A Type System for First-Class Layers with Inheritance, Subtyping, and Swapping[☆]

Hiroaki Inoue, Atsushi Igarashi

Graduate School of Informatics
Kyoto University

Abstract

Context-Oriented Programming (COP) is a programming paradigm to encourage modularization of context-dependent software. Key features of COP are *layers*—modules to describe context-dependent behavioral variations of a software system—and their *dynamic activation*, which can modify the behavior of multiple objects that have already been instantiated. Typechecking programs written in a COP language is difficult because the activation of a layer can even change objects' interfaces. Inoue et al. have informally discussed how to make JCop, an extension of Java for COP by Appeltauer et al., type-safe.

In this article, we formalize a small COP language called ContextFJ$_{<:}$ with its operational semantics and type system and show its type soundness. The language models main features of the type-safe version of JCop, including dynamically activated *first-class* layers, *inheritance* of layer definitions, layer *subtyping*, and layer *swapping*.

Keywords: Context-oriented programming, dynamic layer composition, first-class layers, layer inheritance, type systems

1. Introduction

Software is much more interactive than it used to be: it interacts with not only users but also external resources such as network and sensors and changes its behavior according to inputs from these resources. For example, an e-mail reader may switch to a text-based mode when network throughput is low. Such external information that affects the behavior of software is often referred to as *contexts* and software that is aware of contexts as context-dependent software. However, context-dependent software is hard to develop and maintain, because the description of context-dependent behavior, which we desire to be modularized, often crosscuts with the dominating module structure. To address such a problem from a programming-language perspective, Context-Oriented Programming (COP) has been proposed by Hirschfeld et al [22].

[☆]This is a revised and extended version of the paper "A Sound Type System for Layer Subtyping and Dynamically Activated First-Class Layers," presented at the 13th Asian Symposium on Programming Languages and Systems (APLAS 2015).

Email addresses: hinoue@kuis.kyoto-u.ac.jp (Hiroaki Inoue), igarashi@kuis.kyoto-u.ac.jp (Atsushi Igarashi)

The main language constructs for COP are *layers*, which are modules to specify context-dependent behavior, and their *dynamic layer activation*. A layer is basically a collection of what are called *partial methods*, which add new behavior to existing objects or override existing methods. When a layer is activated at run time by a designated construct, the partial methods defined in it become effective, changing the behavior of objects until the activation ends. Roughly speaking, a layer abstracts a context and dynamic layer activation abstracts change of contexts.

The JCop language [2] is an extension of Java with language constructs for COP. Not only does it support basic COP constructs described above, but also it introduces many advanced features such as inheritance of layer implementations and first-class layers. However, typechecking implemented in the JCop compiler does not take into account the fact that layer activation can change objects' interface by partial methods that add new methods and, as a result, not all "method not found" errors are prevented statically. In our previous work [27], we have studied this problem, proposed a type-safe version of JCop (we call Safe JCop in this paper) with informal discussions on its type system.

In this paper, we formalize most of the ideas proposed in the previous work and prove that they really make the language sound. More concretely, we develop a small COP language called ContextFJ$_{<:}$, which extends ContextFJ by Igarashi, Hirschfeld, and Masuhara [23, 24] to layer inheritance, subtyping of layer types, first-class layers, and a type-safe layer deactivation mechanism called layer swapping [27]; and we prove a type soundness theorem for ContextFJ$_{<:}$. Main issues we have to deal with are (1) the semantics of layer inheritance, which adds another "dimension" to the space of method lookup, (2) sound subtyping for first-class layers, which led us to two kinds of subtyping relation, and (3) layer swapping. A preliminary version of this work has been presented elsewhere [26] under the title "A Sound Type System for Layer Subtyping and Dynamically Activated First-Class Layers." We have extended ContextFJ$_{<:}$ given there with `superproceed` calls, which have been omitted, added proofs, and substantially revised the paper.

The rest of the article is organized as follows. After informally reviewing features of Safe JCop in Section 2, we develop ContextFJ$_{<:}$ with its syntax, operational semantics, and type system in Section 3; and we prove type soundness in Section 4. In Section 5, we discuss related work and then conclude in Section 6.

2. Language Constructs of Safe JCop

In this section, we review language constructs of Safe JCop, first described in [27], including first-class layers, layer inheritance/subtyping, and layer swapping along informal discussions about the type system.

As a running example, we consider programming a graphical computer game called *RetroAdventure* [3]. In this game, a player has a character "hero" that wanders around the game world. Here, we introduce class `Hero` that represents the hero, which has method `move` to walk around, and class `World` that represents the game world.

```
public class Hero {
  Position pos;
  public void move(Direction dir){
    pos = /* changes pos according to dir */;
  }
}
public class World { ... }
```

2.1. Layers and Partial Methods

As mentioned already, a first distinctive feature of COP is *layers*—collections of *partial methods* to modify the behavior of existing objects. A partial method is syntactically similar to an ordinary method declared in a class, except that the name is given in a qualified form `Hero.move()`; this means the partial method is going to override method `move` defined in `Hero` or (if it does not exist) add to `Hero`. A layer can contain partial methods for different classes, so, when it is activated, it can affect objects from various classes at once. Similarly to `super` calls in Java, the body of a partial method can contain `proceed` calls to invoke the original method overridden by this partial method.

Here, suppose that the hero's behavior is influenced by weather conditions in the game world. For example, in a foggy weather, the hero gets slow and, in a stormy weather, the hero cannot move as he likes. Here are layers that denote weathers of the game world.

```
public layer Foggy {
  /* partial method */
  public void Hero.move(Direction dir){
    pos = /* the distance of move is shorter */;
  }
}
public layer Stormy {
  /* partial method */
  public void Hero.move(Direction dir){
    proceed(randomDirection(dir));
  }
  /* baseless partial method */
  public Direction Hero.randomDirection(Direction dir){
    return /* add randomness to dir */;
  }
}
public layer Sunny { ... }
```

`Foggy` and `Stormy` have the definitions of `Hero.move`, which change the behavior of the original definition in different ways. In particular, `Hero.move` in `Stormy` uses `proceed`, replacing the arguments to calls to `move`. It also has `Hero.randomDirection`, used to determine a new randomized direction to which the hero is going to move.

Methods defined in classes are often referred to as *base methods* and partial methods without corresponding base methods as *baseless partial methods*. Notice that activating a layer with baseless partial methods extends object interfaces and `proceed` in a baseless partial method is unsafe unless another layer activation provides a baseless partial method of the same signature.

2.2. Layer Activation and First-Class Layers

In Safe JCop, a layer can be activated by using a layer instance (created by a `new` expression, just as an ordinary Java object, from a layer definition) in a `with` statement. The following code snippet shows how `Foggy` can be activated.

```
with(new Foggy()){
  hero.move(); /* The hero will get slow by Rainy weather. */
}
```

Inside the body of `with`, dynamic method dispatch is affected by the activated layers so that partial methods are looked up first. So, `movement` of the `hero` will be slow.

Layer activation has a dynamic extent in the sense that the behavior of objects changes even in methods called from inside `with`. If more than one layer is activated, a more recent activation has precedence and a `proceed` call in a more recently activated layer may call another partial method (of the same name) in another layer.

In Safe JCop, a layer instance is a first-class citizen and can be stored in a variable, passed to, or returned from a method. A layer name can be used as a type. Combining with layer subtyping discussed later, we can switch layers to activate by a run-time condition. For example, suppose that the game has *difficulty* levels, determined at run time according to some parameters, and each level is represented by an instance of a sublayer of `Difficulty`. Then, we can set the initial difficulty level by code like this:

```
Difficulty diff = /* an expression to compute difficulty */ ;
with(diff){ ... }
```

Moreover, a layer can declare own fields and methods (although we do not model them in layers in this article). So, first-class layers significantly enhance expressiveness of the language.

2.3. Dependencies between Layers

Baseless partial methods and layer activation that has dynamic extent pose a challenge on typechecking because activation of a layer including baseless partial methods can change object interfaces. So, a method invocation, including a `proceed` call, may or may not be safe depending on what layers are activated at the program point. Safe JCop adopts `requires` clauses [24] for layer definitions to express which layers should have been activated before activating each layer (instance). The type system checks whether each activation satisfies the `requires` clause associated to the activated layer and also uses `requires` clauses to estimate interfaces of objects at every program point.

For example, consider another layer `ThunderInStorm`, which expresses an event in a game. It affects the way how the hero's direction is randomized during a storm and includes a baseless partial method with a `proceed` call. To prevent `ThunderInStorm` from being activated in a weather other than a storm, the layer `requires Stormy` as follows:

```
public layer ThunderInStorm requires Stormy {
  public Direction Hero.randomDirection(Direction dir){
    Direction tmpd = proceed(dir);
    ... /* change tmpd to speed up */
    return tmpd;
  }
}
```

An attempt at activating `ThunderInStorm` without activating `Stormy` will be rejected by the type system (unless the activation appears in a layer requiring `Stormy`). Thanks to the `requires` clause, the type system knows that the `proceed` call will not fail. (It will call the partial method of the same name in `Stormy` or some other depending on what layers are activated at run time.)

2.4. Layer Inheritance and Subtyping

In Safe JCop, a layer can inherit definitions from another layer by using the keyword `extends` and the `extends` relation between layers yields subtyping, just like Java classes. If weather lay-

ers have many definitions in common, it is a good idea to define a superlayer Weather and concrete weather layers as its sublayers.

```
public layer Weather {
  public Text People.sayWeather(){ return new Text(""); }
  ...
}
public layer Stormy extends Weather {
  public Text People.sayWeather(){
    Text buf = superproceed();
    buf.setText("It's stormy today.");
    return buf;
  } ...
}
public layer Foggy extends Weather {
  public Text People.sayWeather(){ ... }
  ...
}
```

Here, Weather provides (baseless) partial method sayWeather to the class People, which returns Text data that people say about weather condition. The implementation of People.sayWeather just returns an empty Text and sublayers of the Weather override it. Safe JCop provides superproceed, which calls a partial method overridden because of layer inheritance. The partial method of Stormy sets the contents of the text using superproceed.

Since class subtyping equals to the reflexive and transitive closure of the extends relation, we expect layer subtyping to be the same; an instance of a sublayer can be substituted for that of its superlayer. However, substitutability is more subtle than one might expect and we are led to distinguishing two kinds of substitutability and introducing two kinds of subtyping relation, called weak and normal subtyping. The difference arises from requires clauses. To explain the issue, we define layer Thunder, which is the superlayer of ThunderInStorm and ThunderInFog and a sublayer of a marker layer Event.

```
public layer Event { ... }
public layer Thunder extends Event requires Weather {
  public void change_font(Text label){ label.setFont("Italic"); }
  public Text People.sayWeather(){ change_font(proceed()); }
}
public layer ThunderInStorm extends Thunder requires Stormy {
  public Text People.sayWeather(){
    Text buf = superproceed();
    buf.setText("Escape from here right now!!");
    return buf;
  }
  ...
}
public layer ThunderInFog extends Thunder requires Foggy {
  public Text People.sayWeather(){ ... }
}
```

Thunder changes the font of the text of what People say. It seems natural to set the requires clause of Thunder to be Weather, since its two sublayers require Stormy and Foggy respectively.

Weak subtyping. An instance of a sublayer can be used where a superlayer is `required`, since a sublayer defines more partial methods than its superlayer. For example, to activate the following layer called `Thunder`, which `requires Weather`, it suffices to activate `Foggy`, a sublayer of `Weather`, beforehand.

```
with(new Foggy()){
  // Thunder requires Weather and Foggy extends Weather
  with(new Thunder()){ ... }
}
```

We will formalize substitutability about `requires` as *weak subtyping*, which is the reflexive transitive closure of the `extends` relation between layer types. For the weak subtyping to work, we require that a sublayer declare, at least, what its superlayer `requires` because partial methods inherited from the superlayer may depend on them. We could relax this condition if a sublayer overrides all the partial methods but such a case is expected to be rare and so not taken into account.[1]

Normal subtyping. The above notion of subtyping is called weak because it does *not* guarantee safe substitutability for *first-class* layers. Consider layer `Difficulty` again and assume that it requires no other layers and has sublayers `Easy` and `Hard`. In the following code snippet, the activation of `diff` appears safe because its static type `Difficulty` does not require any layers to have been activated.

```
Difficulty diff = someCondition() ? new Easy() : new Hard();
with(diff){ ... }
```

However, the case where `Easy` or `Hard` requires some layers breaks the expected invariant that the dependency expressed by the `requires` clauses is satisfied at run time. So, for assignments and parameter passing, we need one more condition for subtyping, namely, `requires` of a sublayer must be the same as that of its superlayer. We call this strong notion of subtyping *normal subtyping*.

In Fig. 1, we show the layer subtyping hierarchy of the examples so far. An oval means a layer and the notation `req {X}` beside an oval means its `requiring` layers. Just like `Object` in Java, there is `Base`, which is a superlayer of all layers, in Safe JCop. If a layer omits the `extends` clause, it is implicitly assumed that the layer `extends Base`.

2.5. Layer Swapping and Deactivation

The original JCop provides constructs to *de*activate layers. However, only with `requires`, it is not easy to guarantee that layer deactivation does not lead to an error. For safe deactivation, it has to be checked that there is no layer that `requires` the deactivated layer, but the type system is not designed to keep track of the *absence* of certain layers. Instead of general-purpose layer deactivation mechanisms, Safe JCop introduces a special construct to express one important idiom that uses deactivation, namely *layer swapping* to deactivate some layers and activate a layer at once.

In Safe JCop, we can define a layer as *swappable*, which means that all its sublayers can be swapped with each other, by adding the modifier `swappable`. The `swap` statement for layer swapping is of the following form:

[1] Re-typechecking inherited methods under the new `requires` clause would be another way to relax this condition but this is against modular checking.

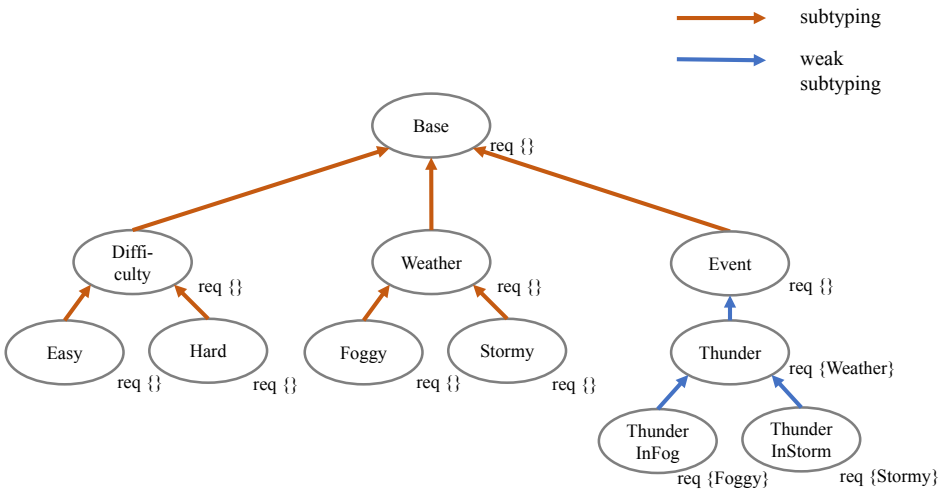

Figure 1: An example of layer subtyping hierarchy.

`swap(`*activation_layer*, *deactivation_layer_type*`){ ... }`

The *activation_layer* is an expression whose static type must be a sublayer of *deactivation_layer_type*, which in turn has to be swappable. It deactivates *all* instances of *deactivation_layer_type* (and its sublayers), and activates the *activation_layer*.

Let's consider `Difficulty` once again. We could define `Difficulty` as a `swappable` layer and use `swap` to switch to another mode temporarily.

```
swappable layer Difficulty { ... }
...
Difficulty diff = someCondition() ? new Easy() : new Hard();
with(diff){
  ...
  swap(new Hard(), Difficulty){
    ... // Enforce hard mode
  }
}
```

Unfortunately, for type safety, the necessary restriction for layer swapping turns out to be stronger than discussed in the previous work [27]: No sublayers of a swappable layer can be `required` by other layers or can change their interfaces or `requires` clauses from the swappable layer.

2.6. Method Lookup

We informally explain how Safe JCop's method lookup mechanism works, before proceeding to the formal calculus.

When method `m` is invoked on an instance of class `C` with layers $\mathtt{L}_1;\cdots;\mathtt{L}_n$ activated, the corresponding method definition is sought as follows: first, the activated layers $\mathtt{L}_n$, $\mathtt{L}_{n-1}$, down

to $\mathtt{L}_1$ are searched (in this order) for a partial method named `C.m`; if `C.m` is not found, the base class `C` is searched for the base definition; if `m` is not found, similar search continues on the `C`'s superclass `D`—namely, the activated layers are searched again for a partial method named `D.m` and the base class `D` is searched for the base definition, and so on. In addition to the usual inheritance chain in class-based object-oriented languages, COP adds another dimension to the space of method lookup. Actually, there is yet another dimension in (Safe) JCop because of layer inheritance: When $\mathtt{L}_i$ is searched for a partial method, its superlayers are searched, too, before going to $\mathtt{L}_{i-1}$. For example, under the following class and layer definitions

```
class C extends D   { }
class D extends E   { void m(){ ... } }
class E             { void m(){ ... } }
layer L1            { void D.m(){ ... } }
layer L2 extends L3 { void E.m(){ ... } }
layer L3            { void C.m(){ ... } }
```

the following statement

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

will execute partial method `C.m` defined in `L3`, whereas the statement

```
with(new L1()) { new C().m(); }
```

will execute `D.m` in `L1`.

