Selective Open Recursion

A Solution to the Fragile Base-Class Problem

Kevin Donnelly, Jonathan Aldrich

Modularity

What?

- Break complex systems into modules with specified interfaces
- Only expose in the interface those aspects of the code that are essential and unlikely to change
- Type systems can help us ensure that the things we want hidden, are hidden

Why?

- Can reason about the correctness of each component in isolation
- Can make changes to the implementation, using only local reasoning, without breaking the system

Modularity

Object-oriented programming languages provide a number of mechanisms for achieving some level of modularity

- Object-Client modularity
  - Using public/private data members to enforce information hiding
  - Public/private methods and use of interfaces, restrict how an object can be used (allowing implementation changes)
- Object-Subclass modularity
  - Private data and methods can be hidden from subclasses

Modularity

In object-oriented programming languages like Java there are a number of ways to inadvertantly break modularity.

- We can break object-client modularity by, for example, inadvertantly returning a pointer to internal data in some toList method. (Ownership type systems tackle this problem)

- We can break object-subclass modularity by exploiting the fragile base-class problem. It is this flaw in modularity that our proposal for selective open recursion addresses.

The Fragile Base Class Problem

The Fragile Base Class Problem exploits the tight coupling between a class and its subclasses caused by inheritence to tell implementation details about the base class.

```
public class CountingSet extends Set {
    private int count;
    public void add(Object o) {
       super.add(o);
       count++;
    }
    public void addAll(Collection c) {
       super.addAll(c);
       count += c.size();
    }
    public int size() {
       return count;
    }
}
```

The Fragile Base Class Problem

The CountingSet implementation is only correct in the usual java semantics if the add and addAll methods are independent. If the Set class is implemented as follows, then CountingSet doesn't work as expected.

```
public class Set {
    List elements;
    public void add(Object o) {
       if (!elements.contains(o))
               elements.add(o);

    }
    public void addAll(Collection c) {
       Iterator i = c.iterator();
       while(i.hasNext()) {
               add(i.next());
       }

    }

}
```

The Problem? Open Recursion

Java Open Recursion Semantics

- No distinction in between self-calls which are merely for convenience of implementation (as the self-call to add from addAll), and which actually aught to be part of the subclassing interface (as for semantic events)
- As a result, all self-calls must be thought of as part of the subclassing interface
- Our experiments on the Java standard library suggest that most self-calls to public methods do no make use of open recursion semantics. Because open recursion is harder to reason about, the fact that it is rarely used suggests that it should not be the default self-call semantics.

Our Proposal: Selective Open Recursion

Selective Open Recursion

- Calls on the object “this” are statically dispatched by default (i.e. self-calls are treated as convenient helper calls, not semantic events of interest to subclasses)
- A new keyword, “open,” which applies to public non-final methods and designates these methods as semantic events.
- Calls to “open” methods use dynamically dispatched open recursion semantics

Analyzing Java Libraries

We have implemented an analysis which, given whole program information, finds which methods must be annotated with “open” in order to preserve their semantics.

- for each public or protected method m of every class C, the program relies on open semantics of m whenever:
  - there is some method m' in $C' <: C$ that m' calls m on `this`
  - there is some $C'' <: C'$ which overrides m
  - C'' either does not override m' or it makes a `super` call to m'

Analyzing Java Libraries

We applied out analysis to a portion of the JDK 1.4.2 Java library (all packages starting with java except for java.nio and java.sql)
Results:

- Open Annotations
  - 9897 method declarations analysed
  - 1394 methods were overridden
  - only 246 methods required open annotations (3% of total, 18% of overridden)
- Optimization Potential
  - 22339 call sites analysed
  - 6852 were self-calls (on `this`)
  - 716 were calls to open methods
  - This means we can inline 6136 self-calls that we would not otherwise be able to (27% of calls)

