ContextFJ

A Core Calculus for Context-Oriented Programming

Atsushi Igarashi
(Kyoto Univ.)

Joint work with
Robert Hirschfeld (HPI)
Hidehiko Masuhara (Univ. 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) languages has been proposed
  - Java, Smalltalk, Common Lisp, JavaScript

This Work

First step towards a formal account of COP langs

- ContextFJ calculus
  - In the style of Featherweight Java [Igarashi et al.'99]
  - Direct operational semantics
    - c.f., encoding COP programs into other formalisms [Molderez et al. '10; Schippers et al. '08]
- (Very Simple) Type System for ContextFJ
  - Proof of Type Soundness

Plan of the Talk

- COP Language Constructs
- ContextFJ
  - Syntax
  - Operational Semantics
  - Simple Type System
- Future Work

COP Language Constructs

- 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
- Dynamically scoped layer (de)activation
  - with/without statements

Example: Personal data class

- Fields: name, address, employer
- Behavioral variations for **`toString()`**
  - **`“Name: ” + name;`**
  - **`“Name: ” + name + “; Addr: ” + address`**
  - **`“Name: ” + name + “; Affil: ” + employer`**
  - …

```
class Person {
  String name, addr, employer;
  Person(String name, String addr,
         String employer){ … }
  String toString() { return "Name: " + name; }

  layer Contact {
    String toString() {
      return proceed() + "; Addr: " + addr;
    }
  }

  layer Employment {
    String toString() {
      return proceed() + "; Affl: " + employer;
    }
  }
 }
```

- Partial method(s) in one layer will be simustaneously activated
- There may be other partial methods defined inside another class

```
cl
  layer Contact {
    String toString() {
      return proceed() + "; Addr: " + addr;
    }
  }

  layer Employment {
    String toString() {
      return proceed() + "; Affl: " + employer;
    }
  }
}
```

- Partial method(s) in one lay… simustaneously activated
- There may be other parti… inside another class

Partial method defs. for toString()

```
layer Contact {
  String toString() {
    return proceed() + "; Addr: " + addr;
  }
  ...
    ...g() {
    ...n proceed() + "; Affl: " + employer;
  }
 }
}
```

Call to the "original" behavior

```
Person me = new Person("Igarashi", "Kyoto",
                       "Kyoto U.");

println(me.toString());   // "Name: Igarashi"

with (Contact) {
  println(me.toString());
         // "Name: Igarashi; Addr: Kyoto"
  f(me); // x.toString() in f(x) will result in
         // the same string as above
}
with (Employment) {
  println(me.toString());
         // "Name: Igarashi; Affl: Kyoto U."
}
```