Now, we turn our attention to the semantics of `super`, `proceed`, and `superproceed`. When a `super`, `proceed` or `superproceed` call is encountered during execution of a (partial) method, it continues to look for a method definition of the same name as follows.

Suppose that `C.m` is found in layer $\mathtt{L}_i$ with layers $\overline{\mathtt{L}} = \mathtt{L}_1; \cdots; \mathtt{L}_n$ activated ($0 < i \leq n$) and that `D` is a superclass of `C`.

- A call `super.m()` starts looking for a partial method `D.m` from $\mathtt{L}_n$ and so on.
- A `proceed` call starts looking for a partial method `C.m` from $\mathtt{L}_{i-1}$ or the base method of class `C` (when $i = 1$), and so on.
- A `superproceed` call starts looking for `C.m` in $\mathtt{L}_i'$ (where $\mathtt{L}_i'$ is the superlayer of $\mathtt{L}_i$), $\mathtt{L}_i''$ (where $\mathtt{L}_i''$ is the superlayer of $\mathtt{L}_i'$), and so on. If `C.m` is not found in the superlayers, it is a run-time error (which the type system will prevent).

For example, consider the following class and layer definitions and suppose `L1`, `L2` and `L3` are activated in this order. (In what follows, the notation `L.C.m` means the partial method `C.m` defined in layer `L`.)

```
class C extends D { }
class D extends E { void m(){ return super.m();} }
class E           { void m(){ return; } }
layer L1 {
  void C.m(){ return e1; }
  void D.m(){ return e2; }
}
layer L2 {
  void C.m(){ return e3; }
}
layer L4 {
  void C.m(){ return e5; }
  void E.m(){ return e6; }
}
layer L3 extends L4 {
  void C.m(){ return e4; }
}
```

- `super.m` calls from `L4.C.m` and `L1.C.m` will invoke `L1.D.m`; and ones from `L1.D.m` and `D.m` will invoke `L4.E.m`, since `L3` inherits `E.m` from `L4`.
- `proceed` calls from `L4.C.m` will invoke `L2.C.m` and ones from `L1.C.m` will invoke `L1.D.m`.
- `superproceed` calls from `L3.C.m` will invoke `L4.C.m`.

Fig. 2 summarizes how `super`, `proceed`, and `superproceed` calls are resolved. Each ball represents a (partial) method definition and its location where it is put. The three axes stands for class inheritance (`C` extends `D` and `D` extends `E`), activated layers (`L1`, `L2`, and `L3` are activated in this order), and layer inheritance (`L3` extends `L4`). Orange arrows represent how `proceed` calls at each method definition are resolved. For example, the top-most long orange arrow means that `proceed` from `L4.E.m` will invoke `E.m`. Green arrows represent `super` and blue `superproceed`.

Finally, we should note that, for `super`, `proceed`, and `superproceed` calls, the activated layers are the same as those when the current method is found. So, `with` or `swap` around `super`, `proceed`, and `superproceed` does not affect which definition is invoked.

3. ContextFJ$_{<:}$

In this section, we formalize a core functional subset of Safe JCop as ContextFJ$_{<:}$ with its syntax, operational semantics and type system. ContextFJ$_{<:}$, a descendant of Featherweight Java (FJ) [25], extends ContextFJ [23, 24] with layer inheritance, `superproceed`, layer subtyping, first-class layers, and swappable layers. JCop features that ContextFJ$_{<:}$ does *not* model for simplicity include: fields and (ordinary) methods in layers, special variable `thislayer` to refer to the current layer instance, `superlayer` to invoke an ordinary method in a superlayer, and declarative layer composition.

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; `x` and `y` over variables, which contains special variable `this`. The abstract syntax of ContextFJ$_{<:}$ is given in Fig. 3.

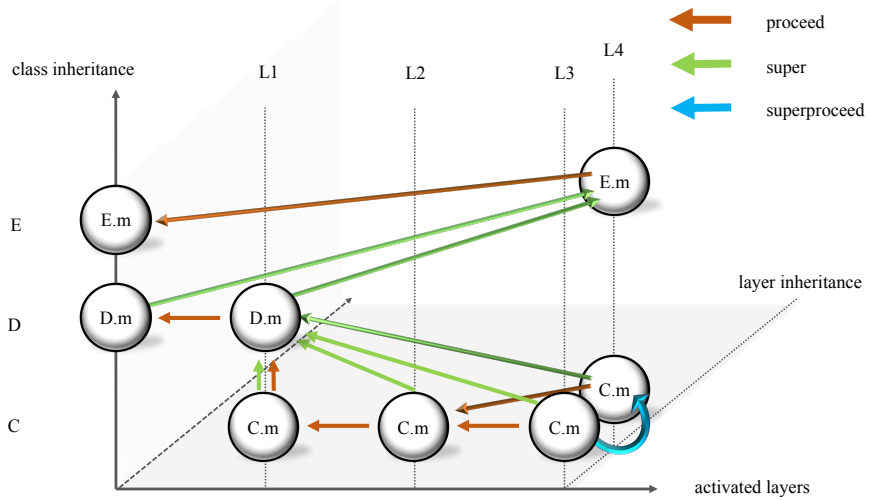

Figure 2: Method Lookup Example.

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 $\overline{\texttt{T}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by $\bullet$. Concatenation of sequences is often denoted by a comma except for layer names, for which we use a semicolon. We also abbreviate pairs of sequences, writing "$\overline{\texttt{T}}\ \overline{\texttt{f}}$" for "$\texttt{T}_1\ \texttt{f}_1, \cdots, \texttt{T}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{T}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for the sequence of declarations "$\texttt{T}_1\ \texttt{f}_1\texttt{;} \ldots \texttt{T}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;} \ldots \texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". Given layer sequence $\overline{\texttt{L}}$, We write $\{\overline{\texttt{L}}\}$ for the set of layers (obtained by ignoring the order). Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

We briefly explain the syntax, focusing on COP-related constructs. A layer definition `LA` consists of optional modifier `swappable`, its name, its superlayer name, layers that it `requires`, and partial methods. A partial method (defined as `PM`) is similar to a method but specifies which `m` to modify by qualifying the simple method name with a class name `C`.

Instantiation can be a layer instance `new L()`, as well as a class instance `new C(`$\overline{\texttt{e}}$`)`. Note that arguments to `new L` are always empty because ContextFJ$_{<:}$ does not model fields of layer instances. In the expression `with` $\texttt{e}_1\ \texttt{e}_2$, $\texttt{e}_1$ stands for the layer to be activated and $\texttt{e}_2$ the body of `with`. In the expression `swap (`$\texttt{e}_1$`, L)` $\texttt{e}_2$, $\texttt{e}_1$ means the layer to be activated, `L` the swappable layer, $\texttt{e}_2$ the body of `swap`. By this expression, during the evaluation of $\texttt{e}_2$, all instances of the swappable layer `L` and its sublayers are deactivated, and $\texttt{e}_1$ is activated. `super.m(`$\overline{\texttt{e}}$`)`, `proceed(`$\overline{\texttt{e}}$`)` and `superproceed(`$\overline{\texttt{e}}$`)` are keywords to invoke methods of the superclass, a previously activated layer, and the superlayer, respectively.

Expressions `new C(`$\overline{\texttt{v}}$`)<D,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)` and `new C(`$\overline{\texttt{v}}$`)<D,L,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)` are special runtime expressions that are related to method invocation mechanism of COP, and not supposed to appear in classes and layers. They basically mean that `m` is going to be invoked on `new C(`$\overline{\texttt{v}}$`)`. The annotation `<D,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>` is used to model `super` and `proceed` whereas `<D,L,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>` is used for `superproceed`. $\overline{\texttt{L}}$ stands for a sequence of activated layers and `D`, `L` and $\overline{\texttt{L}'}$ (which is assumed to

T	::=	C | L	(*types*)
CL	::=	class C ◁ C { $\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$ }	(*classes*)
LA	::=	[swappable] layer L ◁ L req $\overline{\texttt{L}}$ { $\overline{\texttt{PM}}$ }	(*layers*)
K	::=	C($\overline{\texttt{T}}$ $\overline{\texttt{f}}$){ super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$ = $\overline{\texttt{f}}$; }	(*constructors*)
M	::=	T m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){ return e; }	(*methods*)
PM	::=	T C.m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){ return e; }	(*partial methods*)
e, d	::=	x | e.f | e.m($\overline{\texttt{e}}$) | new T($\overline{\texttt{e}}$) | with e e | swap (e,L) e	(*expressions*)
		| proceed($\overline{\texttt{e}}$) | super.m($\overline{\texttt{e}}$) | superproceed($\overline{\texttt{e}}$)	
		| new C($\overline{\texttt{v}}$)<C,$\overline{\texttt{L}}$,$\overline{\texttt{L}}$>.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{v}}$)<C,L,$\overline{\texttt{L}}$,$\overline{\texttt{L}}$>.m($\overline{\texttt{e}}$)	
v, w	::=	new C($\overline{\texttt{v}}$) | new L()	(*values*)

Figure 3: ContextFJ$_{<:}$: Syntax.

be a prefix of $\overline{\texttt{L}}$) play a role of a "cursor" where the method lookup starts from. We explain how they work in detail in Section 3.2.

Program. A ContextFJ$_{<:}$ program (CT, LT, e) consists of a class table CT, a layer table LT and an expression e, which stands for the body of the `main` method. CT maps a class name to a class definition and LT a layer name to a layer definition. A layer definition can be regarded as a function that maps a partial method name C.m to a partial method definition. So, we can view LT as a Curried function, and we often write LT(L)(C.m) for the partial method C.m in L in a program. We assume that the domains of CT and LT are finite. Precisely speaking, the semantics and type system are parameterized over CT and LT but, to lighten the notation, we assume them to be fixed and omit from judgments.

Given CT and LT, `extends` and `requires` clauses are considered relations, written ◁ and `req`, respectively, over class/layer names. Namely, we write L `req` $\texttt{L}_i$ if LT(L) = `layer L req` $\overline{\texttt{L}}$ and $\texttt{L}_i \in \overline{\texttt{L}}$. We also write L `req` {$\overline{\texttt{L}}$} if LT(L) = `layer L req` $\overline{\texttt{L}}$.[2] As usual, we write $\mathcal{R}^+$ for the transitive closure of relation $\mathcal{R}$; similarly for $\mathcal{R}^*$ for the reflexive transitive closure of $\mathcal{R}$. We write L `swappable` if LT(L) is defined with the `swappable` modifier.

We assume the following sanity conditions are satisfied by a given program:

1. CT(C) = `class C ...` for any C ∈ *dom*(CT).
2. `Object` ∉ *dom*(CT).
3. For every class name C (except `Object`) appearing anywhere in CT, C ∈ *dom*(CT).
4. LT(L) = `... layer L ...` for any L ∈ *dom*(LT).
5. `Base` ∉ *dom*(LT).
6. For every layer name L (except `Base`) appearing anywhere in LT, L ∈ *dom*(LT).
7. Both for classes and layers, there are no cycles in the transitive closure of the `extends` clauses.
8. LT(L)(C.m) = `... C.m(...){...}` for any L ∈ *dom*(LT) and C ≠ `Object` and (C.m) ∈ *dom*(LT(L)).

[2]Note that $\texttt{L}_1$ `req` $\texttt{L}_2$ and $\texttt{L}_1$ `req` {$\texttt{L}_2$} have slightly different meanings; the former means $\texttt{L}_2$ is one of the layers required by $\texttt{L}_1$, whereas the latter means $\texttt{L}_2$ is the only layer required by $\texttt{L}_1$.

These sanity conditions are an extension of those of FJ: conditions for layers (4–7) are similar to those for classes (1–3, 7). In Condition 6, like `Object` of classes, layer `Base` is defined as the root of the layer inheritance/subtyping hierarchy. In the condition (8), $\texttt{C} \neq \texttt{Object}$ means that a layer cannot introduce a method to `Object`, which has no base methods. We could allow a layer to add methods to `Object` but doing so would just clutter presentation—there are more rules to deal with the fact that `super` calls cannot be made in partial methods for `Object`.

3.2. Operational Semantics

Lookup Functions. We need a few auxiliary lookup functions to define operational semantics and they are defined in Fig. 4. The function *fields*(C) returns a sequence $\overline{\texttt{T}}\ \overline{\texttt{f}}$ of pairs of a field name and its type by collecting all field declarations from C and its superclasses.

$\boxed{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}$

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

$$\frac{\texttt{class C} \lhd \texttt{D \{}\ \overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; ... \}} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{S}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}} \qquad \text{(F-Class)}$$

$\boxed{\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{x}}\texttt{.e in } \texttt{L}_0}$

$$\frac{\mathit{LT}(\texttt{L})(\texttt{C.m}) = \texttt{T}_0\ \texttt{C.m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e; \}}}{\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{x}}\texttt{.e in L}} \qquad \text{(PMB-Layer)}$$

$$\frac{\mathit{LT}(\texttt{L})(\texttt{C.m}) \text{ undefined} \qquad \texttt{L} \lhd \texttt{L}' \qquad \mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}') = \overline{\texttt{x}}\texttt{.e in } \texttt{L}_0}{\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{x}}\texttt{.e in } \texttt{L}_0} \qquad \text{(PMB-Super)}$$

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

$$\frac{\texttt{class C} \lhd \texttt{D \{ ...}\ \ \texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e; \} ... \}}}{\mathit{mbody}(\texttt{m}, \texttt{C}, \bullet, \overline{\texttt{L}}) = \overline{\texttt{x}}\texttt{.e in C}, \bullet} \qquad \text{(MB-Class)}$$

$$\frac{\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}_0) = \overline{\texttt{x}}\texttt{.e in } \texttt{L}_1}{\mathit{mbody}(\texttt{m}, \texttt{C}, (\overline{\texttt{L}'}; \texttt{L}_0), \overline{\texttt{L}}) = \overline{\texttt{x}}\texttt{.e in C}, (\overline{\texttt{L}'}; \texttt{L}_0)} \qquad \text{(MB-Layer)}$$

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

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

Figure 4: ContextFJ$_{<:}$: Lookup functions.

The function *pmbody*(m, C, L) returns the parameters and body $\overline{\texttt{x}}\texttt{.e}$ of the partial method `C.m` defined in layer L. It also returns the layer name $\texttt{L}_0$ at which `C.m` is found, which will be used in reduction rules to deal with `superproceed`. If partial method `C.m` is not found in L, its superlayer

$\mathtt{L}'$ is searched and so on. 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 sequence $\overline{\mathtt{L}}_2$ keeps track of the layers that are activated when the search initially started. It also returns D and $\overline{\mathtt{L}}''$ (which will be a prefix of $\overline{\mathtt{L}}_2$), information on where the method has been found. For example, in the rule MB-Layer, which means that the method is found in class C and layer $\mathtt{L}_0$ (or its superlayers), *mbody* returns C and $(\overline{\mathtt{L}}'; \mathtt{L}_0)$. Such information will be used in reduction rules to deal with `proceed` and `super`. Readers familiar with ContextFJ will notice that the rules for *mbody* are mostly the same as those in ContextFJ, except that *pmbody*$(\mathtt{m}, \mathtt{C}, \mathtt{L})$ is substituted for *PT*$(\mathtt{m}, \mathtt{C}, \mathtt{L})$ to take layer inheritance into account. By reading the four rules defining the two functions in a bottom-up manner, it is not hard to see the correspondence with the method lookup procedure, informally described in Section 2.6.

Reduction. 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 e′ under the activated layers $\overline{\mathtt{L}}$." The sequence $\overline{\mathtt{L}}$ of layer names stands for nesting of `with` and the rightmost name stands for the most recently activated layer. As for other sequences, $\overline{\mathtt{L}}$ do not contain duplicate names. Note that we put a sequence of layer names $\overline{\mathtt{L}}$ rather than layer instances because layer instances have no fields and `new L()` and L can be identified. If we modelled fields in layer instances, we would have to put instances for layer names.

Reduction rules are found in Fig. 5 and Fig. 6. R-Field is for field access and four rules R-InvkXX are for method invocation: R-Invk initializes the cursor according to the currently activated layers $\overline{\mathtt{L}}$; the rules R-InvkB and R-InvkP represent invocation of a base and partial method, respectively, depending on which kind is found by *mbody*; the rule R-InvkSP deals with the case where the cursor in the receiver object is a quadruple, which occurs when the entire expression was a `superproceed` call. In the last case, *pmbody* is used to find a method body because `superproceed` denotes a partial method in one of the superlayers.

