A Type System for Dynamic Layer Composition

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Robert Hirschfeld
Hasso-Plattner-Institut Potsdam
hirschfeld@hpi.uni-potsdam.de

Hidehiko Masuhara
The University of Tokyo
masuhara@acm.org

Abstract

Dynamic layer composition is one of the key features in context-oriented programming (COP), an approach to improving modularity of behavioral variations that depend on the dynamic context of the execution environment. It allows a layer—a set of new or overriding methods that can belong to several classes—to be added to or removed from existing objects in a disciplined way. We develop a type system for dynamic layer composition, which may change the interfaces of objects at run time, based on a variant of ContextFJ, a core calculus for COP, and prove its soundness.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Language, Theory

Keywords Context-oriented programming, dynamic layer composition, type systems

1. Introduction

Context-oriented programming (COP) is an approach to improving modularity of behavioral variations that depend on dynamic properties of the execution environment [10]. In traditional programming paradigms, such behavioral variations tend to be scattered over several modules, and system architectures that support their dynamic composition are often complicated.

Many COP extensions including those designed on top of Java [2], Smalltalk [9], Common Lisp [6], and JavaScript [15], are based on object-oriented programming languages and introduce *layers* of *partial methods* for defining and organizing behavioral variations and *layer activation mechanisms* for layer selection and composition. A partial method in a layer is, in many cases, a method that can run before, after, or around a (partial) method with the same name and signature defined in a different layer or a class, but it can also be a new method that does not exist in a class yet. A layer groups related partial methods and can be (de)activated at run-time. It so contributes to the specific behavior of a set of objects in response to messages sent and received.

In this paper, we develop a simple type system for such dynamic composition of layers. Although what the type system guarantees is just the absence of "no such method" errors (including the failure of `proceed` calls in around-type partial methods), the existence of partial methods that introduce new behavior to existing classes makes the problem interesting, because layer (de)activation changes the interface of objects at run time. A key idea of our development is the introduction of an explicitly declared inter-layer dependency relation, which plays a role similar to required methods in type systems for mixins [4, 8, 14]. With the help of this dependency, the type system will estimate layers activated at each program point and use this estimation to look up the signature of an invoked method. We formalize the type system for a variant of ContextFJ [11], a COP extension of Featherweight Java [12], and prove its soundness. ContextFJ supports (around-type) partial methods, block-structured dynamic activation of layers, and `proceed` and `super` calls.

We also discuss a few variations of layer activation mechanisms (although we formalize only one of them). Because it turns out that our type system seems *not* to work for the layer activation mechanism used in many COP languages, we prove type soundness for "a variant" of ContextFJ. Nevertheless, we believe that this work is of some value as a clarification about how our simple typing scheme interacts with the design of layer activation mechanisms.

This paper is a continuation of our work [11], in which ContextFJ is formalized. There, an even simpler type system for ContextFJ is discussed, but it is very restrictive because it prohibits layers from adding new methods to existing classes.

The rest of the paper is organized as follows. We first start with reviewing the language mechanisms for COP in Section 2. Section 3 reviews the syntax and operational semantics of ContextFJ and Section 4 defines the type system and proves its soundness. We discuss related and future work in Section 5.

2. Language Constructs for COP

We briefly overview basic constructs along with their usage. Our example is a simplified telecom simulation[1] in which customers make, accept, and terminate phone calls.

2.1 The Base Layer

The base layer consists of standard Java classes and methods, which is always active. The telecom example has `Customer` and `Connection` to represent customers and phone calls between customers, respectively.

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

The following method demonstrates a usage of those classes.

```
Connection simulate() {
```

[1] This simulation is based on an example distributed along with the AspectJ compiler [21].

Copyright is held by the author/owner(s).
FOOL2012 October 22, 2012, Tucson, AZ.
ACM .

```
  Customer atsushi = ..., hidehiko = ...;
  Connection c =
    new Connection(atsushi,hidehiko);
                        // Atsushi calls Hidehiko
  c.complete();         // Hidehiko accepts
  c.drop();             // Hidehiko hangs up
  return c;
}
```

2.2 Layers and Partial Method Definitions

A layer is a collection of methods and fields[2] that are specific to a certain context. Syntactically, they are written as Java classes enclosed by the `layer` construct[3]. Below, the `Timing` layer defines a feature that measures the duration of phone calls. (A COP layer is usually used to group partial methods of more than one class, but as an illustrating example for ContextFJ, partial methods of one class will suffice.)

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

When a layer is active (as explained in Section 2.3), the methods defined in that layer, so called *partial methods*, override those in the base layer. In the above example, `complete` and `drop` are partial methods.

Unlike other COP languages, we also allow a layer to define a method that does not exist in the base layer, which we call a *layer-introduced base method*. In the above example, `getTime` is such a layer-introduced base method.

`proceed(...)` is similar to `super` as it delegates behavior to overridden methods. Whereas `super` changes the starting point of the method lookup to the superclass of the class the (partial) method was defined in, `proceed(...)` will try first to find the next partial or base-level definition of the same method in the same (current) class. If `proceed(...)` cannot find such a partial method in the current receiver class or the active layers associated with it, lookup continues in the superclass of the current lookup class.

Existence of layer-introduced base methods and `proceed` make a type system, which statically guarantees success of `proceed`, complicated. We will show this in Section 4.

2.3 Layer Activation

Many COP languages offer `with` for layer activation and `without` for layer deactivation. In this work, we consider the `ensure` construct, which is similar to but different from `with`, to activate a layer (their differences are described in Section 2.4) but no construct for deactivation.

When we simply call `simulate()`, it merely executes the methods in the base layer because *no layers are activated*.

By using the `ensure` construct, we can activate a layer during the evaluation of its body statement if not already done so. The following example simulates a phone call with the `Timing` layer activated.

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

When simulating calls with the `Timing` layer activated, `complete` and `drop`, the partial methods defined in `Timing`, run instead of the ones in the base layer.

Note that activation of a layer also allows to call layer-introduced base methods. The above example calls `getTime` on the returned `Connection` object inside the `ensure` body, which is not possible without activating `Timing`.

As in most COP language extensions and also in ours, layer compositions are effective for the *dynamic extent* of the execution of the code block enclosed by their corresponding `ensure` statement[4]. So, layers form a stack and are (de)activated in the FIFO manner.

2.4 Order and Dependency between Layers

When we activate a layer while other layers are active, the partial methods in the last-activated layer override those in the earlier-activated layers. The following example explains this order of partial methods. The `Billing` layer below adds a feature that calculates and charges the cost of a phone call when the call ends.

```
layer Billing requires Timing {
  class Connection {
    void charge() {
      int cost =  ...getTime()...;
      ...charge the cost on the caller... }
    void drop() { proceed(); charge(); }
  }
}
```

When we activate the `Timing` and `Billing` layers in this order, the partial method `drop` in `Billing` overrides the one in `Timing` because `Billing` is the most recently activated layer.

```
ensure Timing {
  ensure Billing {
    simulate();
  }
}
```

Since both partial methods have `proceed`, a call to `drop` in `simulate` will stop the timer, perform the base layer's behavior, and then calculate the cost of the call based on the duration of the call obtained through `getTime`.

Note that `Billing` depends on `Timing` as the `charge` method in `Billing` calls `getTime`, which is a layer-introduced base method by `Timing`. We declare this dependency by using the `requires` modifier in the layer declaration. In other words, the following statement, which activates `Billing` without activating `Timing` is incorrect and must be rejected statically. Our type system will detect such an erroneous layer activation.

```
ensure Billing { simulate(); } // incorrect
```

One might wonder why `ensure Billing` activates `Timing` at the same time because it is apparent that `Billing` requires `Timing`. Actually, one layer can depend on more than one layer, in which case it is not always clear in which order they should be activated.

2.5 Comparing `ensure` and `with`

As we mentioned above, many COP languages have a layer activation construct called `with`, which will make sure that the activated layer is always the first layer for which a method is searched. The

[2] The formal model omits fields defined in layers for simplicity. However, as far as type safety is concerned, supporting fields does not cause significant problems.

[3] It is also possible to place those additional definitions in each class to be added, which is the so-called layer-in-class style.

[4] Variants of COP languages allow to manage layer compositions on a per-instance basis [9, 13], which is left as future work in the paper.

difference becomes clear when the same layer is to be activated for the second time—the second activation will move the designated layer to the top of the stack of layers. For example,

```
with Timing {
  with Billing {
    with Timing { simulate(); }
  }
}
```

will invoke the partial method `drop` in `Timing` first[5]. Also, the effect of the outer `with Timing` is disabled until the body of the inner activation finishes. So, `proceed` from `Billing` calls a method in a base class. On the other hand, `ensure` will just make sure the existence of `Timing` without changing the order of already activated layers. So, the code where `with` is replaced by `ensure` is the same as the code without the inner activation of `Timing` (namely, the second code snippet in the last subsection).

The rearrangement of layers caused by `with`, however, destroys the layer ordering in which inter-layer dependency is respected. For example, there is no layer below `Billing`, which requires `Timing`, while the inner `with Timing` is executed. Similarly to double `with`, `without`, which deactivates a designated layer, destroys dependency-respecting layer ordering. Thus, for simplicity, we consider only `ensure` in this paper and leave a sound type system for `with` and `without` for future work. We propose `ensure` mainly to show that, by adopting `ensure`, simple dependency declaration is enough to design a sound type system. Comparisons of `with` and `ensure` from programmers' point of view are interesting but left for future work.

3. Syntax and Semantics of ContextFJ

In this section, we give the syntax and operational semantics of ContextFJ, which is an extension of Featherweight Java (FJ) [12] with (around-type) partial methods, `ensure` for layer activation, `proceed`, and `super`. As already mentioned, we omit fields defined in layers for simplicity. Thus, a layer is a set of partial methods; just as a set of classes is modeled as a class table—a mapping from class names to class definitions—in FJ, a set of layers will be modeled as a mapping from layer and class names to method definitions. The present version of ContextFJ replaces `with` and `without` for layer (de)activation found in the original version [11] with `ensure`. Except for the difference in the layer activation mechanism, the definitions are the same as in the original version.

