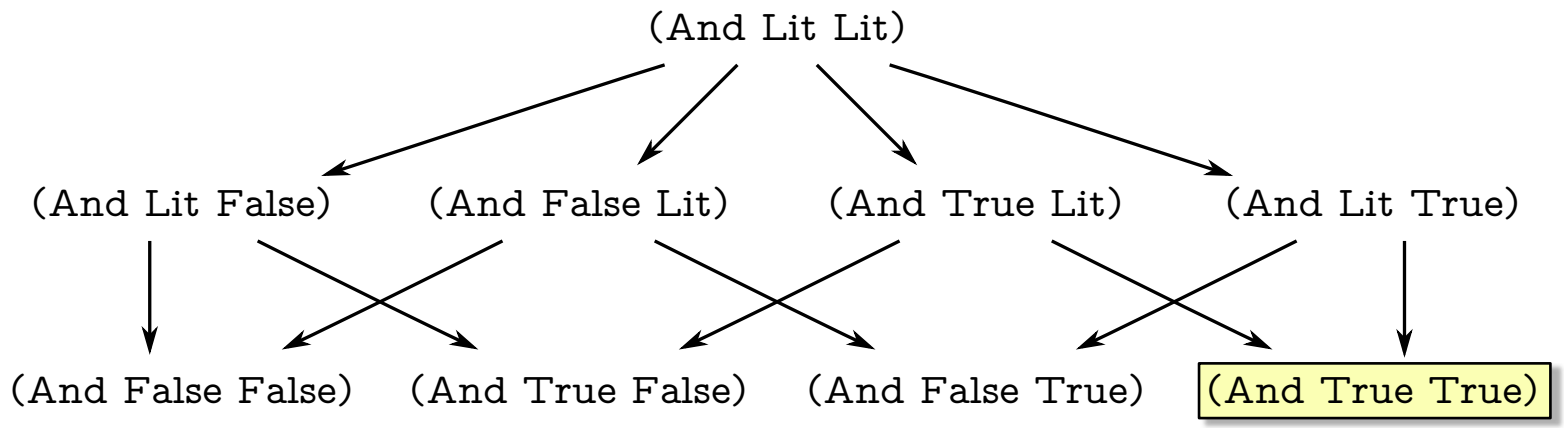


Leaf-Covering

```
class StrictEval extends FC{
        /* ...  */
  Lit combine(And a, True lft, True rht){...}
  Lit combine(And a, Lit lft, Lit rht){...}
}
```

$$covers(\texttt{StrictEval}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$$

Graph Cartesian Product

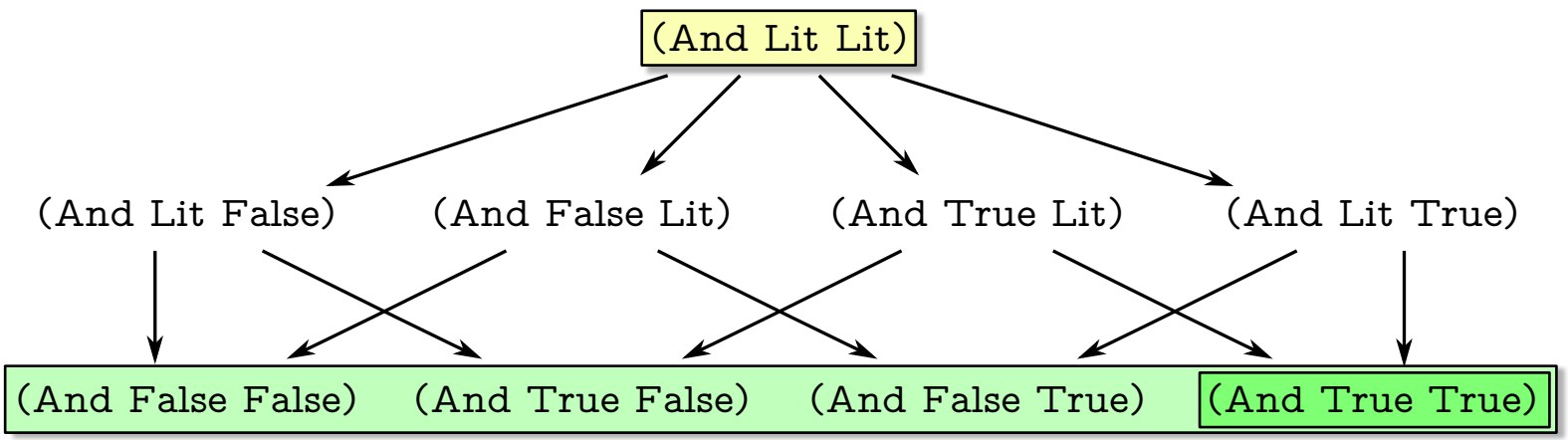


Leaf-Covering

```
class StrictEval extends FC{
        /* ...  */
  Lit combine(And a, True lft, True rht){...}
  Lit combine(And a, Lit lft, Lit rht){...}
}
```

$covers(\texttt{StrictEval}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$

Graph Cartesian Product

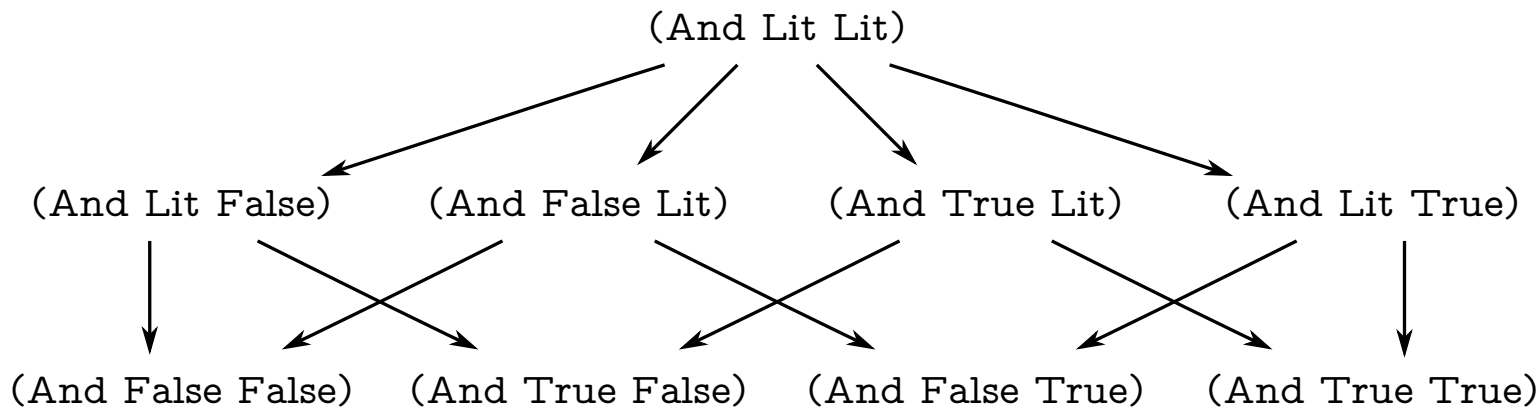


Leaf-Covering

```
class StrictEvalAlt extends FC{
        /* ...  */
  Lit combine(And a, True lft, Lit rht){...}
  Lit combine(And a, False lft, Lit rht){...}
}
```

$$covers(\texttt{StrictEvalAlt}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$$