Note how `this`, `proceed`, `super` and `superproceed` are replaced with the receiver with different cursor locations. For `proceed`, the cursor of triple moves one layer to the left and, for `super`, the cursor moves one level up in the direction of class inheritance, resetting the layers. Thanks to Sanity Condition (8), the superclass D is always found. If we allowed a layer to add baseless partial methods to `Object`, we would have to have special rules, in which there is no substitution for `super` (and typing rules to disallow the use of `super` in such partial methods). Igarashi et al. [24] (as well as the conference version of this article [26]) have overlooked this subtlety. For `superproceed`, the cursor moves one level up in the direction of layer inheritance (generating a quadruple from a triple in R-InvkP). For example, we show how cursors of a triple and a quadruple work using example in Section 2.6. Let e be `new C().m()`. Then, the derivation of $\mathtt{L1};\mathtt{L2};\mathtt{L3} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ will take the form:

$$\dfrac{\dfrac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, (\mathtt{L1};\mathtt{L2};\mathtt{L3}), (\mathtt{L1};\mathtt{L2};\mathtt{L3})) = \bullet.\mathtt{e4}\ \mathtt{in}\ \mathtt{C}, (\mathtt{L1};\mathtt{L2};\mathtt{L3})}{\mathtt{L1};\mathtt{L2};\mathtt{L3} \vdash \mathtt{new\ C<C,(L1;L2;L3),(L1;L2;L3)>().m()} \longrightarrow \mathtt{e}'}\ \text{R-InvkP}}{\mathtt{L1};\mathtt{L2};\mathtt{L3} \vdash \mathtt{new\ C().m()} \longrightarrow \mathtt{e}'}\ \text{R-Invk}$$

where e′ is

$$\left[\begin{array}{ll} \mathtt{new\ C<C,(L1;L2;L3),(L1;L2;L3)>()} & \mathtt{/this} \\ \mathtt{new\ C<D,(L1;L2;L3),(L1;L2;L3)>()} & \mathtt{/super} \\ \mathtt{new\ C<C,(L2;L3),(L1;L2;L3)>().m} & \mathtt{/proceed} \\ \mathtt{new\ C<C,L4,(L1;L2;L3),(L1;L2;L3)>().m} & \mathtt{/superproceed} \end{array}\right] \mathtt{e4}.$$

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

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

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}'',\bullet \qquad \mathtt{class\ C}''\mathtt{\triangleleft\ D\{...\}}}{\overline{\mathtt{L}} \vdash \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}'\mathtt{>.m}(\overline{\mathtt{w}}) \longrightarrow \begin{bmatrix} \mathtt{new\ C}(\overline{\mathtt{v}}) & \mathtt{/this},\\ \overline{\mathtt{w}} & /\overline{\mathtt{x}},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<D},\overline{\mathtt{L}}',\overline{\mathtt{L}}'\mathtt{>/super} & \end{bmatrix}\mathtt{e}_0} \quad \text{(R-InvkB)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}'',(\overline{\mathtt{L}}''';\mathtt{L}_0) \qquad \mathtt{class\ C}''\mathtt{\triangleleft\ D\{...\}} \qquad \mathtt{layer\ L}_0\mathtt{\triangleleft\ L}_1}{\overline{\mathtt{L}} \vdash \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}'\mathtt{>.m}(\overline{\mathtt{w}}) \longrightarrow \begin{bmatrix} \mathtt{new\ C}(\overline{\mathtt{v}}) & \mathtt{/this},\\ \overline{\mathtt{w}} & /\overline{\mathtt{x}},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}'',\overline{\mathtt{L}}''',\overline{\mathtt{L}}'\mathtt{>.m} & \mathtt{/proceed},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<D},\overline{\mathtt{L}}',\overline{\mathtt{L}}'\mathtt{>} & \mathtt{/super},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}'',\mathtt{L}_1,(\overline{\mathtt{L}}''';\mathtt{L}_0),\overline{\mathtt{L}}'\mathtt{>.m/superproceed} & \end{bmatrix}\mathtt{e}_0} \quad \text{(R-InvkP)}$$

$$\frac{\mathit{pmbody}(\mathtt{m},\mathtt{C}',\mathtt{L}_1) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{L}_2 \qquad \mathtt{class\ C}'\mathtt{\triangleleft\ D\{...\}} \qquad \mathtt{layer\ L}_2\mathtt{\triangleleft\ L}_3}{\overline{\mathtt{L}} \vdash \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}',\mathtt{L}_1,(\overline{\mathtt{L}}'';\mathtt{L}_0),\overline{\mathtt{L}}'\mathtt{>.m}(\overline{\mathtt{w}}) \longrightarrow \begin{bmatrix} \mathtt{new\ C}(\overline{\mathtt{v}}) & \mathtt{/this},\\ \overline{\mathtt{w}} & /\overline{\mathtt{x}},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}'\mathtt{>.m} & \mathtt{/proceed},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<D},\overline{\mathtt{L}}',\overline{\mathtt{L}}'\mathtt{>} & \mathtt{/super},\\ \mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{<C}',\mathtt{L}_3,(\overline{\mathtt{L}}'';\mathtt{L}_0),\overline{\mathtt{L}}'\mathtt{>.m/superproceed} & \end{bmatrix}\mathtt{e}_0} \quad \text{(R-InvkSP)}$$

Figure 5: ContextFJ$_{<:}$: Reduction Rules 1.

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

$$\frac{\mathit{swap}(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}}' \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{\overline{\mathtt{L}} \vdash \mathtt{swap\ (new\ L(),}\mathtt{L}_{sw}\mathtt{)\ e} \longrightarrow \mathtt{swap\ (new\ L(),}\mathtt{L}_{sw}\mathtt{)\ e}'} \quad \text{(RC-Swap)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_l \longrightarrow \mathtt{e}_l'}{\overline{\mathtt{L}} \vdash \mathtt{with\ e}_l\ \mathtt{e} \longrightarrow \mathtt{with\ e}_l'\ \mathtt{e}} \quad \text{(RC-WithArg)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_l \longrightarrow \mathtt{e}_l'}{\overline{\mathtt{L}} \vdash \mathtt{swap\ (e}_l\mathtt{,L}_{sw}\mathtt{)\ e} \longrightarrow \mathtt{swap\ (e}_l'\mathtt{,L}_{sw}\mathtt{)\ e}} \quad \text{(RC-SwapArg)}$$

$$\frac{}{\overline{\mathtt{L}} \vdash \mathtt{with\ new\ L()\ v} \longrightarrow \mathtt{v}} \quad \text{(R-WithVal)}$$

$$\frac{}{\overline{\mathtt{L}} \vdash \mathtt{swap\ (new\ L(),\ L}_{sw}\mathtt{)\ v} \longrightarrow \mathtt{v}} \quad \text{(R-SwapVal)}$$

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

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

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

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

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',}\overline{\mathtt{L}}\mathtt{'',}\overline{\mathtt{L}}\mathtt{'>.m(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',}\overline{\mathtt{L}}\mathtt{',}\overline{\mathtt{L}}\mathtt{'>.m(..,e}_i'\mathtt{,..)}} \quad \text{(RC-InvkAArg1)}$$

$$\frac{\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\overline{\mathtt{L}} \vdash \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',L,}\overline{\mathtt{L}}\mathtt{'',}\overline{\mathtt{L}}\mathtt{'>.m(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<C',L,}\overline{\mathtt{L}}\mathtt{',}\overline{\mathtt{L}}\mathtt{'>.m(..,e}_i'\mathtt{,..)}} \quad \text{(RC-InvkAArg2)}$$

Figure 6: ContextFJ$_{<:}$: Reduction Rules 2.

Now, we go back to Fig. 6. The rules RC-With and RC-Swap express layer activation and swapping, respectively. The auxiliary functions $with(\mathtt{L}, \overline{\mathtt{L}})$ and $swap(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}})$ for context manipulation are defined by:

$$with(\mathtt{L}, \overline{\mathtt{L}}) \;=\; (\overline{\mathtt{L}} \setminus \{\mathtt{L}\}); \mathtt{L} \qquad swap(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}}) \;=\; (\overline{\mathtt{L}} \setminus \{\mathtt{L}' \mid \mathtt{L}' \lhd^* \mathtt{L}_{sw}\}); \mathtt{L}$$

The function *with* removes L (if exists) from layer sequence $\overline{\mathtt{L}}$ and adds L to the end of $\overline{\mathtt{L}}$ and *swap* removes all sublayers of $\mathtt{L}_{sw}$ from $\overline{\mathtt{L}}$, and adds L to the end of $\overline{\mathtt{L}}$.[3] The type system checks that $\mathtt{L}_{sw}$ is a `swappable` layer. Based on the above, the rule RC-With means that `with (new L()) e` executes `e` with L activated (as the first layer). The rule RC-Swap is similar; it means that `swap (new L(), L`$_{sw}$`) e` executes by deactivating all sublayers of $\mathtt{L}_{sw}$ and activating layer L. For example, we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathtt{L1;L2;L3} \vdash \mathtt{new\ C().m()} \longrightarrow \mathtt{e'}}\ \text{R-Invk}}{\mathtt{L1;L2} \vdash \mathtt{with\ new\ L3()\ e} \longrightarrow \mathtt{e'}}\ \text{R-With}}{\mathtt{L1} \vdash \mathtt{with\ new\ L2()\ (with\ new\ L3()\ e)} \longrightarrow \mathtt{e'}}\ \text{R-With}}{\bullet \vdash \mathtt{with\ new\ L1()\ (with\ new\ L2()\ (with\ new\ L3()\ e))} \longrightarrow \mathtt{e'}}\ \text{R-With}$$

The rules RC-WithArg and RC-SwapArg are for reduction of expression $\mathtt{e}_l$ that is expected to become a layer instance. Rules RC-WithVal and RC-SwapVal are for final reduction steps of `with` and `swap` blocks, respectively, that pass the value v as it is. Other rules for congruence are same as those of ContextFJ: ContextFJ$_{<:}$ reduction is call by value but the order of reduction of subexpressions is unspecified.

3.3. Type System

As usual, the role of a type system is to ensure the absence of a certain class of run-time errors. Here, they are "field-not-found" and "method-not-found" errors, including the failure of `proceed`, `superproceed` or `super` calls.

As discussed in the last section, the type system takes information on activated layers at every program point into account. We approximate such information by a set Λ of layer names, which mean that, for any layer in Λ, an instance of one of its sublayers has to be activated at run time. This set gives underapproximation of activated layers; other layers might be activated. Activated layers are approximated by sets rather than sequences because the type system is mainly concerned about access to fields and methods and the order of activated layers does not influence which fields and methods are accessible.

In our type system, a type judgment for an expression is of the form $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$, where Γ is a type environment, which records types of variables, and $\mathcal{L}$ stands for where e appears, namely, a method in a class (denoted by `C.m`) or a partial method in a layer (denoted by `L.C.m`). For example, the `proceed` call in the body of the partial method `People.sayWeather()` of layer `Thunder` is typed as follows:

`Thunder.People.sayWeather`; {`Weather`, `Thunder`}; `this` : `People` ⊢ `proceed()` : `Text`

[3] The symbol \ is usually used to remove entities from a set, but we informally use it for a sequence here.

The layer name set {`Weather`, `Thunder`} comes from the fact that `Thunder` `requires` `Weather`. `Thunder` is also included because `Thunder` (or one of its sublayers) is obviously activated when a partial method defined in this very layer is executed.

We start with the definitions of two kinds of layer subtyping discussed in the last section and proceed to functions to look up method types and typing rules.

Subtyping. We define subtyping $\texttt{C} <: \texttt{D}$ for class types, weak subtyping $\texttt{L}_1 <:_w \texttt{L}_2$ and normal subtyping $\texttt{L}_1 <: \texttt{L}_2$ for layer types by the rules in Fig. 7. Class subtyping $\texttt{C} <: \texttt{D}$ is defined as the reflexive and transitive closure of $\lhd$, just as FJ. Weak layer subtyping is also the reflexive and transitive closure of $\lhd$. We extend it to the relation $\Lambda_1 <:_w \Lambda_2$ between layer name sets by LSS-Intro: $\Lambda_1 <:_w \Lambda_2$ if and only if for every element in Λ_2, there must exist a sublayer of it in Λ_1. It is used to check activated layers Λ_1 satisfy the requirement Λ_2 given by a `requires` clause in typechecking a layer activation. Normal subtyping is almost the reflexive and transitive closure of $\lhd$ but there is one additional condition: for $\texttt{L}_1$ to be a normal subtype of $\texttt{L}_2$, the layers they `require` must be the same (LS-Extends). Obviously, if $\texttt{L}_1 <: \texttt{L}_2$, then $\texttt{L}_1 <:_w \texttt{L}_2$ (but not vice versa).

class subtyping $<:$

$$\frac{}{\texttt{C} <: \texttt{C}} \quad \text{(CL-Refl)}$$

$$\frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}} \quad \text{(CL-Trans)}$$

$$\frac{\texttt{class C} \lhd \texttt{D \{...\}}}{\texttt{C} <: \texttt{D}} \quad \text{(CL-Extends)}$$

normal layer subtyping $<:$

$$\frac{}{\texttt{L} <: \texttt{L}} \quad \text{(LS-Refl)}$$

$$\frac{\texttt{L}_1 <: \texttt{L}_2 \qquad \texttt{L}_2 <: \texttt{L}_3}{\texttt{L}_1 <: \texttt{L}_3} \quad \text{(LS-Trans)}$$

$$\frac{\texttt{L} \lhd \texttt{Base} \qquad \texttt{L req } \emptyset}{\texttt{L} <: \texttt{Base}} \quad \text{(LS-Base)}$$

$$\frac{\texttt{L}_1 \lhd \texttt{L}_2 \qquad \texttt{L}_1 \texttt{ req } \Lambda \qquad \texttt{L}_2 \texttt{ req } \Lambda}{\texttt{L}_1 <: \texttt{L}_2} \quad \text{(LS-Extends)}$$

weak layer subtyping $<:_w$

$$\frac{}{\texttt{L} <:_w \texttt{L}} \quad \text{(LSw-Refl)}$$

$$\frac{\texttt{L}_1 <:_w \texttt{L}_2 \qquad \texttt{L}_2 <:_w \texttt{L}_3}{\texttt{L}_1 <:_w \texttt{L}_3} \quad \text{(LSw-Trans)}$$

$$\frac{\texttt{L}_1 \lhd \texttt{L}_2}{\texttt{L}_1 <:_w \texttt{L}_2} \quad \text{(LSw-Extends)}$$

layer set subtyping

$$\frac{\forall \texttt{L}_0 \in \Lambda_0 . \exists \texttt{L}_1 \in \Lambda_1 \text{ s.t. } \texttt{L}_1 <:_w \texttt{L}_0}{\Lambda_1 <:_w \Lambda_0} \quad \text{(LSS-Intro)}$$

Figure 7: ContextFJ$_{<:}$: Subtyping Relations.

Method type lookup. Similarly to *pmbody* and *mbody*, we define two auxiliary functions *pmtype* and *mtype* to look up the signature $\overline{\mathtt{T}}\to\mathtt{T}_0$ (consisting of argument type $\overline{\mathtt{T}}$ and a return type $\mathtt{T}_0$) of a (partial) method. *pmtype*$(\mathtt{m},\mathtt{C},\mathtt{L})$ returns the signature of `C.m` in `L` (or one of its superlayers). *mtype*$(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)$ returns the type of `m` in `C` under the assumption that Λ_1 is activated. The other layer set Λ_2 ($\supseteq\Lambda_1$) is used when the lookup goes on to a superclass. If Λ_1 and Λ_2 are the same, which is mostly the case, we write *mtype*$(\mathtt{m},\mathtt{C},\Lambda_1)$.

$\boxed{\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})=\overline{\mathtt{T}}\to\mathtt{T}_0}$

$$\frac{\mathit{LT}(\mathtt{L})(\mathtt{C.m})=\mathtt{T}_0\ \ \mathtt{C.m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}}{\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})=\overline{\mathtt{T}}\to\mathtt{T}_0}\qquad\text{(PMT-Layer)}$$

$$\frac{\mathit{LT}(\mathtt{L})(\mathtt{C.m})\text{ undefined}\qquad \mathtt{L}\lhd\mathtt{L}'\qquad \mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L}')=\overline{\mathtt{T}}\to\mathtt{T}_0}{\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})=\overline{\mathtt{T}}\to\mathtt{T}_0}\qquad\text{(PMT-Super)}$$

$\boxed{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\to\mathtt{T}_0}$

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

$$\frac{\exists\mathtt{L}\in\Lambda_1.\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})=\overline{\mathtt{T}}\to\mathtt{T}_0}{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\to\mathtt{T}_0}\qquad\text{(MT-PMethod)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C}\lhd\mathtt{D}\ \{\ldots\ \ \overline{\mathtt{M}}\ \}\qquad \mathtt{m}\notin\overline{\mathtt{M}}\\ \forall\mathtt{L}\in\Lambda_1.\mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L})\text{ undefined}\qquad \mathit{mtype}(\mathtt{m},\mathtt{D},\Lambda_2,\Lambda_2)=\overline{\mathtt{T}}\to\mathtt{T}_0\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C},\Lambda_1,\Lambda_2)=\overline{\mathtt{T}}\to\mathtt{T}_0}\qquad\text{(MT-Super)}$$

Figure 8: ContextFJ$_{<:}$: Method Type Lookup functions.