Featherweight OpenJava

Syntax:

$$
\begin{array}{rcl}
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ D\{\bar{C}\bar{f}; K\bar{M}\} \\
K & ::= & C(\bar{D}\bar{g}; \bar{C}\bar{f})\{\ \texttt{super}(\bar{g}); \texttt{this}.\bar{f} = \bar{f}\} \\
M & ::= & [\texttt{open}]\ Cm(\bar{C}\bar{x})\{\texttt{return}\ e;\} \\
e & ::= & x \mid \texttt{new}\ C(\bar{e}) \mid e.f \mid (C)e \\
 & \mid & e.m(\bar{e}) \mid e.C :: m(\bar{e}) \\
 & & \\
v & ::= & \texttt{new}\ C(\bar{v}) \\
\Gamma & ::= & [] \mid \Gamma[x \mapsto C]
\end{array}
$$

Featherweight OpenJava

Static Semantics:

$$\frac{}{C <: C}\ Sub-refl \qquad \frac{C <: D \quad D <: E}{C <: E}\ Sub-trans$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ldots}{C <: D}\ Sub-class$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ T-Var \qquad \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C}\ T-Cast$$

$$\frac{\Gamma \vdash \bar{e} : \bar{C} \quad \Gamma \vdash e_0 : C_0 \quad fields(C) = \bar{D}\bar{f} \quad \bar{C} <: \bar{D}}{\Gamma \vdash \texttt{new}\ C(\bar{e}) : C}\ T-New \qquad \frac{fields(C_0) = \bar{C}\bar{f}}{\Gamma \vdash e_0.f_i : C_i}\ T-Field$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad \Gamma \vdash \bar{e} : \bar{C} \quad mtype(m, C_0) = \bar{D} \to C \quad \bar{C} <: \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C}\ T-Invk$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad \Gamma \vdash \bar{e} : \bar{C} \quad mtype(m, E) = \bar{D} \to C \quad \bar{C} <: \bar{D} \quad C_0 <: E}{\Gamma \vdash e_0.E :: m(\bar{e}) : C}\ T-BoundInvk$$

Featherweight OpenJava

More Static Semantics:

$$\frac{\bar{M}\ OK\ in\ C \qquad fields(D) = \bar{D}\bar{g} \qquad K = C(\bar{D}\bar{g}, \bar{C}\bar{f})\{\texttt{super}(\bar{g}); \texttt{this}.\bar{f} = \bar{f};\}}{\texttt{class}\ C\ \texttt{extends}\ D\{\bar{C}\bar{f}; K\ \bar{M}\}\ OK}\ ClassOK$$

$$\frac{override(m, D, \bar{C} \to C_0) \qquad CT(C) = \texttt{class}\ C\ \texttt{extends}\ D... \qquad \{\bar{x} : \bar{C}, \texttt{this} : C\} \vdash e_0 : E_0 \quad E_0 <: C_0}{C_0\ m(\bar{C}\bar{x})\{\texttt{return}\ e_0;\}\ OK\ in\ C}\ MethOK$$

Featherweight OpenJava

Dynamic Semantics:

$$
\begin{array}{rcl}
Ctx & ::= & \mathsf{new}\ C(v_1, ..., v_{i-1}, \Box, e_{i+1}, ..., e_n) \\
 & | & \Box.f \mid \Box.[C ::]m(\bar{e}) \mid (C)\Box \\
 & | & v.[C ::]m(v_1, ..., v_{i-1}, \Box, e_{i+1}, ..., e_n)
\end{array}
$$

$$\frac{fields(C) = \bar{C}\bar{f}}{\mathsf{new}\ C(\bar{v}).f_i \mapsto v_i}\ R-Field$$

$$\frac{C <: D}{(D)\ \mathsf{new}\ C(\bar{v}) \mapsto\ \mathsf{new}\ C(\bar{v})}\ R-Cast$$

$$\frac{mbody(m, C) = (\bar{x}, e_0)}{\mathsf{new}\ C(\bar{v_f}).m(\bar{v}) \mapsto [\bar{v}/\bar{x}, \mathsf{new}\ C(\bar{v_f})/\mathtt{this}]e_0}\ R-Invk$$

$$\frac{mbody(m, C) = (\bar{x}, e_0)}{v.C :: m(\bar{v}) \mapsto [\bar{v}/\bar{x}, v/\mathtt{this}]e_0}\ R-BoundInvk$$

$$\frac{e \mapsto e'}{Ctx[e] \mapsto Ctx[e']}\ R-Context$$

Featherweight OpenJava

The key to Selective Open Recursion semantics lies in the changes to the $mbody$ helper function.

