# Weaving Generic Programming and Traversal Performance

Bryan Chadwick and Karl Lieberherr

Northeastern University

AOSD '10 March $17^{th}$ 2010

# The Problem

We write programs with...

- Rich, mutually recursive datatypes
- Possibly shifting/changing structures

What we want to accomplish? Make it **easier** to...

- Write complex functions over structures
- Safely reuse for different structures

Main Goals: Flexibility, reuse, and performance

# The Problem: Concretely

## Complex structures: AST

```
Exp  = If | Bin | Num | /* ... */.

If   = <cnd> Exp  <thn> Exp  <els> Exp.
Bin  = <left> Exp  <op> Oper  <right> Exp.
Num  = <val> int.
  /* ... */
```

## Complex function: `Simplify`

- Walk an instance and replace statically computable expressions with constants

$$``(5 + 7)" \rightarrow ``12"$$

# Our Solution: A New Approach, TBGP

## Traversal-based generic programming

- Separate traversal
- Modularize interesting code (Function-classes)
- Put together using *asymmetric multiple-dispatch*
- Function extension = inheritance

## Our Contributions

- Implementation: `DemeterF`
- Powerful, generic base function-classes
- Safety and weaving → performance

Gives us: Flexibility, reuse, performance

# Related Work

| | |
|---|---|
| Visitors | Palsberg and Jay [1998], VanDrunen and Palsberg [2004], Krishnamurthi et al. [1998], Oliveira [2009] |
| Multi-Dispatch | Clifton et al. [2000], Chambers [1992], Chen and Turau [1995] |
| Gen. Prog. | Gibbons [2007], Meijer et al. [1991], Sheard and Fegaras [1993], Jansson and Jeuring [1997], Lämmel and Peyton Jones [2003] |
| AP/Generation | Lieberherr et al. [2004], Orleans [2002], Orleans and Lieberherr [2001], JavaCC [2010], ANTLR [2010] |
| Others | Model-Driven Development (OMG), Event-based/Implicit Invocation (Sullivan and Notkin [1992], Rajan and Leavens [2008]), |

# Outline

1. Traversal-based generic programming
   Introduction
   Details
2. Generic base function-classes
   Building useful functions
3. Weaving traversals and functions
   Traversal generation and inlining
4. Performance results

# What is Traversal-based generic programming?

## **Our** view of AOP

- Base program execution generates events (*join points*)
    - Events are triggered by method call/return
    - Aspects attach *advice* to these events
- Pointcuts select sets of events and bind *context*
- Advice computes with context and *state*

# What is Traversal-based generic programming?

## AOP view of TBGP

- Base program is depth-first traversal
  - Events are triggered by traversal completion
  - Our aspects are function-objects (with `combine` methods)
- Method signatures select events and bind context
- Method bodies compute with context (recursive results)

## Advice chosen based on the dynamic type of recursive results

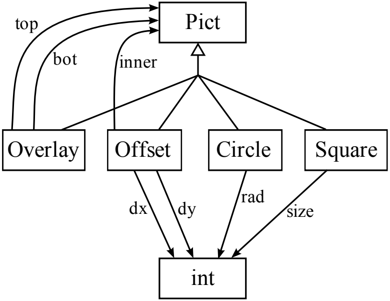
# TBGP Example: Pictures

Pict = Overlay | Offset
| Circle | Square.

Overlay = <top> Pict <bot> Pict.

Offset = <dx> int <dy> int
<inner> Pict.

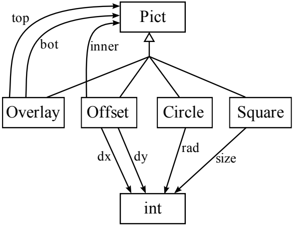
Circle = <rad> int.
Square = <size> int.