Graph Cartesian Product

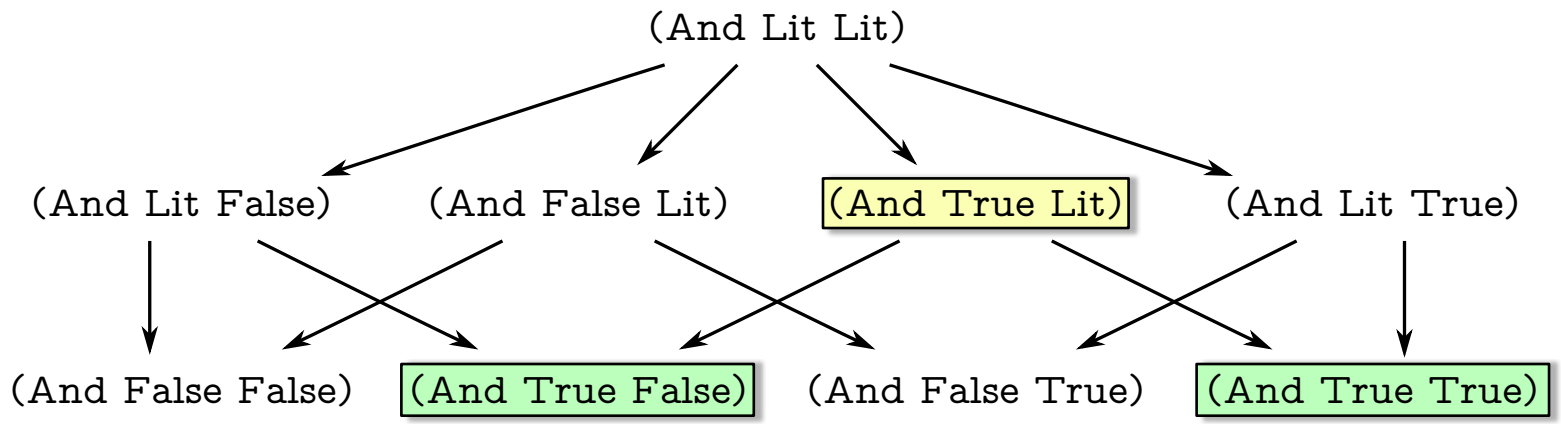


Leaf-Covering

```
class StrictEvalAlt extends FC{
        /* ...  */
  Lit combine(And a, True lft, Lit rht){...}
  Lit combine(And a, False lft, Lit rht){...}
}
```

$covers(\texttt{StrictEvalAlt}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$

Graph Cartesian Product

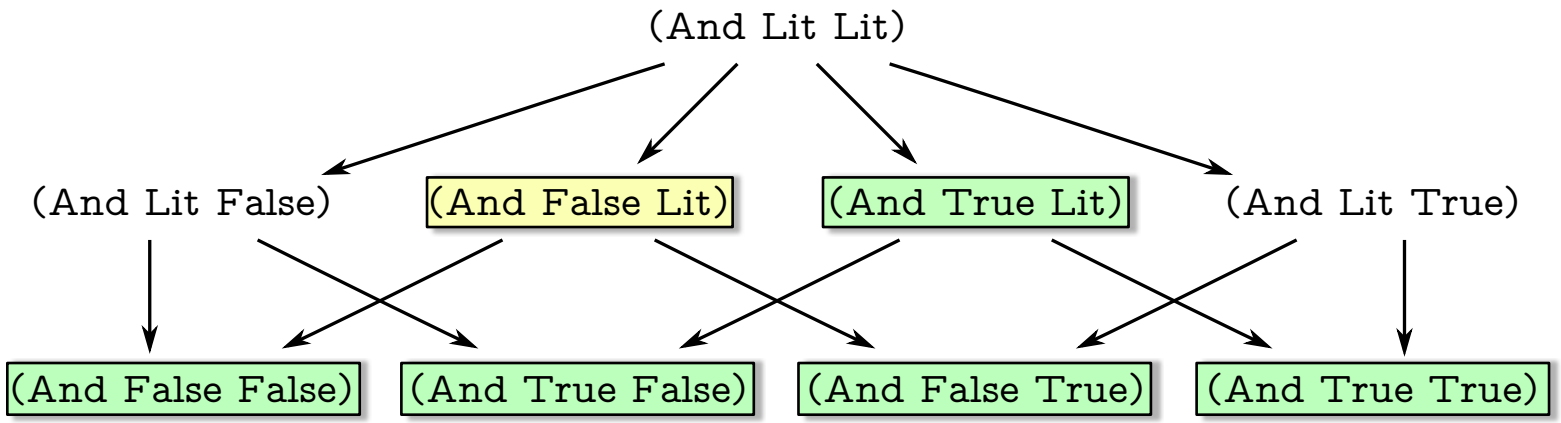


Leaf-Covering

```
class StrictEvalAlt extends FC{
        /* ...  */
  Lit combine(And a, True lft, Lit rht){...}
  Lit combine(And a, False lft, Lit rht){...}
}
```

$covers(\texttt{StrictEvalAlt}, (\texttt{And}, \texttt{Lit}, \texttt{Lit}))$

Graph Cartesian Product



LeafCovering Attributes

LeafCovering is *coNP-Complete*

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

LeafCovering 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

Type Soundness

Theorem: Well typed expressions never get "Stuck"
Or, function selection always succeeds

Lemma: Function specialization: $\forall i \in [0..n] \,.\, T'_i \leq T_i \Rightarrow$
$possibleFs(F, (T'_0 \ldots T'_n)) \subseteq possibleFs(F, (T_0 \ldots T_n))$

Corollary: Reduction preserves $covers$

Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.

* Traversals and Function-Classes/Objects
* Usefulness and Flexibility
* Safety and Types
* Efficiency/Performance

DemeterF Efficiency and Performance

Two possible areas of inefficiencies:

Traversal: walking an object/structure

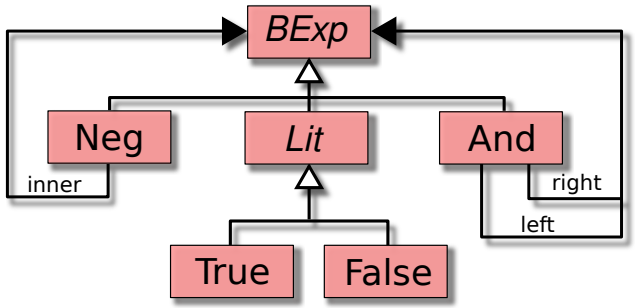
Solution: Use data structures to create traversals

Dispatch: selecting a matching *combine* method

Solution: Use type-checking results to inline dispatch

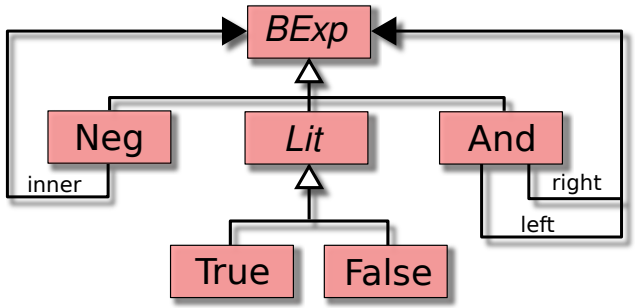
Independent subtraversals = easy parallelization

Efficiency Part 1: Traversals




```
class Traversal{
    FC func;
    Traversal(FC f){ func = f; }

    <R> R traverse(And curr){
        Object left = traverse(curr.left);
        Object right = traverse(curr.right);
        return dispatch(func, curr, left, right);
    }
    /* ...  */
    <R> R traverse(BExp curr){
        if (curr instanceof And) return traverse((And)curr);
        /* ...  */
    }
}
```

