Foundations of Context-Oriented Programming or Typing Dynamic Layer Composition

Atsushi Igarashi (Kyoto U.)

joint work with

Robert Hirschfeld (HPI)

Hidehiko Masuhara (U. Tokyo)

Hiroaki Inoue (Kyoto U.)

Context-Oriented Programming (COP) Language [Costanza, Hirshfeld DLS05] [Hirschfeld, Costanza, Nierstrasz JOT08]

Goal:

Support for *behavioral variations* depending on the *dynamic* context of execution

Example: Mobile email app

When network is fast
inline images are shown

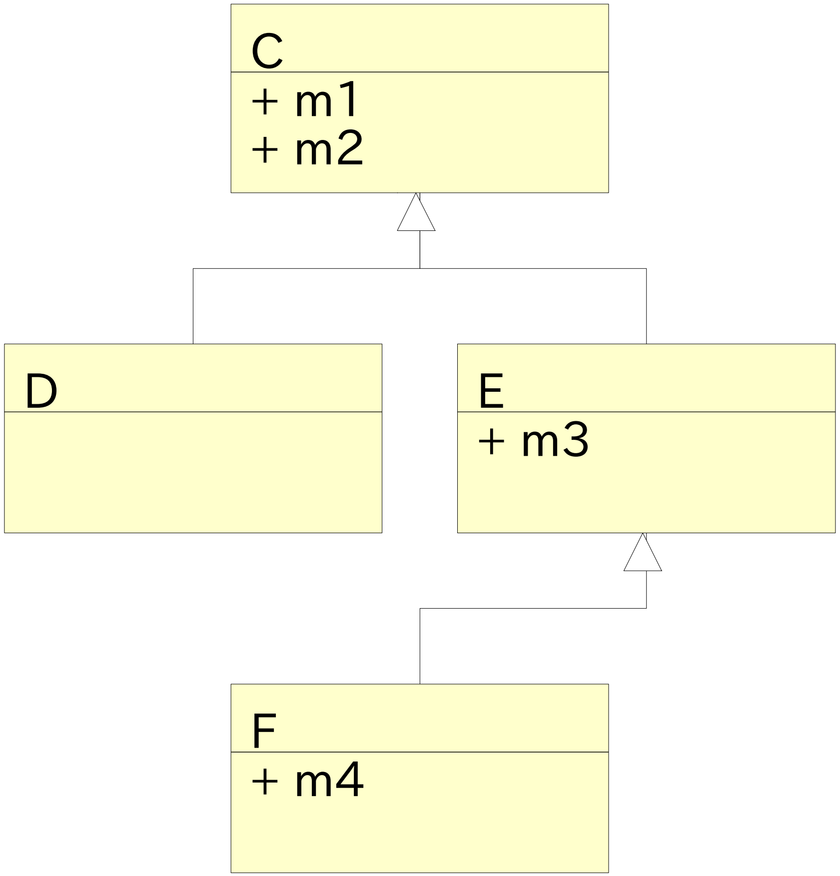
When network is slow
no images are shown