3.1 Syntax

Let metavariables `C`, `D`, and `E` range over class names; `L` over layer names; `f` and `g` over field names; `m` over method names; and `x` and `y` over variables, which contain a special variable `this`. The abstract syntax of ContextFJ is given as follows:

CL	::=	`class C` $\lhd$ `C { C̄ f̄; K M̄ }`	(*classes*)
K	::=	`C(C̄ f̄){ super(f̄); this.f̄ = f̄; }`	(*constructors*)
M	::=	`C m(C̄ x̄){ return e; }`	(*methods*)
e, d	::=	`x` | `e.f` | `e.m(ē)` | `new C(ē)` | `ensure L e` | `proceed(ē)` | `super.m(ē)` | `new C(v̄)<C,L̄,L̄>.m(ē)`	(*expressions*)
v, w	::=	`new C(v̄)`	(*values*)

[5] This code is admittedly artificial but in general, it is not unusual that one layer is to be activated twice in the dynamic extent of a method on code block execution. For example, it can happen that one method activates `Timing` and `Billing` in this order and then calls another method, which activates `Timing` (without knowing it has been activated already).

Following FJ, we use overlines to denote sequences: so, $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$ and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. Layers in a sequence are separated by semicolons. The empty sequence is denoted by $\bullet$. We also abbreviate pairs of sequences, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{C}}\ \overline{\texttt{f}}$;" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1; \ldots \texttt{C}_n\ \texttt{f}_n;$" and "`this.`$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;" for "`this.`$\texttt{f}_1$=$\texttt{f}_1$;...;`this.`$\texttt{f}_n$=$\texttt{f}_n$;". We use commas and semicolons for concatenations. Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

A class definition `CL` consists of its name, its superclass name, field declarations $\overline{\texttt{C}}\ \overline{\texttt{f}}$, a constructor `K`, and method definitions $\overline{\texttt{M}}$. A constructor `K` is a trivial one that takes initial values of all fields and sets them to the corresponding fields. Unlike the examples in the last section, we do not provide syntax for layers; partial methods are registered in a partial method table, explained below. A method `M` takes $\overline{\texttt{x}}$ as arguments and returns the value of expression `e`. As ContextFJ is a functional calculus like FJ, the method body consists of a single return statement and all constructs including `ensure` return values. An expression `e` can be a variable, field access, method invocation, object instantiation, layer activation/deactivation, `proceed`/`super` call, or a special expression `new C(`$\overline{\texttt{v}}$`)<C,`$\overline{\texttt{L}}$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)`, which will be explained shortly. A value is an object of the form `new C(`$\overline{\texttt{v}}$`)`.

The expression `new C(`$\overline{\texttt{v}}$`)<D,`$\overline{\texttt{L}}'$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)`, where $\overline{\texttt{L}}'$ is assumed to be a prefix of $\overline{\texttt{L}}$, is a special run-time expression and not supposed to appear in classes. It basically means that `m` is going to be invoked on `new C(`$\overline{\texttt{v}}$`)`. The annotation `<D,`$\overline{\texttt{L}}'$`,`$\overline{\texttt{L}}$`>`, which is used to model `super` and `proceed`, indicates where method lookup should start. More concretely, the triple `<D,`$(\texttt{L}_1; \cdots; \texttt{L}_i)$`,`$(\texttt{L}_1; \cdots; \texttt{L}_n)$`>` $(i \leq n)$ means that the search for the method definition will start from class `D` of layer $\texttt{L}_i$. So, for example, the usual method invocation `new C(`$\overline{\texttt{v}}$`).m(`$\overline{\texttt{e}}$`)` (without annotation) is semantically equivalent to `new C(`$\overline{\texttt{v}}$`)<C,`$\overline{\texttt{L}}$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)`, where $\overline{\texttt{L}}$ is the active layers when this invocation is to be executed. This triple also plays the role of a "cursor" in the method lookup procedure and move across layers and base classes until the method definition is found. Figure 1 illustrates how a cursor proceeds. Notice that the third element is needed when the method is not found in `D` in any layer including the base: the search continues to layer $\texttt{L}_n$ of `D`'s direct superclass.

With the help of this form, we can give a semantics of `super` and `proceed` by simple substitution-based reduction. For example, consider method invocation `new C().m(`$\overline{\texttt{v}}$`)`. As in FJ, this expression reduces to the method body where parameters and `this` are replaced with arguments $\overline{\texttt{v}}$ and the receiver `new C()`, respectively. Now, what happens to `super` in the method body? It cannot be replaced with the receiver `new C()` since it would confuse `this` and `super`. Method lookup for `super` is different from usual (virtual) method lookup in that it has to start from the direct superclass of *the class in which* `super` *appears*. So, if the method body containing `super.n()` is found in class `D`, then the search for `n` has to start from the direct superclass of `D`. To express this fact, we replace `super` with `new C()<E,...>` where `E` is the direct superclass of `D`. We can deal with `proceed` similarly. Suppose the method body is found in layer $\texttt{L}_i$ in `D`. Then, `proceed(`$\overline{\texttt{e}}$`)` is replaced with `new C()<D,`$(\texttt{L}_1; \cdots; \texttt{L}_{i-1})$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)`, where $\texttt{L}_1; \cdots; \texttt{L}_{i-1}$ are layers activated before $\texttt{L}_i$.

A ContextFJ program $(CT, PT, \texttt{e})$ consists of a class table CT, which maps a class name to a class definition, a partial method table PT, which maps a triple `C`, `L`, and `m` of class, layer, and method names to a method definition, and an expression, which corresponds to the body of the main method. In what follows, we

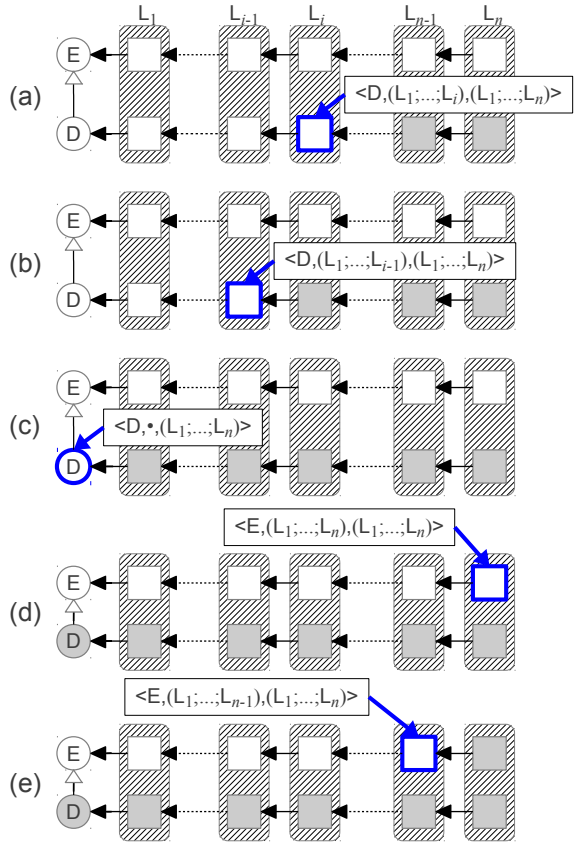

Figure 1. Method lookup in ContextFJ. A circle represents a base class and an arrow with a white head represents subclassing. A shaded round rectangle represent a layer, which contains sets of partial methods (represented by boxes) for base classes. Layers $\mathtt{L}_1$ upto $\mathtt{L}_n$ have been activated in this order. The cursor, represented by an arrow pointing to a thick box, goes left first (subfigures (a)–(c)), goes one level up (to superclass `E`), and restarts lookup from the most recently activated layer $\mathtt{L}_n$ towards the left (subfigures (d)–(e)).

assume CT and PT to be fixed and satisfy the following sanity conditions:

1. $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for any $\mathtt{C} \in dom(CT)$.
2. $\mathtt{Object} \notin dom(CT)$.
3. For every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$;
4. There are no cycles in the transitive closure of the `extends` clauses.
5. $PT(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \mathtt{\ldots\ m(\ldots)\{\ldots\}}$ for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in dom(PT)$.

We introduce dependency between layers expressed by `requires` clauses in the next section, where a type system is defined.

Lookup functions. As in FJ, we define a few auxiliary functions to look up field and method definitions. They are defined by the rules in Figure 2. The function $fields(\mathtt{C})$ returns a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of pairs of a field name and its type by collecting all field declarations

$$\boxed{fields(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

$$fields(\mathtt{Object}) = \bullet \qquad \text{(F-Object)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ \overline{C}\ \overline{f};\ \ldots\ \}} \qquad fields(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{fields(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}} \qquad \text{(F-Class)}$$

$$\boxed{mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}}''}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ \ldots\ C_0\ m(\overline{C}\ \overline{x})\{\ return\ e;\ \}\ \ldots\}}}{mbody(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ C}, \bullet} \qquad \text{(MB-Class)}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{C}_0\ \ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}}{mbody(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}}'; \mathtt{L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ C}, (\overline{\mathtt{L}}'; \mathtt{L}_0)} \qquad \text{(MB-Layer)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ \ldots\ \overline{M}\ \}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad mbody(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ E}, \overline{\mathtt{L}}'}{mbody(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ E}, \overline{\mathtt{L}}'} \qquad \text{(MB-Super)}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)\ \text{undefined} \qquad mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}}''}{mbody(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}}'; \mathtt{L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}}''} \qquad \text{(MB-NextLayer)}$$

Figure 2. ContextFJ: Lookup functions.

from `C` and its superclasses. The function $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)$ returns the parameters and body $\overline{\mathtt{x}}.\mathtt{e}$ of method `m` in class `C` when the search starts from $\overline{\mathtt{L}}_1$; the other layer names $\overline{\mathtt{L}}_2$ keep track of the layers that are activated when the search initially started. It also returns the information on where the method has been found—the information will be used in reduction rules to deal with `proceed` and `super`. As we mentioned already, the method definition is searched for in class `C` in all activated layers and the base definition and, if there is none, then the search continues to `C`'s superclass. By reading the rules in a bottom-up manner, we can read off a recursive search procedure. The rule MB-Class means that `m` is found in the base class definition `C` (notice the third argument is $\bullet$) and the rule MB-Layer that `m` is found in layer $\mathtt{L}_0$. The rule MB-Super, which deals with the situation where `m` is not found in a base class (expressed by the condition $\mathtt{m} \notin \overline{\mathtt{M}}$), motivates the fourth argument of *mbody*. The search goes on to `C`'s superclass `D` and has to take all activated layers into account; so, $\overline{\mathtt{L}}$ is copied to the third argument in the premise. The rule MB-NextLayer means that, if `C` of $\mathtt{L}_0$ does not have `m`, then the search goes on to the next layer (in $\overline{\mathtt{L}}'$) leaving the class name unchanged.