# TBGP Example: Pictures (ToString)

```
class ToString extends ID{
    String combine(Circle c, int rad)
    { return "Circle("+rad+")"; }
    String combine(Overlay o, String top, String bot)
    { return "Overlay("+top+","+bot+")"; }
    /* ... */

    String toString(Pict p)
    { return new Traversal(this).<String>traverse(p); }
}
```

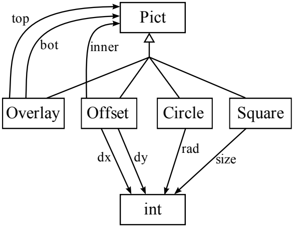


- * combine methods are like pointcuts and advice
- * Adaptive depth-first traversal

# TBGP Example: Pictures (ToString)

```
class ToString extends ID{
  String combine(Circle c, int rad)
  { return "Circle("+rad+")"; }
  String combine(Overlay o, String top, String bot)
  { return "Overlay("+top+","+bot+")"; }
  /* ... */

  String toString(Pict p)
  { return new Traversal(this).<String>traverse(p); }
}
```



* combine methods are like pointcuts and advice

* Adaptive depth-first traversal

# TBGP Example: Execution

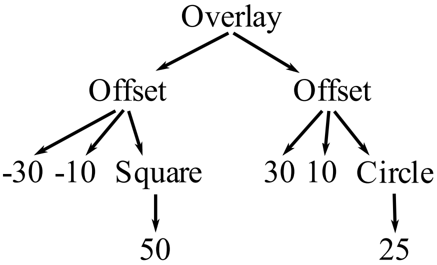
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

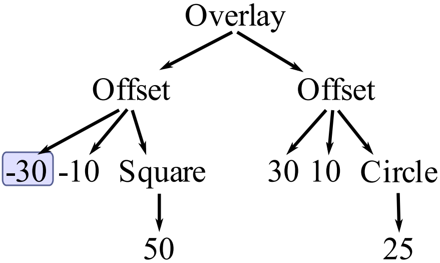
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

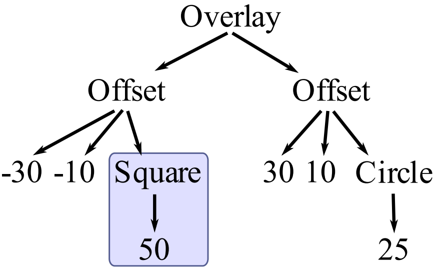
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return "Overlay("+top+","+bot+")"; }

  String combine(Offset o, int dx, int dy, String inner)
  { return "Offset("+dx+","+dy+","+inner+")"; }

  String combine(Circle c, int rad)
  { return "Circle("+rad+")"; }

  String combine(Square s, int size)
  { return "Square("+size+")"; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

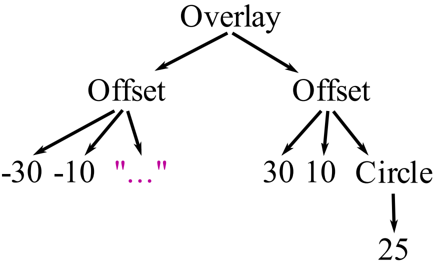
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return "Overlay("+top+","+bot+")"; }

  String combine(Offset o, int dx, int dy, String inner)
  { return "Offset("+dx+","+dy+","+inner+")"; }

  String combine(Circle c, int rad)
  { return "Circle("+rad+")"; }

  String combine(Square s, int size)
  { return "Square("+size+")"; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

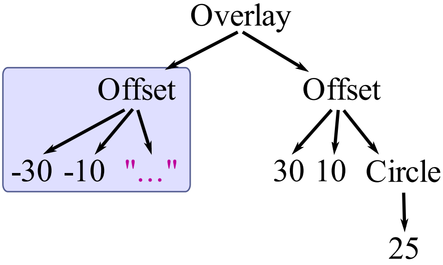
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return "Overlay("+top+","+bot+")"; }

  String combine(Offset o, int dx, int dy, String inner)
  { return "Offset("+dx+","+dy+","+inner+")"; }

  String combine(Circle c, int rad)
  { return "Circle("+rad+")"; }

  String combine(Square s, int size)
  { return "Square("+size+")"; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

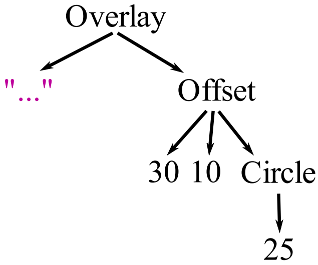
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

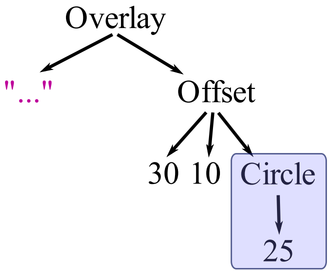
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