Common COP language features

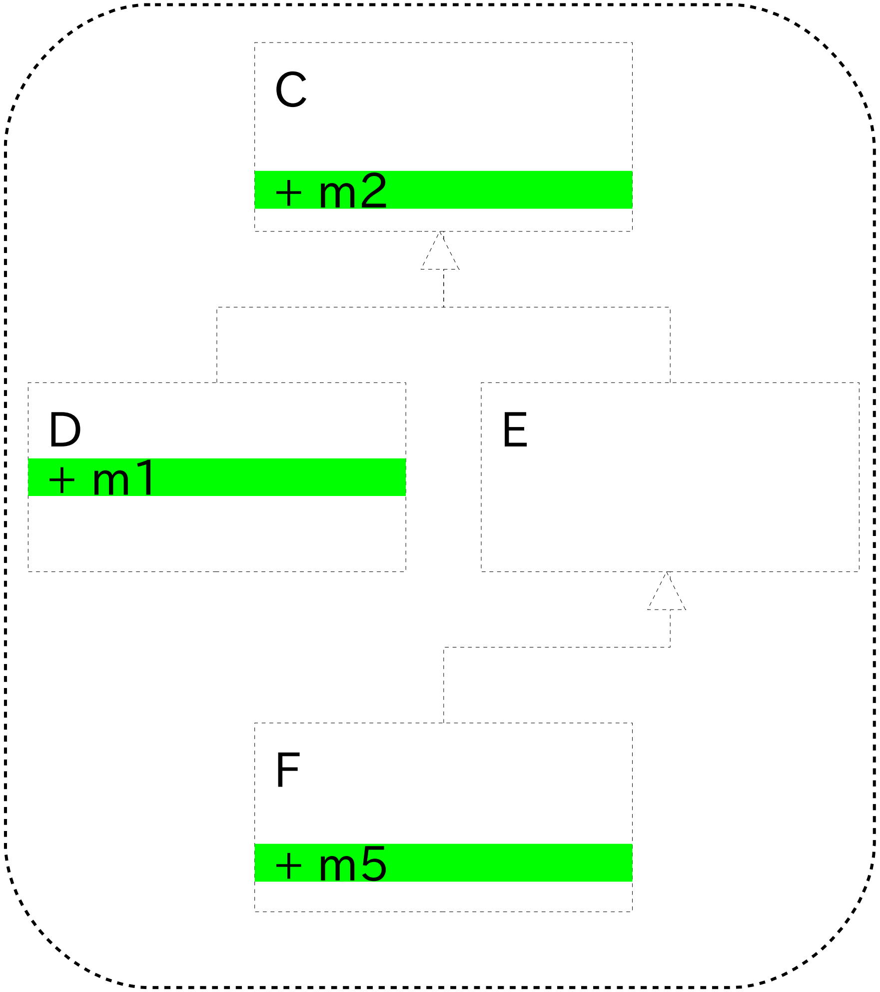
- Layer
  - A unit of behavioral variations, consisting of *partial* method definitions for multiple classes
  - (Loose) correspondence to contexts
  - A unit of cross-cutting modularity
- Dynamic layer activation
  - To change the behavior of a set of objects at the same time

Dynamic Layer Activation in COP

Base class hiearchy

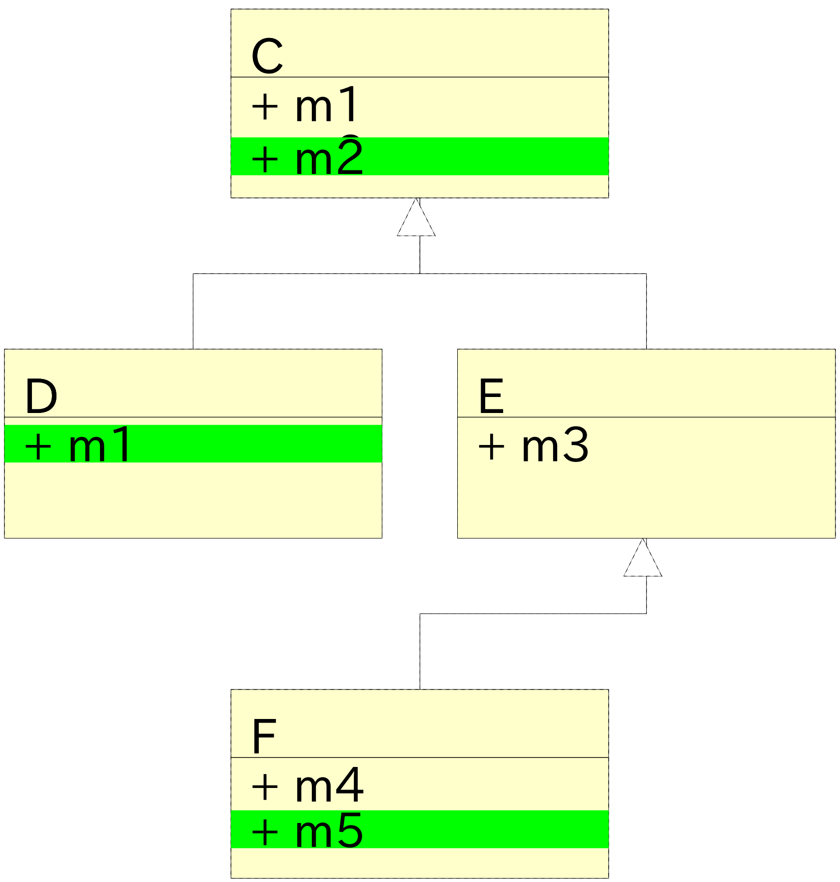


Layer of partial methods



Dynamic Layer Activation in COP

Base class hiearchy



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 Talk

- Quick tour on JCop [Appeltauer+], a specific implementation of COP on top of Java
  - With a more concrete example
  - (Comparison with AOP using pointcut/advice)
- Foundations for COPL
  - (Operational) Semantics
  - Type System