$$\frac{CT(C) = \texttt{class}\ \ C\ \ \texttt{extends}\ \ D\{\bar{C}\bar{f}; K\bar{M}\} \quad m \text{ is not defined in } \bar{M}}{mbody(m, C) = mbody(m, D)}$$

$$\frac{CT(C) = \texttt{class}\ \ C...\{\bar{C}\bar{f}; K\bar{M}\} \quad (D\ m(\bar{D}\bar{x})\{\texttt{return}\ \ e;\}) \in \bar{M}}{mbody(m, C) = (\bar{x}, body(C, e))}$$

The $body$ function replaces normal method calls to `this` on non-open methods with bounded method calls

$$\frac{\begin{array}{c}(e \neq \texttt{this} \vee open(C, m)) \\ body(C, e) = e' \quad \bar{e'} = body(C, \bar{e})\end{array}}{body(C, e.m(\bar{e})) = e'.m(\bar{e'})}\ Body - Meth$$

$$\frac{\bar{e'} = body(C, \bar{e}) \quad \neg open(C, m)}{body(C, \texttt{this}.m(\bar{e})) = \texttt{this}.C :: m.(\bar{e'})}\ Body - Self$$

Featherweight OpenJava

Type Soundness (Exactly the same as for Featherweight Java)

Theorem 1 (Type Preservation)

If $\emptyset \vdash e : C$ *and* $e \mapsto e'$, *then there exists a* $D <: C$ *such that* $\emptyset \vdash e' : D$

Proof: *By induction over the derivation of* $e \mapsto e'$. ■

Theorem 2 (Progress)

If $\emptyset \vdash e : C$ *then either*

- e *is a value, or*
- $e = Ctx[(D)\ \mathsf{new}\ \ E(\bar{v})]$ *with* $E \not<: D$, *or*
- $e \mapsto e'$ *for some* e'

Proof: *By induction over the derivation of* $\emptyset \vdash e : C$. ■

Equivalence

Local Equivalence:

$$\frac{\begin{array}{c} domain(CT_1) = domain(CT_2) \\ \forall C \in domain(CT_1) : (CT_1, C) \simeq (CT_2, C) \end{array}}{CT_1 \simeq CT_2} \; Eq - CT$$

$$\frac{\begin{array}{c} CT_1(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\bar{f}; K\ \bar{M}\} \\ CT_2(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\bar{f}; K\ \bar{M'}\} \\ |M| = |M'| \\ \forall i \in \{1..|M|\}.(CT_1, D, M_i) \simeq (CT_2, D, M'_i) \end{array}}{(CT_1, C) \simeq (CT_2, C)} \; Eq - Cls$$

$$\frac{(CT_1, [\bar{v}/\bar{x}]body(D, e)) \simeq (CT_2, [\bar{v}/\bar{x}]body(D, e))}{(CT_1, D, C\ m(\bar{C}\bar{x})\{\texttt{return}\ e;\}) \simeq (CT_2, D, C\ m(\bar{C}\bar{x})\{\texttt{return}\ e';\})} \; Eq - Meth$$

Equivalence

Local Equivalence:

$$\frac{}{(CT_1, v) \approx (CT_2, v)}\ Eq-Val$$

$$\frac{E \not<: D \quad E' \not<: D'}{\begin{array}{c}(CT_1, Ctx_1[(D)\mathsf{new}\ \ E(\bar{v})]) \\ \approx (CT_2, Ctx_2[(D')\mathsf{new}\ \ E'(\bar{v'})])\end{array}}\ Eq-Error$$

$$\frac{\begin{array}{c}CT_1 \vdash e_1 \overset{stat^*}{\mapsto} e_1' \\ CT_2 \vdash e_2 \overset{stat^*}{\mapsto} e_2' \\ (CT_1, e_1') \approx (CT_2, e_2')\end{array}}{(CT_1, e_1) \approx (CT_2, e_2)}\ Eq-Stat$$