3.2 Operational Semantics

The operational semantics of ContextFJ is given by a reduction relation of the form $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to $\mathtt{e}'$ under the activated layers $\overline{\mathtt{L}}$". Here, $\overline{\mathtt{L}}$ do not contain duplicate names, as we noted earlier. The main rules are shown in Figure 3.

The first four rules are the main computation rules for field access and method invocation. The rule R-Field for field access is straightforward: *fields* tells which argument to `new C(..)` corresponds to $\mathtt{f}_i$. The next three rules are for method invocation. The rule R-InvK is for method invocation where the cursor of the method lookup procedure has not been "initialized"; the cursor is set to be at the receiver's class and the currently activated layers. In the rule R-InvkB, the receiver is `new C(`$\overline{\mathtt{v}}$`)` and `<C'`$,\overline{\mathtt{L}}'',\overline{\mathtt{L}}'$`>` is the location of the cursor. When the method body is found in the base-layer class $\mathtt{C}''$ (denoted by "`in C''`$, \bullet$"), the whole expression

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).f}_i \longrightarrow \mathtt{v}_i} \quad \text{(R-FIELD)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C,\overline{L},\overline{L}\rangle.m(\overline{w})} \longrightarrow \mathtt{e'}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v}).m(\overline{w})} \longrightarrow \mathtt{e'}} \quad \text{(R-INVK)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C'}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}'', \bullet \qquad \mathtt{class\ C''} \lhd \mathtt{D\{...\}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C',\overline{L}'',\overline{L}'\rangle.m(\overline{w})} \longrightarrow \left[\begin{array}{ll} \mathtt{new\ C(\overline{v})} & /\mathtt{this}, \\ \mathtt{\overline{w}} & /\mathtt{\overline{x}}, \\ \mathtt{new\ C(\overline{v})\langle D,\overline{L}',\overline{L}'\rangle} & /\mathtt{super} \end{array}\right] \mathtt{e}_0} \quad \text{(R-INVKB)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C'}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}'', (\overline{\mathtt{L}}'''; \mathtt{L}_0) \quad \mathtt{class\ C''} \lhd \mathtt{D\{...\}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C',\overline{L}'',\overline{L}'\rangle.m(\overline{w})} \longrightarrow \left[\begin{array}{ll} \mathtt{new\ C(\overline{v})} & /\mathtt{this}, \\ \mathtt{\overline{w}} & /\mathtt{\overline{x}}, \\ \mathtt{new\ C(\overline{v})\langle C'',\overline{L}''',\overline{L}'\rangle.m} & /\mathtt{proceed}, \\ \mathtt{new\ C(\overline{v})\langle D,\overline{L}',\overline{L}'\rangle} & /\mathtt{super} \end{array}\right] \mathtt{e}_0} \quad \text{(R-INVKP)}$$

$$\frac{\mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e'}}{\overline{\mathtt{L}} \vdash \mathtt{ensure\ L\ e} \longrightarrow \mathtt{ensure\ L\ e'}} \quad \text{(RC-ENSURE)}$$

$$\frac{}{\overline{\mathtt{L}} \vdash \mathtt{ensure\ L\ v} \longrightarrow \mathtt{v}} \quad \text{(R-ENSUREVAL)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0\mathtt{'}}{\overline{\mathtt{L}} \vdash \mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}_0\mathtt{'.f}} \quad \text{(RC-FIELD)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0\mathtt{'}}{\overline{\mathtt{L}} \vdash \mathtt{e}_0\mathtt{.m(\overline{e})} \longrightarrow \mathtt{e}_0\mathtt{'.m(\overline{e})}} \quad \text{(RC-INVKRECV)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i\mathtt{'}}{\overline{\mathtt{L}} \vdash \mathtt{e}_0\mathtt{.m(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{e}_0\mathtt{.m(..,e}_i\mathtt{',..)}} \quad \text{(RC-INVKARG)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i\mathtt{'}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{new\ C(..,e}_i\mathtt{',..)}} \quad \text{(RC-NEW)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i\mathtt{'}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(\overline{v})\langle C',\overline{L}',\overline{L}''\rangle.m(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{new\ C(\overline{v})\langle C',\overline{L}',\overline{L}''\rangle.m(..,e}_i\mathtt{',..)} \quad \text{(RC-INVKAARG)}$$

Figure 3. ContextFJ: Reduction rules.

reduces to the method body where the formal parameters $\overline{\mathtt{x}}$ and `this` are replaced by the actual arguments $\overline{\mathtt{w}}$ and the receiver, respectively. Furthermore, `super` is replaced by the receiver with the cursor pointing to the superclass of $\mathtt{C}''$. The rule R-INVKP, which is similar to R-INVKB, deals with the case where the method body is found in layer $\mathtt{L}_0$ in class $\mathtt{C}''$. In this case, `proceed` in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layer $\overline{\mathtt{L}}'''$ (dropping $\mathtt{L}_0$). Since the meaning of the annotated invocation is not affected by the layers in the context (note that $\overline{\mathtt{L}}$ are not significant in these rules), the substitution for `super` and `proceed` also means that their meaning is the same throughout a given method body, even when they appear inside `ensure`. Note that, unlike FJ, reduction in ContextFJ is call-by-value, requiring receivers and arguments to be values. This evaluation strategy reflects the fact that arguments should be evaluated under the caller-side context.

The following rules are related to context manipulation. The rule RC-ENSURE means that `e` in `ensure L e` should be executing by activating `L`. The auxiliary function $\mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}})$, defined by:

$$\begin{array}{lcll} \mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}}) & = & \overline{\mathtt{L}} & (\text{if } \mathtt{L} \in \overline{\mathtt{L}}) \\ \mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}}) & = & \overline{\mathtt{L}}\mathtt{;L} & (\text{otherwise}) \end{array}$$

adds `L` to the end of $\overline{\mathtt{L}}$ if `L` is not in $\overline{\mathtt{L}}$ (or returns $\overline{\mathtt{L}}$ otherwise). It only ensures the existence of `L` without changing the order of already activated layers. As we have already discussed, this is in contrast to `with` statements, which other COP languages usually provide. A `with` statement activates a layer but, if the layer is already activated in the middle of the layer stack, it will be moved to the top, changing the order of activated layers.

The next rule R-ENSUREVAL means that, once the evaluation of the body of `ensure` is finished, it returns the value of the body.

There are other trivial congruence rules to allow subexpressions to reduce. Note that ContextFJ reduction is call by value, but the order of reduction of subexpressions is unspecified.

4. Type System

In this section, we give a type system for ContextFJ. As usual, the role of a type system is to guarantee type soundness, namely, to prevent statically field-not-found and method-not-found errors from happening at run time. In ContextFJ, it also means that a type system should ensure that every `proceed()` or `super()` call succeeds.

A key idea in this type system is to keep track of an (under-) approximation of layers activated at each program point. Such approximation gives information on what methods are made available by layer activation (in addition to those defined in the base layer). Roughly speaking, a type judgment for an expression is of the form $\Lambda; \Gamma \vdash \mathtt{e} : \mathtt{C}$, where Γ is a type environment, which records types of variables, and a set Λ of layers that are assumed to be activated when `e` is evaluated. This approximated layer information Λ will be used to typecheck a method invocation expression. For example, the call to `getTime` in partial method `charge` in `Billing` in Section 2 is valid because layer `Timing` provides `getTime`. It could be represented by a type judgment

$$\{\mathtt{Timing}\}; \mathtt{this} : \mathtt{Connection} \vdash \mathtt{this.getTime()} : \mathtt{int}$$

where $\bullet$ is the empty type environment. On the other hand,

$$\emptyset; \mathtt{this} : \mathtt{Connection} \vdash \mathtt{this.getTime()} : \mathtt{int}$$

is not a valid type judgment because no layer is assumed (represented by $\emptyset$) and the base definition of `Connection` does not give method `getTime`.

As we have already discussed, a program will be typechecked with information on dependency between layers. Let $\mathcal{R}$ be a binary relation on layer names; $(\mathtt{L}_1, \mathtt{L}_2) \in \mathcal{R}$ intuitively means that layer $\mathtt{L}_1$ `requires` $\mathtt{L}_2$, that is, when $\mathtt{L}_1$ is to be activated, $\mathtt{L}_2$ has to have been activated already. The dependency relation $\mathcal{R}$ is used in typechecking the entry point of each partial method: when a partial method in layer `L` is typechecked, all the layers related to `L` by $\mathcal{R}$ are assumed in a type judgment for the body of the partial method. In what follows, we assume a fixed dependency relation and write $\mathtt{L\ req}\ \Lambda$, read "layer `L` requires layers Λ", when $\Lambda = \{\mathtt{L}' \mid (\mathtt{L}, \mathtt{L}') \in \mathcal{R}\}$.