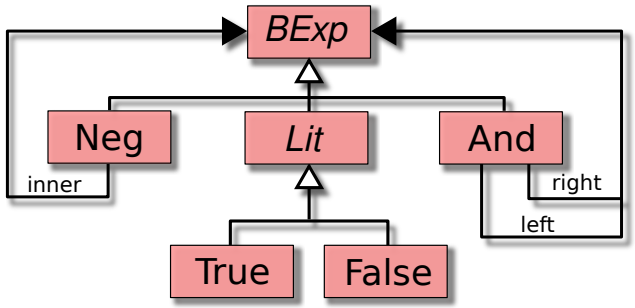
Efficiency Part 1: Traversals




```
class Traversal{
    FC func;
    Traversal(FC f){ func = f; }

    <R> R traverse(And curr){
        Object left = traverse(curr.left);
        Object right = traverse(curr.right);
        return dispatch(func, curr, left, right);
    }
    /* ...  */
    <R> R traverse(BExp curr){
        if (curr instanceof And) return traverse((And)curr);
        /* ...  */
    }
}
```

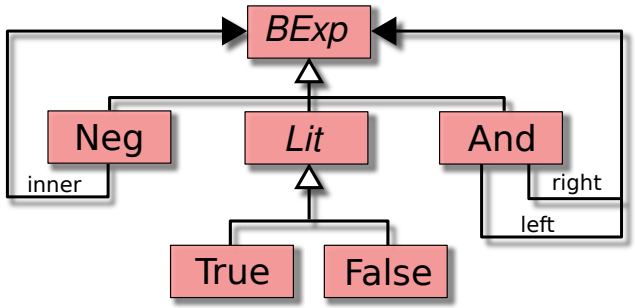
- Type info
- Method info

Efficiency Part 1: Dispatch



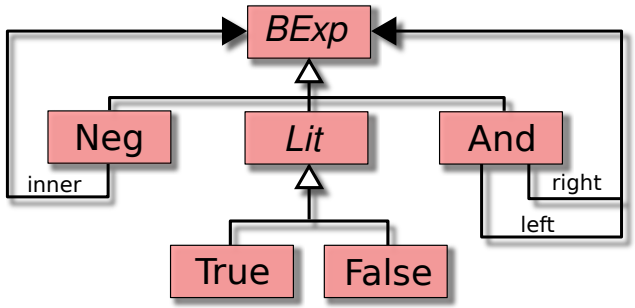

```
class Traversal{
    StrictEval func;
    Traversal(StrictEval f){ func = f; }

    Lit traverse(And curr){
        Lit left = traverse(curr.left);
        Lit right = traverse(curr.right);
        if (left instanceof True)
            if (right instanceof True)
                return func.combine(curr, (True)left, (True)right);
            else return func.combine(curr, left, right);
        else return func.combine(curr, left, right);
    }
    /* ...  */
}
```

Efficiency Part 1: Dispatch



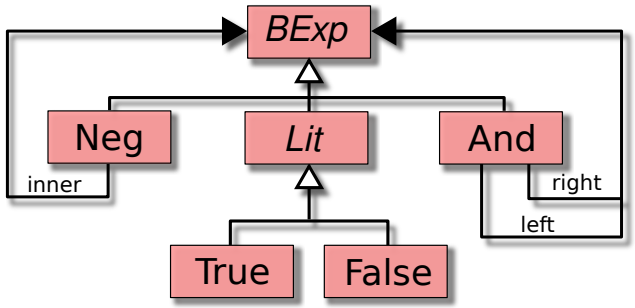

```
class Traversal{
    StrictEval func;
    Traversal(StrictEval f){ func = f; }

    Lit traverse(And curr){
        Lit left = traverse(curr.left);
        Lit right = traverse(curr.right);
        if (left instanceof True)
            if (right instanceof True)
                return func.combine(curr, (True)left, (True)right);
            else return func.combine(curr, left, right);
        else return func.combine(curr, left, right);
    }
    /* ...  */
}
```

Efficiency Part 1: Dispatch




```
class Traversal{
    StrictEvalAlt func;
    Traversal(StrictEvalAlt f){ func = f; }

    Lit traverse(And curr){
        Lit left = traverse(curr.left);
        Lit right = traverse(curr.right);
        if (left instanceof True)
            return func.combine(curr, (True)left, right);
        else return func.combine(curr, (False)left, right);
    }
    /* ...  */
}
```

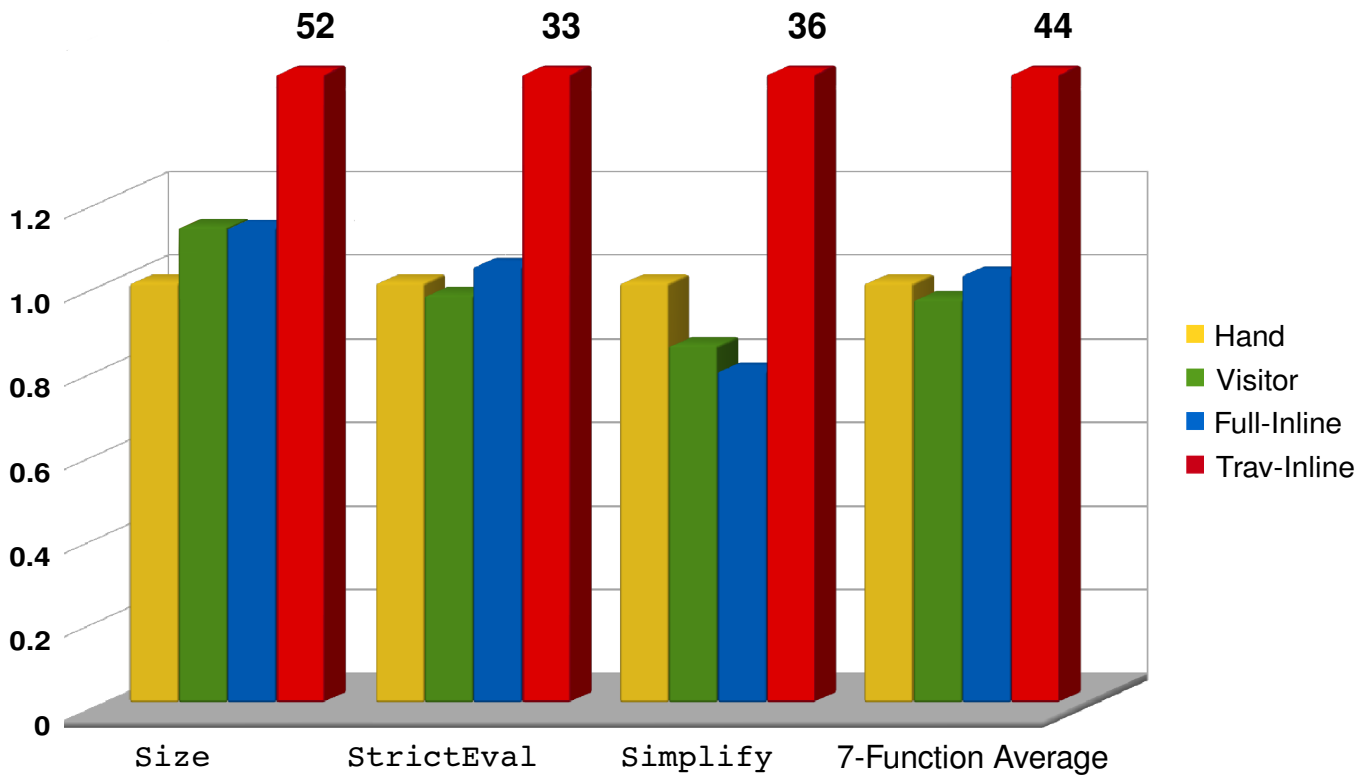
Efficiency Part 1: Dispatch




```
class Traversal{
    StrictEvalAlt func;
    Traversal(StrictEvalAlt f){ func = f; }

    Lit traverse(And curr){
        Lit left = traverse(curr.left);
        Lit right = traverse(curr.right);
        if (left instanceof True)
            return func.combine(curr, (True)left, right);
        else return func.combine(curr, (False)left, right);
    }
    /* ...  */
}
```