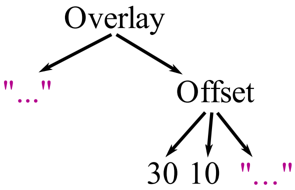
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

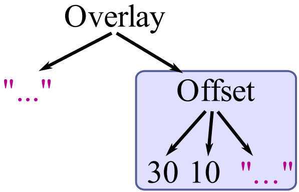
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return "Overlay("+top+","+bot+")"; }

  String combine(Offset o, int dx, int dy, String inner)
  { return "Offset("+dx+","+dy+","+inner+")"; }

  String combine(Circle c, int rad)
  { return "Circle("+rad+")"; }

  String combine(Square s, int size)
  { return "Square("+size+")"; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

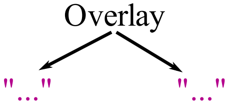
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

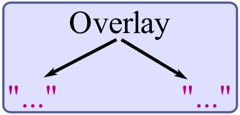
```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```



# TBGP Example: Execution

```
class ToString extends ID{
  String combine(Overlay o, String top, String bot)
  { return ”Overlay(”+top+”,”+bot+”)”; }

  String combine(Offset o, int dx, int dy, String inner)
  { return ”Offset(”+dx+”,”+dy+”,”+inner+”)”; }

  String combine(Circle c, int rad)
  { return ”Circle(”+rad+”)”; }

  String combine(Square s, int size)
  { return ”Square(”+size+”)”; }
}
```

"Overlay(...)"

```
/* Provided by DemeterF */
class ID{
  int combine(int i){ return i; }
  /* ... */
}
```

# TBGP: Key points

## What did we do?

- Separate, functional traversal
  - Structural recursion factored out
  - Follows from our data structures
  - Supports different traversal implementations
- Modularized interesting functionality
  - Limit scattering
- Implicit dispatch selects advice
  - Recursive return values determine choice

# TBGP: Implicit Dispatch

Only one advice...

- The "most specific" signature
- Based on runtime types:

  (host, ... *recursive results* ...)
- The *host* is our leftmost argument

  - So we give left-to-right precedence

Termed: Asymmetric multiple-dispatch

- No runtime ambiguities

What do we gain?

# TBGP: Implicit Dispatch

## Gives us Abstraction

```
String combine(Pict p, int i)
{ /*.. Applies to multiple cases ..*/ }
```

## And Overloading/Overriding

```
Number combine(Pict p, Number lft, Number rht)
{ /*.. Applies to more general cases ..*/ }

Integer combine(Overlay o, Integer lft, Double rht)
{ /*.. Applies to specific case ..*/ }
```

## How do we know methods work together?

# TBGP: Dispatch Safety

## What can go wrong?

- No applicable advice, means no recursive result (!)

## How can we ensure safety?

- Compute the return types over the structure
- Make sure we have at least one applicable method
- Argument signatures must cover all cases (be *complete*)

## Determine (statically) which methods *may* be called

- Calculate runtime dispatch residue

# Outline

1. Traversal-based generic programming
   - Introduction
   - Details
2. Generic base function-classes
   - Building useful functions
3. Weaving traversals and functions
   - Traversal generation and inlining
4. Performance results

# TBGP: Generic Programming

Two useful generic cases (Lämmel [2003])

Type Preserving, TP (Rebuild/copy)

$\text{traverse}_{\text{TP}} : \forall\, T\, .\, T \rightarrow T$

Type Unifying, TU (Deep Fold)

$\text{traverse}_{\text{TU}}{<}\alpha{>} : \forall\, T\, .\, T \rightarrow \alpha$

Also called *transformations* and *queries*

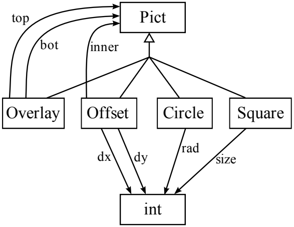
Scrap Your Boilerplate (Lämmel and Peyton Jones [2003])

# TBGP: Generic Programming (TP)

## Functional updates for Picts

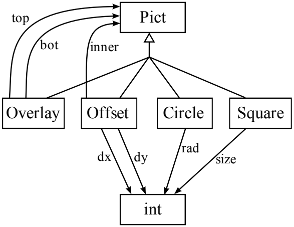
```
// Triple the size of all Picts
class Triple extends TP{
    int combine(int i){ return i*3; }
}
```