The type system also has to guarantee that the layers assumed in type judgments are really activated at run time. It is guaranteed by the typing rule for `ensure L` below:

$$\frac{\mathtt{L\ req}\ \Lambda' \qquad \Lambda' \subseteq \Lambda \qquad \Lambda \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0}{\Lambda; \Gamma \vdash \mathtt{ensure\ L\ e}_0 : \mathtt{C}_0}$$

First, the third premise means that layer L (which is to be activated) can be assumed in addition to the already activated layers Λ when typechecking the body $\mathtt{e}_0$ of `ensure`. Second, the first two premises guarantee that layers Λ' that L requires have been already activated (that is, are included in Λ) when L is activated. Since such "activated-before" relation is preserved (remember that layers are always manipulated in the FIFO manner) during program execution, all calls (including `proceed`) from L will succeed.

To summarize key technical points, (1) a type judgment is augmented with approximation of activated layers; (2) the method type lookup function takes activated layers into account; and (3) the typing rule for `ensure` guarantees the assumed layers are really activated at run time. Keeping these in mind, we proceed to a formal type system.

4.1 Subtyping

The subtyping relation `C <: D`, which is the same as that in FJ, is defined as the reflexive and transitive closure of the `extends` clauses.

$$\frac{}{\mathtt{C} <: \mathtt{C}} \quad \text{(S-REFL)}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}} \quad \text{(S-TRANS)}$$

$$\frac{\texttt{class C} \lhd \mathtt{D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}} \quad \text{(S-EXTENDS)}$$

4.2 Method type lookup

We define another auxiliary function to look up the type of a method. The function $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2)$, defined by the rules below, takes a method name m, a class name C, and two sets Λ_1 and Λ_2 of layer names and returns a pair, written $\overline{\mathtt{C}} \to \mathtt{C}_0$, of argument types $\overline{\mathtt{C}}$ and a return type $\mathtt{C}_0$. The sets Λ_1 and Λ_2 stand for statically known activated layers, in which m is looked for. The first set is used to look up m in C, whereas the second is used when m is not found in C and the search continues to C's superclass. Although these two sets are the same in most uses of *mtype* (in fact, we write $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda)$ for $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda, \Lambda)$), we need to distinguish them for typing `proceed`, because `proceed` cannot proceed to where it is executed, but may proceed to a method of the same name in a superclass in the same layer.

$$\frac{\texttt{class C} \lhd \texttt{D}\ \{\ldots\ \mathtt{C}_0\ \texttt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \texttt{return e;}\ \}\ \ldots\}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{C}} \to \mathtt{C}_0} \quad \text{(MT-CLASS)}$$

$$\frac{\mathtt{L} \in \Lambda_1 \qquad \mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \mathtt{C}_0\ \texttt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \texttt{return e;}\ \}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{C}} \to \mathtt{C}_0} \quad \text{(MT-PMETHOD)}$$

$$\frac{\texttt{class C} \lhd \texttt{D}\ \{\ldots\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \forall \mathtt{L} \in \Lambda_1.\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L})\ \text{undefined} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_2, \Lambda_2) = \overline{\mathtt{C}} \to \mathtt{C}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{C}} \to \mathtt{C}_0} \quad \text{(MT-SUPER)}$$

The rule MT-CLASS is used when m is defined in the base layer; the rule MT-PMETHOD is used when m is defined in one of the activated layers; and the rule MT-SUPER is used when m is not defined in class C. Notice that, in the premise of MT-SUPER, both third and fourth arguments to *mtype* are Λ_2.

Remark. Note that these rules by themselves do *not* define *mtype* as a (set-theoretic) function on $(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2)$ in the sense that it may be the case that $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{C}} \to \mathtt{C}_0$ and $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{D}} \to \mathtt{D}_0$ are derived but $\overline{\mathtt{C}}, \mathtt{C}_0 \neq \overline{\mathtt{D}}, \mathtt{D}_0$. We later enforce the signature of a method in a class to be the same in every layer (including the base) by a typing rule for a program.

4.3 Typing

A *type environment*, denoted by Γ, is a finite mapping from variables to class names, which are also types in ContextFJ. We write $\overline{\mathtt{x}}{:}\overline{\mathtt{C}}$ for a type environment Γ such that $\mathit{dom}(\Gamma) = \{\overline{\mathtt{x}}\}$ and $\Gamma(\mathtt{x}_i) = \mathtt{C}_i$ for any i. We use $\mathcal{L}$ to stand for a *location*, which is either $\bullet$ (the main expression), `C.m` (the body of method m in class C in the base layer), or `L.C.m` (the body of method m in class C in layer L). The typing rules for expressions, methods, classes, and programs are shown in Figure 4.

Expression typing. A type judgment for expressions is of the form $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{C}$, read "expression e is given type C under context Γ, location $\mathcal{L}$, and activated layers Λ." Activated layers Λ are supposed to be a subset of layers actually activated when the expression is evaluated at run time. Also note that Λ is a set rather than a sequence; it means that the type system does not know in what order layers are activated.

The first four rules T-VAR for variables, T-INVK for method invocation, T-FIELD for field access, T-NEW for object instantiation are mostly straightforward adaptations of those of FJ. Note that, in T-INVK, Λ is used to look up the type of method m in C. As discussed already, the rule T-ENSURE for `ensure` requires that layers Λ' that the newly activated layer L requires be activated; the body e is checked under the assumption in which L is added to the set of activated layers.

The next three rules are concerned about `super` and `proceed` calls. The rule T-SUPERB is for `super` in a method in a base class C, as represented in the location of the type judgment. The method type is retrieved as if the receiver type is E, which is the direct superclass of C, where the present expression appears. The set of activated layers passed to *mtype* is assumed to be empty because base classes cannot assume any layers to be activated. Note that it is always empty no matter how many `ensures` surround this `super` call. This corresponds to the operational semantics that the behavior of `super` is not affected by `ensures` in the callee. In the other two rules, the sets of layers given to *mtype* have also nothing to do with that in type judgments. The rule T-SUPERP for `super` in a partial method defined in `L.C` is similar. The only difference is that it can assume the existence of layers Λ' that L requires and also L itself. The rule T-PROCEED for `proceed` is also similar; the method name to be looked up is taken from the location `L.C.m`. Note that the third argument to *mtype* is just Λ', which means that a `proceed` call cannot proceed to the same method recursively (but can proceed to a method of the same name in a superclass of the same layer).

We defer the typing rule for method invocation `new C(`$\overline{\mathtt{v}}$`)<D,`$\overline{\mathtt{L}}$`,`$\overline{\mathtt{L}'}$`>.m(`$\overline{\mathtt{w}}$`)` on an object with a cursor to the discussion about type soundness.

Method/class/program typing. A type judgment for methods is of the form `M ok in C` (for methods in a base class) or `M ok in L.C` (for partial methods), read "method M is well formed in C (or L.C, respectively)." The typing rules T-METHOD and T-PMETHOD are straightforward. Both rules check that the method body is well typed under the type environment that formal parameters $\overline{\mathtt{x}}$ are given declared types $\overline{\mathtt{C}}$ and `this` is given the name of the class name where the method appears. The type of the method body has to be a subtype of the declared return type. For methods in a base class, the method body has to be well typed without assuming any activated layers, whereas, for partial methods, layers (Λ) that the current layer (L) requires can be assumed (as well as the current layer itself). Unlike FJ, valid method overriding is not checked here because it requires the whole program to check.

Expression typing: $\boxed{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}}$

$$\frac{(\Gamma = \overline{\mathtt{x}}{:}\overline{\mathtt{C}})}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{x}_i : \mathtt{C}_i} \quad \text{(T-VAR)}$$

$$\frac{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{C}_i} \quad \text{(T-FIELD)}$$

$$\frac{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{C}_0} \quad \text{(T-NEW)}$$

$$\frac{\mathtt{L}\ \mathtt{req}\ \Lambda' \qquad \Lambda' \subseteq \Lambda \qquad \mathcal{L};\Lambda \cup \{\mathtt{L}\};\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{ensure}\ \mathtt{L}\ \mathtt{e}_0 : \mathtt{C}_0} \quad \text{(T-ENSURE)}$$

$$\frac{\mathtt{class}\ \mathtt{C} \lhd \mathtt{E}\ \{\ldots\} \qquad \mathit{mtype}(\mathtt{m}', \mathtt{E}, \emptyset) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathtt{C.m};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathtt{C.m};\Lambda;\Gamma \vdash \mathtt{super.m}'(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-SUPERB)}$$

$$\frac{\mathtt{class}\ \mathtt{C} \lhd \mathtt{E}\ \{\ldots\} \qquad \mathtt{L}\ \mathtt{req}\ \Lambda' \qquad \mathit{mtype}(\mathtt{m}', \mathtt{E}, \Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathtt{L.C.m};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathtt{L.C.m};\Lambda;\Gamma \vdash \mathtt{super.m}'(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-SUPERP)}$$

$$\frac{\mathtt{L}\ \mathtt{req}\ \Lambda' \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda', \Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \qquad \mathtt{L.C.m};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{D}}}{\mathtt{L.C.m};\Lambda;\Gamma \vdash \mathtt{proceed}(\overline{\mathtt{e}}) : \mathtt{D}_0} \quad \text{(T-PROCEED)}$$

Method/class typing: $\boxed{\mathtt{M\ ok\ in\ C}}$

$$\frac{\mathtt{C.m};\emptyset;\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{D}_0 \qquad \mathtt{D}_0 <: \mathtt{C}_0}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{ok\ in\ C}} \quad \text{(T-METHOD)}$$

$\boxed{\mathtt{M\ ok\ in\ L.C}}$

$$\frac{\mathtt{L}\ \mathtt{req}\ \Lambda \qquad \mathtt{L.C.m};\Lambda \cup \{\mathtt{L}\};\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{D}_0 \qquad \mathtt{D}_0 <: \mathtt{C}_0}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{ok\ in\ L.C}} \quad \text{(T-PMETHOD)}$$