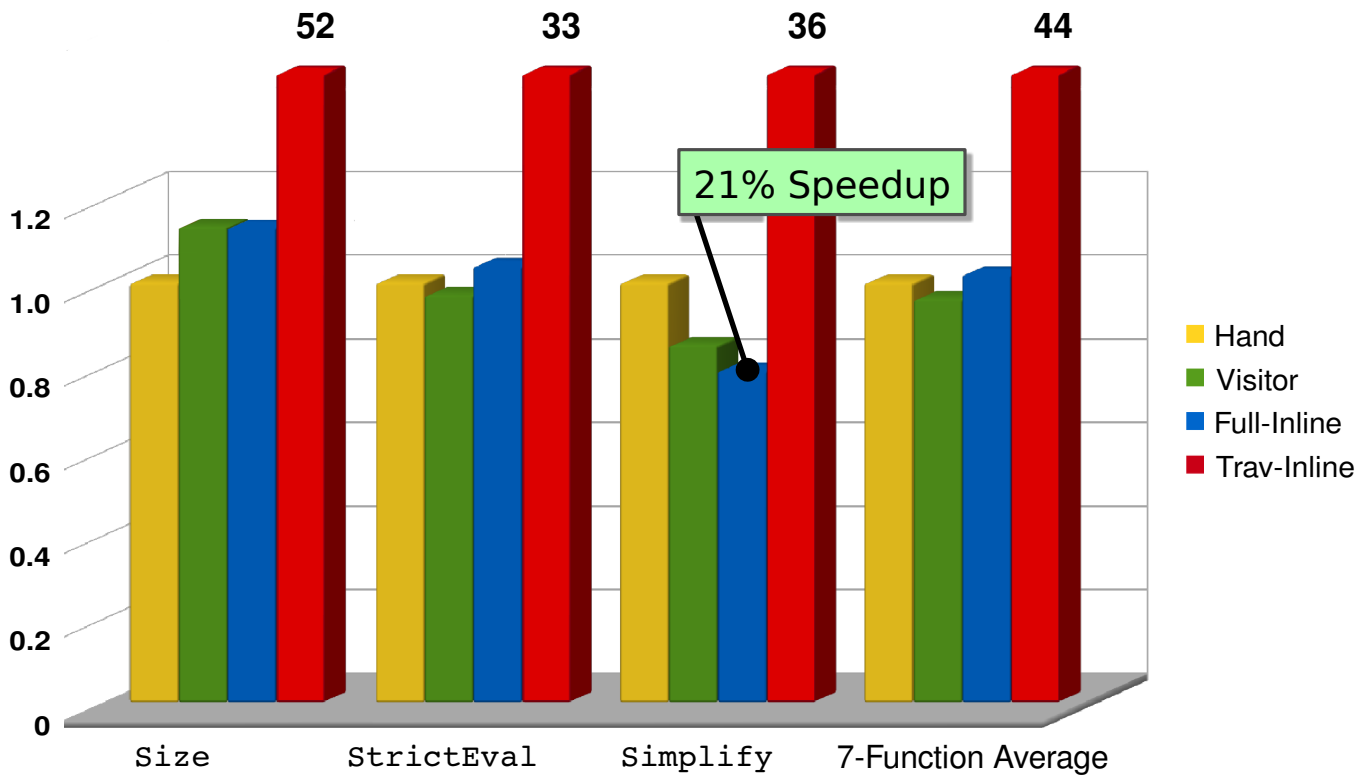
Experimental Results - 1

Hand-written vs. DemeterF



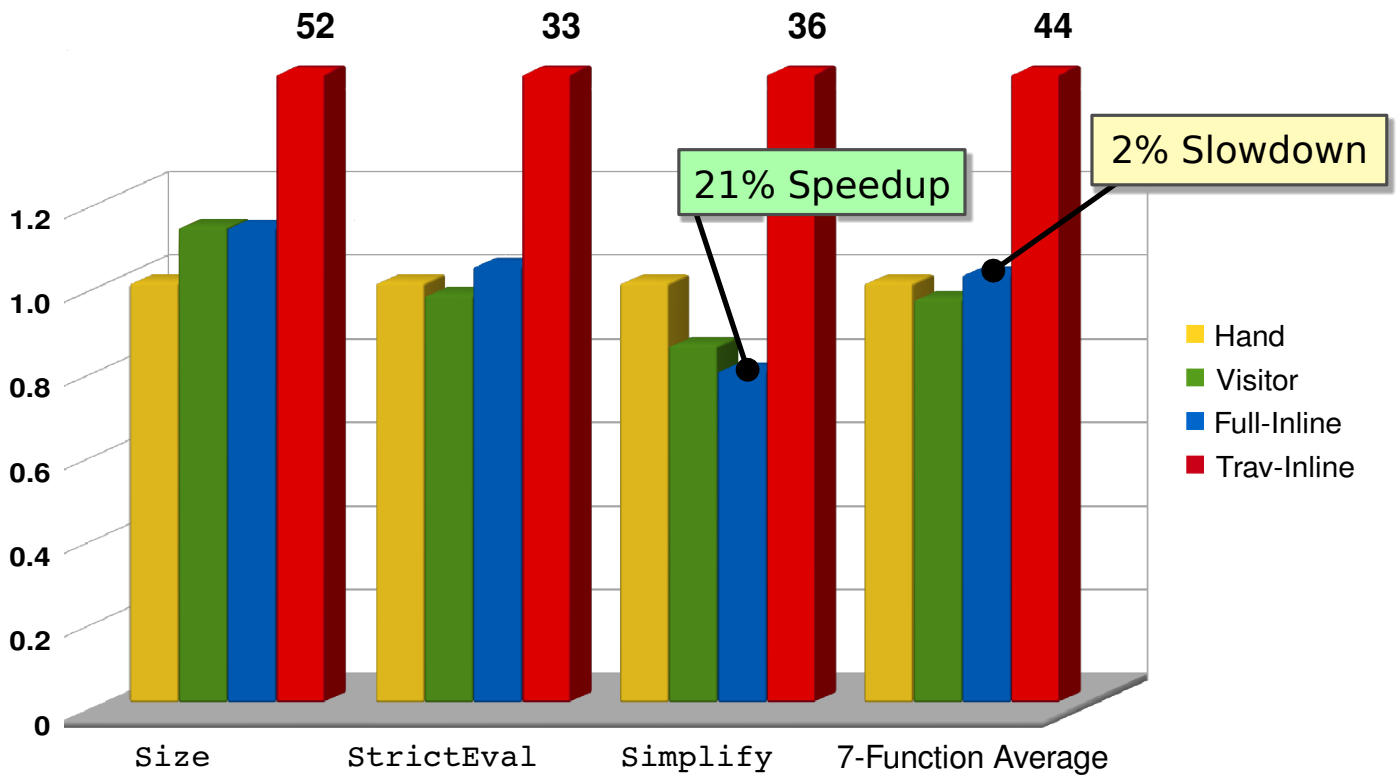
Experimental Results - 1

Hand-written vs. DemeterF



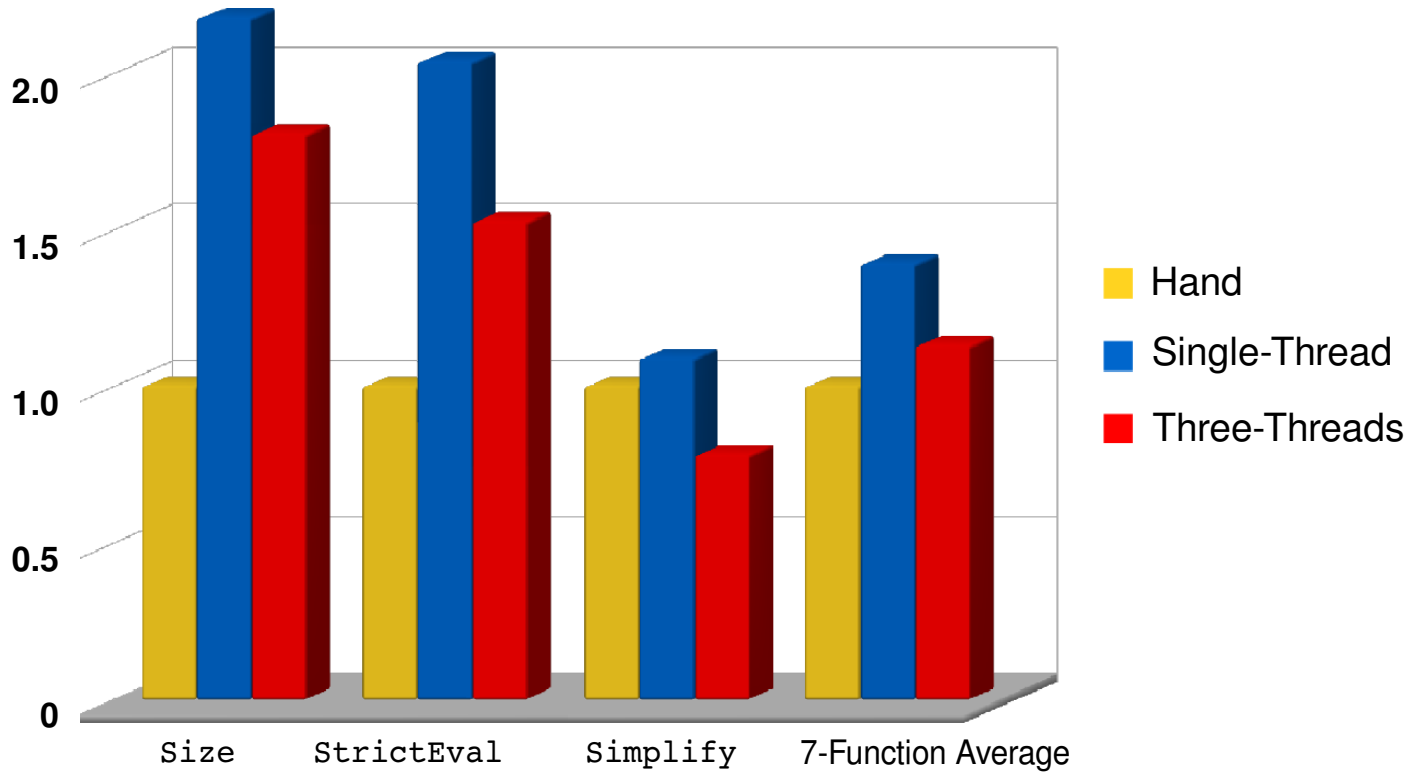
Experimental Results - 1

Hand-written vs. DemeterF



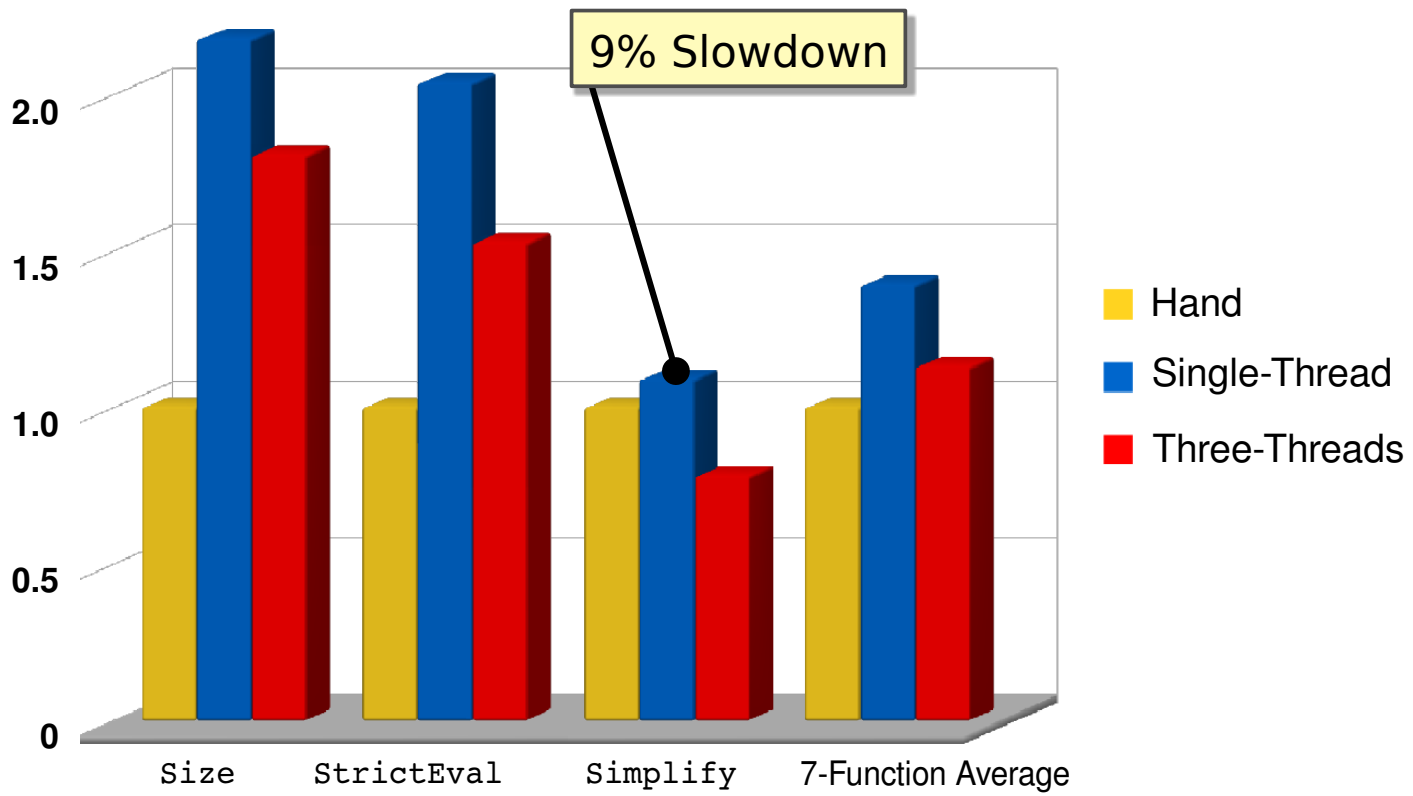
Experimental Results - 2

Hand-written vs. DemeterF Parallel



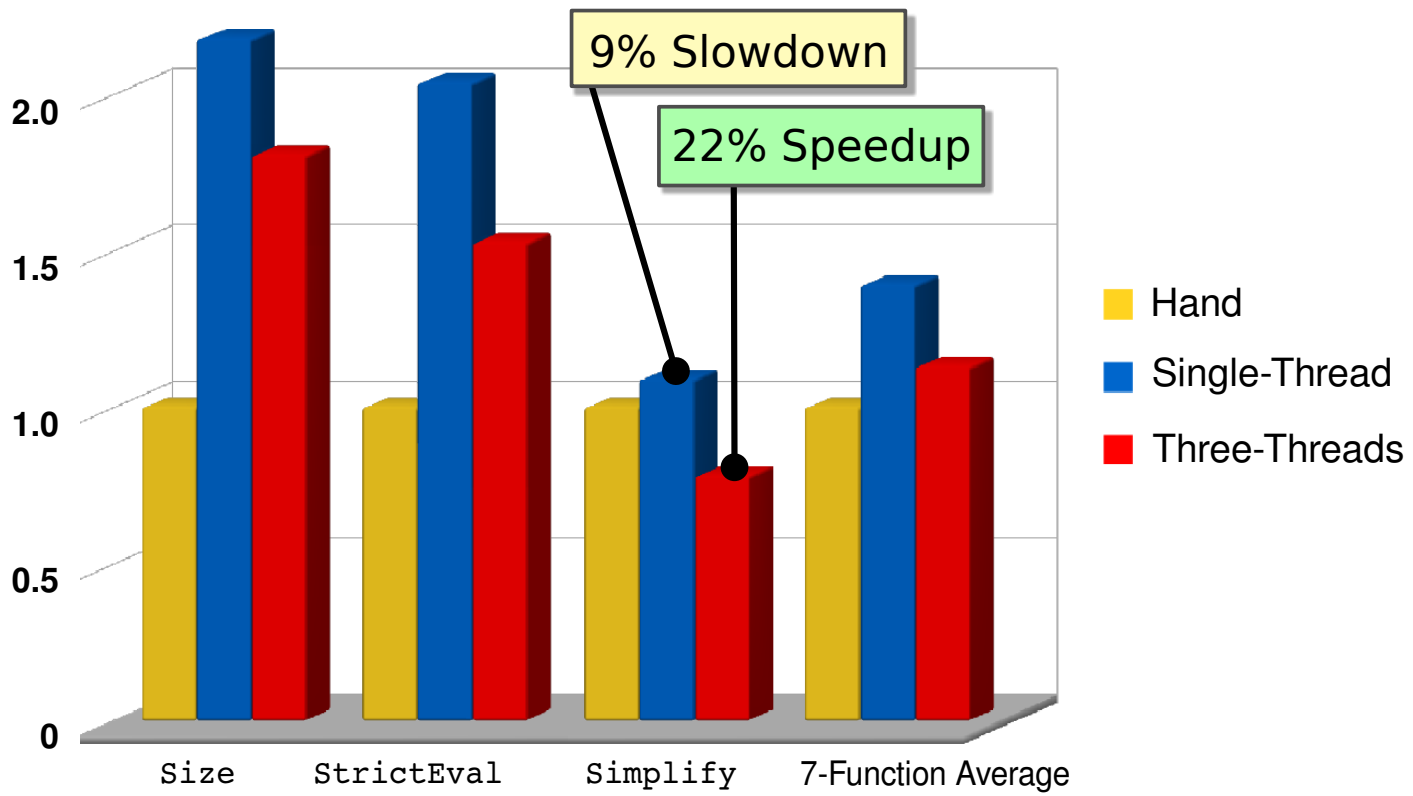
Experimental Results - 2