```
// Flip top/bot ordering
class Flip extends TP{
    Overlay combine(Overlay o, Pict top, Pict bot)
    { return new Overlay(bot, top); }
}
```



- Special cases for int and Overlay
- TP *rebuilds* other cases

# TBGP: Generic Programming (TU)



## Deep Fold for Picts

```
// Collect the Circles
class CollectCircs extends TU<List<Circle>>{
   List<Circle> combine(){ return List.create(); }
   List<Circle> fold(List<Circle> a, List<Circle> b){ return a.append(b); }
   List<Circle> combine(Circle c){ return List.create(c); }
}
```

- Default `combine()` returns the empty-list (leafs)
- `fold` merges two results
- Special case for Circle
- TU calls `fold` for composite cases

# TBGP: Generic Programming

## Benefits

- Overriding/overloading is easy
- Exploit commonalities, write our own base classes
- Function-classes are *near-sighted*

## TP/TU in particular

- Functions adapt by way of TP/TU
- TP/TU are structure-based
    - We can generate concrete versions

# Outline

1. Traversal-based generic programming
   - Introduction
   - Details
2. Generic base function-classes
   - Building useful functions
3. Weaving traversals and functions
   - Traversal generation and inlining
4. Performance results

# Weaving traversals and functions

## Traversal is structure-based

- Produce an implementation of structural recursion
- Inline method selection residue (if any)

## How do we implement traversal?

1. Abstract: choose between direct subclasses
2. Concrete: traverse each field, select/apply a `combine` method
3. Use types, methods, and residue from type checking

# Weaving traversals and functions

## What does it look like for `ToString`?

```
class InlineToString{
    ToString func;
        /* ... */
}
```

Pict = Overlay | ... .

```
String traversePict(Pict h){
    if(h instanceof Overlay) return traverseOverlay((Overlay)h);
        /* ... */
    throw new RuntimeException(”Unknown Pict”);
}
```

Overlay = <top> Pict <bot> Pict.

```
String traverseOverlay(Overlay h){
    String top = traversePict(h.top);
    String bot = traversePict(h.bot);
    return func.combine(h, top, bot);
}
```

# Weaving traversals and functions

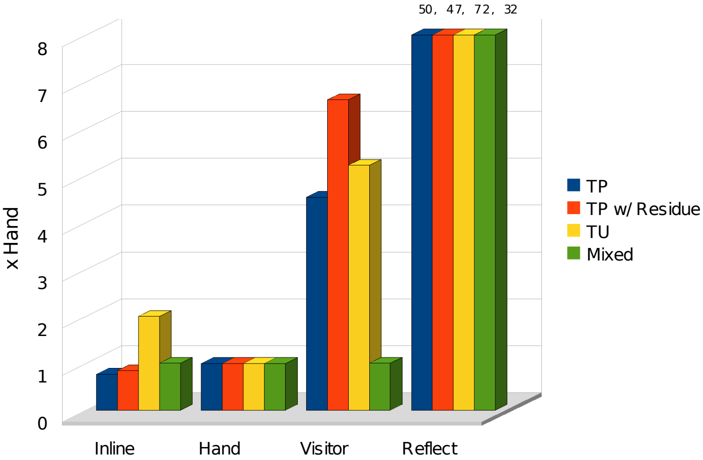
## Separate Traversals/Functions means:

- Less redundant information
- New traversal implementations only require regeneration
- Good for parallelism and/or eliminating stack use

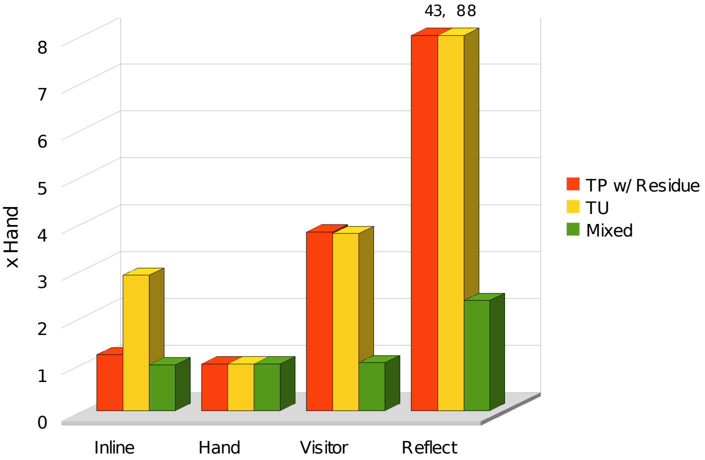
## Inlining benefits

- Automatic generation (structure + function → code)
- Direct replacement for reflective traversal
- Performs *much* better

# Performance: Picts



# Performance: Exp. Compiler



# Conclusions

## Traversal-Based Generic Prog. (TBGP)

- Separate/abstract structural recursion
- Function-classes modularize interesting code
- Combine the two with implicit, asymmetric multiple-dispatch
- Reuse/safety is check-able

## Extensible base function-classes

- TP/TU for starters

## Weave together traversal and functions for performance

# Conclusions

## TBGP: Important Points

- Enable powerful, extensible, generic functions
- Flexibility of reflection, with the safety and performance of hand-written, structural recursion

# Conclusions

## TBGP: Important Points

- Enable powerful, extensible, generic functions
- Flexibility of reflection, with the safety and performance of hand-written, structural recursion

## Thank You

| | |
|---|---|
| Bryan Chadwick: | chadwick@ccs.neu.edu |
| Karl Lieberherr: | lieber@ccs.neu.edu |
| DemeterF Home: | http://www.ccs.neu.edu/~chadwick/demeterf/ |

# TBGP: Type Preserving Details