$\boxed{\mathtt{CL\ ok}}$

$$\frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\ \mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}{=}\overline{\mathtt{f}};\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{ok\ in\ C}}{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{ok}} \quad \text{(T-CLASS)}$$

Valid overriding: $\boxed{\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2)}$ $\boxed{\mathit{override}^h(\mathtt{L}, \mathtt{C})}$ $\boxed{\mathit{override}^v(\mathtt{C}, \mathtt{D})}$

$$\frac{\forall \mathtt{m}, \mathtt{C}.\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}_1) = \mathtt{C}_0\ \ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ldots\} \text{ and } \mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}_2) = \mathtt{D}_0\ \ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{y}})\{\ldots\}, \text{ then } \overline{\mathtt{C}}, \mathtt{C}_0 = \overline{\mathtt{D}}, \mathtt{D}_0}{\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2)}$$

$$\frac{\forall \mathtt{m}.\ \text{if } \mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \mathtt{C}_0\ \ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ldots\}\ \ldots\ \} \text{ and } \mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \mathtt{D}_0\ \ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{y}})\{\ldots\}, \text{ then } \overline{\mathtt{C}}, \mathtt{C}_0 = \overline{\mathtt{D}}, \mathtt{D}_0}{\mathit{override}^h(\mathtt{L}, \mathtt{C})}$$

$$\frac{\forall \mathtt{m}.\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{C}, \mathit{dom}(\mathit{PT}), \mathit{dom}(\mathit{PT})) = \overline{\mathtt{C}} \to \mathtt{C}_0 \text{ and } \mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathit{dom}(\mathit{PT}), \mathit{dom}(\mathit{PT})) = \overline{\mathtt{D}} \to \mathtt{D}_0 \text{ and } \mathtt{C} <: \mathtt{D}, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 <: \mathtt{D}_0}{\mathit{override}^v(\mathtt{C}, \mathtt{D})}$$

Program typing: $\boxed{\vdash (\mathit{CT}, \mathit{PT}, \mathtt{e}) : \mathtt{C}}$

$$\frac{\begin{array}{c}\forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathit{CT}(\mathtt{C})\ \ \mathtt{ok} \qquad \forall (\mathtt{m}, \mathtt{C}, \mathtt{L}) \in \mathit{dom}(\mathit{PT}).\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L})\ \ \mathtt{ok\ in\ L.C} \\ \bullet;\emptyset;\bullet \vdash \mathtt{e} : \mathtt{C} \\ \forall \mathtt{L}_1, \mathtt{L}_2 \in \mathit{dom}(\mathit{PT}).\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2) \\ \forall \mathtt{C} \in \mathit{dom}(\mathit{CT}), \mathtt{L} \in \mathit{dom}(\mathit{PT}).\mathit{override}^h(\mathtt{L}, \mathtt{C}) \qquad \forall \mathtt{C}, \mathtt{D} \in \mathit{dom}(\mathit{CT}).\mathit{override}^v(\mathtt{C}, \mathtt{D})\end{array}}{\vdash (\mathit{CT}, \mathit{PT}, \mathtt{e}) : \mathtt{C}} \quad \text{(T-PROG)}$$

Figure 4. ContextFJ: Typing rules.

A class is well formed (written CL OK) when the constructor matches the field declarations and all methods are well formed.

Finally, a program is well formed when all classes are well formed, all partial methods are well formed, and the main expression is well typed under the empty assumption. The other conditions mean that no two layers provide conflicting methods and all method overriding (by a subclass or a partial method in a layer) is valid. The predicate $\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2)$ means that there are no conflicting methods in $\mathtt{L}_1$ and $\mathtt{L}_2$; the predicate $\mathit{override}^h(\mathtt{L}, \mathtt{C})$ (h stands for "horizontally") mean that all overriding partial methods in L have the same signatures as the corresponding methods in C; and the predicate $\mathit{override}^v(\mathtt{C}, \mathtt{D})$ (v stands for "vertically") mean that method override by a subclass C of D is valid. Note that when *noconflict* and $\mathit{override}^h$ hold for any combination of layers and classes, *mtype* is a (set-theoretic) function.

It is interesting to see that covariant overriding of the return type is allowed only by a based method in a subclass. In fact, we cannot allow covariant overriding by a partial method because the order of layer composition varies at run time. The last premise checks that a method in a subclass correctly overrides a method of the same name in a superclass; we can allow covariant overriding of the return type here.

Note that, unlike *noconflict* and $\mathit{override}^h$, checking $\mathit{override}^v$ for a given pair of classes needs type information on *all* the layers, as $\mathit{dom}(PT)$ is used for the third and fourth argument to *mtype*. We need to take all the layers into account in case a subclass C defines m, which is not present in its superclass D, and some layer adds (not overrides) m to D.

4.4 Type Soundness

This type system is sound with respect to the operational semantics given in the last section. Type soundness is shown via subject reduction and progress properties [16, 22]. In order to state these properties, though, we need to formalize the condition when a statically assumed layer set matches a run-time layer configuration. We write $\overline{\mathtt{L}}$ *wf*, read "a run-time layer configuration $\overline{\mathtt{L}}$ is well formed", which is defined as follows:

$$\bullet \ \mathit{wf}$$

$$\frac{\overline{\mathtt{L}} \ \mathit{wf} \qquad \mathtt{L}\ \mathtt{req}\ \Lambda \qquad \Lambda \subseteq \{\overline{\mathtt{L}}\}}{\overline{\mathtt{L}};\mathtt{L} \ \mathit{wf}}$$

The first rule means that the empty sequence of layers is well formed and the second that a sequence $\overline{\mathtt{L}};\mathtt{L}$ is well formed if the prefix $\overline{\mathtt{L}}$ is well formed and the layers Λ that L requires have already been activated ($\Lambda \subseteq \{\overline{\mathtt{L}}\}$).[6]

We also need to give a typing rule for expressions that appear only at run time, i.e., method invocation on an object with a cursor.

$$\frac{\begin{array}{c}\overline{\mathtt{L}}' \text{ is a prefix of } \overline{\mathtt{L}}'' \qquad \overline{\mathtt{L}}'' \ \mathit{wf} \\ \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}} \\ \mathtt{C}_0 <: \mathtt{D} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}''\}) = \overline{\mathtt{F}} \to \mathtt{F}_0 \\ \mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \overline{\mathtt{E}} <: \overline{\mathtt{F}}\end{array}}{\mathcal{L};\Lambda;\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{D},\overline{\mathtt{L}}',\overline{\mathtt{L}}''\texttt{>}.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{F}_0} \quad \text{(T-InvkA)}$$

This rule is basically thought as a combination of T-Invk and T-New. One notable point is that the cursor information D, $\overline{\mathtt{L}}'$, and $\overline{\mathtt{L}}''$ is used to look up the type of m (instead of the receiver's run-time class $\mathtt{C}_0$ and the assumed set of activated layers Λ).

Now, the soundness theorem is stated below. Proofs of subject reduction and progress are given in Appendix A; type soundness follows easily from them.

[6] The notation $\{\overline{\mathtt{L}}\}$ is used to ignore the order in a sequence; formally, it denotes the set consisting of all elements of $\overline{\mathtt{L}}$.

Theorem 1 (Subject Reduction). *Suppose given class and partial method tables are well-formed. If* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \mathtt{e} : \mathtt{C}$ *and* $\overline{\mathtt{L}}$ *wf and* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \mathtt{e}' : \mathtt{D}$ *for some* D *such that* $\mathtt{D} <: \mathtt{C}$.

Theorem 2 (Progress). *Suppose given class and partial method tables are well-formed. If* $\bullet;\{\overline{\mathtt{L}}\};\bullet \vdash \mathtt{e} : \mathtt{C}$ *and* $\overline{\mathtt{L}}$ *wf, then either* e *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

Theorem 3 (Type Soundness). *If* $\vdash (CT, PT, \mathtt{e}) : \mathtt{C}$ *and* e *reduces to a normal form, then* e *is* $\mathtt{new}\ \mathtt{D}(\overline{\mathtt{v}})$ *for some* $\overline{\mathtt{v}}$ *and* D *such that* $\mathtt{D} <: \mathtt{C}$.

5. Discussion

We have formalized a type system for dynamic layer composition and proved its soundness. One key idea is to approximate activated layers at each program point with the help of explicitly declared dependencies between layers. Our result shows that such a dependency relation is sufficient for a particular layer activation mechanism, namely ensure, which does not change the order of already activated layers.

We discuss other possible layer (de)activation mechanisms. Many COP languages have with L {...}, which always activates L as the first layer to be executed by *changing the order of layers* when L has been already activated, and without L {...}, which temporarily deactivates L during the execution of the body. One motivation for with is that a programmer may want to ensure partial methods in the activated layer are executed first. The present type system is not sufficient for such order-changing layer manipulation because it statically estimates only a *lower bound* of activated layers (whose ordering is lost). For example, consider without L1 {...} when Λ in the type judgment for this expression is $\{\mathtt{L}_1, \mathtt{L}_2\}$. Since $\{\mathtt{L}_1, \mathtt{L}_2\}$ gives only a lower-bound, the run-time layer configuration can be $\mathtt{L}_1;\mathtt{L}_2$, $\mathtt{L}_2;\mathtt{L}_1$, or even $\mathtt{L}_2;\mathtt{L}_1;\mathtt{L}_3$. It is unsafe, however, to remove $\mathtt{L}_1$ from $\mathtt{L}_2;\mathtt{L}_1;\mathtt{L}_3$ if $\mathtt{L}_3$ requires $\mathtt{L}_1$. Similarly, with $\mathtt{L}_1$ may cause trouble when the run-time configuration is $\mathtt{L}_2;\mathtt{L}_1;\mathtt{L}_3$ and $\mathtt{L}_3$ requires $\mathtt{L}_1$: because it will move $\mathtt{L}_1$ and change the configuration to $\mathtt{L}_2;\mathtt{L}_3;\mathtt{L}_1$, where $\mathtt{L}_3$'s requirement is no longer satisfied. In short, with and without are difficult because they may break the well-formedness condition on a run-time layer configuration.