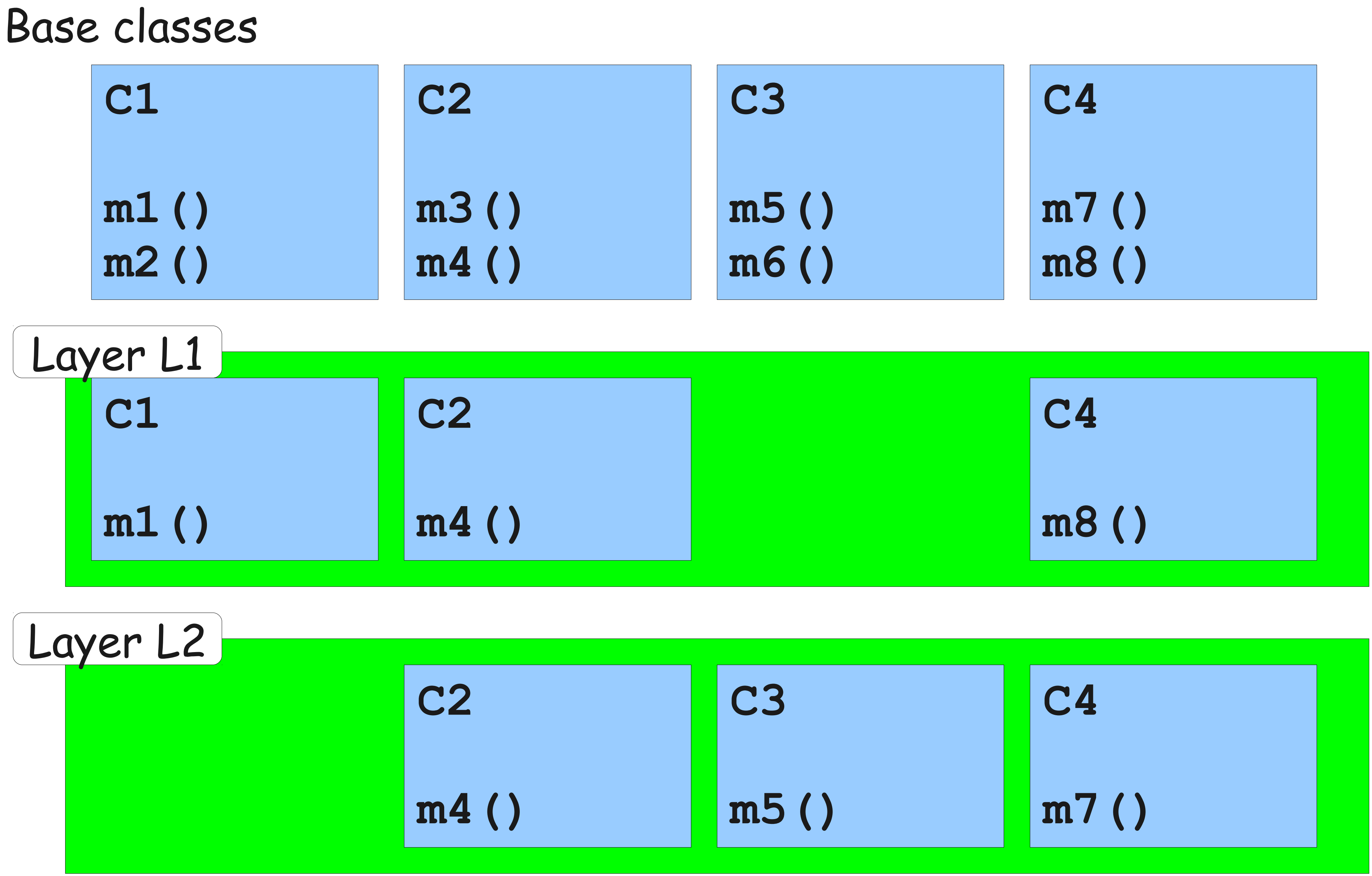
Activation of Contact

```
Person me = new Pe                                     yoto”,
                                                     ;

println(me.toString());    // “Name: Igarashi”

with (Contact) {
  println(me.toString());
          // “Name: Igarashi; Addr: Kyoto”
  f(me); // x.toString() in f(x) will result in
          // the same string as above
}
with (Employment) {
  println(me.toString());
          // “Name: Igarashi; Affl: Kyoto U.”
}
```