This Talk

- Quick tour on JCop [Appeltauer+], a specific implementation of COP on top of Java
  - With a more concrete example
  - (Comparison with AOP using pointcut/advice)
- Foundations for COPL
  - (Operational) Semantics
  - Type System

Example: Telecom simulation
(adapted from AOP example)

- Class `Conn` to represent connection between two `Customers`
  - `complete()` when a connection has been established
  - `drop()` when the customers are disconnected
- Behavioral variations to consider
  - Recording the lengths of conversations
  - Billing

Base Program

```
class Conn { // Connection
  Conn(Customer a, Customer b) { … }
  void complete() { … }
  void drop() { … }
    // details are not important ...
}
```

```
Conn simulate() {
  Customer robert = …, hidehiko = …;
  Conn c = new Conn(robert, hidehiko);
                     // Robert calls Hidehiko
  c.complete();      // Hidehiko accepts
  c.drop();          // and hangs up
  return c;
}
```

Layer for Measuring Time

```
layer Timing {
  Timer timer = …;
  void Conn.complete(){ proceed(); timer.start();}
  void Conn.drop(){ timer.stop(); proceed(); }
  int Conn.getTime() { return timer.getTime(); }
}
```

- The two methods in `Conn` are modified by *partial* method definitions to operate the timer
  - The original behavior is represented by **`proceed()`**
- **`getTime()`** is newly introduced
  - but also called "partial" method

Layer Activation with `with`

```
with (new Timing()) {  // layer activation!
   Conn c = simulate();
   System.out.println(c.getTime());
}
```

- **`with`** block to activate a layer
- Activation is effective even in methods invoked inside the block
- A layer *instance* has to be created
  - Layer instances are also first-class objects

Layer for Billing

```
layer Billing {
  void Conn.drop() { proceed(); charge(); }
  void Conn.charge() { … getTime(); … }
}
```

```
with (new Timing()) {
  with (new Billing()) {
    Connection c = simulate();
} }
```

- Recently activated layer has priority
  - **drop()** will stop the timer, hang the call, and charge

Not in this example, but...

- One layer can contain partial methods belonging to different classes
  - c.f. Mixin layers [Smaragdakis&Batory 98]
- **`super()`** is also supported
- Layer inheritance/subtyping

Layer Inheritance/Subtyping

- Implementation of different billing policies, switched by run-time conditions

```
abstract layer AbsBilling {
  void Conn.drop();
  void Conn.charge();
}
layer Billing1 extends AbsBilling { … }
layer Billing2 extends AbsBilling { … }
```

```
AbsBilling b =
  some_cond ? new Billing1():new Billing2();

with(b) { … }
```

Very rough Comparison with PA-style AOP

	COP	AOP
Unit of behavior	partial meth.	advice
Oblivious?	No	Yes
Join points	Meth. exec.	Many kinds
Pointcut	cflow + execution	Many kinds

Some Foundational Questions

- What is the semantics of method invocations?
  - What happens when the same layer is activated more than once?
  - How do `proceed`, `super`, and `with` interact with each other?
- How can types prevent `NoSuchMethodError`?
  - Object interface can change dynamically!
  - Only overriding partial methods can `proceed`

This Talk

- Quick tour on COP language features
  - With a more concrete example
- Foundations for COPL
  - (Operational) Semantics
  - Type System

A core calculus of COP: ContextFJ

[Hirschfeld, I., Masuhara FOAL11]

ContextFJ = Featherweight Java [I.,Pierce,Wadler'99]

+ partial methods

+ proceed(), super()

+ with expressions

- layers are global and second-class

- no layer inheritance

ContextFJ<:

[I., Inoue APLAS'15]

ContextFJ<: = Featherweight Java

+ partial methods
+ proceed(), super()
+ with expressions

+ first-class layers (w/o fields)