Hand-written vs. DemeterF Parallel



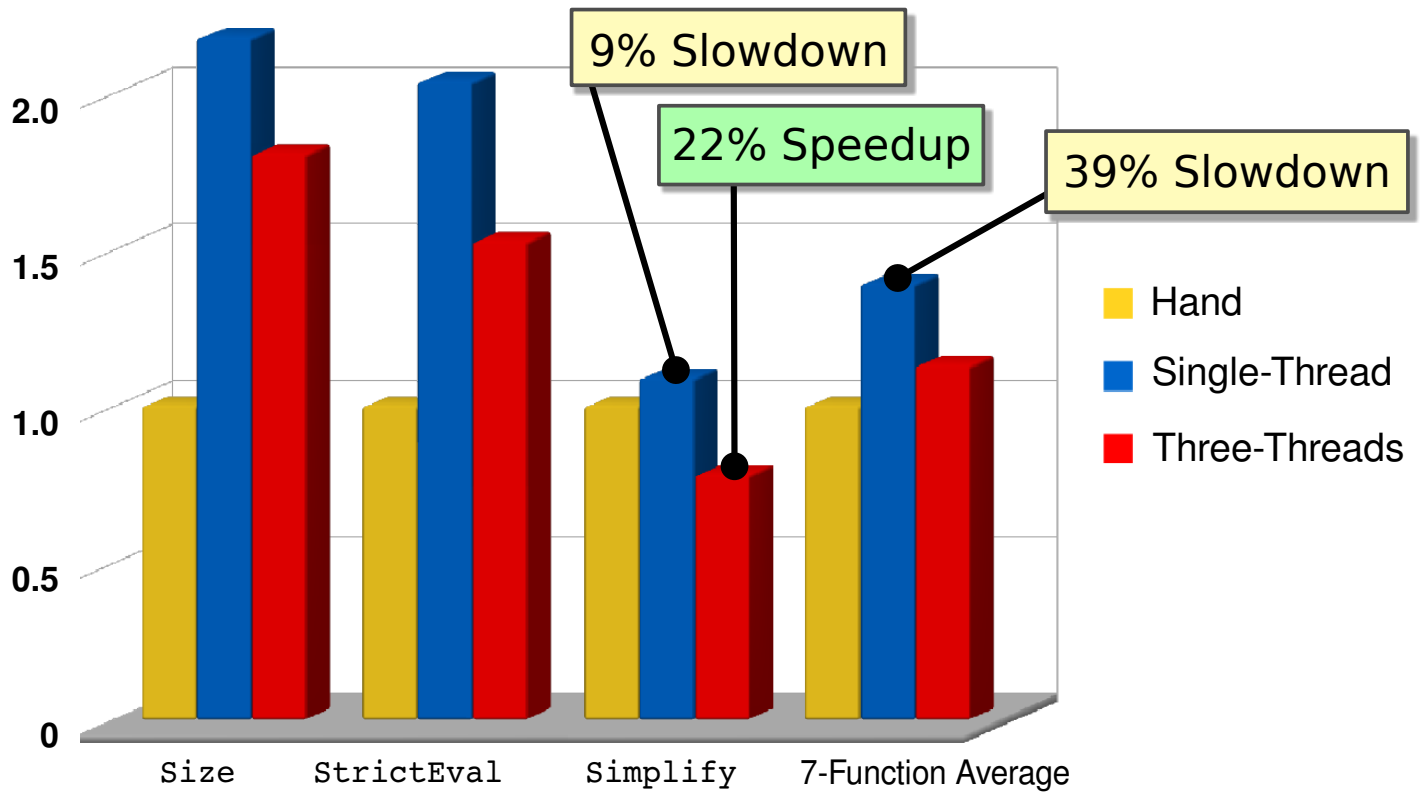
Experimental Results - 2

Hand-written vs. DemeterF Parallel



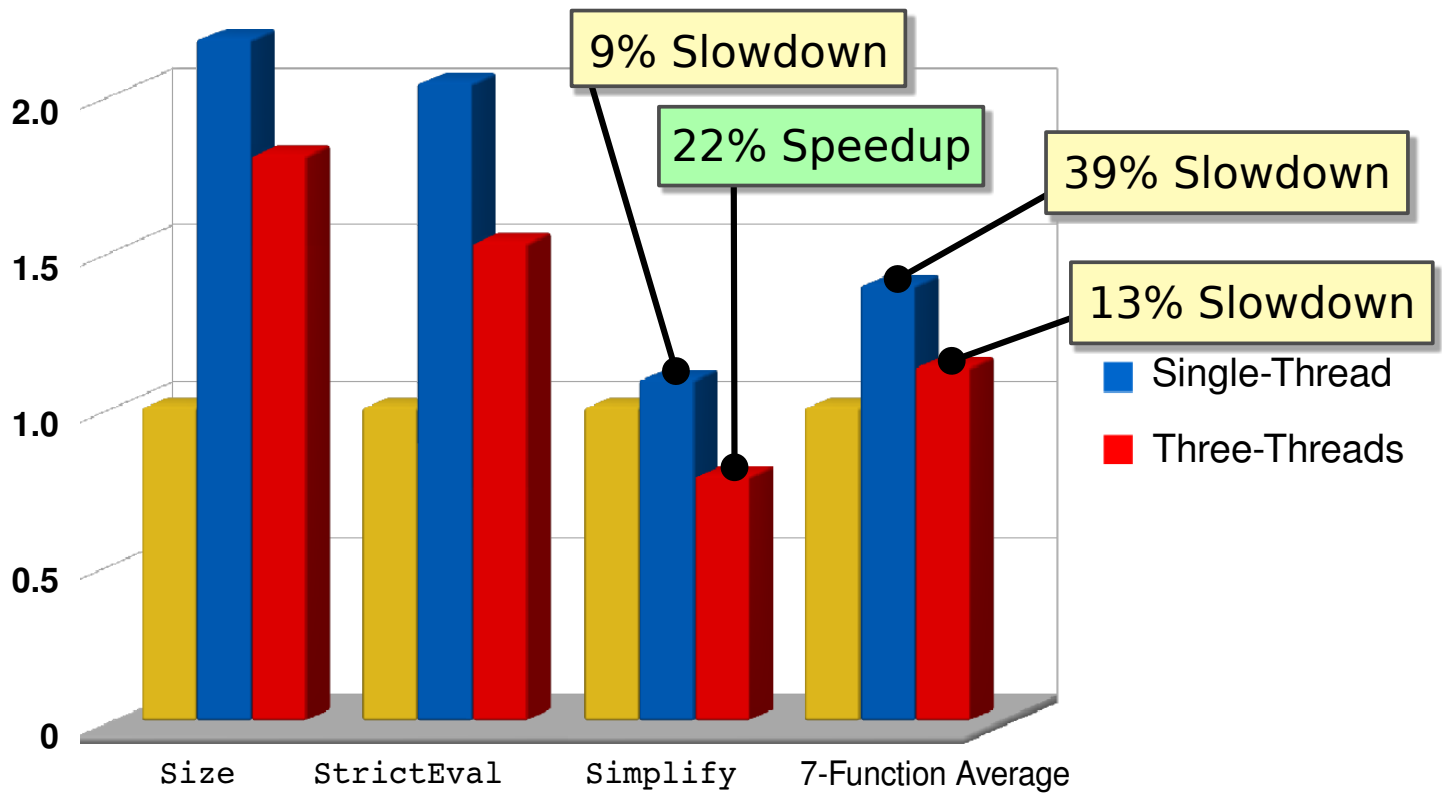
Experimental Results - 2

Hand-written vs. DemeterF Parallel



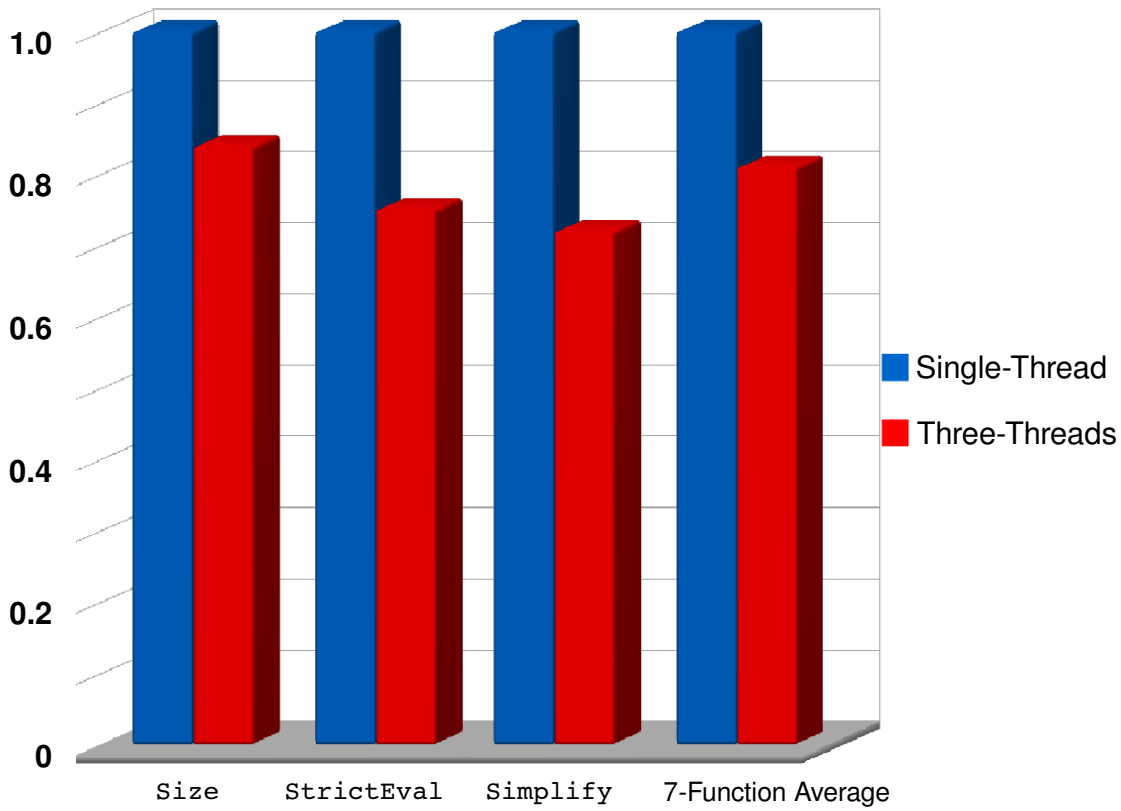
Experimental Results - 2

Hand-written vs. DemeterF Parallel



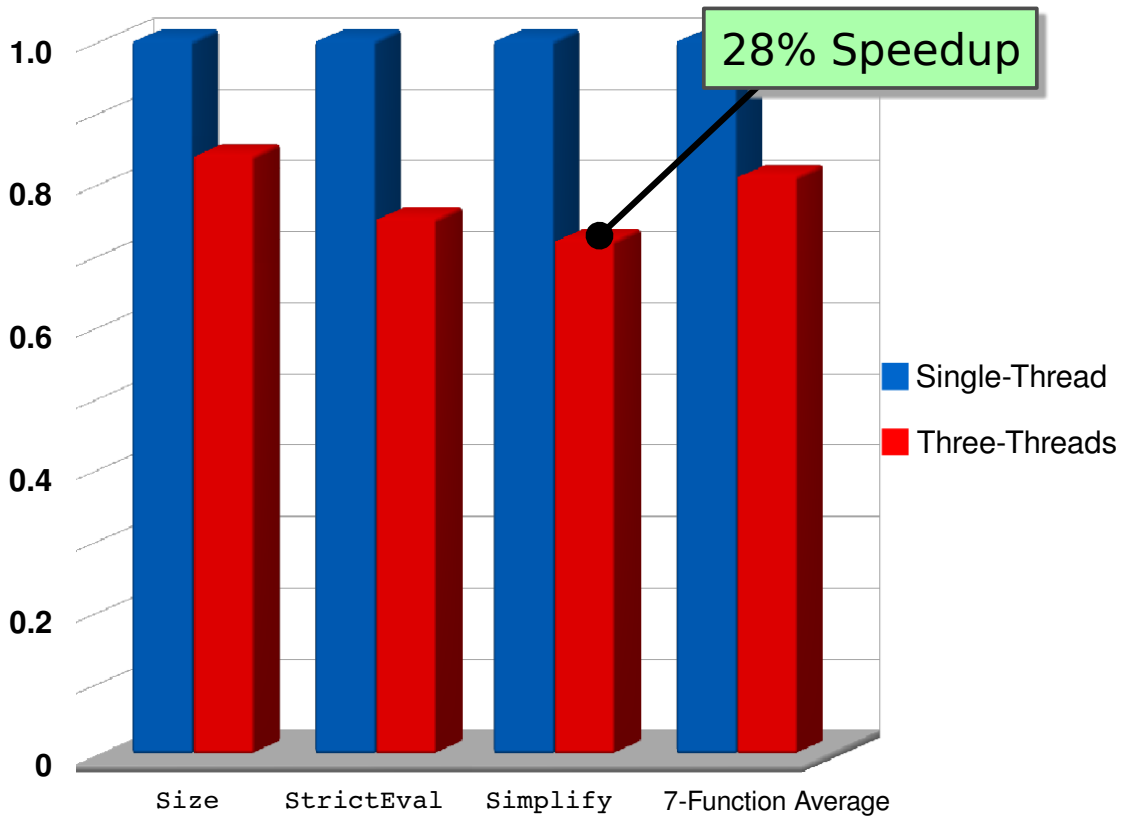
Experimental Results - 3

Single vs. Multi-Threaded



