A Type System for Functional Traversal-Based Aspects

Bryan Chadwick and Karl Lieberherr

Northeastern University

March 2^{nd} 2009

Outline

Intro: Traversals

AOP Modularizes Crosscutting Concerns

Traversal is an Important Concern

- Walk a Data Structure... Do Some Work
- Tedious to Write
- Crosscuts Data Definitions

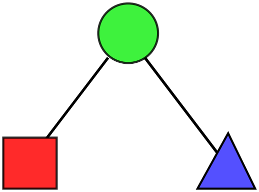
Intro: Functional Traversal

Functional Traversal

- Compute Without Mutation
- Multi-threading
- Safe But Flexible
- Eliminate Implicit Ordering

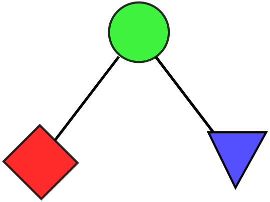
Intro: Functional Traversal

Tree Contraction



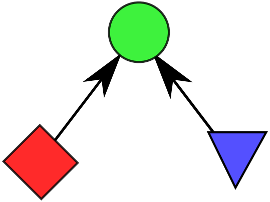
Intro: Functional Traversal

Tree Contraction



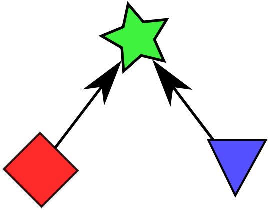
Intro: Functional Traversal

Tree Contraction



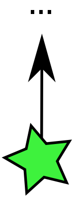
Intro: Functional Traversal

Tree Contraction



Intro: Functional Traversal

Tree Contraction



Goals: Modularity and Safety

Separate Traversals and Computation

- Traversal Flexibility/Control
- Freedom of Implementation
- Write Once or Generate from Structures

Enforce Safety With Types

- Express More of the Computation
- Assumptions Are Checked

Functional Traversal-Based Aspects

Sets of Functions Over a Depth-First Traversal

- Each Function Is **Advice**
- **Join Points** Are *After* Subtraversals Complete
- **Pointcuts** Are Function Signatures

Benefits

- Functional !
- Type Sound

Surface Syntax

$$
\begin{array}{rcl}
x & ::= & \text{variable names} \\
C & ::= & \text{concrete type names} \\
A & ::= & \text{abstract type names} \\
T & ::= & C \mid A \\
\\
P & ::= & D_1 \ \ldots \ D_n \ \ e \\
D & ::= & \mathtt{concrete}\ C\ \mathtt{(}\,T_1, \ \ldots, \ T_n\,\mathtt{)} \\
 & \mid & \mathtt{abstract}\ A\ \mathtt{(}\,T_1, \ \ldots, \ T_n\,\mathtt{)} \\
\\
e & ::= & x \mid \mathtt{new}\ C\,\mathtt{(}\,e_1, \ \ldots, \ e_n\,\mathtt{)} \mid \mathtt{traverse(}\,e_0, \ F\,\mathtt{)} \\
F & ::= & \mathtt{funcset(}f_1 \ \ldots \ f_n\mathtt{)} \\
f & ::= & (T_0\ x_0, \ \ldots, \ T_n\ x_n)\{\ \mathtt{return}\ \ e;\ \}
\end{array}
$$

Example: Boolean Expressions

Data Definitions

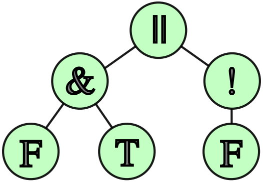
```
abstract Exp (Lit, Neg, And, Or)

abstract Lit   (True, False)
concrete True  ()
concrete False ()

concrete Neg (Exp)
concrete And (Exp, Exp)
concrete Or  (Exp, Exp)
```

Example: Boolean Expressions

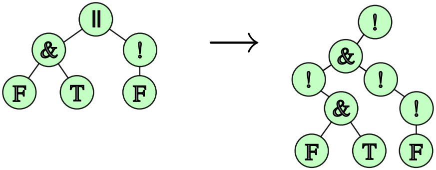
An Expression




```
new Or(new And(new False(),
                new True()),
       new Neg(new False()))
```

Example: Boolean Expressions

Or Elimination

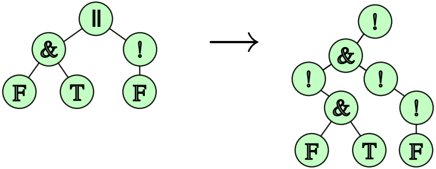



```
funcset(
  (True t){ return t; }
  (False f){ return f; }
  (Neg n, Exp e) { return new Neg(e); }
  (And a, Exp l, Exp r){ return new And(l,r); }

  (Or o, Exp l, Exp r){
    return new Neg(new And(new Neg(l),
                           new Neg(r)));
  }
)
```

Example: Boolean Expressions

Or Elimination



Other Functions Just Reconstruct

```
(Or o, Exp l, Exp r){
  return new Neg(new And(new Neg(l),
                         new Neg(r)));
}
```

Semantics

Traversals and Dispatch

- Adaptive Depth-First Traversal
- Dispatch on First Argument Type
- Enforce Safety With Types