These rules by themselves do not define *mtype* as a function, because different layers may contain partial methods of the same name with different signatures. So, precisely speaking, it should rather be understood as a relation; in a well-typed program, it will behave as a function, though.

Expression Typing. As mentioned already, the type judgment for expressions is of the form $\mathcal{L};\Lambda;\Gamma\vdash\mathtt{e}:\mathtt{T}$, read "`e` is given type `T` under context Γ, location $\mathcal{L}$ and layer set Λ". In addition to `C.m` and `L.C.m`, $\mathcal{L}$ can be $\bullet$, which means the top-level (i.e., under execution). Typing rules are given in Fig. 9. We defer typing rules for run-time expressions `new C(`$\overline{\mathtt{v}}$`)<D,`$\overline{\mathtt{L}'}$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{e}}$`)` and `new C(`$\overline{\mathtt{v}}$`)<D,L,`$\overline{\mathtt{L}'}$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\mathtt{e}}$`)` to the next section and focus on expressions that appear class and layer definitions.

Rules T-Var, T-Field are easy. T-New and T-NewL are for instance of classes and instance of layers, respectively. The rule T-Invk is straightforward: the method signature $\overline{\mathtt{T}}\to\mathtt{T}_0$ is retrieved from the receiver type $\mathtt{C}_0$ and Λ; the types of the actual arguments must be subtypes of $\overline{\mathtt{T}}$; and the whole expression is given the method return type $\mathtt{T}_0$. The rule T-With checks, by $\Lambda<:_w\Lambda'$, that the layers `required` by `L`—the type of the layer to be activated—are already activated and that the body $\mathtt{e}_0$ is well typed under the assumption that `L` is additionally activated. T-Swap is

$\boxed{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}}$

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

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

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

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

$$\frac{}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{L}_0() : \mathtt{L}_0} \quad \text{(T-NewL)}$$

$$\frac{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_l : \mathtt{L} \qquad \mathtt{L}\ \mathtt{req}\ \Lambda' \qquad \Lambda <:_w \Lambda' \qquad \mathcal{L}; \Lambda \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{with}\ \mathtt{e}_l\ \mathtt{e}_0 : \mathtt{T}_0} \quad \text{(T-With)}$$

$$\frac{\begin{array}{c}\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e}_l : \mathtt{L} \qquad \mathtt{L} <:_w \mathtt{L}_{sw} \qquad \mathtt{L}_{sw}\ \mathtt{swappable} \qquad \mathtt{L}\ \mathtt{req}\ \Lambda' \\ \Lambda_{rm} = \Lambda \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\} \qquad \Lambda_{rm} <:_w \Lambda' \qquad \mathcal{L}; \Lambda_{rm} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0\end{array}}{\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{swap}\ (\mathtt{e}_l, \mathtt{L}_{sw})\ \mathtt{e}_0 : \mathtt{T}_0} \quad \text{(T-Swap)}$$

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

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

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

$$\frac{\mathtt{L} \lhd \mathtt{L}' \qquad \mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}') = \overline{\mathtt{T}} \to \mathtt{T}_0 \qquad \mathtt{L}.\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}}}{\mathtt{L}.\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \mathtt{superproceed}(\overline{\mathtt{e}}) : \mathtt{T}_0} \quad \text{(T-SuperProceed)}$$

Figure 9: ContextFJ$_{<:}$: Expression typing.

similar; the set Λ_{rm} stands for the set of layers after deactivation and must be a weak subtype of the required set Λ'. The last four rules are for `super`, `proceed`, and `superproceed` calls and so they are similar to T-Invk. Differences are in how the method signature is obtained. In the rules T-SuperB and T-SuperP for a `super` call in a method defined in a class and in a partial method, respectively, the superclass `E` is given to *mtype*. Layer names are taken from the `requires` clause instead of Λ—corresponding to the fact that the method to be invoked is not affected by `with` or `swap` surrounding `super` (a class cannot require any layer, hence the empty set). In the rule T-Proceed for a `proceed` call, the current class name `C` is used. Similarly to T-SuperP, layer names are taken from the `require` clause. The last argument to *mtype* is $\Lambda \cup \{\mathtt{L}\}$ because a `proceed` call can proceed to a partial method `D.m` (where `D` is a superclass of `C`) defined in the same layer `L`. In the rule T-SuperProceed, *pmtype* is used instead of *mtype*.

`M ok in C`

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

`PM ok in L`

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

`CL ok`

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\overline{\mathtt{S}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\ \mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this.}\overline{\mathtt{f}}=\overline{\mathtt{f}};\ \} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{ok\ in\ C}\end{array}}{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{ok}} \quad \text{(T-Class)}$$

`LA ok`

$$\frac{\begin{array}{c}\mathtt{L}\text{ is not sublayer of any }\mathtt{swappable}\text{ layer} \\ \mathtt{L'\ req}\ \Lambda' \qquad \{\overline{\mathtt{L}}\} <:_w \Lambda' \qquad \overline{\overline{\mathtt{PM}}}\ \mathtt{ok\ in\ L}\end{array}}{[\mathtt{swappable}]\ \mathtt{layer\ L\ req}\ \overline{\mathtt{L}} \triangleleft \mathtt{L'}\ \{\ \overline{\overline{\mathtt{PM}}}\ \}\ \mathtt{ok}} \quad \text{(T-Layer)}$$

$$\frac{\begin{array}{c}\mathtt{L} \triangleleft^{+} \mathtt{L}_{sw} \qquad \mathtt{L}_{sw}\ \mathtt{swappable} \\ \mathtt{L'\ req}\ \Lambda' \qquad \{\overline{\mathtt{L}}\} = \Lambda' \\ \overline{\overline{\mathtt{PM}}}\ \mathtt{ok\ in\ L} \qquad \forall \mathtt{C.m} \in \{\overline{\overline{\mathtt{PM}}}\}.\ \mathit{pmtype}(\mathtt{m},\mathtt{C},\mathtt{L}_{sw})\text{ defined} \\ \neg\exists \mathtt{L}_2 \in \mathit{dom}(\mathit{LT}).\mathtt{L}_2\ \mathtt{req\ L}\end{array}}{\mathtt{layer\ L\ req}\ \overline{\mathtt{L}} \triangleleft \mathtt{L'}\ \{\ \overline{\overline{\mathtt{PM}}}\ \}\ \mathtt{ok}} \quad \text{(T-LayerSW)}$$

Figure 10: ContextFJ$_{<:}$: Method/Class/Layer typing.

In Igarashi et al. [24], in which a type system for ContextFJ is developed, another layer activation construct called `ensure` is adopted. The difference from `with` is that, if an already activated layer is to be activated, `ensure` does not change the activated layer sequence, whereas `with` will pull that layer to the head of the sequence so that partial methods in it are invoked first. For example, activating layers `L1`, `L2`, `L1` in this order results in `L1;L2` with `ensure` but in `L2;L1`

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

$$\frac{\forall \mathtt{m}, \mathtt{C}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0.\ \text{if}\quad \mathit{LT}(\mathtt{L}_1)(\mathtt{C}.\mathtt{m}) = \mathtt{T}_0\ \ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ldots\} \\ \text{and } \mathit{LT}(\mathtt{L}_2)(\mathtt{C}.\mathtt{m}) = \mathtt{S}_0\ \ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{y}})\{\ldots\}, \text{then } \overline{\mathtt{T}}, \mathtt{T}_0 = \overline{\mathtt{S}}, \mathtt{S}_0}{\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2)}$$

$$\frac{\forall \mathtt{m}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0, \overline{\mathtt{x}}.\ \text{if}\quad \mathit{LT}(\mathtt{L})(\mathtt{C}.\mathtt{m}) = \mathtt{S}_0\ \ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ldots\} \\ \text{and } \mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \to \mathtt{T}_0, \text{then } \overline{\mathtt{T}}, \mathtt{T}_0 = \overline{\mathtt{S}}, \mathtt{S}_0}{\mathit{override}^h(\mathtt{L}, \mathtt{C})}$$

$$\frac{\forall \mathtt{m}, \mathtt{D}, \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{S}}, \mathtt{S}_0.\ \text{if } \mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\ \ \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \mathtt{x})\{\ldots\}\ldots\} \\ \text{and } \mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathit{dom}(\mathit{LT}), \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \to \mathtt{T}_0, \\ \text{then } \overline{\mathtt{T}} = \overline{\mathtt{S}} \text{ and } \mathtt{S}_0 <: \mathtt{T}_0}{\mathit{override}^v(\mathtt{C})}$$

$\vdash (\mathit{CT}, \mathit{LT})\ \mathtt{ok}$ $\vdash (\mathit{CT}, \mathit{LT}, \mathtt{e}) : \mathtt{T}$

$$\frac{\forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathit{CT}(\mathtt{C})\ \mathtt{ok} \qquad \forall \mathtt{L} \in \mathit{dom}(\mathit{LT}).\mathit{LT}(\mathtt{L})\ \mathtt{ok} \\ \forall \mathtt{L}_1, \mathtt{L}_2 \in \mathit{dom}(\mathit{LT}).\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_2) \\ \forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathtt{L} \in \mathit{dom}(\mathit{LT}).\mathit{override}^h(\mathtt{L}, \mathtt{C}) \qquad \forall \mathtt{C} \in \mathit{dom}(\mathit{CT}).\mathit{override}^v(\mathtt{C})}{\vdash (\mathit{CT}, \mathit{LT})\ \mathtt{ok}} \quad \text{(T-Table)}$$

$$\frac{\vdash (\mathit{CT}, \mathit{LT})\ \mathtt{ok} \qquad \bullet; \emptyset; \bullet \vdash \mathtt{e} : \mathtt{T}}{\vdash (\mathit{CT}, \mathit{LT}, \mathtt{e}) : \mathtt{T}} \quad \text{(T-Prog)}$$

Figure 11: ContextFJ$_{<:}$: Program typing.

with the `with` statement. Igarashi et al. argue that the rearrangement of layers by `with` destroys the layer ordering in which interlayer dependency is respected. For example, if `L2 requires L1`, then `L2;L1` violates the `require` clause in the sense that the layers that `L2` requires do not precede `L2` in the sequence. So, for simplicity, Igarashi et al. considered only `ensure`, which does not have this problem.

Our discovery is that, in fact, this anomaly caused by `with` is not really a problem for type soundness and essentially the same typing rule works—Our typing rule T-With for `with` is indeed very similar to that for `ensure` in ContextFJ; the only difference is the use of $\subseteq$ in the place of weak subtyping $<:_w$ (ContextFJ does not have layer subtyping). The reason why a layer sequence like `L2;L1` is not problematic can be explained as follows. Actually, problematic would be a partial method defined in `L2` calling another (partial) method, say `C.m`, that exists only in `L1`—that is, one that is undefined in a base class—via `proceed`.[4] Such a dangling partial method cannot be executed, however: `C.m` in `L1` cannot contain `proceed`, which leads to execution of the dangling partial method, because `L1` is activated first, meaning that `L1` does not `require` any other layer, but it is assumed here that `m` is not defined in base class `C`.

[4] Invoking `m` via `this` or `super` will find `m` in `L1`.

Typing for Methods, Partial Methods, Classes, Layers, and Programs. Typing rules for (partial) methods, layers, and classes and are given in Fig. 10. The rule T-Method is standard. Readers familiar with FJ may notice that a condition for valid overriding is missing; it is put in elsewhere–see below. The rule T-PMethod for a partial method means that the method body $\mathtt{e}_0$ is typed under the layer set `required` by this layer. The rule T-Layer is for layers that are not sublayers of any `swappable` layer and demands that the `requires` clause of the layer be *covariant* and all partial methods are well formed. The rule T-LayerSW is for sublayers of `swappable` layers. It demands, in addition to the conditions described in T-Layer, that the `requires` clause of this layer be the same as those of its parent `swappable` layer, that no partial method be newly introduced, and that this layer be not required by other layers. The last condition requires a global program analysis.

It is worth elaborating the rule T-LayerSW in more detail. First, if the condition $\{\overline{\mathtt{L}}\} = \Lambda'$ were $\{\overline{\mathtt{L}}\} <:_w \Lambda'$ (as in T-Layer), the type system would be unsound. A counterexample is below:

```
class C {}
swappable layer L0 { int C.m() { return 0; } }
layer L1 extends L0 {}
layer L2 extends L0 requires L { int C.m() { return proceed(); } }
layer L requires L0 { int C.m() { return proceed(); } }
```

Layer `L2` additionally requires `L`, which requires `L0`, a swappable superclass of `L2`. The condition $\{\overline{\mathtt{L}}\} <:_w \Lambda'$ would be trivially satisfied for `L2` because the `requires` clause of `L0` is empty. The partial methods in `L2` and `L` are well formed because `L` and `L0`, respectively, provide definitions to `proceed`. Under these classes and layers, the following expression

```
with (new L1())
  with (new L())          // fulfills "requires L0"
    swap(L0, new L2())    // fulfills "requires L"
      new C().m()
```

is well typed, because `L1`, which is a subclass of `L0`, is activated before activating `L`, and `L` is activated before activating `L2`. However, the `swap` expression executed under `L1;L` would get stuck as follows:

$$
\begin{aligned}
&\mathtt{L1;L} \vdash \mathtt{swap(L0,\ new\ L2())\ new\ C().m()} \\
&\quad \longrightarrow \mathtt{swap(L0,\ new\ L2())\ new\ C<C,L,(L;L2)>().m()} \\
&\quad \longrightarrow \mathtt{swap(L0,\ new\ L2())\ new\ C<C,\bullet,(L;L2)>().m()} \\
&\quad \not\longrightarrow
\end{aligned}
$$

The method invocation would take place under `L;L2`, both of which have `C.m` but the second `proceed` call goes nowhere.

Second, if a subclass of a swappable layer were allowed to define a new method (which is not defined in the swappable), then the type system would be unsound, too. Consider the following classes and layers.

```
class C {}
class D extends C {}

swappable layer L0 {}
layer L1 extends L0 {}
layer L2 extends L0 {
  int C.m() { return this.m(); }
  int D.m() { return swap(L, new L2()) super.m(); }
}
```

Layer `L2` defines new partial methods `C.m` and `D.m`. They are well formed: in particular, `super.m()` is well typed because `L2` itself provides `C.m`. The following expression

```
with (new L2()) new D().m()
```

is well typed, since `D.m` invoked with `L2` activated. However, reduction of `new D().m()` under `L2` would get stuck:

$$
\begin{aligned}
&\texttt{L2} \vdash \texttt{new D().m()} \\
&\quad \longrightarrow \texttt{swap(L, new L1()) new D<C,L2,L2>().m()} \\
&\quad \longrightarrow \texttt{swap(L, new L1()) new D().m()} \\
&\quad \not\longrightarrow
\end{aligned}
$$

Since `super` calls are not affected by `swap`, `super.m()` in `D.m` succeeds but, by the time `this.m()` is executed, `L2` will be swapped out.

Fig. 11 is for program typing; a program is well typed if all classes and layers in *CT* and *LT*, respectively, are well formed and the main expression `e` is typed (at the top-level $\bullet$).

The most involved is the rule to check valid method overriding used in T-Table. The predicate *noconflict* means that for two partial methods of the same (qualified) name must have the same signature. The predicate $override^h$ means that, for any partial method, the overridden method (base method in `C` or partial methods for `C`'s superclass) must have the same signature. The predicate $override^v$ means that a base method can override a (partial) method in its superclass (or layers modifying it) with a covariant return type. Note that, unlike Java, checking valid method overriding requires a whole program because a layer may add a new method to a base class, one of whose subclass may accidentally define a method of the same name without knowing of that layer.

4. Type Soundness

In this section, we prove type soundness of ContextFJ$_{<:}$ via subject reduction and progress [38]. Strictly speaking, we should present typing rules for run-time expressions first before stating these properties but, for ease of understanding, we will reverse the order and start with the statements of the properties.

Since we model the execution of a `main` method starting with no layers activated, we are mainly interested in the case where $\mathcal{L}$ is $\bullet$ and the layer sequence is empty. However, we have to strengthen the statements of these properties so that the layer sequence can be nonempty. We introduce the notion of *well-formed layer sets* for this purpose.

We define the relation $\{\overline{\texttt{L}}\}\,wf$, read "layer set $\{\overline{\texttt{L}}\}$ is well formed," by the rules in Fig. 12. Intuitively, a set of layers is well-formed if one can obtain the layers by activating them one by

$$\frac{}{\emptyset\ wf} \qquad \text{(WF-EMPTY)}$$

$$\frac{\Lambda\ wf \qquad \mathtt{L}_a\ \mathtt{req}\ \Lambda' \qquad \Lambda <:_w \Lambda'}{\Lambda \cup \{\mathtt{L}_a\}\ wf} \qquad \text{(WF-WITH)}$$

$$\frac{\Lambda\ wf \qquad \mathtt{L}_{sw}\ \mathtt{swappable} \qquad \mathtt{L} <:_w \mathtt{L}_{sw} \qquad \mathtt{L}\ \mathtt{req}\ \Lambda' \qquad \Lambda_{rm} = \Lambda \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\} \qquad \Lambda_{rm} <:_w \Lambda'}{\Lambda_{rm} \cup \{\mathtt{L}\}\ wf} \qquad \text{(WF-SWAP)}$$

Figure 12: ContextFJ$_{<:}$: Layer set well-formedness.

one so that `requires` clauses are satisfied. We ignore the order of activation because the `with` statement can change the order of activated layers by activating an already activated layer again.