$$\frac{e_1\ and\ e_2\ diverge\ according\ to\ Eq-Stat\ and\ Eq-Dispatch}{(CT_1, e_1) \approx (CT_2, e_2)}\ Eq-Diverge$$

$$\frac{\begin{array}{c}(CT_1, Ctx_1 : C) \approx (CT_2, Ctx_2 : C) \\ CT_1; \emptyset \vdash v.m(\bar{v}) : C\end{array}}{(CT_1, Ctx_1[v.m(\bar{v})]) \approx (CT_2, Ctx_2[v.m(\bar{v})])}\ Eq-Dispatch$$

$$\frac{\begin{array}{c}\forall v'\ such\ that\ \emptyset \vdash v' : C'\ and\ C' <: C. \\ (CT_1, Ctx_1[v']) \approx (CT_2, Ctx_2[v'])\end{array}}{(CT_1, Ctx_1 : C) \approx (CT_2, Ctx_2 : C)}\ Eq-Context$$

Equivalence

Global Equivalence:

$$\frac{}{(CT_1, v) \cong (CT_2, v)}\ Bi - Val \qquad \frac{E \not<: D \quad E' \not<: D'}{\begin{array}{c}(CT_1, Ctx_1[(D)\mathsf{new}\ \ E(\bar{v})]) \\ \cong (CT_2, Ctx_2[(D')\mathsf{new}\ \ E'(\bar{v'})])\end{array}}\ Bi - Error$$

$$\frac{\begin{array}{c}CT_1 \vdash e_1 \overset{stat^*}{\mapsto} e_1' \\ CT_2 \vdash e_2 \overset{stat^*}{\mapsto} e_2' \\ (CT_1, e_1') \cong (CT_2, e_2')\end{array}}{(CT_1, e_1) \cong (CT_2, e_2)}\ Bi - Stat \qquad \frac{\begin{array}{c}e_1\ and\ e_2\ diverge\ according \\ to\ Bi - Stat\ and\ Bi - Dispatch\end{array}}{(CT_1, e_1) \cong (CT_2, e_2)}\ Bi - Diverge$$

$$\frac{\begin{array}{c}CT_1 \vdash Ctx_1[v_1.m(\bar{v_1})] \overset{stat^*}{\mapsto} e_1' \\ CT_2 \vdash Ctx_2[v_2.m(\bar{v_2})] \overset{stat^*}{\mapsto} e_2' \\ (CT_1, e_1') \cong (CT_2, e_2')\end{array}}{(CT_1, Ctx_1[v_1.m(\bar{v_1})]) \cong (CT_2, Ctx_2[v_2.m(\bar{v_2})])}\ Bi - Dispatch$$

Abstraction

The abstraction theorem states that if two libraries, CT_1 and CT_2 are locally equivalent, then any well-typed client code e will execute in a globally equivalent way with both libraries.

Theorem 3 (Client Abstraction)

If $CT_1 \simeq CT_2$*, then* $\forall e$ *such that* $CT_1, \emptyset \vdash e : C$ *and* $CT_2, \emptyset \vdash e : C$*, we have* $(CT_1, e) \cong (CT_2, e)$*.*

Theorem 4 (Library Abstraction)

if $CT_1 \simeq CT_2$*, then* $\forall CT'$ *such that* $CT_1 \cup CT'\ OK$ *and* $CT_2 \cup CT'\ OK$*, we have* $CT_1 \cup CT' \simeq CT_2 \cup CT'$*.*

Conclusions

- The fragile base class problem is caused by the failure to distinguish internal calls that are for convenience and those that are extension points for subclasses.
- Our proposal makes this distinction clear by the use of `open`
- Experiments on the Java library have shown that not using open recursion by default does not add much annotation burden and allows for potentially significant optimization gains
- The abstraction theorem shows that locally equivalent implementation of a library cannot be distinguished by the library's clients
  - This allows more freedom to library designer's to change implementation without breaking subclass code.