The present type system works if layer manipulation constructs do not break well-formedness. One compromise between ensure and with could be to activate the designated layer always as the first one *and leave the already activated one as it is*, resulting in two copies of the same layer in the run-time layer configuration. However, it may cause a partial method in a layer to run twice for one method call, leaving programmers surprised (especially when layers have side effects, which is the case in real COP language extensions).

Related Work A layer in COP languages is essentially a set of mixins (or a mixin layer [19]), which can be composed or decomposed at run time. An idea similar to our requires clauses can be found in type systems for mixins [4, 8, 14], where a mixin specifies the interface of classes to be composed. Our require clauses can be considered an extension of this idea to a set of interfaces. In a language with mixins, however, once an object is instantiated, composed mixins are never "deactivated". Nevertheless, as this paper shows, a similar idea works—to some extent—even for dynamic (de)composition. Our requires clauses are not very flexible, because one has to specify a *single* set of layers, which are tied to specific implementations. So, one cannot express dependency like "this layer requires either $\mathtt{L}_1$ or $\mathtt{L}_2$" or even "this layer requires any layer that provides a method of this signature." It would not be hard to extend our type system so that dependency can be specified via a

set of method signatures [4, 14] or Java-like interfaces adapted for layers.

In fact, Clarke and Sergey [5] independently formalize a core language (also called ContextFJ) for context-oriented programming (with both `with` and `without` but no inheritance) and develop such a type system. In their type system, each partial/base method (rather than a layer) is equipped with a set of the signatures of the methods that it may call as dependency information, which is very fine-grained. However, their type system turns out to be unsound because it does not handle removal of layers (caused by `without`) properly (personal communication with Clarke and Sergey).

Feature-oriented programming (FOP) [3] and delta-oriented programming (DOP) [17] also advocate the use of layers or delta modules respectively to describe behavioral variations. In both approaches, however, composition with base classes is only *static*, namely, happens at compile time. Their type systems [1, 7, 18] also use explicitly declared dependencies, often called feature models, for modular typechecking. The languages to specify dependencies between layers or delta modules are richer than ours, which is just a set of `requires` clauses. It is interesting future work to incorporate feature models in our type system.

Typestate checking [20] is a technique to keep track of state transition of computational resources (such as files and sockets) during program execution statically. Our type system, which might be considered a kind of typestate checking of layer configurations, is simpler than typestate checking in that there is only one global resource but inexact because only an approximation of a layer configuration can be obtained. Note that typestate checking is not directly applicable because it is usually based on a finite state transition system, whereas the state space of layer configurations is infinite.

Acknowledgments. Comments from anonymous reviewers of FOOL2012 helped us improve the presentation of the paper. We thank Dave Clarke and Ilya Sergey for answering questions on their work and members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research No. 22240002 (Igarashi and Masuhara).

References

[1] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *Proc. of GPCE*, 2008. doi:10.1145/1449913.1449931.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, January 2011.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 2004. doi:10.1109/TSE.2004.23.

[4] Viviana Bono, Amit Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and mixins. In *Proc. of ECOOP'99*, 1999.

[5] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *Proc. of 1st International Workshop on Context-Oriented Programming (COP'09)*, Genova, Italy, 2009.

[6] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming - an overview of ContextL. In *Proc. of DLS*, 2005. doi:10.1145/1146841.1146842.

[7] Benjamin Delaware, William Cook, and Don Batory. A machine-checked model of safe composition. In *Proc. of International Workshop on Foundations of Aspect-Oriented Languages*, 2009. doi:10.1145/1509837.1509846.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. of ACM POPL*, pages 171–183, 1998.

[9] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *Proc. of GTTSE*, 2008.

[10] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 2008.

[11] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: A minimal core calculus for context-oriented programming. In *Proc. of International Workshop on Foundations of Aspect-Oriented Languages (FOAL2011)*, Pernambuco, Brazil, March 2011.

[12] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 2001. doi:10.1145/503502.503505.

[13] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011.

[14] Tetsuo Kamina and Tetsuo Tamai. A core calculus for mixin-types. In *Proc. of FOOL11*, 2004.

[15] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming, Special Issue on Software Evolution*, 2010. doi:10.1016/j.scico.2010.11.013.

[16] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2001.

[17] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of Software Product Line Conference*, 2010.

[18] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proc. of AOSD*, 2011.

[19] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP'98*, pages 550–570, 1998.

[20] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, January 1986.

[21] The AspectJ Team. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`. Site visited Aug. 12, 2012.

[22] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A. Proofs

LEMMA 1 (Weakening).

1. *If* $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$, *then* $\mathcal{L};\Lambda;\Gamma,\mathtt{x}\colon \mathtt{D} \vdash \mathtt{e} : \mathtt{C}$.
2. *If* $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$, *then* $\mathcal{L};\Lambda \cup \{\mathtt{L}\};\Gamma \vdash \mathtt{e} : \mathtt{C}$.

Proof: By straightforward induction on $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{e} : \mathtt{C}$. □

LEMMA 2 (Strengthening for values). *If* $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{v} : \mathtt{C}$, *then* $\mathcal{L}';\Lambda';\Gamma' \vdash \mathtt{v} : \mathtt{C}$.

Proof: By straightforward induction on $\mathcal{L};\Lambda;\Gamma \vdash \mathtt{v} : \mathtt{C}$. □

LEMMA 3. *If* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ *and* $\mathtt{D} <: \mathtt{C}$, *then* $\mathit{fields}(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}\ \overline{\mathtt{g}}$ *for some* $\overline{\mathtt{D}}$ *and* $\overline{\mathtt{g}}$.

Proof: By straightforward induction on $\mathtt{D} <: \mathtt{C}$. □

LEMMA 4. *If* $\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda) = \overline{\mathtt{D}}\to\mathtt{D}_0$ *and* $\mathtt{D} <: \mathtt{C}$, *then* $\mathit{mtype}(\mathtt{m},\mathtt{D},\Lambda) = \overline{\mathtt{D}}\to\mathtt{E}_0$ *and* $\mathtt{E}_0 <: \mathtt{D}_0$ *for some* $\mathtt{E}_0$.

Proof: By induction on $\mathtt{D} <: \mathtt{C}$. □

LEMMA 5 (Substitution). *If* $\mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}}\colon\overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\mathcal{L};\Lambda;\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}}$ *and* $\overline{\mathtt{D}} <: \overline{\mathtt{C}}$, *then* $\mathcal{L};\Lambda;\Gamma \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{D}_0$ *and* $\mathtt{D}_0 <: \mathtt{C}_0$ *for some* $\mathtt{D}_0$.

Proof: By induction on $\mathcal{L};\Lambda;\Gamma,\overline{\mathtt{x}}\colon\overline{\mathtt{C}} \vdash \mathtt{e}_0 : \mathtt{C}_0$. □

LEMMA 6 (Substitution for `super` and `proceed`).

1. *If* $\mathtt{L}\ \mathtt{req}\ \Lambda'$ *and* $\Lambda' \subseteq \{\overline{\mathtt{L}'}\}$ *and* $\overline{\mathtt{L}'};\mathtt{L}$ *is a prefix of* $\overline{\mathtt{L}}$ *and* $\overline{\mathtt{L}}$ *wf and* $\mathtt{L.C.m};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\mathtt{D}_0 <: \mathtt{C}$ *and* $\mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ *and* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{E}}$ *and* $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$ *and* $\mathtt{class}\ \mathtt{C} \lhd \mathtt{D}$, *then* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash S\mathtt{e}_0 : \mathtt{C}_0$ *where*
$$S = \begin{bmatrix} \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m/proceed}, \\ \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\quad\mathtt{/super} \end{bmatrix}.$$
2. *If* $\overline{\mathtt{L}}$ *wf and* $\mathtt{C.m};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\mathtt{D}_0 <: \mathtt{C}$ *and* $\mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ *and* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{E}}$ *and* $\overline{\mathtt{E}} <: \overline{\mathtt{D}}$ *and* $\mathtt{class}\ \mathtt{C} \lhd \mathtt{D}$, *then* $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash [\mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>/super}]\mathtt{e}_0 : \mathtt{C}_0$.

Proof: 1. By induction on $\mathtt{L.C.m};\Lambda;\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$ with case analysis on the last typing rule used. We show only main cases below.

Case T-SUPERP**:** $\mathtt{e}_0 = \mathtt{super.m}'(\overline{\mathtt{e}})$, $\mathit{mtype}(\mathtt{m}',\mathtt{D},\Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{F}}\to\mathtt{C}_0$, $\mathtt{L.C.m};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}}$, $\overline{\mathtt{G}} <: \overline{\mathtt{F}}$

Since $S\mathtt{e}_0 = \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m}'(S\overline{\mathtt{e}})$, it suffices to show that $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}'(S\overline{\mathtt{e}}) : \mathtt{C}_0$. By the induction hypothesis, we have $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash S\overline{\mathtt{e}} : \overline{\mathtt{G}}$. Since $\mathtt{D}_0 <: \mathtt{C}$ and $\mathtt{class}\ \mathtt{C} \lhd \mathtt{D}$, we have $\mathtt{D}_0 <: \mathtt{D}$. Then, T-INVKA finishes the case.

Case T-PROCEED**:** $\mathtt{e}_0 = \mathtt{proceed}(\overline{\mathtt{e}})$, $\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda',\Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{F}}\to\mathtt{C}_0$, $\mathtt{L.C.m};\Lambda;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}}$, $\overline{\mathtt{G}} <: \overline{\mathtt{F}}$

Since $S\mathtt{e}_0 = \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}(S\overline{\mathtt{e}})$, it suffices to show that
$$\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{v}})\mathtt{<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}(S\overline{\mathtt{e}}) : \mathtt{C}_0$$
but it is easy to show by T-INVKA and the induction hypothesis.