Aside from layer well-formedness, the statements of subject reduction, progress, and type soundness are standard:

Theorem 1 (Subject Reduction). *Suppose* $\vdash (CT, LT)$ `ok`. *If* $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf and* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{S}$ *for some* S *such that* $\mathtt{S} <: \mathtt{T}$.

Theorem 2 (Progress). *Suppose* $\vdash (CT, LT)$ `ok`. *If* $\bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf, then* e *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

Theorem 3 (Type Soundness). *If* $\vdash (CT, LT, \mathtt{e}) : \mathtt{T}$ *and* e *reduces to a normal form under the empty set of layers, then the normal form is* `new` $\mathtt{S}(\overline{\mathtt{v}})$ *for some* $\overline{\mathtt{v}}$ *and* S *such that* $\mathtt{S} <: \mathtt{T}$.

4.1. Typing Rules for Run-time Expressions

To prove the theorems above, we have to give typing rules for run-time expressions of the forms `new` $\mathtt{C}(\overline{\mathtt{v}})$<D,$\overline{\mathtt{L}'}$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$) and `new` $\mathtt{C}(\overline{\mathtt{v}})$<D,L,$\overline{\mathtt{L}'}$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), which are not supposed to appear in a class/layer table. The typing rules with the rules for a few auxiliary judgments are given in Fig. 13:

In the rule T-InvkA for `new` $\mathtt{C}_0(\overline{\mathtt{v}})$<$\mathtt{D}_0$,$\overline{\mathtt{L}'}$,$\overline{\mathtt{L}}$>.m($\overline{\mathtt{e}}$), the premises except for $\mathtt{C}_0$.m $\vdash$ <$\mathtt{D}_0$,$\overline{\mathtt{L}'}$,$\overline{\mathtt{L}}$> `ok` and $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$—they are explained in detail below—are similar to T-Invk. The method signature is obtained by using the current cursor <$\mathtt{D}_0$,$\overline{\mathtt{L}'}$,$\overline{\mathtt{L}}$>. The rule T-InvkAL for a method invoked by `superproceed` is similar. One difference is that the method signature is obtained by using *pmtype*; the receiver is derived from a `superproceed` call that originated from a superlayer of $\mathtt{L}_0$, hence $\mathtt{L}_0 <:_{sw} \mathtt{L}_1$.

The condition $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$ relates the layer sequence $\overline{\mathtt{L}}$ in the cursor and Λ, which intuitively represents the set of layers activated at this program point. In many cases, $\Lambda = \{\overline{\mathtt{L}}\}$ holds but if `super` and `proceed` calls are surrounded by `with` or `swap`, they can be different. The relation $<:_{sw}$ is similar to $<:_w$ but the additional clauses $\exists \mathtt{L}_2 \in dom(LT).\mathtt{L}_2$ `swappable` and $\mathtt{L}_0 <:_w \mathtt{L}_2$ and $\mathtt{L}_1 <:_w \mathtt{L}_2$ take into account the possibility that a layer in Λ may be activated by swapping layers in $\{\overline{\mathtt{L}}\}$ out.

The judgment $\mathtt{C.m} \vdash$ <D,$\overline{\mathtt{L}_1}$,$\overline{\mathtt{L}_2}$> `ok`, which means that the cursor is well formed with respect to method m in class C, is defined by WF-Cursor. It requires that D to be a superclass of C and $\overline{\mathtt{L}_2}$ to be well formed. The last condition $ndp(\mathtt{m}, \mathtt{D}_0, \overline{\mathtt{L}'}, \overline{\mathtt{L}})$ (standing for "non-dangling `proceed`")

$$\frac{\bullet;\Lambda;\Gamma \vdash \texttt{new}\ \texttt{C}_0(\overline{\texttt{v}}) : \texttt{C}_0 \qquad \texttt{C}_0.\texttt{m} \vdash \texttt{<}\texttt{D}_0,\overline{\texttt{L}'},\overline{\texttt{L}}\texttt{>}\ \texttt{ok} \qquad \Lambda <:_{sw} \{\overline{\texttt{L}}\} \quad \mathit{mtype}(\texttt{m},\texttt{D}_0,\{\overline{\texttt{L}'}\},\{\overline{\texttt{L}}\}) = \overline{\texttt{T}'}\to\texttt{T}_0 \qquad \bullet;\Lambda;\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \overline{\texttt{S}} <: \overline{\texttt{T}'}}{\bullet;\Lambda;\Gamma \vdash \texttt{new}\ \texttt{C}_0(\overline{\texttt{v}})\texttt{<}\texttt{D}_0,\overline{\texttt{L}'},\overline{\texttt{L}}\texttt{>}.\texttt{m}(\overline{\texttt{e}}) : \texttt{T}_0} \quad \text{(T-InvkA)}$$

$$\frac{\bullet;\Lambda;\Gamma \vdash \texttt{new}\ \texttt{C}_0(\overline{\texttt{v}}) : \texttt{C}_0 \qquad \texttt{C}_0.\texttt{m} \vdash \texttt{<}\texttt{D}_0,(\overline{\texttt{L}''};\texttt{L}_0),\overline{\texttt{L}}\texttt{>}\ \texttt{ok} \qquad \Lambda <:_{sw} \{\overline{\texttt{L}}\} \quad \texttt{L}_0 <:_w \texttt{L}_1 \qquad \mathit{pmtype}(\texttt{m},\texttt{D}_0,\texttt{L}_1) = \overline{\texttt{T}'}\to\texttt{T}_0 \qquad \bullet;\Lambda;\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \overline{\texttt{S}} <: \overline{\texttt{T}'}}{\bullet;\Lambda;\Gamma \vdash \texttt{new}\ \texttt{C}_0(\overline{\texttt{v}})\texttt{<}\texttt{D}_0,\texttt{L}_1,(\overline{\texttt{L}''};\texttt{L}_0),\overline{\texttt{L}}\texttt{>}.\texttt{m}(\overline{\texttt{e}}) : \texttt{T}_0} \quad \text{(T-InvkAL)}$$

$$\frac{\forall \texttt{L}_0 \in \Lambda_0.\exists \texttt{L}_1 \in \Lambda_1.(\ \texttt{L}_1 <:_w \texttt{L}_0 \text{ or } \exists \texttt{L}_2 \in \mathit{dom}(LT).\texttt{L}_2\ \texttt{swappable and } \texttt{L}_0 <:_w \texttt{L}_2 \text{ and } \texttt{L}_1 <:_w \texttt{L}_2)}{\Lambda_1 <:_{sw} \Lambda_0} \quad \text{(LSSW-Intro)}$$

$$\frac{\texttt{C} <: \texttt{D} \qquad \{\overline{\texttt{L}_2}\}\ \mathit{wf} \qquad \mathit{ndp}(\texttt{m},\texttt{D},\overline{\texttt{L}_1},\overline{\texttt{L}_2})}{\texttt{C}.\texttt{m} \vdash \texttt{<}\texttt{D},\overline{\texttt{L}_1},\overline{\texttt{L}_2}\texttt{>}\ \texttt{ok}} \quad \text{(Wf-Cursor)}$$

$$\frac{\texttt{class C \{..}\ \texttt{C}_0\ \texttt{m(..)\{..\}}\ \texttt{..\}}}{\mathit{ndp}(\texttt{m},\texttt{C},\overline{\texttt{L}_1},(\overline{\texttt{L}_1};\overline{\texttt{L}_2}))} \quad \text{(NDP-Class)}$$

$$\frac{\exists \texttt{L}_0 \in \overline{\texttt{L}_1}.\texttt{proceed} \notin \mathit{pmbody}(\texttt{m},\texttt{C},\texttt{L}_0)}{\mathit{ndp}(\texttt{m},\texttt{C},\overline{\texttt{L}_1},(\overline{\texttt{L}_1};\overline{\texttt{L}_2}))} \quad \text{(NDP-Layer)}$$

$$\frac{\texttt{C} \lhd \texttt{D} \qquad \mathit{ndp}(\texttt{m},\texttt{D},(\overline{\texttt{L}_1};\overline{\texttt{L}_2}),(\overline{\texttt{L}_1};\overline{\texttt{L}_2}))}{\mathit{ndp}(\texttt{m},\texttt{C},\overline{\texttt{L}_1},(\overline{\texttt{L}_1};\overline{\texttt{L}_2}))} \quad \text{(NDP-Super)}$$

Figure 13: ContextFJ$_{<:}$: Runtime expression typing.

intuitively means "a chain of `proceed` calls from the given cursor location $\texttt{<D}_0\texttt{,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>}$ eventually reaches a (partial) method that does *not* call `proceed`" and is defined by the rules NDP-Class, NDP-Layer and NDP-Super, which are straightforward. (Here, "$\texttt{proceed} \notin \mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}_0)$" means that there is no `proceed` calls in the method body obtained by $\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}_0)$.) This predicate represents an invariant condition throughout a chain of `proceed` calls and ensures there will not be a dangling `proceed` call.

4.2. Subject Reduction

The proof of subject reduction is done by induction on $\overline{\texttt{L}} \vdash \texttt{e} \longrightarrow \texttt{e}'$. Similarly to FJ, one main lemma is the Substitution Lemma, which is used in the case where `e` is a method invocation and states substitution of values of types $\overline{\texttt{T}}$ for variables of types $\overline{\texttt{S}}$, where $\overline{\texttt{S}}$ are subtypes of $\overline{\texttt{T}}$, in a well typed term preserves typing. Another important lemma here is Lemma 5, which states substitution for `proceed`, `super`, and `superproceed` preserves typing.

We state several main lemmas to prove the theorems above; their proofs as well as other lemmas and proofs are found in Appendix. We fix *CT* and *LT* and assume (CT, LT) `ok` in the rest of this section.

As usual, adding an unused variable to the type environment preserves typing (Weakening). Narrowing usually refers to the property that replacing the type of a variable in the type environment with its subtype preserves typing; here, we need narrowing with respect to (extended) layer set subtyping $<:_{sw}$. The next lemma states that a well typed value remains well typed regardless of its typing context $(\mathcal{L}; \Lambda; \Gamma)$.

Lemma 1 (Weakening). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \texttt{e} : \texttt{T}$*, then* $\mathcal{L}; \Lambda; \Gamma, \texttt{x}\colon \texttt{S} \vdash \texttt{e} : \texttt{T}$.

Lemma 2 (Layer Set Narrowing). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \texttt{e} : \texttt{T}$ *and* $\Lambda' <:_{sw} \Lambda$*, then* $\mathcal{L}; \Lambda'; \Gamma \vdash \texttt{e} : \texttt{T}$.

Lemma 3 (Strengthening for values). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \texttt{v} : \texttt{T}$ *then,* $\mathcal{L}'; \Lambda'; \Gamma' \vdash \texttt{v} : \texttt{T}$.

The statement of the Substitution Lemma is straightforward.

Lemma 4 (Substitution). *If* $\mathcal{L}; \Lambda; \Gamma, \overline{\texttt{x}}\colon \overline{\texttt{T}} \vdash \texttt{e} : \texttt{T}$ *and* $\mathcal{L}; \Lambda; \Gamma \vdash \overline{\texttt{v}} : \overline{\texttt{S}}$ *and* $\overline{\texttt{S}} <: \overline{\texttt{T}}$*, then* $\mathcal{L}; \Lambda; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e} : \texttt{S}$ *and* $\texttt{S} <: \texttt{T}$ *for some* S.

The next lemma states that substitution for `proceed`, `super`, and `superproceed` preserves typing. The first item is for an invocation of a partial method, which may contain `proceed` and `superproceed` calls as well as `super` calls; the second is for a base method, which may contain only `super` calls. The conditions, which look rather complicated, correspond to the premises of T-InvkA and T-InvkAL.

Lemma 5 (Substitution for `super`, `proceed` and `superproceed`).

1. *If* $\bullet; \Lambda; \Gamma \vdash \texttt{new } \texttt{C}_0(\overline{\texttt{v}}) : \texttt{C}_0$ *and* $\texttt{L.C.m}; \Lambda; \Gamma \vdash \texttt{e} : \texttt{T}$ *and* $\texttt{C}_0\texttt{.m} \vdash \texttt{<C,}(\overline{\texttt{L}}';\texttt{L}'')\texttt{,}\overline{\texttt{L}}\texttt{>}$ `ok` *and* $\texttt{C} \triangleleft \texttt{D}$ *and* $\texttt{L}'' <:_w \texttt{L} \triangleleft \texttt{L}'$ *and* $\Lambda <:_{sw} \{\overline{\texttt{L}}\}$ *and* $\texttt{proceed} \in \texttt{e} \implies \mathit{ndp}(\texttt{m}, \texttt{C}, \overline{\texttt{L}}', \overline{\texttt{L}})$*, then* $\bullet; \Lambda; \Gamma \vdash S\,\texttt{e} : \texttt{T}$ *where*

$$S = \begin{bmatrix} \texttt{new } \texttt{C}_0(\overline{\texttt{v}})\texttt{<C,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>.m} & \texttt{/proceed,} \\ \texttt{new } \texttt{C}_0(\overline{\texttt{v}})\texttt{<D,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>} & \texttt{/super,} \\ \texttt{new } \texttt{C}_0(\overline{\texttt{v}})\texttt{<C,L',(}\overline{\texttt{L}}';\texttt{L}'')\texttt{,}\overline{\texttt{L}}\texttt{>.m} & \texttt{/superproceed} \end{bmatrix}.$$

2. *If* $\bullet; \Lambda; \Gamma \vdash$ `new` $\mathtt{C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$ *and* $\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{C}_0.\mathtt{m} \vdash$ `<`$\mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}}$`>` `ok` *and* $\mathtt{C} \lhd \mathtt{D}$ *and* $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$*, then* $\bullet; \Lambda; \Gamma \vdash$ `[new` $\mathtt{C}_0(\overline{\mathtt{v}})$`<`$\mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}$`>/super]`$\mathtt{e} : \mathtt{T}$.

The next two lemmas state method bodies obtained by *pmbody* and *mbody* are well typed according to the type information obtained by *pmtype* and *mtype*, respectively.

Lemma 6 (Inversion for partial method body). *If* $\mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0$ `in` $\mathtt{L}'$ *and* $\mathtt{L}$ `req` Λ *and* $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$*, then* $\mathtt{L}.\mathtt{C}.\mathtt{m}; \Lambda \cup \{\mathtt{L}\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0$ *for some* $\mathtt{S}_0 <:_w \mathtt{T}_0$.

Lemma 7 (Inversion for method body). *Suppose* $\{\overline{\mathtt{L}}\}$ *wf and* $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *in* $\mathtt{C}', \overline{\mathtt{L}''}$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$ *and* $\mathit{ndp}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}})$.

1. *If* $\overline{\mathtt{L}''} = \overline{\mathtt{L}'''}; \mathtt{L}_0$*, then* $\mathtt{L}_0$ `req` Λ *and* $\mathtt{L}_0.\mathtt{C}'.\mathtt{m}; \Lambda \cup \{\mathtt{L}_0\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{U}_0$ *and* $\mathtt{C} <: \mathtt{C}'$ *and* $\mathtt{U}_0 <: \mathtt{T}_0$ *and* $\mathit{ndp}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}''}, \overline{\mathtt{L}})$ *for some* Λ *and* $\mathtt{U}_0$.
2. *If* $\overline{\mathtt{L}''} = \bullet$*, then* $\mathtt{C}'.\mathtt{m}; \emptyset; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}' \vdash \mathtt{e}_0 : \mathtt{U}_0$ *and* $\mathtt{C} <: \mathtt{C}'$ *and* $\mathtt{U}_0 <: \mathtt{T}_0$ *and* $\mathit{ndp}(\mathtt{m}, \mathtt{C}', \bullet, \overline{\mathtt{L}})$ *for some* $\mathtt{U}_0$.

We also need additional lemmas derived from runtime conditions. Layer-set wellformedness Λ *wf* provides two important properties. The first states that a well formed layer set is closed under the `requires` clause and the second that, if method `m` is found in `C` (under the assumption that Λ activated) but not in its direct superclass `D`, then at least one of those methods does not call `proceed`. This lemma is used to prove the next lemma (Lemma 10), which derives *ndp* for an initial cursor of the form `<`$\mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}}$`>`.

Lemma 8. *] If* Λ *wf, then* $\forall \mathtt{L} \in \Lambda, \forall \mathtt{L}'$ *s.t.* $\mathtt{L}$ `req` $\mathtt{L}', \exists \mathtt{L}'' \in \Lambda. \mathtt{L}'' <:_w \mathtt{L}'$.

Lemma 9. *If* Λ *wf and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda)$ *defined and* $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda)$ *undefined and* $\mathtt{C} \lhd \mathtt{D}$*, then* $(\exists \mathtt{L}' \in \Lambda.$`proceed` $\notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}'))$ *or* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda)$ *defined.*

Lemma 10. *If* $\{\overline{\mathtt{L}}\}$ *wf and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$*, then* $\mathit{ndp}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}})$.