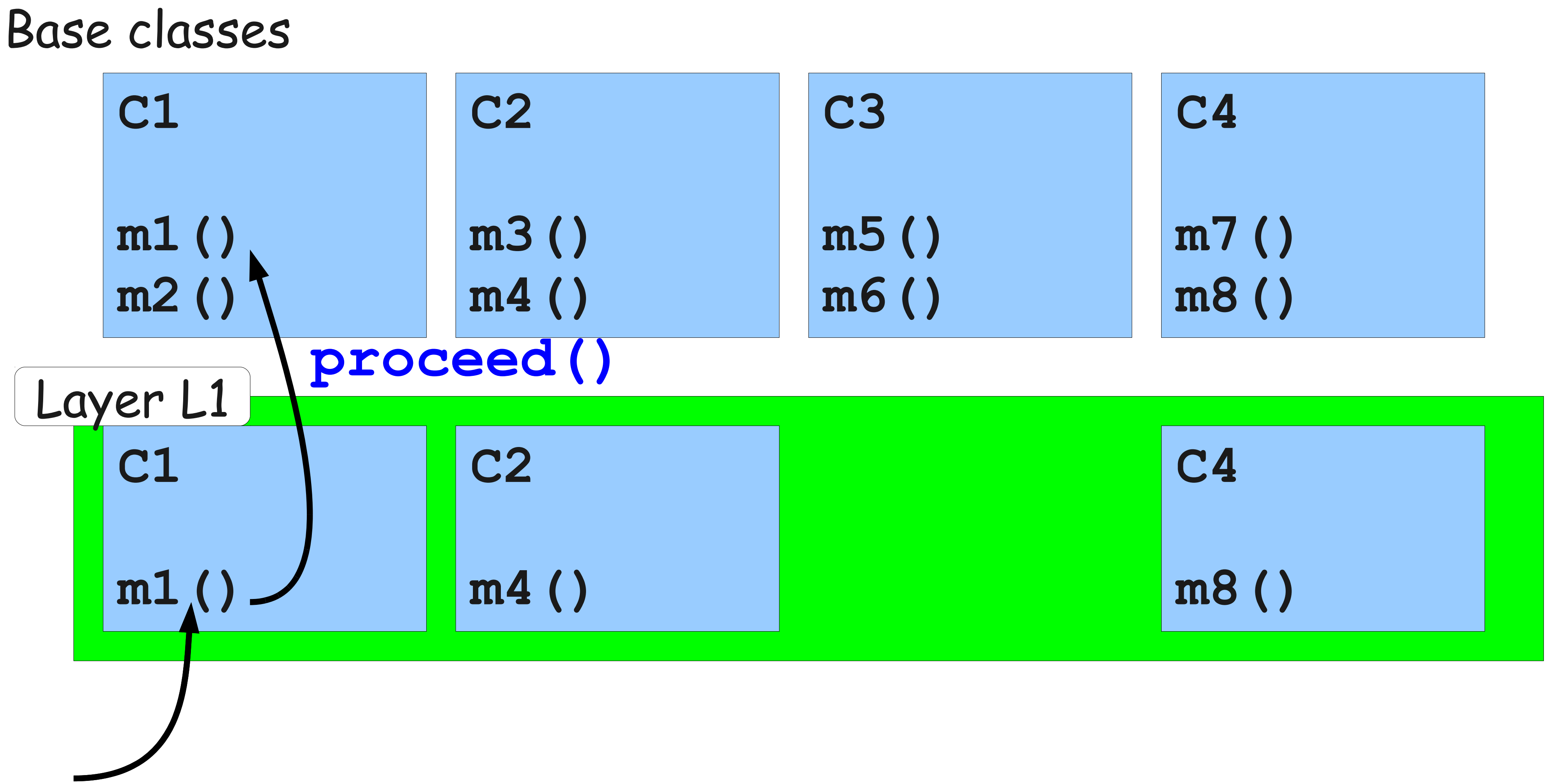
- Layer precedence depends on activation order

```
with (Contact) {
  with (Employment) {
    println(me.toString());
// "Name: Igarashi; Addr: Kyoto; Affl: Kyoto U."
} }

with (Employment) {
  with (Contact) {
    println(me.toString());
// "Name: Igarashi; Affl: Kyoto U.; Addr: Kyoto"
}  }
```