2. Similar. Note that the case T-PROCEED cannot happen. □

LEMMA 7. *Suppose* $\overline{\mathtt{L}}'$ *is a prefix of* $\overline{\mathtt{L}}''$ *and* $\overline{\mathtt{L}}''$ *wf and* $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathit{in}\ \mathtt{C}',\overline{\mathtt{L}}$ *and* $\mathit{mtype}(\mathtt{m},\mathtt{C},\{\overline{\mathtt{L}}'\},\{\overline{\mathtt{L}}''\}) = \overline{\mathtt{D}}\to\mathtt{D}_0$.

1. *If* $\overline{\mathtt{L}} = \overline{\mathtt{L}}''';\mathtt{L}_0$, *then* $\mathtt{L}_0\mathtt{.C}'\mathtt{.m};\Lambda' \cup \{\mathtt{L}_0\};\overline{\mathtt{x}}\colon\overline{\mathtt{D}},\mathtt{this} : \mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{E}_0$ *and* $\mathtt{L}_0\ \mathtt{req}\ \Lambda'$ *and* $\Lambda' \subseteq \{\overline{\mathtt{L}}'''\}$ *and* $\mathtt{C} <: \mathtt{C}'$ *and* $\mathtt{E}_0 <: \mathtt{D}_0$ *for some* $\mathtt{E}_0$ *and* Λ'.
2. *If* $\overline{\mathtt{L}} = \bullet$, *then* $\mathtt{C}'\mathtt{.m};\emptyset;\overline{\mathtt{x}}\colon\overline{\mathtt{D}},\mathtt{this} : \mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{E}_0$ *and* $\mathtt{C} <: \mathtt{C}'$ *and* $\mathtt{E}_0 <: \mathtt{D}_0$ *for some* $\mathtt{E}_0$.

Proof: By induction on $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}',\overline{\mathtt{L}}$.

Case MB-CLASS**:** $\overline{\mathtt{L}}' = \bullet$, $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\ \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \ldots$, $\mathtt{C}' = \mathtt{C}$, $\overline{\mathtt{L}} = \bullet$

By T-CLASS, T-METHOD and MT-CLASS, it must be the case that
$$\mathtt{C}_0,\overline{\mathtt{C}} = \mathtt{D}_0,\overline{\mathtt{D}}$$
$$\mathtt{C.m};\emptyset;\overline{\mathtt{x}} : \overline{\mathtt{D}},\mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0$$
$$\mathtt{E}_0 <: \mathtt{D}_0$$
for some $\mathtt{E}_0$, finishing the case.

Case MB-LAYER**:** $\overline{\mathtt{L}}' = \overline{\mathtt{L}}''';\mathtt{L}_0$, $\mathit{PT}(\mathtt{m},\mathtt{C},\mathtt{L}_0) = \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}$, $\mathtt{C}' = \mathtt{C}$, $\overline{\mathtt{L}} = \overline{\mathtt{L}}'$

By T-PMETHOD, it must be the case that
$$\mathtt{C}_0,\overline{\mathtt{C}} = \mathtt{D}_0,\overline{\mathtt{D}}$$
$$\mathtt{L}_0\ \mathtt{req}\ \Lambda$$
$$\mathtt{L}_0\mathtt{.C.m};\Lambda \cup \{\mathtt{L}_0\};\overline{\mathtt{x}} : \overline{\mathtt{D}},\mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0$$
$$\mathtt{E}_0 <: \mathtt{D}_0$$
for some $\mathtt{E}_0$. By $\overline{\mathtt{L}}''$ *wf*, we have $\overline{\mathtt{L}}'$ *wf* and so $\Lambda \subseteq \{\overline{\mathtt{L}}'''\}$, finishing the case.

Case MB-SUPER**:** $\overline{\mathtt{L}}' = \bullet$, $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \overline{\mathtt{M}}\ \}$, $\mathtt{m} \notin \overline{\mathtt{M}}$, $\mathit{mbody}(\mathtt{m},\mathtt{D},\overline{\mathtt{L}}'',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}',\overline{\mathtt{L}}$

By MT-SUPER, it must be the case that $\mathit{mtype}(\mathtt{m},\mathtt{D},\{\overline{\mathtt{L}}''\},\{\overline{\mathtt{L}}''\}) = \overline{\mathtt{D}}\to\mathtt{D}_0$. The induction hypothesis and transitivity of subtyping finish the case.

Case MB-NEXTLAYER**:** $\overline{\mathtt{L}}' = \overline{\mathtt{L}}''';\mathtt{L}_0$, $\mathit{PT}(\mathtt{m},\mathtt{C},\mathtt{L}_0)$ undefined, $\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}''',\overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}',\overline{\mathtt{L}}$

The induction hypothesis finishes the case. □

Proof of Theorem 1: By induction on $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the last reduction rule used.

Case R-FIELD**:** $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\mathtt{.f}_i$, $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, $\mathtt{e}' = \mathtt{v}_i$

By T-FIELD and T-NEW, it must be the case that
$$\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} \qquad \overline{\mathtt{D}} <: \overline{\mathtt{C}} \qquad \mathtt{C} = \mathtt{C}_i$$
and, in particular, $\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \mathtt{v}_i : \mathtt{D}_i$ and $\mathtt{D}_i <: \mathtt{C}_i$, finishing the case.

Case R-INVK**:** $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\mathtt{.m}(\overline{\mathtt{w}})$, $\overline{\mathtt{L}} \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})\mathtt{<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}(\overline{\mathtt{w}}) \longrightarrow \mathtt{e}'$

By T-INVK and T-NEW, it must be the case that
$$\bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} \qquad \mathit{fields}(\overline{\mathtt{C}}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \overline{\mathtt{D}} <: \overline{\mathtt{C}}$$
$$\mathit{mtype}(\mathtt{m},\mathtt{C}_0) = \overline{\mathtt{E}}\to\mathtt{C} \qquad \bullet;\{\overline{\mathtt{L}}\};\Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{F}} \qquad \overline{\mathtt{F}} <: \overline{\mathtt{E}}.$$

Since $\mathtt{C}_0 <: \mathtt{C}_0$, we have

$$\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<}\mathtt{C}_0,\overline{\mathtt{L}},\overline{\mathtt{L}}\mathtt{>.m}(\overline{\mathtt{w}}) : \mathtt{C}$$

by T-INVKA. By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{D}$ for some $\mathtt{D} <: \mathtt{C}$, finishing the case.

Case R-INVKP**:**

$$\begin{array}{l}
\mathtt{e} = \mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<C',}\overline{\mathtt{L}}',\overline{\mathtt{L}}''\mathtt{>.m}(\overline{\mathtt{w}}) \\
\mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}'') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C''}, (\overline{\mathtt{L}}'''; \mathtt{L}_0) \\
\mathtt{class\ C''} \lhd \mathtt{D\ \{...\}} \\
\mathtt{e}' = \left[\begin{array}{ll}
\mathtt{new\ C}_0(\overline{\mathtt{v}}) & /\mathtt{this} \\
\overline{\mathtt{w}} & /\overline{\mathtt{x}} \\
\mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<C'',}\overline{\mathtt{L}}''',\overline{\mathtt{L}}''\mathtt{>.m} & /\mathtt{proceed} \\
\mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}'',\overline{\mathtt{L}}''\mathtt{>} & /\mathtt{super}
\end{array}\right] \mathtt{e}_0
\end{array}$$

By T-INVKA, it must be the case that

$$\begin{array}{ll}
\overline{\mathtt{L}}' \text{ is a prefix of } \overline{\mathtt{L}}'' & \overline{\mathtt{L}}'' \ \textit{wf} \\
\textit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} & \textit{mtype}(\mathtt{m}, \mathtt{C}', \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}''\}) = \overline{\mathtt{F}}\to\mathtt{C} \\
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} & \overline{\mathtt{D}} <: \overline{\mathtt{C}} \qquad \mathtt{C}_0 <: \mathtt{C}' \\
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{E}} & \overline{\mathtt{E}} <: \overline{\mathtt{F}}.
\end{array}$$

By T-NEW, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{new\ C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$.

By Lemma 7,

$$\mathtt{L}_0\mathtt{.C''.m}; \Lambda' \cup \{\mathtt{L}_0\}; \overline{\mathtt{x}} : \overline{\mathtt{F}}, \mathtt{this} : \mathtt{C''} \vdash \mathtt{e}_0 : \mathtt{E}_0$$

and $\mathtt{C}' <: \mathtt{C}''$ and $\mathtt{E}_0 <: \mathtt{C}$ and $\mathtt{L}_0\ \mathtt{req}\ \Lambda'$ and $\Lambda' \subseteq \{\overline{\mathtt{L}}'''\}$ for some $\mathtt{E}_0$ and Λ'. By S-TRANS, $\mathtt{C}_0 <: \mathtt{C}''$.

By Lemmas 1, 2, 5, and 6, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{E}_0{}'$ for some $\mathtt{E}_0{}' <: \mathtt{E}_0$. By S-TRANS, $\mathtt{E}_0{}' <: \mathtt{C}$, finishing the case.

Case R-INVKB**:**

Similar to the case for R-INVKP.

Case RC-ENSURE**:** $\mathtt{e} = \mathtt{ensure\ L\ e}_0 \qquad \mathtt{e}' = \mathtt{ensure\ L\ e}_0{}'$
$\overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$

By T-ENSURE, it must be the case that $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}$. By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0{}' : \mathtt{D}$ for some $\mathtt{D} <: \mathtt{C}$. By T-ENSURE, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{D}$, finishing the case.

Case R-ENSUREVAL**:** $\mathtt{e} = \mathtt{ensure\ L\ v}_0 \qquad \mathtt{e}' = \mathtt{v}_0$

By T-ENSURE, it must be the case that $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{v}_0 : \mathtt{C}$. Then, by Lemma 2, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{v}_0 : \mathtt{C}$, finishing the case.

