A Type System for Dynamic Layer Composition

Atsushi Igarashi  
(Kyoto Univ.)

Joint work with
Robert Hirschfeld (HPI)  
Hidehiko Masuhara (Univ. of Tokyo)
Context-oriented Programming (COP)  
[Hirschfeld, Costanza, Nierstrasz JOT08]

- Goal: modularizing behavioral variations depending on the dynamic context of execution
  - e.g., editor key binding depending on buffer modes
- Several COP extensions of existing (OOP) langs
- Common language features
  - Layers of partial methods
  - Dynamic layer activation mechanism
Dynamic Layer Activation in COP

Base class hierarchy

- C
  - m1
  - m2
- D
- E
  - m3
- F
  - m4

Layer of partial methods

- C
  - m2
- D
  - m1
- E
- F
  - m5
Dynamic Layer Activation in COP

Base class hierarchy

Layer of partial methods

- Layer activation changes behavior of objects *that have been already instantiated*

- Partial methods can call the original behavior by `proceed()`
This Work

Type system to prevent "NoSuchMethod" incl. dangling proceed calls
“Sounds like an old problem. What is a challenge?”

- Object interfaces can change as layers are (de)activated!

Overriding partial method

“Layer-introduced” method, which can dynamically change the object interface!
This Work

Type system to prevent “NoSuchMethod” incl. dangling proceed calls

• By keeping track of which layers are activated at each program point
  • Using declaration of dependency between layers

Restriction:

• Activation/deactivation constructs are (slightly) different from the original
Technical Contributions

- Formal type system for (a variant of) ContextFJ [Hirschfeld, I., Masuhara @FOAL'11]
  - FJ-style calculus modeling COP features
- Proof of type soundness
Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
- Conclusion and Future Work
**COP Language Constructs Considered in This Work**

- **Partial methods**
  - Smallest unit to describe behavioral variations
  - Comparable to advice in AOP
- **Layers**
  - A bunch of partial methods
  - Unit of modularity/cross-cutting concerns
- **Block-structured *global* layer activation**
  - *ensure statements* (a variant of *with*)
class Connection {
    Connection(Customer a, Customer b) { ... }
    void complete() { ... }
    void drop() { ... }
}

Connection simulate() {
    Customer robert = ..., hidehiko = ...;
    Connection c = new Connection(robert, hidehiko);
    // Robert calls Hidehiko
    c.complete();  // Hidehiko accepts
    c.drop();      // and hangs up
    return c;
}
Example: Telecom simulation

```java
class Connection {
    Connection(Customer a, Customer b) { ... }
    void complete() { ... }
    void drop() { ... }
}

Layer Timing {
    class Connection {
        Timer timer;
        void complete() { proceed(); timer.start(); }
        void drop() { timer.stop(); proceed(); }
        int getTime() { return timer.getTime(); }
    }
}

ensure (Timing) { // layer activation!
    Connection c = simulate();
    System.out.println(c.getTime());
}
```

- Layer activation is valid inside `simulate()`
Example: Telecom simulation

- Layer activation is valid inside `simulate()`
- Recently activated layer has priority
Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
- Conclusion and Future Work
Main Problem and Key Ideas

Main Problem:

• Layer activation can change object interface

Key Ideas:

• Approximation of activated layers at each program point
  • With the help of explicit “requires” declarations to specify inter-layer dependency
Telecom example, revisited

- For `charge()` in Billing to work, layer-introduced base method `getTime()` defined in Timing should be active beforehand
For **charge()** in Billing to work, layer-introduced base method **getTime()** defined in Timing should be active beforehand.

In other words, Billing requires Timing.
Meaning of requires

When layer $L$ requires $L_1, ..., L_n$

- Partial method in $L$ can invoke methods defined in any of $L_1, ..., L_n$ (or base)

- Partial method $m$ in $L$ can proceed when $m$ is defined in any of $L_1, ..., L_n$ (or base)

- All of $L_1, ..., L_n$ must have been already activated before activating $L$
A Bit of Formalism

ContextFJ calculus [Hirschfeld, I., Masuhara @FOAL'11]

\[
\begin{align*}
L &::= \text{class } C \text{ extends } C \{ \sim C \sim f; \sim M \} \\
M &::= C \text{ } m(\sim C \sim x)\{ \text{return } e; \} \\
e &::= x | e.f | e.m(\sim e) | \text{new } C(\sim e) \\
&\quad | \text{ensure } L \text{ } e | \text{proceed}(\sim e) | \text{super}.m(\sim e)
\end{align*}
\]

- A ContextFJ program is \((CT, PT, e)\), where
  - Class table: \(CT(C) = L\)
  - Partial method table: \(PT(m,C,L) = M\)
Type Judgment \[ \Lambda; \Gamma \vdash e : C \]

"Under set \( \Lambda \) of activated layers and type env. \( \Gamma \), exp \( e \) is given type \( C \)"

- \{\}; \text{c: Conn.} \vdash \text{c.getTime()} : \text{int}
- \{\text{Timing}\}; \text{c: Conn.} \vdash \text{c.getTime()} : \text{int}
- \{\}; \text{c: Conn.} \vdash \text{ensure Timing c.getTime()} : \text{int}
- \{\}; \text{c: Conn.} \vdash \text{ensure Billing c.drop()} : \text{void}
- \{\text{Timing}\}; \text{c: Conn.} \vdash \text{ensure Billing c.drop()} : \text{void}
Main Typing Rules

• Typing rule for method invocation

\[
\Lambda; \Gamma \vdash e_0 : C_0 \quad \text{mtype}(m,C_0,\Lambda) = \sim D \rightarrow C
\]

\[
\Lambda; \Gamma \vdash \sim e : \sim C \quad \sim C <: \sim D
\]

\[
\Lambda; \Gamma \vdash e_0 . m(\sim e) : C
\]