+ layer inheritance

+ layer subtyping

Syntax (1/2)

"~" for sequences

$T ::= C \mid L$	*types*
$CL ::=$ class $C < D$ { $\sim T \sim f$; $\sim M$ }	classes
$LA ::=$ layer $L < L$ { $\sim PM$; }	layers
$M ::= T\ m(\sim T \sim x)${ return e; }	methods
$PM ::= T\ C.m(\sim T \sim x)${ return e; }	partial meth.
$e ::= x \mid e.f \mid e.m(\sim e) \mid$ new $T(\sim e)$	expressions
$\mid$ with e e	layer activation
$\mid$ proceed($\sim e$)	proceed call
$\mid$ super.$m(\sim e)$	super call

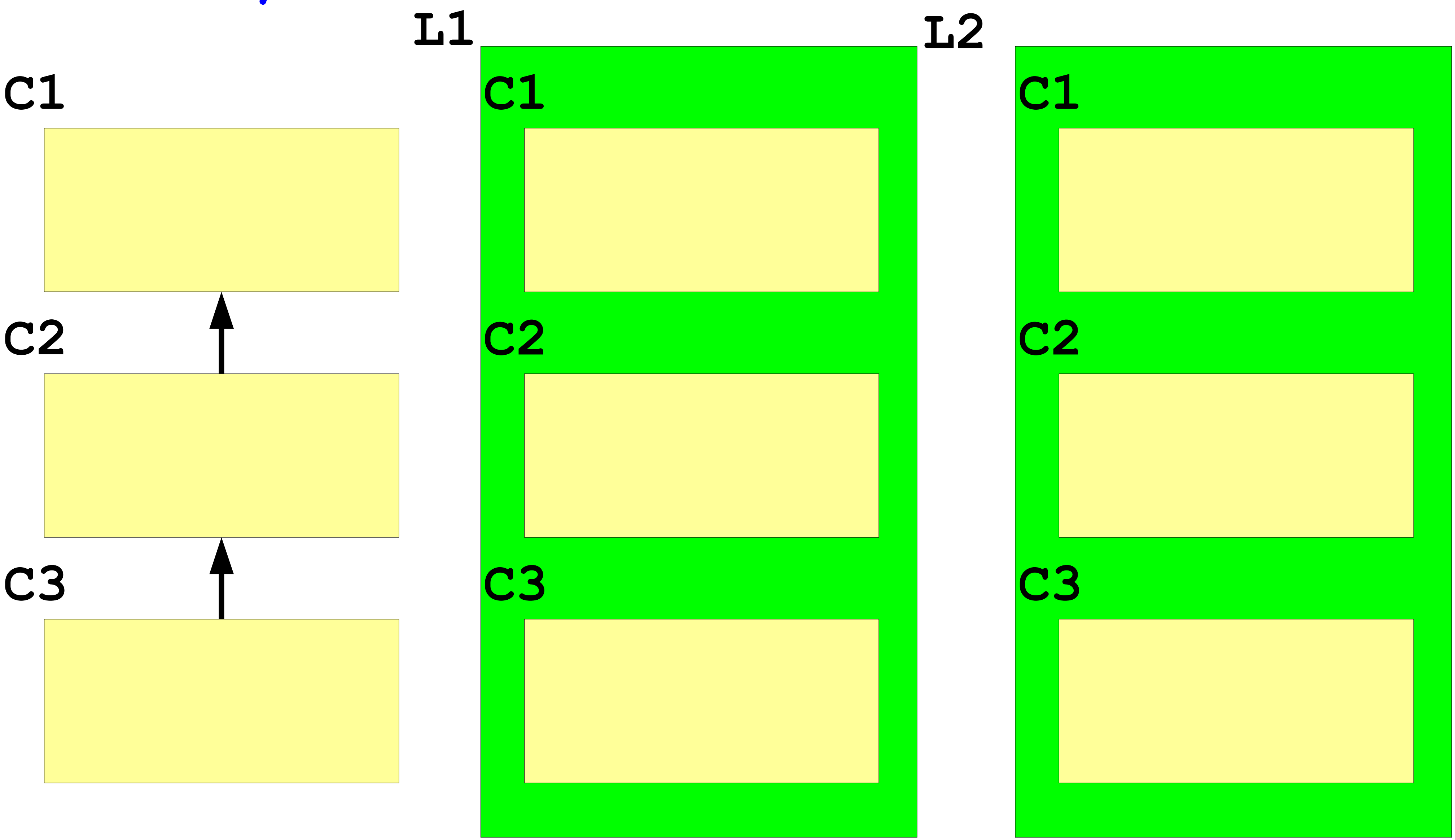
Syntax (2/2)