Functions

- Depend on Argument Order
- Can implement “Field” Access

Semantics: Expanded

Expanded Dispatch Semantics

- Asymmetric Multiple Dispatch
- Pattern Matching with Safety

Similar to CLOS Generic Functions

Example: Refactored

Abstract Binary Exp

```
abstract Exp (Lit, Neg, Bin)

                ...

concrete Bin (Op, Exp, Exp)

abstract Op  (And, Or)
concrete And ()
concrete Or  ()
```

Example: Refactored Evaluation

```
funcset(
  (Lit l){ return l; }
  (Neg n, True t) { return new False(); }
  (Neg n, False t){ return new True(); }

  (Op o){ return o; }

  (Bin b, And a, True l, True r){ return l; }
  (Bin b, And a, Lit l, Lit r)
   { return new False(); }

  (Bin b, Or o, False l, False r){ return l; }
  (Bin b, Or o, Lit l, Lit r)
   { return new True(); }
)
```

Benefits of Extended Dispatch

- Abstraction

  ```
  (Lit l){...}
  (Bin b, And a, Lit l, Lit r){...}
  ```

  Handle Multiple Cases

- Type Flexibility

  ```
  (Bin b, ...) {...} → Lit
        (Op o){...} → Op
  ```

  Not Strictly Type "*Unifying*" or "*Preserving*"

- Type Safety

  Adding `XOr` to `Op`

  ⇒ `*error* (Bin b, XOr x, ...)` not handled

Type System

Connect Static and Dynamic Worlds

- Advice Lookup Never Fails
- Advice Application Never Fails

Enables Flexible Traversal

- Dynamic (Reflection)
- Heap Based, With/Without Stack
- Complete Inlining

Typing Rules

[T-VAR]

$$\frac{x : T \in \Gamma}{\Gamma \vdash_e x : T}$$

[T-NEW]

$$\frac{\texttt{concrete}\ C\ \texttt{(}\, T_1, \ldots, T_n\, \texttt{)} \in P \qquad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i \ \text{ for all } i \in 1..n}{\Gamma \vdash_e \texttt{new}\ C\ \texttt{(}\, e_1,\ \ldots,\ e_n\, \texttt{)} : C}$$

[T-FUNC]

$$\frac{\overline{x_i : T_i} \vdash_e e_0 : T}{\vdash_F \texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)}\{\ \texttt{return}\ \ e_0\,\texttt{;}\ \} : T}$$

[T-TRAV]

$$\frac{\Gamma \vdash_e e_0 : T_0 \quad \emptyset \vdash_{\mathcal{T}} \langle T_0,\ F \rangle : T; \emptyset}{\texttt{traverse(}\, e_0\,\texttt{,}\ F\,\texttt{)} : T}$$

Traversal Typing Rules

[T-CTRAV]

$$\frac{\begin{array}{c}\texttt{concrete}\ C\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \\ \mathit{types}(\mathit{choose}(F, C)) = (C,\ T''_1,\ \ldots\ , T''_n) \qquad \vdash_F \mathit{choose}(F, C) : T \\ \text{for all } i \in 1..n\ \ T_i \notin \mathcal{X} \Rightarrow\ \mathcal{X} \cup \{C\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T'_i; \Phi_i\ \wedge\ T'_i \leq T''_i \\ (C,\ T') \in (\Phi_1 \cup \cdots \cup \Phi_n)\ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\ T''_j)\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \\ \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n) \backslash (C, \_)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T; \Phi'}$$

[T-ATRAV]

$$\frac{\begin{array}{c}\texttt{abstract}\ A\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \\ \text{for all } i \in 1..n\ \ T_i \notin \mathcal{X} \Rightarrow\ \mathcal{X} \cup \{A\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T'_i; \Phi_i\ \wedge\ T'_i \leq T \\ (A,\ T') \in (\Phi_1 \cup \cdots \cup \Phi_n)\ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\ T)\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \\ \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n) \backslash (A, \_)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T; \Phi'}$$

Soundness

Type System: Rules Out Runtime Errors

- Advice Lookup Never Fails
- Advice Application Never Fails
- Correctly Predicts Program Result

Notables

- Complete Functions
- Subtype Traversals Return Subtypes

Related Work

AOP Semantics: Bruns et al. [2004] Jagadeesan et al. [2003] Wand et al. [2004]

AOP Soundness: Kammüller and Voesgen [2009] Walker et al. [2003]

OO Type Soundness: Igarashi et al. [1999] Flatt et al. [1998]

Constraint Type Systems: Palsberg and Schwartzbach [1991]

Next Steps

Adding Features to the Model

- Multiple-dispatch
- Function Set Extension
- Traversal Control and Abstraction

Towards Traditional Adaptive Programming

Next Steps

Full Language Implementation

- Independent
- Or in a Future Functional Language

Implementation Features

- Type Directed Inlining
- Type Directed Traversal Generation
- Performance results

Contact

Bryan Chadwick: chadwick@ccs.neu.edu

Karl Lieberherr: lieber@ccs.neu.edu

DemeterF Home: http://www.ccs.neu.edu/~chadwick/demeterf/

References

Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer-Verlag, 2004.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *In POPL*, pages 171–183. ACM Press, 1998.

Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *TOPLAS*, pages 132–146, 1999.

Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, pages 54–73, 2003.

Florian Kammüller and Matthias Voesgen. Towards type safety of aspect-oriented languages. In *AOSD 2006, FOAL Workshop*, 2009.

Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: http://doi.acm.org/10.1145/117954.117965.

David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP*, pages 127–139, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944718.

Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.