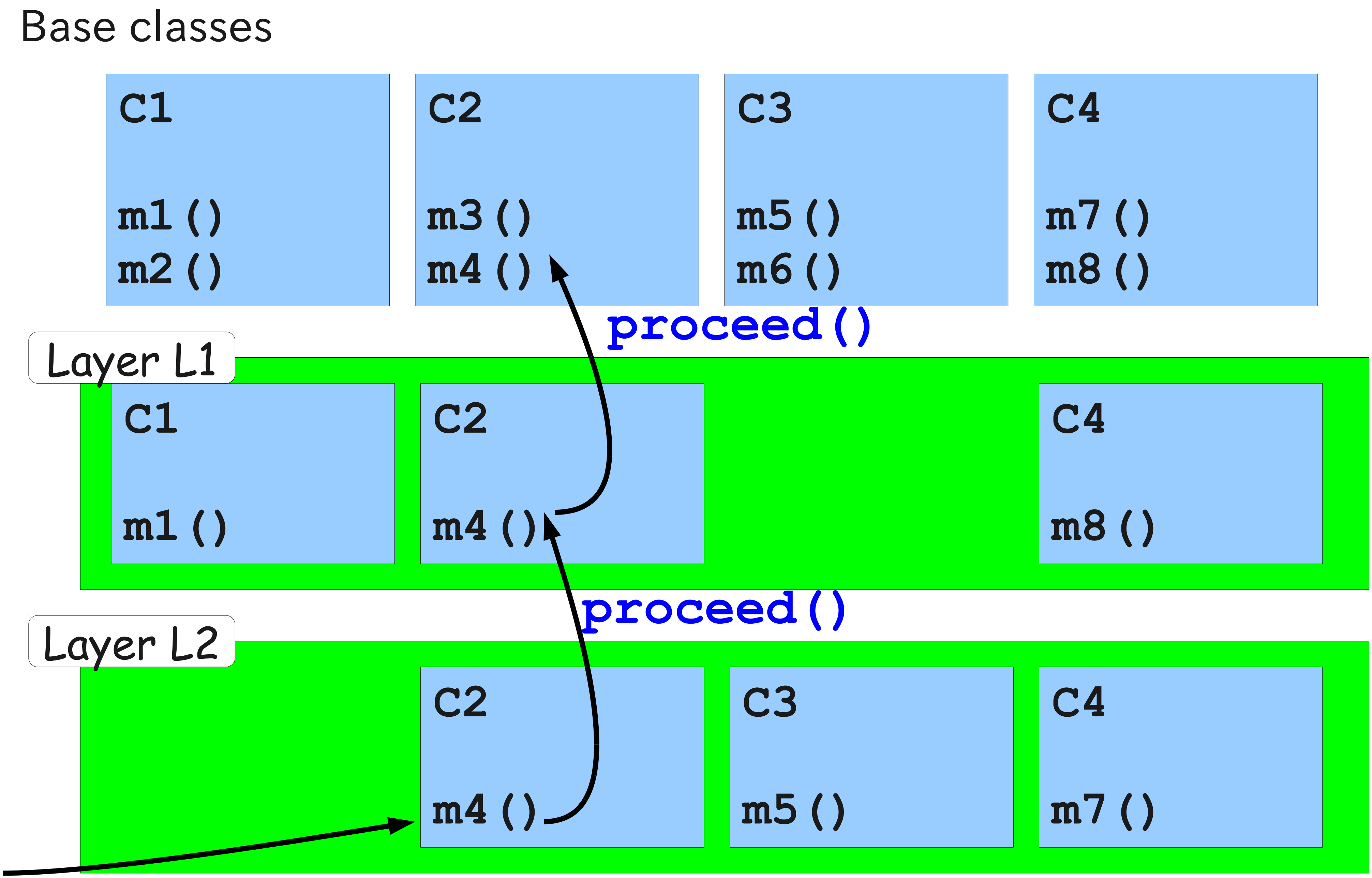
How a COP program is organized



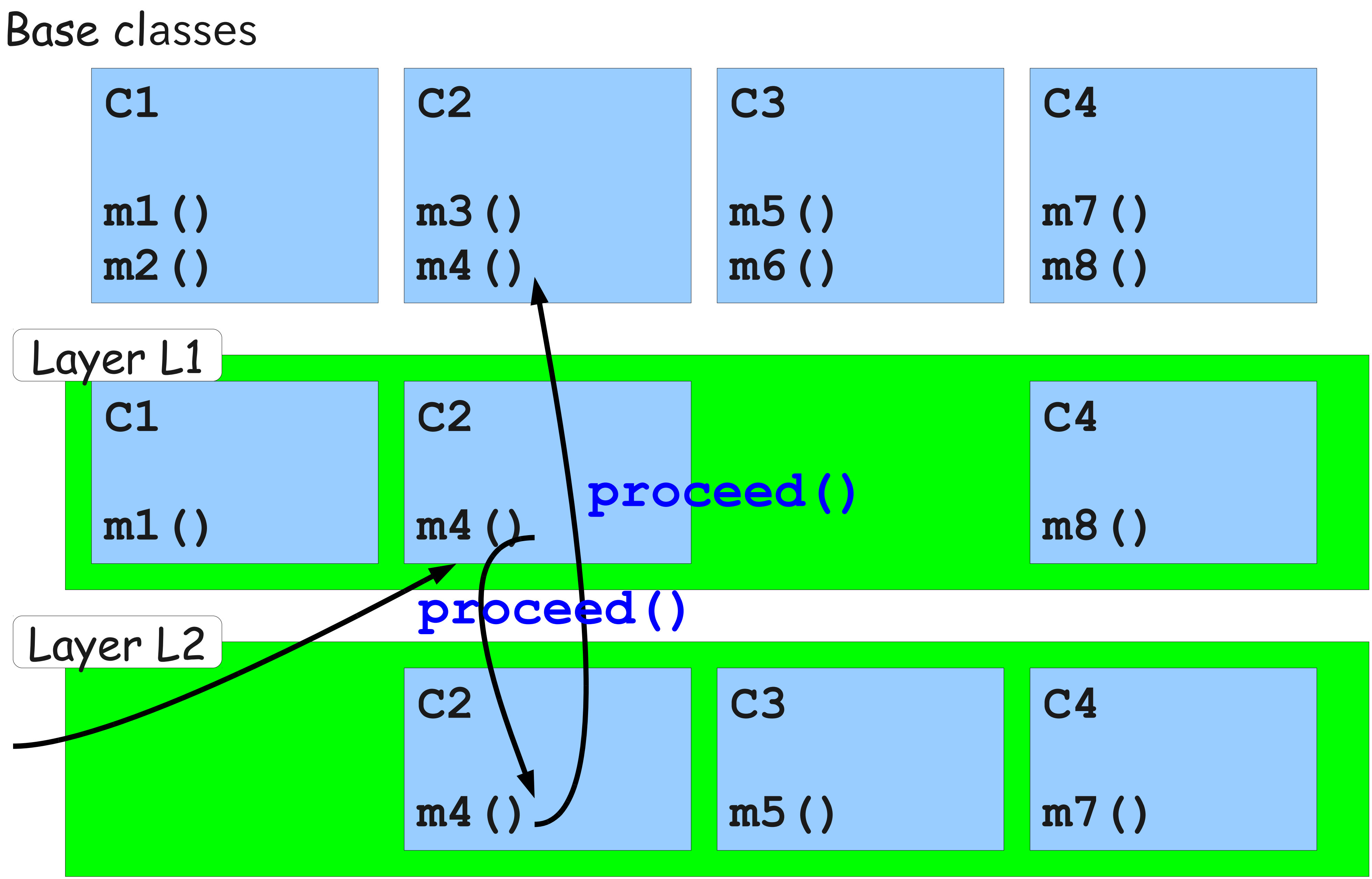
with (L1) { ... }


Base classes
C1
m1()
m2()
C2
m3()
m4()
C3
m5()
m6()
C4
m7()
m8()
proceed()
Layer L1
C1
m1()
C2
m4()
C4
m8()

with (L1) { with (L2) { ... } }



with (L2) { with (L1) { ... } }



Plan of the Talk

- COP Language Constructs
- ContextFJ
  - Syntax
  - Operational Semantics
  - Simple Type System
- Future Work

ContextFJ

- ContextFJ =

Featherweight Java [Igarashi,Pierce,Wadler'99]

+ Partial methods

+ proceed(), super()

+ with/without expressions

Syntax (1/2)

CL ::= class C < D { ~C ~f; ~M }	classes
M ::= C m(~C ~x){ return e; }	methods
e ::= x | e.f | e.m(~e) | new C(~e)	expressions
| with L e	layer activation
| without L e	layer deactivation
| proceed(~e)	proceed call
| super.m(~e)	super call

Syntax (2/2)

ContextFJ program: (*CT*, *PT*, e)

- Class table: *CT*(C) = CL
- Partial method table: *PT*(m,C,L) = M
- Main expression: e

Operational Semantics

FJ

- Lookup function:
  mbody(m,C) = ~x.e
- Reduction relation:
  e → e'