• Typing rule for layer activation

\[
L \text{ req } \Lambda' \quad \Lambda' \subseteq \Lambda \quad \Lambda \cup \{L\}; \Gamma \vdash e : C
\]

\[
\Lambda; \Gamma \vdash \text{ensure } L e : C
\]
Main Typing

- Typing rule for method invocation

\[
\Lambda; \Gamma \vdash e_0 : C_0 \quad \text{mtype}(m, C_0, \Lambda) = \sim D \rightarrow C
\]

\[
\Lambda; \Gamma \vdash \sim e : \sim C \quad \sim C < : \sim D
\]

\[
\Lambda; \Gamma \vdash e_0.m(\sim e) : C
\]

- Typing rule for layer activation

\[
L \ \text{req} \quad \Lambda' \subseteq \Lambda \quad \Lambda \cup \{L\}; \Gamma \vdash e : C
\]

\[
\Lambda; \Gamma \vdash \text{ensure} \quad L \ e : C
\]
Main Typing

- Typing rule for method invocation

\[ \Lambda; \Gamma \vdash e_0 : C_0 \quad \text{mtype}(m, C_0, \Lambda) = \sim D \rightarrow C \]

\[ \vdash \sim e : \sim C \]

\[ \Gamma \vdash e_0.m(\sim e) : C \]

- Typing rule for layer activation

\[ L \text{ req } \Lambda' \quad \Lambda' \subseteq \Lambda \quad \Lambda \cup \{L\}; \Gamma \vdash e : C \]

\[ \Lambda; \Gamma \vdash \textbf{ensure } L e : C \]

Method lookup takes activated layers into account

Layers that L requires are already activated

Body of ensure is typed under additional assumption
Type Soundness

- **Reduction:**
  \[ L_1; \ldots; L_n \vdash e \rightarrow e' \]

- **Thm. (Subject Reduction):**
  - If \( \{L_1, \ldots, L_n\}; \Gamma \vdash e : C \) and \( L_1; \ldots; L_n \vdash e \rightarrow e' \) and \( L_1; \ldots; L_n \) is well formed,
    then \( \exists D. \{L_1, \ldots, L_n\}; \Gamma \vdash e' : D \) and \( D <: C \)

- **Thm. (Progress)**
  - If \( \{L_1, \ldots, L_n\}; \cdot \vdash e : C \),
    then \( e \) is a value or \( \exists e'. e \rightarrow e' \)
Type Soundness

- **Reduction:** $L_1;...;L_n \vdash e$ 

- **Thm. (Subject Reduction):**
  - If $\{L_1,...,L_n\}; \Gamma \vdash e : C$ and $L_1;...;L_n \vdash e \rightarrow e'$ and $L_1;...;L_n$ is well formed,
    then $\exists D. \{L_1,...,L_n\}; \Gamma \vdash e' : D$ and $D <: C$

- **Thm. (Progress):**
  - If $\{L_1,...,L_n\}; \cdot \vdash e : C$,
    then $e$ is a value or $\exists e'. e \rightarrow e'$

- **Empty seq. is wf.**
- **$L_1;...;L_n$ is wf.**
  - if $L_1;...;L_{n-1}$ is wf.
  - and $L_n$ (req; $\subseteq$) $\{L_1,...,L_{n-1}\}$

*Sequence of active layers*
- Empty seq. is wf.
- $L_1;...;L_n$ is wf.
  - if $L_1;...;L_{n-1}$ is wf.
  - and $L_n$ (req; $\subseteq$) $\{L_1,...,L_{n-1}\}$
Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
  - Activation constructs
  - Related work
- Conclusion and Future Work
Constructs for (de)activation

The original version of ContextFJ, as well as implementation (ContextJ, JCOP) has different constructs for (de)activation

- with for activation
- without for (block-structured) deactivation
Comparing `ensure` and `with`

- Difference emerges when the same layer is to be activated twice

```
ensure Timing {
  ensure Billing {
    ensure Timing {
      ...
  }
}

with Timing {
  with Billing {
    with Timing {
      ...
    }
  }
```

```
Billing
Timing
...
...
...
...
```
Why not with and without?

- Because they break the invariant (i.e. well-formedness) enforced by the type system!

```plaintext
ensure Timing {
  ensure Billing {
    without Timing {
      ... 
    }
  }
}

with Timing {
  with Billing {
    with Timing {
      ... 
    }
  }
}
```
Related Work

• Type System for COP [Clarke & Sergey@COP'09]

• ContextFJ
  – proposed independently of us
  – no inheritance, subtly different semantics

• Set of method signatures as method-wise dependency information
  – Finer-grained specification

• No proof of soundness
  – In fact, the type system turns out to be flawed (personal communication), due to without
Related Work, contd.

• **Type Systems for Mixins** [Bono et al., Flatt et al., Kamina&Tamai, etc.]

  • Interfaces of classes to be composed
    - Structural type information

• Composition is fixed once an object is instantiated

• A similar idea works (to some extent ;-) also for more dynamic composition as in COP

• Types for FOP, DOP
Related Work, contd.\(^2\)

- **Typestate checking** \([\text{Strom}&\text{Yemini}\,'86, etc.}\]
- Checking state transition for computational resources (such as files and sockets)
- Layer configuration can be considered a state
  - But it's global and unique
Conclusion

Type system for dynamic layer composition

- Estimation of (globally) activated layers at each program point
- Explicit requires clauses to help typechecking
- Soundness proof

Future work: Dealing with the originally proposed layer activation mechanism (with and without)