Case RC-FIELD**:** $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \qquad \overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$
$\mathtt{e}' = \mathtt{e}_0{}'\mathtt{.f}_i$

By T-FIELD, it must be the case that

$$\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \textit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathtt{C} = \mathtt{C}_i.$$

By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0{}' : \mathtt{D}_0$ for some $\mathtt{D}_0 <: \mathtt{C}_0$. By Lemma 3, $\textit{fields}(\mathtt{D}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{D}}$ and $\overline{\mathtt{g}}$. By T-FIELD, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0{}'\mathtt{.f}_i : \mathtt{C}_i$, finishing the case.

Case RC-INVKRECV**:** $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \qquad \overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$
$\mathtt{e}' = \mathtt{e}_0{}'\mathtt{.m}(\overline{\mathtt{e}})$

By T-INVK, it must be the case that

$$\begin{array}{ll}
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 & \textit{mtype}(\mathtt{m}, \mathtt{C}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{D}}\to\mathtt{C} \\
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} & \overline{\mathtt{E}} <: \overline{\mathtt{D}}.
\end{array}$$

By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0{}' : \mathtt{D}_0$ for some $\mathtt{D}_0 <: \mathtt{C}_0$. By Lemma 4, $\textit{mtype}(\mathtt{m}, \mathtt{D}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{D}}\to\mathtt{D}$ and $\mathtt{D} <: \mathtt{C}$ for some $\mathtt{D}$. By T-INVK, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0{}'\mathtt{.m}(\overline{\mathtt{e}}) : \mathtt{D}$, finishing the case.

Case RC-INVKARG**:** $\mathtt{e} = \mathtt{e}_0\mathtt{.m(..,e}_i\mathtt{,..)} \qquad \overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i{}'$
$\mathtt{e}' = \mathtt{e}_0\mathtt{.m(..,e}_i{}'\mathtt{,..)}$

By T-INVK, it must be the case that

$$\begin{array}{ll}
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 & \textit{mtype}(\mathtt{m}, \mathtt{C}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{D}}\to\mathtt{C} \\
\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} & \overline{\mathtt{E}} <: \overline{\mathtt{D}}.
\end{array}$$

By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_i{}' : \mathtt{F}_i$ for some $\mathtt{F}_i <: \mathtt{E}_i$. By S-TRANS, $\mathtt{F}_i <: \mathtt{D}_i$. So, by T-INVK, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{C}$, finishing the case.

Case RC-NEW, RC-INVKAARG**:**

Similar to the case above. □

LEMMA 8. *If* $\textit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{D}}\to\mathtt{D}_0$ *and* $\overline{\mathtt{L}}$ *is a prefix of* $\overline{\mathtt{L}}'$ *and* $\Lambda_1 \subseteq \{\overline{\mathtt{L}}\}$ *and* $\Lambda_1 \subseteq \Lambda_2 \subseteq \{\overline{\mathtt{L}}'\}$ *and* $\overline{\mathtt{L}}'$ *wf, then there exist* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *and* $\overline{\mathtt{L}}''$ *and* $\mathtt{C}'$ *(*$\neq \mathtt{Object}$*) such that* $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathit{in}\ \mathtt{C}', \overline{\mathtt{L}}''$ *and the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{D}}$ *are equal.*

Proof: By lexicographic induction on $\textit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{D}}\to\mathtt{D}_0$ and the length of $\overline{\mathtt{L}}$.

Case: $\overline{\mathtt{L}} = \bullet$
$\mathtt{class\ C} \lhd \mathtt{D\ \{...\ C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\mathtt{\{\ return\ e}_0\mathtt{;\ \}\ ...\}}$

By MT-CLASS, it must be the case that $\overline{\mathtt{D}}, \mathtt{D}_0 = \overline{\mathtt{C}}, \mathtt{C}_0$ and the lengths of $\overline{\mathtt{C}}$ and $\overline{\mathtt{x}}$ are equal. Then, by MB-CLASS, $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}, \bullet$.

Case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}''', \mathtt{L}_0$
$\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{E}_0\ \mathtt{m}(\overline{\mathtt{E}}\ \overline{\mathtt{x}})\mathtt{\{\ return\ e}_0\mathtt{;\ \}}$

By T-PMETHOD, it must be the case that $\overline{\mathtt{E}}, \mathtt{E}_0 = \overline{\mathtt{D}}, \mathtt{D}_0$ and the lengths of $\overline{\mathtt{D}}$ and $\overline{\mathtt{x}}$ are equal. By MB-LAYER, $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}, \overline{\mathtt{L}}$.

Case: $\overline{\mathtt{L}} = \bullet \qquad \mathtt{class\ C} \lhd \mathtt{D\ \{...}\ \overline{\mathtt{M}}\ \mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}$
$\Lambda_1 = \emptyset$

By MT-SUPER, we have $\textit{mtype}(\mathtt{m}, \mathtt{D}, \emptyset, \Lambda_2) = \overline{\mathtt{D}}\to\mathtt{D}_0$. By the induction hypothesis, there exist $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ and $\overline{\mathtt{L}}''$ and $\mathtt{C}'$ ($\neq \mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}}''$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are equal. By MB-SUPER, $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}}''$, finishing the case.

Case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}'''; \mathtt{L}_0 \qquad \mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)$ undefined

By the induction hypothesis, there exist $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ and $\overline{\mathtt{L}}''$ and $\mathtt{C}'$ ($\neq \mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}''', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}}''$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are equal. By MB-NEXTLAYER, $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}}''$, finishing the case. □

Proof of Theorem 2: By induction on $\bullet; \Lambda; \bullet \vdash \mathtt{e} : \mathtt{C}$ with case analysis on the last typing rule used.

Case T-VAR, T-SUPER, T-PROCEED**:**

Cannot happen.

Case T-FIELD**:** $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \qquad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e}_0 : \mathtt{C}_0$
$\textit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathtt{C} = \mathtt{C}_i$

By the induction hypothesis, either $\mathtt{e}_0$ is a value or there exists $\mathtt{e}_0{}'$ such that $\overline{\mathtt{L}} \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$. In the latter case, RC-FIELD finishes the case. In the former case where $\mathtt{e}_0$ is a value, by T-NEW, we have

$$\mathtt{e}_0 = \mathtt{new\ C}_0(\overline{\mathtt{v}}) \qquad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} \qquad \overline{\mathtt{D}} <: \overline{\mathtt{C}}.$$

So, we have $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{v}_i$, finishing the case.

Case T-INVK**:** $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \qquad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e}_0 : \mathtt{C}_0$
$\textit{mtype}(\mathtt{m}, \mathtt{C}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{D}}\to\mathtt{C} \qquad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$
$\overline{\mathtt{E}} <: \overline{\mathtt{D}}$

By the induction hypothesis, there exist $i \geq 0$ and $\mathtt{e}_i{}'$ such that $\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i{}'$, in which case RC-INVKRECV or RC-INVKARG finishes the case, or all $\mathtt{e}_i$'s are values $\mathtt{v}_0, \overline{\mathtt{v}}$. Then, by T-NEW, $\mathtt{v}_0 = \mathtt{new\ C}_0(\overline{\mathtt{w}})$ for some values $\overline{\mathtt{w}}$. By Lemma 8, there exist $\overline{\mathtt{x}}$, $\mathtt{e}_0{}'$, $\overline{\mathtt{L}}''$, and $\mathtt{C}'$ ($\neq \mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}}''$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{D}}$ are the same. Since $\mathtt{C}' \neq \mathtt{Object}$, there exists $\mathtt{D}'$ such that $\mathtt{class\ C}' \lhd \mathtt{D}'\ \mathtt{\{...\}}$. We have two subcases here depending on whether $\overline{\mathtt{L}}''$ is empty or not. We will show the case where $\overline{\mathtt{L}}''$ is not empty; the other

case is similar. Let $\overline{\mathtt{L}}'' = \overline{\mathtt{L}}'''; \mathtt{L}_0$ for some $\overline{\mathtt{L}}'''$ and $\mathtt{L}_0$. Then, the expression

$$\mathtt{e}' = \left[\begin{array}{ll} \mathtt{new\ C_0(\overline{w})} & /\mathtt{this} \\ \overline{\mathtt{v}} & /\overline{\mathtt{x}} \\ \mathtt{new\ C_0(\overline{w})\langle C', \overline{L}''', \overline{L}\rangle.m} & /\mathtt{proceed} \\ \mathtt{new\ C_0(\overline{w})\langle D', \overline{L}, \overline{L}\rangle} & /\mathtt{super} \end{array}\right] \mathtt{e_0}'$$

is well defined (note that the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{v}}$ are equal). Then, by R-INVKP and R-INVK, $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$.

Case T-NEW**:** $\mathtt{e} = \mathtt{new\ C(\overline{e})}$ $\quad$ $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$
$\bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}}$ $\quad$ $\overline{\mathtt{D}} <: \overline{\mathtt{C}}$

By the induction hypothesis, either (1) $\overline{\mathtt{e}}$ are all values, in which case $\mathtt{e}$ is also a value; or (2) there exist i and $\mathtt{e}_i{}'$ such that $\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i{}'$, in which case RC-NEW finishes the case.

Case T-ENSURE**:** $\mathtt{e} = \mathtt{ensure\ L\ e_0}$ $\quad$ $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \bullet \vdash \mathtt{e_0} : \mathtt{C}$
$\mathtt{L\ req}\ \Lambda'$ $\quad$ $\Lambda' \subseteq \{\overline{\mathtt{L}}\}$

By the induction hypothesis, either $\mathtt{e_0}$ is a value, in which case R-ENSUREVAL finishes the case, or there exists $\mathtt{e_0}'$ such that $\mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}}) \vdash \mathtt{e_0} \longrightarrow \mathtt{e_0}'$, in which case RC-ENSURE finishes the case (notice that $\mathit{ensure}(\mathtt{L}, \overline{\mathtt{L}})$ *wf*).

Case T-INKVA**:**

Similar to (the second half of) the case for T-INVK. □