ContextFJ

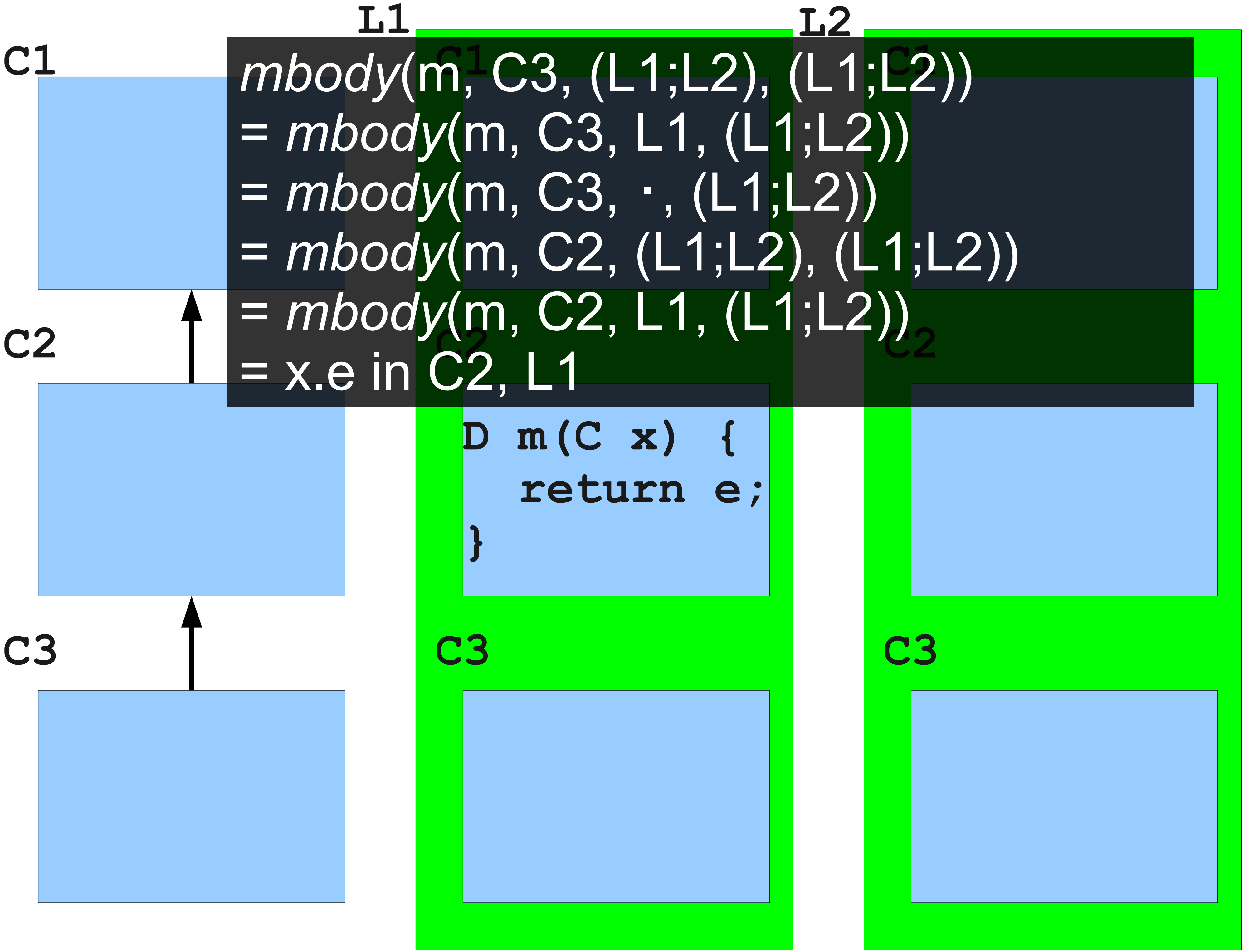
- Lookup function:
  mbody(m,C,~L1,~L2)
  = ~x.e in D,~L3
- Reduction relation:
  ~L ├ e → e'