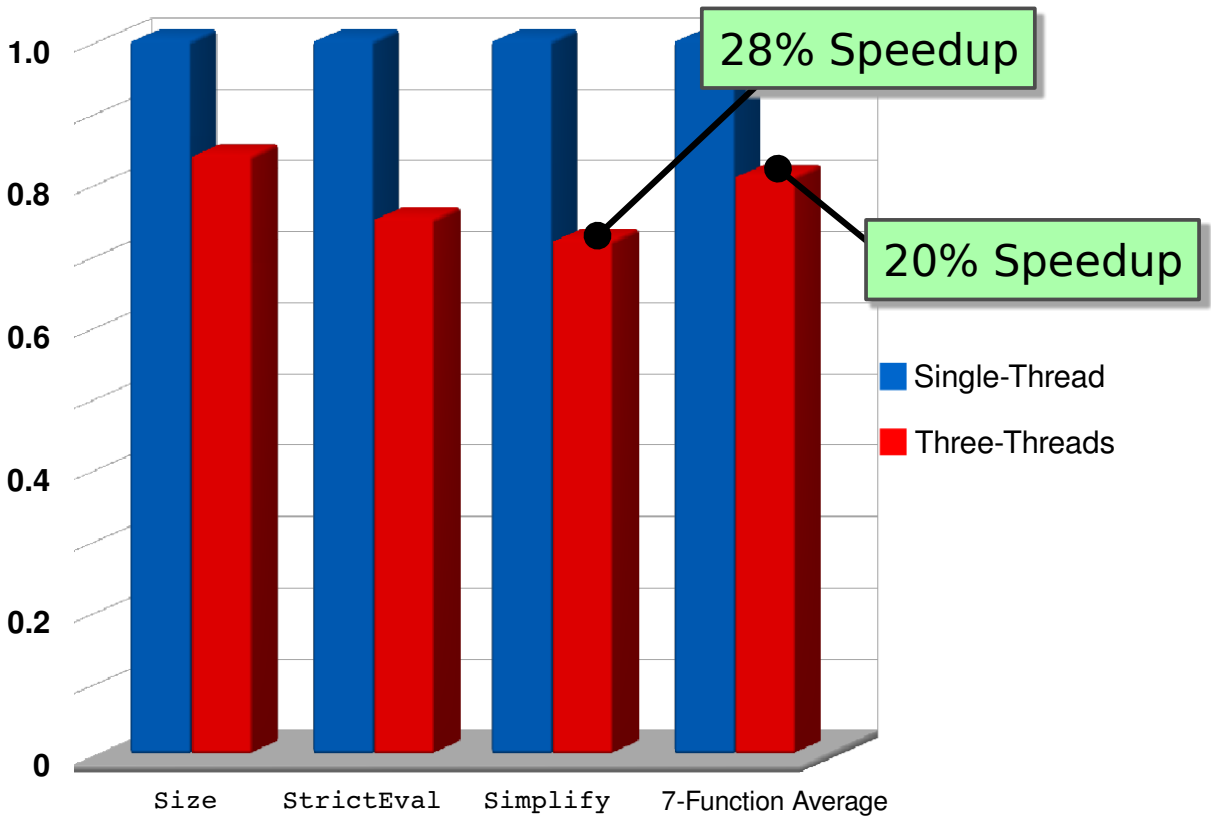
Experimental Results - 3

Single vs. Multi-Threaded



Experimental Results - 3

Single vs. Multi-Threaded



Related Work

AP and Visitors

- Lieberherr, (1996)
- Palsberg and Jay, (1998)
- Lieberherr, Patt-Shamir, and Orleans (2004)
- Oliveira, Wang, and Gibbons, (2008)

Multiple Dispatch

- Chen, Turau, and Klas, (1994)
- Chambers and Leavens, (1995)
- Millstein and Chambers, (1999)

Related Work

Generalized Folds

- Meijer, Fokkinga, and Paterson, (1991)
- Sheard and Fegaras, (1993)
- Lämmel, Visser, and Kort, (2000)

Attribute Grammars

- Knuth, (1968)
- Bochmann, (1976)
- Engelfriet and Filé, (1989)

Related Work

Generic Functional Programming

- Jansson and Jeuring, (1997)
- Hinze, (2000)
- Lämmel and Peyton Jones, (2003)

Strategic Programming

- Visser, (2001)
- Lämmel, (2003)
- Lämmel, Visser, and Visser, (2003,2004)

Future Work

Integration

- Language of function-classes/traversals
- Implementation in typed/untyped languages

Enhancements

- Include *strategies* in type system
- Improve DemeterF/Tool usability
- Increase parallel performance

Conclusion/Contributions

- New function traversal-based approach
- Generic programming ideas for OOP
- Type system for Adaptive Programming
- Tools and algorithms for safe, efficient, functional AP

The End

Thank You