ContextFJ program: (CT, LT, e)

- Class table: $CT(C) = CL$
- Layer table: $LT(L) = LA$
- Main expression: e

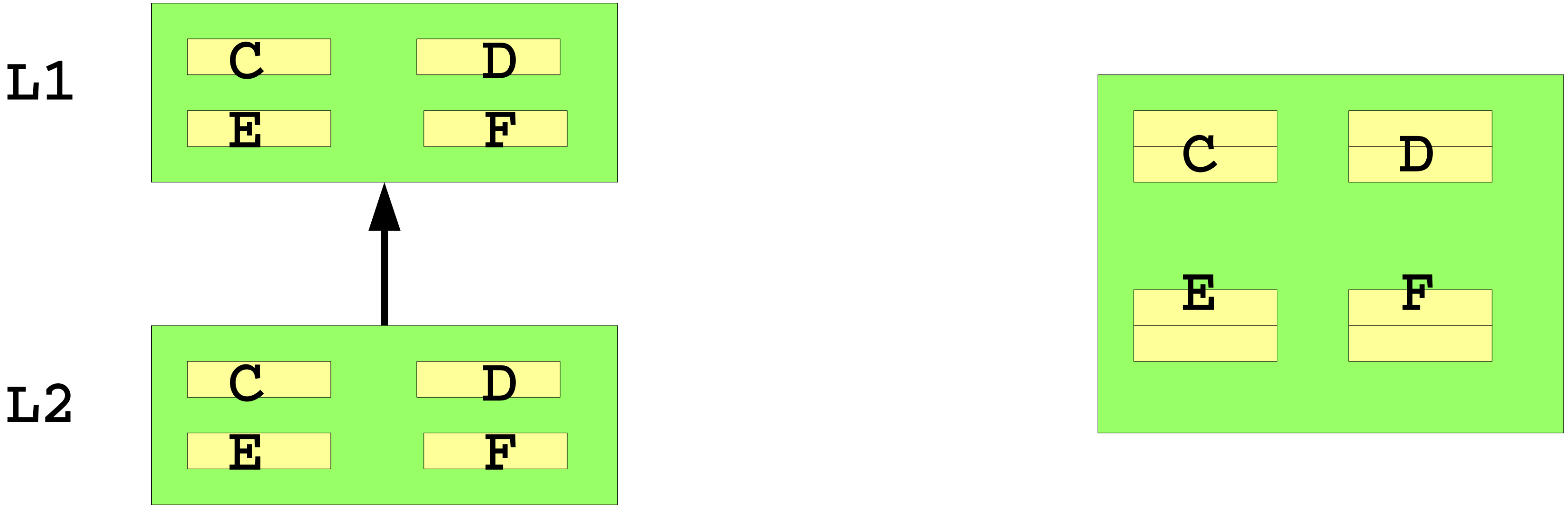
Semantics of Method Dispatch w/o Layer Inheritance

```
with (new L1()) {
  with (new L2()) {
    c3.m(…);
} }
```



Semantics with Layer Inheritance

- "3D" dispatching
- Each layer can be thought of as the result of (possibly overriding) composition of superlayers



Lookup function: *mbody*

$mbody(m, C, \sim L_1, \sim L_2) = \sim x.e$ in D, $\sim L_3$

- "Body of method *m* in *C* is *e* with params $\sim x$"
- $\sim L_2$ is the list of activated layers
- C, $\sim L_1$ denote the currently focused position
  - Acting like a cursor
- D, $\sim L_3$ denote where $\sim x.e$ is found