As stated below, the predicate *ndp* ensures the existence of a method:

Lemma 11. *If* $\mathit{ndp}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}})$ *holds for some* `m`*,* `C`*,* $\overline{\mathtt{L}'}$ *and* $\overline{\mathtt{L}}$*, then* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$ *for some* $\overline{\mathtt{T}}$ *and* $\mathtt{T}_0$.

4.3. Progress

To prove the Progress Theorem, we need the following two lemmas, which show the existence of a method body from well definedness of *pmtype* and *mtype*.

Lemma 12. *If* $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$*, then there exist* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *and* $\mathtt{L}'$ *(*$\neq$ `Base`*) such that* $\mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *in* $\mathtt{L}'$ *and the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{T}}$ *are equal and* $\mathtt{L} <:_w \mathtt{L}'$.

Lemma 13. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}_0$ *and* $\overline{\mathtt{L}'}$ *is a prefix of* $\overline{\mathtt{L}}$ *and* $\{\overline{\mathtt{L}}\}$ *wf, then there exist* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *and* $\overline{\mathtt{L}''}$ *and* $\mathtt{C}'$ *(*$\neq$ `Object`*) such that* $\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}'}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *in* $\mathtt{C}', \overline{\mathtt{L}''}$ *and the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{T}}$ *are equal and, if* $\overline{\mathtt{L}''}$ *is not empty, the last layer name of* $\overline{\mathtt{L}''}$ *is not* `Base`.

5. Related Work

Foundation of Context-Oriented Programming. Our work is a direct descendant of Igarashi, Hirschfeld, and Masuhara [23, 24], where a tiny COP language ContextFJ is developed and its type system is proved to be sound. ContextFJ is not equipped with layer inheritance, layer subtyping, or first-class layers but allows baseless partial methods to be declared in the second type system [24], in which `requires` declarations are first introduced into COP.

Our swappable layers resemble atomic layers in ContextL [13], in which mutual exclusion between layers can be specified and activation of an atomic layer may automatically deactivates another layer in conflict. Our syntax is a little verbose in that the swappable layer name such as `Weather` has to be explicit because a layer may have more than one swappable layer in its superlayers. It may be a reasonable idea to disallow a sublayer of a swappable layer to be swappable for the sake of syntactic conciseness.

Similarly to our swappable layers for layer deactivation, Kamina et al. [28, 32] also show another approach to safe layer deactivation mechanism and formalized its semantics and type safety with an extension of ContextFJ. Their approach is also based on its `requires` relationship. The key idea is to modify the method lookup so that it searches not only activated layers but all layers that are required by those activated layers.

Besides block-style layer activation mechanisms as in JCop, there are other mechanisms such as imperative activation of Subjective-C [20], event-based activation of EventCJ [29], and implicit activation of Flute [4]. The original JCop also supports implicit layer activation [3], but currently we omit it from our formalization. ServalCJ [30] provides a generalized layer activation mechanism that can treat the layer activation mechanisms above uniformly. However, though some studies provide formal semantics such as in EventCJ [1] and ServalCJ [31], they do not discuss type soundness of languages with baseless partial methods; e.g., ServalCJ does not support baseless partial methods.

There are several studies to enrich description of relationships between contexts. Subjective-C [20], an extension of Objective-C with COP, adopts imperative context activation with imperative context relationship description, which supports various kinds of declarations of dependency between layers, such as implication, requirement, and exclusion. Context Petri Nets [9, 10] is a context-oriented extension of Petri Nets, and helps formalization of description of context dependencies in Subjective-C. ML_{CoDa} [16, 17] provides two kinds of components; one for declarative description of context dependencies and the other for functional computation. It also provides a type and effect system and a loading-time verification mechanism that detects failures in adaptation.

Dynamic Software Product Line. Software product line (SPL) is a paradigm of industrial software development that enables to create various variations of software by mostly reusing common modules. Programming languages for SPL, such as Feature-Oriented Programming [35, 5] and Delta-Oriented Programming (DOP) [37], have been studied. They provide modules that refine existing classes and combine them according to given configuration at compile-time or build-time.

Recent studies [21, 36] reveal that SPL also needs dynamic reconfiguration of software, and so dynamic delta-oriented programming [15, 14] is proposed. Dynamic DOP provides mainly three kinds of modules; a delta module for describing refinement of classes (similar to a layer of COP), a product-line declaration for describing valid configurations, and a dynamic reconfiguration graph for replacing heap objects dynamically. Unlike COP, the composition order of

delta modules is determined uniquely by a given product line declaration; this property is called *unambiguity*. A type system of dynamic DOP also ensures that all valid reconfigurations lead to type-safe products.

Type systems for advanced composition mechanisms of OOP. There are many type systems proposed for advanced composition mechanisms such as mixins [7, 19], traits [34], open classes (a.k.a. inter-type declarations) [12], and revisers [11]. A common idea is to let programmers declare dependency between modules as required interfaces; our `requires` declarations basically follow it. In most work, however, composition is done at compile or link time unlike COP languages. We think that it is interesting that the same idea works even for dynamic composition found in COP languages.

Kamina and Tamai [33] propose McJava, in which mixin-based composition can be deferred to object instantiation. In fact, `new` expressions can specify a class and mixins to instantiate an object. So, the type of an object also consists of a class name and a sequence of mixin names. Whereas composition is per-instance basis in McJava, it is global in ContextFJ$_{<:}$. However, in McJava, composition cannot be changed once an object is instantiated.

Drossopoulou et al. [18] proposed *Fickle*$_{\text{II}}$, a class-based object-oriented language with *dynamic reclassification*, which allows an object to change its class at run time. Their idea of root classes, which serve as interface, is similar to our swappable layers; their restriction that state classes cannot be used as type for fields is similar to ours that a sublayer of a swappable cannot be `required` by any other layer.

Bettini et al. [6] developed a type system for *dynamic trait replacement*, which allows methods in an object to be exchanged at run time. They introduce the notion of *replaceable* to describe the signatures of replaceable methods; a replaceable appears as part of the type of an object and the trait to replace methods of the object has to provide the methods in that replaceable. The roles of replaceables and traits are somewhat similar to those of swappable layers, which provide interfaces common to swapped layers, and sublayers of swappable.

Although not a type system, Burton and Sekerinski [8] studies interference problem of dynamic mixin composition, in which some order of mixin composition breaks required specification of class methods. They develop a refinement calculus in order to formalize dynamic mixin composition.

6. Concluding Remarks

We have developed a formal type system for a small COP language with layer inheritance, layer subtyping, swappable layers, and first-class layers, and shown that the type system is sound with respect to the operational semantics. As in previous work, `requires` declarations are important to guarantee safety in the presence of baseless partial methods. Subtyping for first-class layers is subtle because there are two kinds of substitutability. We have introduced weak subtyping for checking whether a `requires` clause is satisfied and normal subtyping for usual substitutability.

In JCop, a layer definition can contain field and (ordinary) method declarations so that a layer instance can act just like an ordinary object. Typechecking accesses to these members of layer instances is the same as ordinary objects. If we model fields of layer instances, we will have to modify the reduction relation so that the sequence of activated layers consists of layer instances (with their field values) rather than layer names.

JCop also provides special variable `thislayer`, which can be used in partial methods and is similar to `this` of classes. It represents the layer instance in which the invoked partial method is found at run time and can be used to access fields and methods of that layer instance. In operational semantics, the layer instance would be substituted for `thislayer`, similarly to `this`. Typing `thislayer` is also similar to `this` in the sense that it is given the name of the layer in which it appears but `thislayer` cannot be used for layer activation because, at run time, it may be bound to an instance of a *weak* subtype.

We have not fully investigated the interaction between our type system with other features in Java, such as concurrency, generics, and lambda, although we expect most of them are orthogonal.

Acknowledgments.. We thank Tomoyuki Aotani, Malte Appeltauer, Robert Hirschfeld, and Tetsuo Kamina for valuable discussions on the subject. This work was supported in part by Kyoto University Design School (Inoue) and MEXT KAKENHI Grant Number 23220001 (Igarashi).

[1] T. Aotani, T. Kamina, and H. Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*. ACM, 2011.

[2] M. Appeltauer and R. Hirschfeld. *The JCop language specification: Version 1.0, April 2012*. Number 59. Universitätsverlag Potsdam, 2012.

[3] M. Appeltauer, R. Hirschfeld, and J. Lincke. Declarative layer composition with the JCop programming language. *Journal of Object Technology*, 12, 2013.

[4] E. Bainomugisha, J. Vallejos, C. De Roover, A. Lombide Carreton, and W. De Meuter. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! 2012, pages 67–84. ACM, 2012.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, 2004.

[6] L. Bettini, S. Capecchi, and F. Damiani. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.

[7] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *ECOOP'99–Object-Oriented Programming*, pages 43–66. Springer, 1999.

[8] E. Burton and E. Sekerinski. The safety of dynamic mixin composition. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1992–1999. ACM, 2015.

[9] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, and T. D'Hondt. Modeling and analyzing self-adaptive systems with context Petri nets. In *Proc. of TASE*, pages 191–198. IEEE, 2013.

[10] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, J. Vallejos, and T. D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71–94, 2015.

[11] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proc. of the ACM OOPSLA*, pages 539–554, 2010.

[12] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3):517–575, 2006.

[13] P. Costanza and T. D'Hondt. Feature descriptions for context-oriented programming. In *Proc. of SPLC (2)*, pages 9–14, 2008. Workshop on Dynamic Software Product Lines (DSPL 2008).

[14] F. Damiani, L. Padovani, I. Schaefer, and C. Seidl. A core calculus for dynamic delta-oriented programming. *Acta Informatica*, pages 1–39, 2017.

[15] F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 34. ACM, 2011.

[16] P. Degano, G.-L. Ferrari, and L. Galletta. A two-phase static analysis for reliable adaptation. In *International Conference on Software Engineering and Formal Methods*, pages 347–362. Springer, 2014.

[17] P. Degano, G.-L. Ferrari, and L. Galletta. A two-component language for adaptation: Design, semantics and program analysis. *IEEE Transactions on Software Engineering*, 42(6):505–529, 2016.

[18] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle$_{\text{II}}$. *ACM Trans. Prog. Lang. Syst.*, 24(2):153–191, 2002.

[19] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of the ACM POPL*, pages 171–183. ACM, 1998.

[20] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-C. In *International Conference on Software Language Engineering*, pages 246–265. Springer, 2010.
[21] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.
[22] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
[23] R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: A minimal core calculus for context-oriented programming. In *Proc. of Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2011.
[24] A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. In *Proc. of FOOL*, Oct. 2012.
[25] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
[26] H. Inoue and A. Igarashi. A sound type system for layer subtyping and dynamically activated first-class layers. In *Asian Symposium on Programming Languages and Systems*, pages 445–462. Springer, 2015.
[27] H. Inoue, A. Igarashi, M. Appeltauer, and R. Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In *Proc. of the Workshop on Context-Oriented Programming*, pages 7:1–7:6. ACM, 2014.
[28] T. Kamina, T. Aotani, and A. Igarashi. On-demand layer activation for type-safe deactivation. In *Proc. of COP'14*, Uppsala, Sweden, July 2014.
[29] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *Proc. of ACM AOSD*, pages 253–264. ACM, 2011.
[30] T. Kamina, T. Aotani, and H. Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Proceedings of the 14th International Conference on Modularity*, pages 14–28. ACM, 2015.
[31] T. Kamina, T. Aotani, and H. Masuhara. *Generalized Layer Activation Mechanism for Context-Oriented Programming*, pages 123–166. Springer International Publishing, Cham, 2016.
[32] T. Kamina, T. Aotani, H. Masuhara, and A. Igarashi. Method safety mechanism for asynchronous layer deactivation. Submitted for publication.
[33] T. Kamina and T. Tamai. McJava–a design and implementation of Java with mixin-types. In *Proc. of APLAS*, pages 398–414, 2004.
[34] L. Liquori and A. Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Trans. Prog. Lang. Syst.*, 30(2):11, 2008.
[35] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97–Object-Oriented Programming*, pages 419–443. Springer, 1997.
[36] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011.
[37] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010.
[38] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

A. Proofs

We fix CT and LT and assume (CT, LT) $\texttt{ok}$ throughout this section.

Lemma A.1. *If* $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *and* $\mathtt{L}_1 <:_w \mathtt{L}_2$*, then* $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_1) = \overline{\mathtt{T}} \to \mathtt{T}_0$.

Proof. By induction on $\mathtt{L}' <:_w \mathtt{L}$, using $\mathit{noconflict}(\mathtt{L}', \mathtt{L})$ in the case where $\mathtt{L}' \lhd \mathtt{L}$. □

Lemma A.2. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *and* $\Lambda_3 <:_{sw} \Lambda_1$ *and* $\Lambda_4 <:_{sw} \Lambda_2$ *and* $\Lambda_1 \subseteq \Lambda_2$ *and* $\Lambda_3 \subseteq \Lambda_4$*, then* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}} \to \mathtt{T}_0$.

Proof. By induction on the derivation of $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$ with case analysis on the last rule used.

Case MT-Class**:** $\texttt{class C} \lhd \texttt{D \{...}\ \ \mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\texttt{\{ return e; \} ...\}}$

MT-Class finishes the case.

Case MT-PMethod**:** $\exists \mathtt{L}_1 \in \Lambda_1.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_1) = \overline{\mathtt{T}} \to \mathtt{T}_0$

By $\Lambda_3 <:_{sw} \Lambda_1$, there exists $\mathtt{L}_3 \in \Lambda_3$ such that either (1) $\mathtt{L}_3 <:_w \mathtt{L}_1$ or (2) there exists $\mathtt{L}$ such that $\mathtt{L}$ $\texttt{swappable}$ and $\mathtt{L}_1, \mathtt{L}_3 <:_w \mathtt{L}$. In the case (1), Lemma A.1 and MT-PMethod finish the case. In the case (2), by T-LayerSW and $\mathit{noconflict}(\mathtt{L}_1, \mathtt{L}_3)$, it is easy to show $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_3) = \overline{\mathtt{T}} \to \mathtt{T}_0$. Then, Lemma A.1 and MT-PMethod finish the case.

Case MT-Super**:** $\texttt{class C} \lhd \texttt{D \{...}\ \ \overline{\mathtt{M}}\texttt{\}}\quad \mathtt{m} \notin \overline{\mathtt{M}}$
$\forall \mathtt{L} \in \Lambda_1.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ undefined $\quad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_2, \Lambda_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$

If $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ is undefined for all $\mathtt{L} \in \Lambda_3$, then the induction hypothesis and MT-Super finish the case. Otherwise, we have $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\texttt{\{...\}}$ for some $\mathtt{L} \in \Lambda_3$. Then, $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{S}} \to \mathtt{S}_0$ holds by MT-PMethod and also there exists $\mathtt{L}'$ such that $\mathit{LT}(\mathtt{L}')(\mathtt{C.m}) = \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\texttt{\{...\}}$.

By the induction hypothesis, $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathit{dom}(\mathit{LT}), \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \to \mathtt{T}_0$ (since $\mathit{dom}(\mathit{LT}) <:_w \Lambda_2$). By MT-Super, $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}} \to \mathtt{T}_0$. Finally, $\overline{\mathtt{S}}$, $\overline{\mathtt{S}}_0 = \overline{\mathtt{T}}$, $\mathtt{T}_0$ follows from $\mathit{override}^h(\mathtt{L}', \mathtt{C})$, finishing the case. □

Lemma A.3. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}}' \to \mathtt{T}_0'$*, then* $\overline{\mathtt{T}}$, $\mathtt{T}_0 = \overline{\mathtt{T}}'$, $\mathtt{T}_0'$.

Proof. By induction on the derivation of $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_2) = \overline{\mathtt{T}} \to \mathtt{T}_0$ with case analysis on the last rule used.

Case MT-Class**:** $\texttt{class C} \lhd \texttt{D \{...}\ \ \mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\texttt{\{...\} ...\}}$

Easy. Use $\mathit{override}^h(\mathtt{L}, \mathtt{C})$ for $\mathtt{L} \in \mathit{dom}(\mathit{LT})$ if $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}}' \to \mathtt{T}_0'$ is derived by MT-PMethod, in which case there exists $\mathtt{L}'$ such that $\mathtt{L} <:_w \mathtt{L}'$ and $\mathit{LT}(\mathtt{L}')(\mathtt{C.m}) = \mathtt{T}_0'\ \mathtt{m}(\overline{\mathtt{T}}'\ \overline{\mathtt{x}})\ \texttt{\{...\}}$. (Note that $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}}' \to \mathtt{T}_0'$ cannot be derived by MT-Super.)

Case MT-PMethod**:** $\exists \mathtt{L}_1 \in \Lambda_1.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_1) = \overline{\mathtt{T}} \to \mathtt{T}_0$

There exists $\mathtt{L}_1'$ such that $\mathtt{L}_1 <:_w \mathtt{L}_1'$ and $\mathit{LT}(\mathtt{L}_1')(\mathtt{C.m}) = \mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \texttt{\{...\}}$. Further case analysis on $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}}' \to \mathtt{T}_0'$.