Lookup function: *mbody*

mbody(m,C,~L1,~L2) = ~x.e in D, ~L3

- "Body of method m in C is e with params ~x"
- ~L2 is the list of activated layers
- C, ~L1 denote the currently focused position
- D, ~L3 denote where ~x.e is found

mbody(m, C3, (L1;L2), (L1;L2))
= *mbody*(m, C3, L1, (L1;L2))
= *mbody*(m, C3, ·, (L1;L2))
= *mbody*(m, C2, (L1;L2), (L1;L2))
= *mbody*(m, C2, L1, (L1;L2))
= x.e in C2, L1



$$\frac{\mathit{PT}(m,C,L0) = C0\ m(\sim C\ \sim x)\{\ \text{return}\ e;\ \}}{\mathit{mbody}(m,C,(\sim L1;L0),\ \sim L2) = \sim x.e\ \text{in}\ C,\ (\sim L1;\ L0)}$$

$$\frac{\begin{array}{c}\mathit{PT}(m,C,L0)\ \text{undefined} \\ \mathit{mbody}(m,C,\sim L1,\sim L2) = \sim x.e\ \text{in}\ D,\ \sim L3\end{array}}{\mathit{mbody}(m,C,(\sim L1;L0),\ \sim L2) = \sim x.e\ \text{in}\ D,\ \sim L3}$$

- *mbody*(toString, Person, ·, ·) =
  ().(“Name: ” + this.name) in Person, ·
- *mbody*(toString, Person, Contact, Contact) =
  ().(proceed() + “Addr: “+ this.addr)
  in Person, Contact

Reduction: ~L ├ e → e'

- "e reduces to e' in one step under activated layers ~L"
- e.g,
  - ├ new Person(...).toString()
    → "Name: " + new Person(...).name
  - Contact ├ new Person(...).toString()
    → proceed() + "Affl: " + new Person(...).addr
    - … actually, not quite correct! (Wait for a few slides!)

Reduction rule for layer activation

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

- Activated layer L always comes at the top
  - Even when it's already been activated
- *e.g.*,

├ with Contact (new Person(...).toString())
→ with Contact (
proceed() + "Affil: " + new Person(...).addr
)

Run-time expression to deal with proceed and super

e ::= … | new C(~e)<D,~L1,~L2>

- Essentially new C(~e)
- Annotation <D,~L1,~L2> remembers
  - where method lookup starts next time (D, ~L1)
  - what layers have been activated (~L2)
- Contact ├ new Person(...).toString()
  → new Person<Person, ･, Contact>().toString()
  + “Affl: “ + new Person(...).addr

Reduction Rules for Method Invocation

~L ├ new C(~v)<C,~L,~L>.m(~w) → e'

~L ├ new C(~v).m(~w) → e'

mbody(m, D, ~L1, ~L2) = ~x.e in E, (~L3,L)
class E < F

~L4 ├ new C(~v)<D,~L1,~L2>.m(~w) →
[new C(~v) / this,
~w / ~x,
new C(~v)<E, ~L3, ~L2>.m / proceed,
new C(~v)<F, ~L2, ~L2> / super] e

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(~v) / this,
~w / ~x,
new C(~v)<E, ~L3, ~L2>.m / proceed,
new C(~v)<F, ~L2, ~L2> / super] e