Reduction: $\sim L \vdash e \rightarrow e'$

- "*e* reduces to *e'* under activated layers $\sim L$"
  - Instances from the same layer are not really distinguished (because there are no states)
- *e.g,*
  - `Timing` $\vdash$ `new Conn(...).drop()` $\rightarrow$ `new Conn(...).timer.stop(); proceed()`
  - `Timing; Billing` $\vdash$ `new Conn(...).drop()` $\rightarrow$ `proceed(); charge()`
    - … actually, **`proceed`** is replaced at this point (see next slides)

Reduction rule for layer activation

$$\frac{remove(L, \sim L) = \sim L' \quad \sim L'; L \vdash e \rightarrow e'}{\sim L \vdash \text{with } L\ e \rightarrow \text{with } L\ e'}$$

- The body e is reduced under the context where L is added
  - Activated layer L *always* comes at the top
    - Even when it's already been activated

```
Timing ⊢
  with Billing (new Conn(...).drop())
 → with Billing (proceed(); charge())
```

Run-time expression to deal with proceed and super

e ::= … | new *C*<*D*,~*L1*,~*L2*>(~*v*)

- Essentially new *C*(~*v*).*m*(~*e*)
- Annotation <*D*,~*L1*,~*L2*> remembers
  - where method lookup starts next time (*D*, ~*L1*)
  - what layers have been activated (~*L2*)

```
Timing; Billing ⊢ new Conn(...).drop() →
 new Conn<Conn,Timing,(Timing;Billing)>
   (...).drop();
 charge()
```

Reduction Rules for Method Invocation

$$\frac{\sim L \vdash \text{new } C(\sim v)\langle C,\sim L,\sim L\rangle.m(\sim w) \to e'}{\sim L \vdash \text{new } C(\sim v).m(\sim w) \to e'}$$

$$\frac{\begin{array}{l} mbody(m, D, \sim L_1, \sim L_2) = \sim x.e \text{ in } E, (\sim L_3; L) \\ \text{class } E < F \end{array}}{\begin{array}{l} \sim L_4 \vdash \text{new } C(\sim v)\langle D,\sim L_1,\sim L_2\rangle.m(\sim w) \to \\ \quad [\text{new } C(\sim v) / \text{this}, \\ \quad \sim w \qquad / \sim x, \\ \quad \text{new } C\langle E, \sim L_3, \sim L_2\rangle(\sim v).m \;\; / \text{ proceed}, \\ \quad \text{new } C\langle F, \sim L_2, \sim L_2\rangle(\sim v) \;\; / \text{ super} \;\;]\; e \end{array}}$$

Reduction Rules for Method Invocation

$$\frac{\sim L \vdash \text{new } C(\sim v)\langle C, \sim L, \sim L\rangle.m(\sim w) \rightarrow e'}{\sim L \vdash \text{new } C(\sim v).m(\sim w) \rightarrow e'}$$

Invocation on an "unannotated" object
is affected by currently activated layers ~L