Subcase MT-Class**:**

Similar to the above case.

Subcase MT-PMethod**:** $\exists \mathtt{L}_2 \in \Lambda_3.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}_2) = \overline{\mathtt{T}}'\to\mathtt{T}_0'$

There exists $\mathtt{L}_2'$ such that $\mathtt{L}_2 <:_w \mathtt{L}_2'$ and $\mathit{LT}(\mathtt{L}_2')(\mathtt{C}.\mathtt{m}) = \mathtt{T}_0'\ \mathtt{m}(\overline{\mathtt{T}}'\ \overline{\mathtt{x}})\ \{\ldots\}$. Then, $\mathit{noconflict}(\mathtt{L}_1', \mathtt{L}_2')$ finishes the case.

Subcase MT-Super**:** $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\ \overline{\mathtt{M}}\}$ $\quad \mathtt{m} \notin \overline{\mathtt{M}}$
$\forall \mathtt{L} \in \Lambda_3.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ undefined $\quad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_4, \Lambda_4) = \overline{\mathtt{T}}'\to\mathtt{T}_0'$

In this case, $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \mathit{dom}(\mathit{LT})) = \overline{\mathtt{T}}'\to\mathtt{T}_0'$ because we can show that

$$\begin{aligned}\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_4, \Lambda_4) &= \mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathit{dom}(\mathit{LT}), \mathit{dom}(\mathit{LT})) \\ &= \mathit{mtype}(\mathtt{m}, \mathtt{C}, \emptyset, \mathit{dom}(\mathit{LT})).\end{aligned}$$

by Lemma A.2 and MT-Super. Then, $\mathit{override}^h(\mathtt{L}_1', \mathtt{C})$ finishes the case.

Case MT-Super**:** $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\ \overline{\mathtt{M}}\}$ $\quad \mathtt{m} \notin \overline{\mathtt{M}}$
$\forall \mathtt{L} \in \Lambda_1.\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L})$ undefined $\quad \mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_2, \Lambda_2) = \overline{\mathtt{T}}\to\mathtt{T}_0$

Further case analysis on $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda_3, \Lambda_4) = \overline{\mathtt{T}}'\to\mathtt{T}_0'$.

Subcase MT-PMethod**:**

Similar to the subcase MT-Super above.

Subcase MT-Class**:**

Cannot happen.

Subcase MT-Super**:**

By the induction hypothesis, $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_2, \Lambda_2) = \mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda_4, \Lambda_4)$. □

Lemma A.4. *If* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{D} <: \mathtt{C}$*, then* $\mathit{fields}(\mathtt{D}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ *for some* $\overline{\mathtt{S}}$, $\overline{\mathtt{g}}$.

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

Lemma A.5 (Weakening, Lemma 1). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$*, then* $\mathcal{L}; \Lambda; \Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$.

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

Lemma A.6. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda) = \overline{\mathtt{T}}\to\mathtt{T}_0$ *and* $\mathtt{D} <: \mathtt{C}$*, then* $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda) = \overline{\mathtt{T}}\to\mathtt{S}_0$ *and* $\mathtt{S}_0 <: \mathtt{T}_0$ *for some* $\mathtt{S}_0$.

Proof. By induction on $\mathtt{D} <: \mathtt{C}$. We show only the case where $\mathtt{D}$ extends $\mathtt{C}$. If $\mathtt{class\ D} \lhd \mathtt{C}\ \{\ldots\ \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \ldots\}$, then $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda) = \overline{\mathtt{S}}\to\mathtt{S}_0$ for some $\overline{\mathtt{S}}$ by MT-Class. By $\mathit{override}^v(\mathtt{D})$, $\overline{\mathtt{S}} = \overline{\mathtt{T}}$ and $\mathtt{S}_0 <: \mathtt{T}_0$. If $\exists \mathtt{L} \in \Lambda.\mathit{pmtype}(\mathtt{m}, \mathtt{D}, \mathtt{L}) = \overline{\mathtt{S}}\to\mathtt{S}_0$, then we have $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda) = \overline{\mathtt{S}}\to\mathtt{S}_0$ by MT-PMethod. By $\mathit{override}^h(\mathtt{L}, \mathtt{D})$, we get $\overline{\mathtt{S}} = \overline{\mathtt{T}}$ and $\mathtt{S}_0 = \mathtt{T}_0$. Otherwise, $\mathit{mtype}(\mathtt{m}, \mathtt{D}, \Lambda) = \overline{\mathtt{T}}\to\mathtt{T}_0$ by MT-Super. □

Lemma A.7 (Narrowing, Lemma 2). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Lambda' <:_{sw} \Lambda$*, then* $\mathcal{L}; \Lambda'; \Gamma \vdash \mathtt{e} : \mathtt{T}$.

Proof. By induction on $\mathcal{L}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$. We show only some representative cases.

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

By Lemma A.2, $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \Lambda') = \overline{\mathtt{D}}\to\mathtt{C}$. Then, the induction hypothesis and T-Invk finish the case.

Case T-With: $\texttt{e} = \texttt{with e}_l\ \texttt{e}_0$ $\quad \mathcal{L}; \Lambda; \Gamma \vdash \texttt{e}_l : \texttt{L}$ $\quad \texttt{L req } \Lambda''$ $\quad \Lambda <:_w \Lambda''$ $\quad \mathcal{L}; \Lambda \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$

By LSSW-Intro, we have $\Lambda' \cup \{\texttt{L}\} <:_{sw} \Lambda \cup \{\texttt{L}\}$. By the induction hypothesis, $\mathcal{L}; \Lambda' \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$.

It is easy to show that $<:_{sw}$ is transitive and so $\Lambda' <:_{sw} \Lambda''$. By the induction hypothesis, we also have $\mathcal{L}; \Lambda'; \Gamma \vdash \texttt{e}_l : \texttt{L}$.

Moreover, in a well-formed program, $\texttt{L req } \Lambda''$ means that Λ'' does not contain any sublayer of `swappable` layers. By these facts and LSS-Intro, we get $\Lambda' <:_w \Lambda''$. Then, by T-With, $\mathcal{L}; \Lambda'; \Gamma \vdash \texttt{with e}_l\ \texttt{e}_0 : \texttt{T}$, finishing the case.

Case T-Swap: $\texttt{e} = \texttt{swap (e}_l\texttt{,L}_{sw}\texttt{) e}_0$ $\quad \mathcal{L}; \Lambda; \Gamma \vdash \texttt{e}_l : \texttt{L}$ $\quad \texttt{L}_{sw}\ \texttt{swappable}$ $\quad \texttt{L} <:_w \texttt{L}_{sw}$ $\quad \texttt{L req } \Lambda''$ $\quad \Lambda_{rm} = (\Lambda \backslash \{\texttt{L}' \mid \texttt{L}' <:_w \texttt{L}_{sw}\})$ $\quad \Lambda_{rm} <:_w \Lambda''$ $\quad \mathcal{L}; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$

It is easy to show that $(\Lambda' \backslash \{\texttt{L}' \mid \texttt{L}' <:_w \texttt{L}_{sw}\}) \cup \{\texttt{L}\} <:_{sw} \Lambda_{rm} \cup \{\texttt{L}\}$ from $\Lambda' <:_{sw} \Lambda$. By the induction hypothesis, $\mathcal{L}; (\Lambda' \backslash \{\texttt{L}' | \texttt{L}' <:_w \texttt{L}_{sw}\}) \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$.

By LSSW-Intro, we have $(\Lambda' \backslash \{\texttt{L}' | \texttt{L}' <:_w \texttt{L}_{sw}\}) \cup \{\texttt{L}\} <:_{sw} \Lambda''$. Moreover, in a well-formed program, $\texttt{L req } \Lambda''$ means that Λ'' does not have any sublayer of `swappable` layers. By these facts and LSS-Intro, we get $(\Lambda' \backslash \{\texttt{L}' | \texttt{L}' <:_w \texttt{L}_{sw}\}) \cup \{\texttt{L}\} <:_w \Lambda''$. By the induction hypothesis, we also have $\mathcal{L}; \Lambda'; \Gamma \vdash \texttt{e}_l : \texttt{L}$. Then, by T-Swap, $\mathcal{L}; \Lambda'; \Gamma \vdash \texttt{swap (e}_l\texttt{,L}_{sw}\texttt{) e}_0 : \texttt{T}$, finishing the case. □

Lemma A.8 (Strengthening for values, Lemma 3)). *If* $\mathcal{L}; \Lambda; \Gamma \vdash \texttt{v} : \texttt{T}$ *then,* $\mathcal{L}'; \Lambda'; \Gamma' \vdash \texttt{v} : \texttt{T}$.

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

Lemma A.9 (Substitution, Lemma 4). *If* $\mathcal{L}; \Lambda; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e} : \texttt{T}$ *and* $\mathcal{L}; \Lambda; \Gamma \vdash \overline{\texttt{v}} : \overline{\texttt{S}}$ *and* $\overline{\texttt{S}} <: \overline{\texttt{T}}$, *then* $\mathcal{L}; \Lambda; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e} : \texttt{S}$ *and* $\texttt{S} <: \texttt{T}$ *for some* S.

Proof. By induction on $\mathcal{L}; \Lambda; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e} : \texttt{T}$ with case analysis on the last rule used. We show main cases of T-With and T-Swap.

Case T-With: $\texttt{e} = \texttt{with e}_l\ \texttt{e}_0$ $\quad \mathcal{L}; \Lambda; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_l : \texttt{L}$ $\quad \texttt{L req } \Lambda'$ $\quad \Lambda <:_w \Lambda'$ $\quad \mathcal{L}; \Lambda \cup \{\texttt{L}\}; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_0 : \texttt{T}$

By the induction hypothesis, $\mathcal{L}; \Lambda; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_l : \texttt{L}_0$ and $\texttt{L}_0 <: \texttt{L}$ for some $\texttt{L}_0$. By induction on $\texttt{L}_0 <: \texttt{L}$, it is easy to show that $\texttt{L}_0\ \texttt{req } \Lambda'$. Since $\texttt{L}_0 <: \texttt{L}$, we also have $\texttt{L}_0 <:_w \texttt{L}$ and so it is easy to show $\Lambda \cup \{\texttt{L}_0\} <:_w \Lambda \cup \{\texttt{L}\}$. By the induction hypothesis, $\mathcal{L}; \Lambda \cup \{\texttt{L}\}; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_0 : \texttt{S}$ and $\texttt{S} <: \texttt{T}$ for some S. Then, by Lemma A.7, $\mathcal{L}; \Lambda \cup \{\texttt{L}_0\}; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_0 : \texttt{S}$; and T-With finish the case.

Case T-Swap: $\texttt{e} = \texttt{swap (e}_l\texttt{, L}_{sw}\texttt{) e}_0$ $\quad \mathcal{L}; \Lambda; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_l : \texttt{L}$ $\quad \texttt{L}_{sw}\ \texttt{swappable}$ $\quad \texttt{L} <:_w \texttt{L}_{sw}$ $\quad \texttt{L req } \Lambda'$ $\quad \Lambda_{rm} = \Lambda \setminus \{\texttt{L}' \mid \texttt{L}' <:_w \texttt{L}_{sw}\}$ $\quad \Lambda_{rm} <:_w \Lambda'$ $\quad \mathcal{L}; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_0 : \texttt{T}$

By the induction hypothesis, $\mathcal{L}; \Lambda; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_l : \texttt{L}_0$ and $\texttt{L}_0 <: \texttt{L}$ for some $\texttt{L}_0$. By induction on $\texttt{L}_0 <: \texttt{L}$, it is easy to show $\texttt{L}_0\ \texttt{req } \Lambda'$. Since $\texttt{L}_0 <: \texttt{L}$, we have $\texttt{L}_0 <:_w \texttt{L}$ and $\texttt{L}_0 <:_w \texttt{L}_{sw}$ and $\Lambda_{rm} \cup \{\texttt{L}_0\} <:_w \Lambda_{rm} \cup \{\texttt{L}\}$. By the induction hypothesis, $\mathcal{L}; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_0 : \texttt{S}$ and $\texttt{S} <: \texttt{T}$ for some S. Then, by Lemma A.7, $\mathcal{L}; \Lambda_{rm} \cup \{\texttt{L}_0\}; \Gamma \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}_0 : \texttt{S}$; and T-With finishes the case. □

Lemma A.10. *If* $\mathtt{L}_1 <:_w \mathtt{L}_2$ *and* $\mathtt{L}_1$ `req` Λ_1 *and* $\mathtt{L}_2$ `req` Λ_2*, then* $\Lambda_1 <:_w \Lambda_2$.

Proof. By induction on $\mathtt{L}_1 <:_w \mathtt{L}_2$. Use T-Layer in the case for LSw-Extends. □

We prove a stronger property than Lemma 8; in the statement below, $(<:_w; \mathtt{req})$ stands for the composition of the two relations $<:_w$ and `req`.

Lemma A.11. *If* Λ *wf, then* $\forall \mathtt{L} \in \Lambda, \forall \mathtt{L}'$ *s.t.* $\mathtt{L}$ $(<:_w; \mathtt{req})$ $\mathtt{L}', \exists \mathtt{L}'' \in \Lambda.\mathtt{L}'' <:_w \mathtt{L}'$.

Proof. Induction on the derivation of Λ *wf*.

Case Wf-Empty**:**

Trivial.

Case Wf-With**:** $\Lambda = \Lambda_0 \cup \{\mathtt{L}_a\}$ Λ_0 *wf* $\mathtt{L}_a$ `req` Λ' $\Lambda_0 <:_w \Lambda'$

By the induction hypothesis, we have $\forall \mathtt{L} \in \Lambda_0.\forall \mathtt{L}'$ s.t. $\mathtt{L}$ $(<:_w; \mathtt{req})$ $\mathtt{L}'.\exists \mathtt{L}'' \in \Lambda_0.\mathtt{L}'' <:_w \mathtt{L}'$. By $\Lambda_0 <:_w \Lambda'$ and Lemma A.10, we have $\forall \mathtt{L}'$ s.t. $\mathtt{L}_a$ $(<:_w; \mathtt{req})$ $\mathtt{L}'.\exists \mathtt{L}'' \in \Lambda_0.\mathtt{L}'' <:_w \mathtt{L}'$. So, $\forall \mathtt{L} \in \Lambda.\forall \mathtt{L}'$ s.t. $\mathtt{L}$ $(<:_w; \mathtt{req})$ $\mathtt{L}'.\exists \mathtt{L}'' \in \Lambda.\mathtt{L}'' <:_w \mathtt{L}'$.

Case Wf-Swap**:** $\Lambda = \Lambda_{rm} \cup \{\mathtt{L}_a\}$ Λ_0 *wf* $\mathtt{L}_{sw}$ `swappable` $\mathtt{L}_a <:_w \mathtt{L}_{sw}$ $\mathtt{L}_a$ `req` Λ_a $\Lambda_{rm} = \Lambda_0 \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\}$ $\Lambda_{rm} <:_w \Lambda_a$

By the induction hypothesis, we have

$$\forall \mathtt{L} \in \Lambda_0.\forall \mathtt{L}' \text{ s.t. } \mathtt{L} \ (<:_w; \mathtt{req}) \ \mathtt{L}'.\exists \mathtt{L}'' \in \Lambda_0.\mathtt{L}'' <:_w \mathtt{L}',$$

and so

$$\forall \mathtt{L} \in \Lambda_{rm}.\forall \mathtt{L}' \text{ s.t. } \mathtt{L} \ (<:_w; \mathtt{req}) \ \mathtt{L}', \exists \mathtt{L}'' \in \Lambda_0 \text{ s.t. } \mathtt{L}'' <:_w \mathtt{L}'.$$

In fact, we can show that

$$\forall \mathtt{L} \in \Lambda_{rm}.\forall \mathtt{L}' \text{ s.t. } \mathtt{L} \ (<:_w; \mathtt{req}) \ \mathtt{L}', \exists \mathtt{L}'' \in (\Lambda_{rm} \cup \{\mathtt{L}_a\}) \text{ s.t. } \mathtt{L}'' <:_w \mathtt{L}' :$$

if $\mathtt{L}'' \in \{\mathtt{L}_b \mid \mathtt{L}_b <:_w \mathtt{L}_{sw}\}$ for given $\mathtt{L}$ and $\mathtt{L}'$, then it must be the case that $\mathtt{L}_{sw} <:_w \mathtt{L}'$ because $\mathtt{L}'$ is required by some weak supertype of $\mathtt{L}$ and so must not be a sublayer of a swappable and that $\mathtt{L}_a <:_w \mathtt{L}'$.