```
// Specific TP for Picts
class TP_Pict{
  Overlay combine(Overlay o, Pict t, Pict b)
  { return new Overlay(t,b); }

  Offset combine(Offset o, int dx, int dy, Pict in)
  { return new Offset(dx,dy,in); }

  Circle combine(Circle c, int r){ return new Circle(r); }
  Square combine(Square s, int sz){ return new Square(sz); }

  int combine(int i){ return i; }
}
```

# TBGP: Type Unifying Details

```
// Specific TU for Picts
class TU_Pict<X>{
  abstract X combine();       //** Default result
  abstract X fold(X a, X b); //** Merge two results

  X combine(Offset o, X dx, X dy, X inner)
  { return fold(dx, fold(dy, inner)); }
  X combine(Overlay o, X top, X bot){ return fold(top, bot); }
  X combine(Circle c, X rad){ return rad; }
  X combine(Square s, X size){ return size; }

  X combine(int i){ return combine(); }
}
```

# Function-classes from a CD

## Only need Concrete classes

C = <$f_1$> $D_1$ ... <$f_n$> $D_n$.

```
// TP methods
C combine(C c, $D_1$ $f_1$, ..., $D_n$ $f_n$){
   return new C($f_1$, ..., $f_n$);
}

// TU methods
X combine(C c, X $f_1$, ..., X $f_n$){
   return fold($f_1$, fold(..., $f_n$));
}
```

# References (1)

ANTLR. ANother Tool for Language Recognition. Website, 2010. `http://www.antlr.org/`.

Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92*, pages 33–56. Springer-Verlag, 1992.

Weimin Chen and Volker Turau. Multiple-dispatching based on automata. *Theor. Pract. Object Syst.*, 1(1):41–59, 1995.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00*, pages 130–145, 2000.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97*, pages 470–482. ACM Press, 1997.

JavaCC. The Java Compiler Compiler™. Website, 2010. `https://javacc.dev.java.net/`.

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113, London, UK, 1998. Springer Verlag.

# References (2)

Ralf Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. volume 38, pages 26–37. ACM Press, March 2003. TLDI '03.

Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA '91*, volume 523, pages 124–144. Springer Verlag, Berlin, 1991.

Bruno C. Oliveira. Modular visitor components. In *ECOOP '09*, pages 269–293. Springer-Verlag, 2009.

Doug Orleans. Incremental programming with extensible decisions. In *AOSD '02*, pages 56–64, New York, NY, USA, 2002. ACM.

Doug Orleans and Karl J. Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001*, Kyoto, Japan, September 2001. Springer Verlag.

Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98*, Washington, DC, USA, 1998.

Hridesh Rajan and Gary T. Leavens Ptolemy: A Language with Quantified, Typed Events. In *ECOOP '08*, pages 155–179. Springer-Verlag, 2008.

# References (3)

Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *FPCA '93*, pages 233–242. ACM Press, New York, 1993.

Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, 1992.

Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. *FOOL '04*, January 2004.