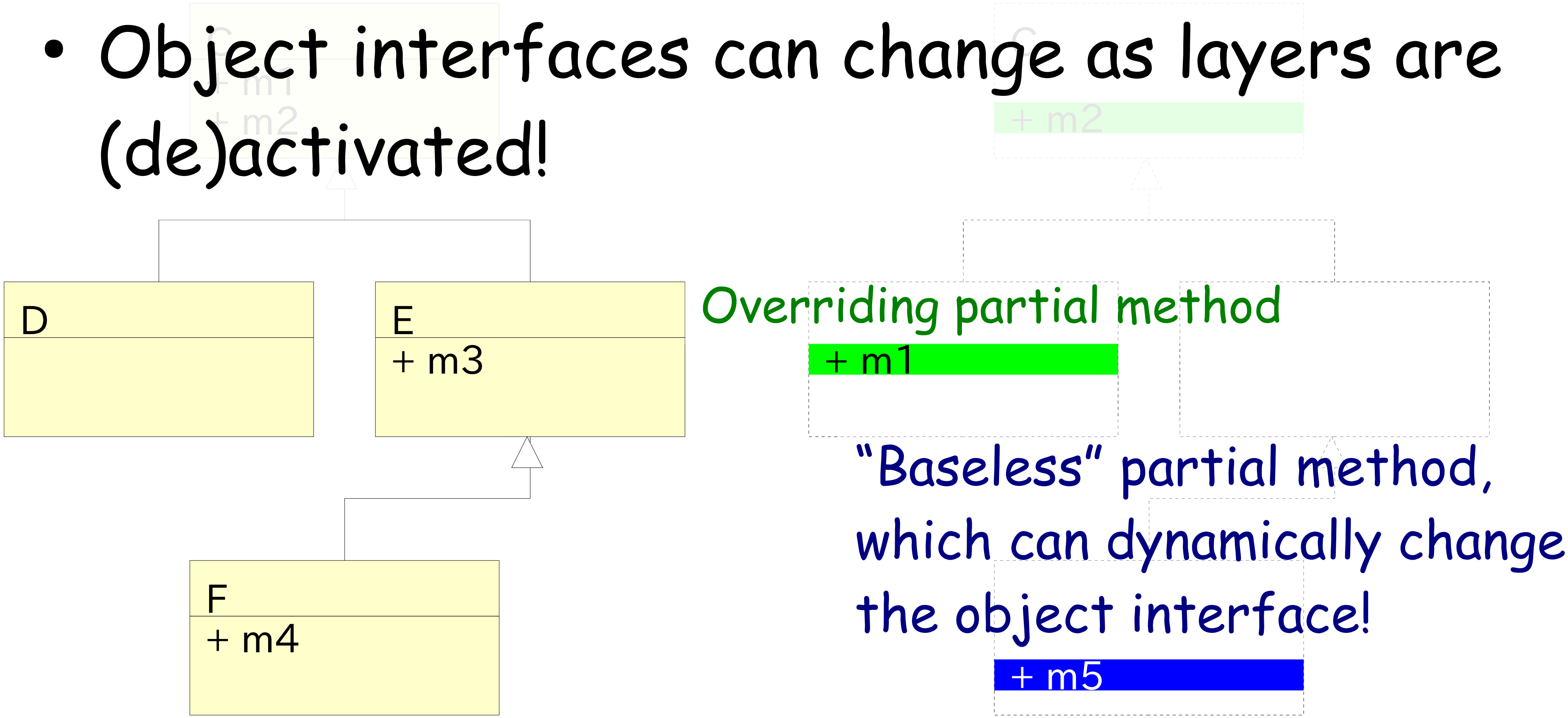
[new $C(\sim v)$ / this,
$\sim w$ / $\sim x$,
new $C\langle E, \sim L_3, \sim L_2\rangle(\sim v).m$ / proceed,
new $C\langle F, \sim L_2, \sim L_2\rangle(\sim v)$ / super] e

This Talk

- Quick tour on COP language features
  - With a more concrete example
- Foundations for COPL
  - (Operational) Semantics
  - Type System
    - To prevent "NoSuchMethodError" including dangling `proceed` calls

"Sounds like an old problem. What is a challenge?"

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



Key Idea (1/2)

Approximating activated layers at each program point

- With the help of explicit "**`requires`**" declarations to specify inter-layer dependency
  - Static analysis could dispense with such explicit declarations, though

Key Idea (2/2)

Two kinds of substitutability for layers

- When one layer L1 `requires` layer L2, does a sublayer of L2 can satisfy L1's requirement?
- When is it safe to pass an instance of a layer to where a supertype is expected?

should be distinguished

Telecom example, revisited

```
class Conn {
  Conn(Custome
  void complet
  void drop()
}
```

```
layer Timing {
  Timer Conn.tim
  void Conn.comp
  void Conn.drop
  int Conn.getTi
}
```

```
layer Billing                                  {


  void Conn.drop(){ proceed(); cl

  void Conn.charge(){ … getTime()
}
```

- For **charge()** in **Billing** to work, baseless partial method **getTime()** defined in **Timing** should be active beforehand

Telecom example, revisited

```
class Conn {
  Conn(Custome
  void complet
  void drop()
}
```

```
layer Timing {
  Timer Conn.tim
  void Conn.comp
  void Conn.drop
  int Conn.getTi
}
```

```
layer Billing requires Timing  {


  void Conn.drop(){ proceed(); ch

  void Conn.charge(){ … getTime(
}
```

- For `charge()` in `Billing` to work, baseless method `getTime()` defined in `Timing` should be active beforehand
- In other words, `Billing requires Timing`

Meaning of `requires`

When layer L **`requires`** $L_1, \ldots, L_n$

- All of $L_1, \ldots, L_n$ must have been already activated (in any order) before activating L
- Partial method in L can invoke methods defined in any of $L_1, \ldots, L_n$ (or base class)
- Partial method m in L can **`proceed`** when m is defined in any of $L_1, \ldots, L_n$ (or base class)

Coeffect system?