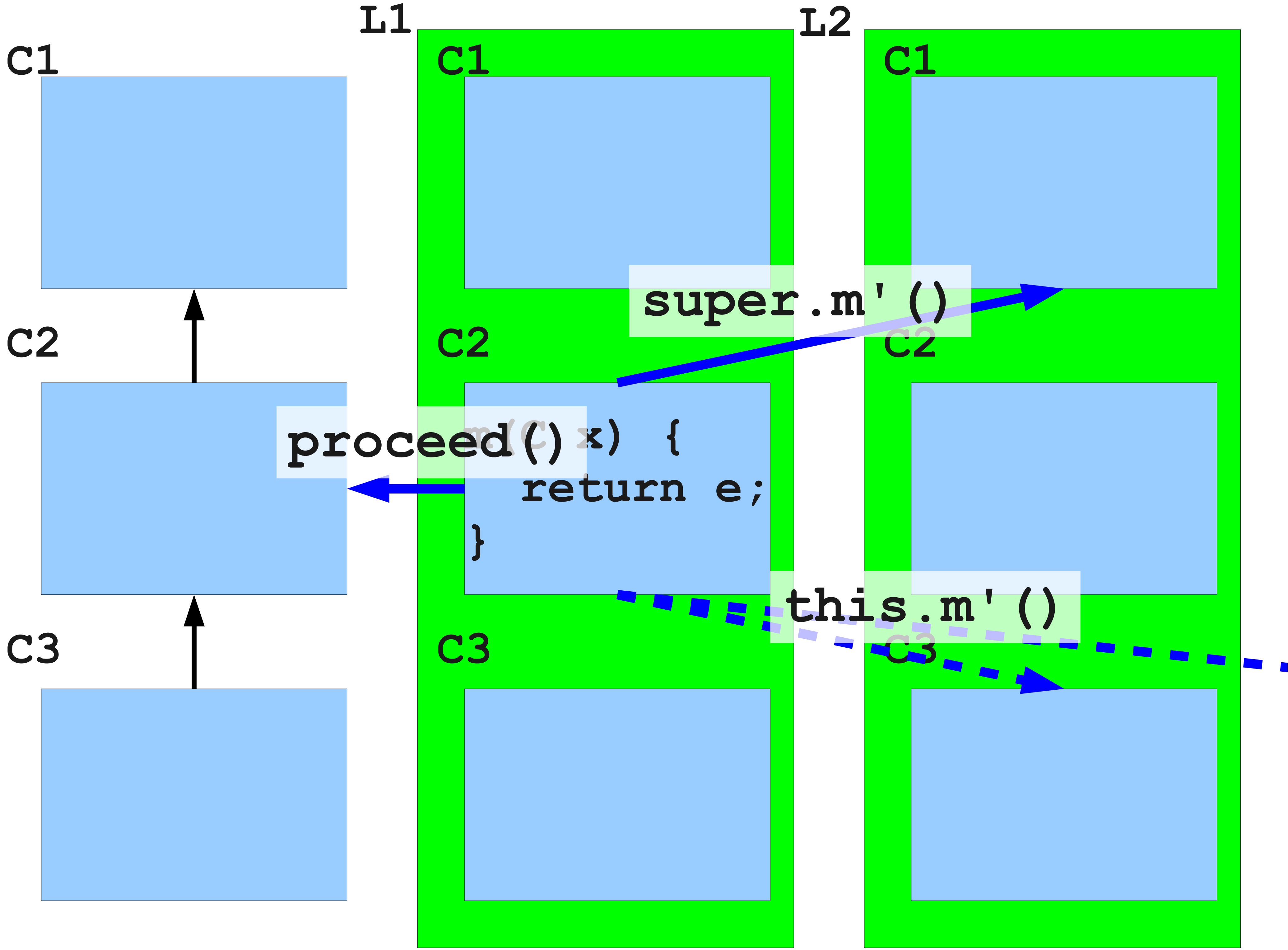
Reduction Rules

Self calls will be affected by with/without in e
but super/proceed calls won't

~L ├ new C(~v).m(~w) → e'

mbody(m, D, ~L1, ~L2) = ~x.e in E, (~L3,L)
class E < F

~L4 ├ new C(~v)<D,~L1,~L2>.m(~w) →
[new C(~v) / this,
~w / ~x,
new C(~v)<E, ~L3, ~L2>.m / proceed,
new C(~v)<F, ~L2, ~L2> / super] e



L1
L2
C1
C2
C3
C1
C2
C3
C1
C2
C3
super.m'()
proceed()
m(C x) {
return e;
}
this.m'()

Type System for ContextFJ

- Main problem: ensure proceed() to succeed
  - Non-trivial as layer configuration changes dynamically!
- A simple (but restrictive) answer: every partial method has to override one in a base class
  - rather than to introduce new behavior

⇒ Mostly the same type system as FJ!

  - Covariant return type overriding only for base methods
  - Type Soundness by Preservation + Progress

Summary

- Language Constructs for COP
  - Partial methods in layers
  - Layer (de)activation
- ContextFJ for direct account of COP programs
  - Operational semantics
  - Simple and sound type system

Summary

“We can talk about COP languages
at Starbucks, even without Mac! ”
- Hirschfeld

- Layer (de)activation
- ContextFJ for direct account of COP programs
  - Operational semantics
  - Simple and sound type system

Future work

- Sophisticated type system to allow partial methods to *introduce* new behavior
  - c.f., FOP type systems
- Formal accounts of advanced COP features
  - Stateful layers
  - First-class layers