Type Judgment $\Lambda; \Gamma \vdash e : T$

"Under set Λ of activated layers and type env. Γ, exp e is given type C"

- {}; c: **`Conn`** $\vdash$ **`c.getTime()`** : **`int`**
- {**`Timing`**}; c: **`Conn`** $\vdash$ **`c.getTime()`** : **`int`**
- {}; c: **`Conn`** $\vdash$ **`with (new Timing())c.getTime()`** : **`int`**
- {}; c: **`Conn`** $\vdash$ **`with (new Billing())c.drop()`** : **`void`**
- {**`Timing`**}; c: **`Conn`**

  $\vdash$ **`with (new Billing()) c.drop()`** : **`void`**

Main Typing Rules

- Typing rule for method invocation

$$\frac{\Lambda; \Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0, \Lambda) = \sim T \rightarrow T_0 \qquad \Lambda; \Gamma \vdash \sim e : \sim S \qquad \sim S <: \sim T}{\Lambda; \Gamma \vdash e_0 . m (\sim e) : T_0}$$

- Typing rule for layer activation

$$\frac{\Lambda; \Gamma \vdash e1 : L \qquad \Lambda \cup \{L\}; \Gamma \vdash e2 : T}{\Lambda; \Gamma \vdash \texttt{with}\ e1\ e2 : T}$$

Inheritance and `requires`

- Sublayer can't require fewer layers than its parent
  - Otherwise, requirement by inherited partial methods may be invalidated
- It seems natural to allow a sublayer to require more layers ...

...Or, maybe not!

```
AbsBilling b =
  some_condition ? new Billing1():new Billing2();

with(b) { … }
```

- The type system seems to always allow `with(b)` (if `AbsBilling` requires no layer)
- But, what if `Billing2` requires more layers than `AbsBilling`?
  - At run time, dependecy is broken!!

Our Solution: Two subtyping rels for layer types

- Weak subtyping: reflexive transitive closure of `extends`
- Normal subtyping: reflexive transitive closure of `extends` *with invariant* `requires`

Main Typing Rules, revisited

- Typing rule for method invocation

$$\frac{\Lambda; \Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0, \Lambda) = \sim T \to T_0 \qquad \Lambda; \Gamma \vdash \sim e : \sim S \qquad \sim S <: \sim T}{\Lambda; \Gamma \vdash e_0\texttt{.}m\,\texttt{(}\sim e\texttt{)} : T_0}$$

- Typing rule for layer activation

$$\frac{L\ \texttt{req}\ \Lambda' \qquad \Lambda <:_w \Lambda' \qquad \Lambda; \Gamma \vdash e1 : L \qquad \Lambda \cup \{L\}; \Gamma \vdash e2 : T}{\Lambda; \Gamma \vdash \texttt{with}\ e1\ e2 : T}$$

Other notable features

- Checking correct method overriding requires the whole program
  - Accidental conflict between partial layers

Type Soundness

- Thm. (Type Soundness):
  - If $\vdash e : T$ and $\vdash e \rightarrow^* e'$ (normal), then e' = new $S(\sim v)$ and $S <: T$
- Proof by showing Preservation and Progress
  - Induction is trickier than you might expect

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 [Strom&Yemini'86, etc.]
  - Checking state transition for computational resources (such as files and sockets)
  - Layer configuration can be considered a state
  - Resources are first-class whereas layers are global

Conclusion

- Dynamic layer composition for describing context-dependent behavioral change concisely and modularly
- Core calculus ContextFJ<: for formal semantics and type system
  - Estimation of (globally) activated layers
  - Explicit `requires` clauses to help typechecking
  - Two kinds of subtyping
  - (Layer swapping)
- (Implementation will be available)

Future work

- Interfaces for layers
  - Specifying layer names makes your program too implementation-specific
- Formal accounts of advanced COP features
  - Other activation mechanisms, e.g., in EventCJ [Kamina, Aotani, Masuhara AOSD'11]
- More formal connection to coeffects?
- Verification?