By $\mathtt{L}_a$ `req` Λ_a and $\Lambda_{rm} <:_w \Lambda_a$, we finally have

$$\forall \mathtt{L} \in \Lambda.\forall \mathtt{L}' \text{ s.t. } \mathtt{L} \ (<:_w; \mathtt{req}) \ \mathtt{L}', \exists \mathtt{L}'' \in (\Lambda_{rm} \cup \{\mathtt{L}_a\}) \text{ s.t. } \mathtt{L}'' <:_w \mathtt{L}'.$$

□

Lemma A.12 (Lemma 9). *If* Λ *wf and mtype*$(\mathtt{m}, \mathtt{C}, \Lambda)$ *defined and mtype*$(\mathtt{m}, \mathtt{D}, \Lambda)$ *undefined and* $\mathtt{C} \triangleleft \mathtt{D}$*, then* $(\exists \mathtt{L}' \in \Lambda.\mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}'))$ *or mtype*$(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda)$ *defined.*

Proof. We prove by induction on the derivation of *wf* a stronger property: If Λ *wf* and *mtype*$(\mathtt{m}, \mathtt{C}, \Lambda)$ defined and *mtype*$(\mathtt{m}, \mathtt{D}, \Lambda)$ undefined and $\mathtt{C} \triangleleft \mathtt{D}$, then $(\exists \mathtt{L}' \in \Lambda.\mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}') \wedge (\forall \mathtt{L}'', \mathtt{L}'''$ s.t. $\mathtt{L}' <:_w \mathtt{L}'' \wedge \mathtt{L}''$ `swappable` $\wedge \mathtt{L}''' <:_w \mathtt{L}''.\mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}'''))))$ or *mtype*$(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda)$ is defined.

In what follows, we define predicate *npr*$(\mathtt{m}, \mathtt{C}, \Lambda)$ by $(\exists \mathtt{L}' \in \Lambda.\mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}') \wedge (\forall \mathtt{L}'', \mathtt{L}'''$ s.t. $\mathtt{L}' <:_w \mathtt{L}'' \wedge \mathtt{L}''$ `swappable` $\wedge \mathtt{L}''' <:_w \mathtt{L}''.\mathtt{proceed} \notin \mathit{pmbody}(\mathtt{m}, \mathtt{C}, \mathtt{L}'''))))$ or *mtype*$(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda)$ is defined.

Case WF-EMPTY**:**

Trivial.

Case WF-WITH**:** $\Lambda = \Lambda_0 \cup \{\mathtt{L}_a\}$ $\Lambda_0\ wf$ $\mathtt{L}_a\ \mathtt{req}\ \Lambda'$ $\Lambda_0 <:_w \Lambda'$

If $mtype(\mathtt{m}, \mathtt{C}, \Lambda_0)$ is defined, by the induction hypothesis, $npr(\mathtt{m}, \mathtt{C}, \Lambda_0)$ holds. Since $\Lambda = \Lambda_0 \cup \{\mathtt{L}_a\}$, $npr(\mathtt{m}, \mathtt{C}, \Lambda)$ also holds.

Otherwise, it must be the case that $mtype(\mathtt{m}, \mathtt{C}, \Lambda_0)$ undefined and $mtype(\mathtt{m}, \mathtt{C}, \{\mathtt{L}_a\})$ defined. Since $\Lambda <:_w \Lambda_0 <:_w \Lambda'$ and neither $mtype(\mathtt{m}, \mathtt{C}, \Lambda_0)$ nor $mtype(\mathtt{m}, \mathtt{D}, \Lambda)$ is defined, $mtype(\mathtt{m}, \mathtt{C}, \Lambda', \Lambda' \cup \{\mathtt{L}_a\})$ is undefined. Then, $\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}_a)$ holds since if the partial method had `proceed`, it would contradict the fact that $\mathtt{L}_a$ is well-typed (in particular, $mtype(\mathtt{m}, \mathtt{C}, \Lambda', \Lambda' \cup \{\mathtt{L}_a\})$ would not be defined, as opposed to what T-PROCEED requires). If $\mathtt{L}_a$ is a sublayer of swappable layer $\mathtt{L}_{sw}$, for all $\mathtt{L}_b <:_w \mathtt{L}_{sw}$, $\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}_b)$ through the same argument (note that $\mathtt{L}_b\ \mathtt{req}\ \Lambda'$). Then, $npr(\mathtt{m}, \mathtt{C}, \Lambda)$ holds.

Case WF-SWAP**:** $\Lambda = \Lambda_{rm} \cup \{\mathtt{L}_a\}$ $\Lambda_0\ wf$ $\mathtt{L}_{sw}\ \mathtt{swappable}$ $\mathtt{L}_a <:_w \mathtt{L}_{sw}$ $\mathtt{L}_a\ \mathtt{req}\ \Lambda_a$ $\Lambda_{rm} = \Lambda_0 \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\}$ $\Lambda_{rm} <:_w \Lambda_a$

It is easy to show $\Lambda <:_{sw} \Lambda_0$ and vice versa. By Lemma A.2, $mtype(\mathtt{m}, \mathtt{C}, \Lambda_0)$ is defined and $mtype(\mathtt{m}, \mathtt{D}, \Lambda_0)$ is undefined. By the induction hypothesis, $npr(\mathtt{m}, \mathtt{C}, \Lambda_0)$, that is, either (1) $mtype(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda_0)$ is defined, or (2) $(\exists \mathtt{L}' \in \Lambda_0.\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}') \wedge (\forall \mathtt{L}'', \mathtt{L}'''$ s.t. $\mathtt{L}' <:_w \mathtt{L}'' \wedge \mathtt{L}''\ \mathtt{swappable} \wedge \mathtt{L}''' <:_w \mathtt{L}''.\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}'''))$).

We show $npr(\mathtt{m}, \mathtt{C}, \Lambda)$ by case analysis. In the case (1), we have $mtype(\mathtt{m}, \mathtt{C}, \emptyset, \Lambda)$ defined by Lemma A.2. The case (2) is also easy: if $\mathtt{L}' \in \Lambda_{rm}$, then $\mathtt{L}' \in \Lambda$; otherwise, $\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}_a)$ because $\mathtt{L}' <:_w \mathtt{L}_{sw}$ and $\mathtt{L}_a <:_w \mathtt{L}_{sw}$ and $\mathtt{L}_{sw}\ \mathtt{swappable}$, hence $npr(\mathtt{m}, \mathtt{C}, \Lambda)$. □

Lemma A.13 (Lemma 10). *If* $\{\overline{\mathtt{L}}\}\ wf$ *and* $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$*, then* $ndp(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}})$.

PROOF. By induction on the length of $\mathtt{C} <: \mathtt{D} <: \cdots \mathtt{Object}$. The case where the length is zero is trivial.

Case: $\mathtt{C} \lhd \mathtt{D}$ $mtype(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}\})$ undefined

By $\{\overline{\mathtt{L}}\}\ wf$ and Lemma A.12, we have $mtype(\mathtt{m}, \mathtt{C}, \emptyset, \{\overline{\mathtt{L}}\})$ is defined or $\exists \mathtt{L}_1 \in \{\overline{\mathtt{L}}\}.\mathtt{proceed} \notin pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}_1)$. If $mtype(\mathtt{m}, \mathtt{C}, \emptyset, \{\overline{\mathtt{L}}\})$ is defined, class C must have the definition of method m since $mtype(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}\})$ is undefined, and so NDP-CLASS finishes the case. In the other case, NDP-LAYER finishes the case.

Case: $\mathtt{C} \lhd \mathtt{D}$ $mtype(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}\})$ defined

By the induction hypothesis, $ndp(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}})$ holds. Then, NDP-SUPER finishes the case. □

Lemma A.14 (Lemma 11). *If* $ndp(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}})$*, then* $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *for some* $\overline{\mathtt{T}}$ *and* $\mathtt{T}_0$.

PROOF. By induction on $ndp(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}})$. □

Lemma A.15. *If* $\mathtt{L}.\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{L}' <:_w \mathtt{L}$*, then* $\mathtt{L}'.\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$.

PROOF. Suppose that $\mathtt{L}\ \mathtt{req}\ \Lambda_0$ and $\mathtt{L}'\ \mathtt{req}\ \Lambda_1$. Since L and L′ are well-formed, $\Lambda_1 <:_w \Lambda_0$. We proceed by induction on $\mathtt{L}.\mathtt{C}.\mathtt{m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$. We show only main cases.

Case T-SUPERP: $\mathtt{e} = \mathtt{super.m'}(\overline{\mathtt{e}})$ $\quad \mathtt{class\ C} \triangleleft \mathtt{D}$ $\quad \mathtt{L\ req}\ \Lambda_0$
$mtype(\mathtt{m'}, \mathtt{D}, \Lambda_0 \cup \{\mathtt{L}\}) = \overline{\mathtt{T}} \to \mathtt{T}$ $\quad \mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ $\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

Since $\mathtt{L'} <:_w \mathtt{L}$ and $\Lambda_1 <:_w \Lambda_0$, we have $\Lambda_1 \cup \{\mathtt{L'}\} <:_w \Lambda_0 \cup \{\mathtt{L}\}$. Then, by Lemma A.2, $mtype(\mathtt{m'}, \mathtt{D}, \Lambda_1 \cup \{\mathtt{L'}\}) = \overline{\mathtt{T}} \to \mathtt{T}$. The induction hypothesis and T-SUPERP finish the case.

Case T-PROCEED: $\mathtt{e} = \mathtt{proceed}(\overline{\mathtt{e}})$ $\quad \mathtt{L\ req}\ \Lambda_0$ $\quad mtype(\mathtt{m}, \mathtt{C}, \Lambda_0, \Lambda_0 \cup \{\mathtt{L}\}) = \overline{\mathtt{T}} \to \mathtt{T}$
$\mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ $\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

Since $\Lambda_1 <:_w \Lambda_0$, we have $\Lambda_1 \cup \{\mathtt{L'}\} <:_w \Lambda_0 \cup \{\mathtt{L}\}$. Then, by Lemma A.2, $mtype(\mathtt{m}, \mathtt{C}, \Lambda_1, \Lambda_1 \cup \{\mathtt{L'}\}) = \overline{\mathtt{T}} \to \mathtt{T}$. The induction hypothesis and T-PROCEED finish the case.

Case T-SUPERPROCEED: $\mathtt{e} = \mathtt{superproceed}(\overline{\mathtt{e}})$ $\quad \mathtt{L} \triangleleft \mathtt{L''}$ $\quad pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L''}) = \overline{\mathtt{T}} \to \mathtt{T}$
$\mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ $\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

We have that for some $\mathtt{L'''}$, $\mathtt{L'} \triangleleft \mathtt{L'''}$. Then, $\mathtt{L'''} <:_w \mathtt{L''}$ and $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L'''}) = \overline{\mathtt{T}} \to \mathtt{T}$ by Lemma A.1. The induction hypothesis and T-SUPERPROCEED finish the case. □

Lemma A.16 (Inversion for partial method body, Lemma 6). *If* $pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ L'}$ *and* $\mathtt{L\ req}\ \Lambda$ *and* $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{T}} \to \mathtt{T}_0$*, then* $\mathtt{L.C.m}; \Lambda \cup \{\mathtt{L}\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0$ *for some* $\mathtt{S}_0 <:_w \mathtt{T}_0$.

PROOF. By induction on $pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ L'}$.

Case PMB-SUPER: $LT(\mathtt{L})(\mathtt{C.m})$ undefined $\quad \mathtt{L} \triangleleft \mathtt{L''}$ $\quad pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L''}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ L'}$

By $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ and PMT-SUPER, it must be the case that $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L''}) = \overline{\mathtt{T}} \to \mathtt{T}_0$. By the induction hypothesis,

$$\mathtt{L''.C.m}; \Lambda \cup \{\mathtt{L''}\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0$$

for some $\mathtt{S}_0 <:_w \mathtt{T}_0$. Lemmas A.7 and A.15 finish the case.

Case PMB-LAYER: $LT(\mathtt{L})(\mathtt{C.m}) = \mathtt{T}_0\ \mathtt{C.m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}$ $\quad \mathtt{L'} = \mathtt{L}$
By T-PMETHOD, it must be the case that

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

for some $\mathtt{S}_0$ s.t. $\mathtt{S}_0 <:_w \mathtt{T}_0$, finishing the case. □

Lemma A.17 (Substitution for `super`, `proceed` and `superproceed`, Lemma 5).

1. *If* $\bullet; \Lambda; \Gamma \vdash \mathtt{new\ C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$ *and* $\mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{C}_0\mathtt{.m} \vdash \mathtt{<C,}(\overline{\mathtt{L'}}\mathtt{;L''})\mathtt{,}\overline{\mathtt{L}}\mathtt{>\ ok}$ *and* $\mathtt{C} \triangleleft \mathtt{D}$ *and* $\mathtt{L''} <:_w \mathtt{L} \triangleleft \mathtt{L'}$ *and* $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$ *and* $\mathtt{proceed} \in \mathtt{e} \implies ndp(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L'}}, \overline{\mathtt{L}})$*, then* $\bullet; \Lambda; \Gamma \vdash S\,\mathtt{e} : \mathtt{T}$ *where*

$$S = \left[\begin{array}{ll} \mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<C,}\overline{\mathtt{L'}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m} & \mathtt{/proceed,} \\ \mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>} & \mathtt{/super,} \\ \mathtt{new\ C}_0(\overline{\mathtt{v}})\mathtt{<C,L',(}\overline{\mathtt{L'}}\mathtt{;L'')\mathtt{,}}\overline{\mathtt{L}}\mathtt{>.m} & \mathtt{/superproceed} \end{array}\right].$$

2. *If* $\bullet; \Lambda; \Gamma \vdash \mathtt{new\ C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$ *and* $\mathtt{C.m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{C}_0\mathtt{.m} \vdash \mathtt{<C,}\overline{\mathtt{L'}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>\ ok}$ *and* $\mathtt{C} \triangleleft \mathtt{D}$ *and* $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$*, then* $\bullet; \Lambda; \Gamma \vdash \mathtt{[new\ C}_0(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>/super]e} : \mathtt{T}$.

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

Case T-SuperB**:**

Cannot happen.

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

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{v}})\mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m'}(S\,\overline{\mathtt{e}})\ :\ \mathtt{T}$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{v}})\ :\ \mathtt{C_0}$. Next, we show $\mathtt{C_0.m} \vdash \mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\ \mathtt{ok}$. By $\mathtt{C_0.m} \vdash \mathtt{<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\ \mathtt{ok}$, we have $\mathtt{C_0} <: \mathtt{C}$, from which $\mathtt{C_0} <: \mathtt{D}$ follows, and $\{\overline{\mathtt{L}}\}\ wf$. By Lemma A.11 and $\mathtt{L''} \in \{\overline{\mathtt{L}}\}$ and $\mathtt{L''} <:_w \mathtt{L}$, for any $\mathtt{L_1}$ such that $\mathtt{L\ req\ L_1}$, there exists $\mathtt{L_2} \in \{\overline{\mathtt{L}}\}$ such that $\mathtt{L_2} <:_w \mathtt{L_1}$; so, $\{\overline{\mathtt{L}}\} <:_w \Lambda' \cup \{\mathtt{L}\}$. Then, by $\mathit{mtype}(\mathtt{m'}, \mathtt{D}, \Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{T}}'\to\mathtt{T}$ and Lemma A.2, we have $\mathit{mtype}(\mathtt{m'}, \mathtt{D}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}}'\to\mathtt{T}$; moreover, by Lemma A.13, $\mathit{ndp}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}})$. So, $\mathtt{C_0.m} \vdash \mathtt{<D,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\ \mathtt{ok}$. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\,\overline{\mathtt{e}} : \overline{\mathtt{S}}'$ and, by assumption, $\overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$. Finally, T-InvkA finishes the case.

Case T-Proceed**:** $\mathtt{e} = \mathtt{proceed}(\overline{\mathtt{e}})$ $\quad \mathit{mtype}(\mathtt{m}, \mathtt{C}, \Lambda', \Lambda' \cup \{\mathtt{L}\}) = \overline{\mathtt{T}}'\to\mathtt{T}$
$\mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}'$ $\quad \mathtt{L\ req}\ \Lambda'$ $\quad \overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>}(\overline{\mathtt{v}})\mathtt{.m}(S\,\overline{\mathtt{e}})\ :\ \mathtt{T}$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{v}})\ :\ \mathtt{C_0}$. Since $\mathtt{proceed} \in \mathtt{e}$, we have $\mathit{ndp}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}})$, from which $\mathtt{C_0.m} \vdash \mathtt{<C,}\overline{\mathtt{L}}'\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\ \mathtt{ok}$ and follow. By Lemmas A.14 and A.3, we have $\mathit{mtype}(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}}'\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}}'\to\mathtt{T}$, too. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\,\overline{\mathtt{e}} : \overline{\mathtt{S}}'$ and, by assumption, $\overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$. Finally, T-InvkA finishes the case.

Case T-SuperProceed**:** $\mathtt{e} = \mathtt{superproceed}(\overline{\mathtt{e}})$ $\quad \mathtt{L} \lhd \mathtt{L}'$
$\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}') = \overline{\mathtt{T}}'\to\mathtt{T}$
$\mathtt{L.C.m}; \Lambda; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}'$ $\quad \overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0<C,L'},(\overline{\mathtt{L}}'\mathtt{;L''})\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m}(S\,\overline{\mathtt{e}})\ :\ \mathtt{T}$.
By assumption, we have $\bullet; \Lambda; \Gamma \vdash \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{v}})\ :\ \mathtt{C_0}$ and $\mathtt{C_0.m} \vdash \mathtt{<C,}(\overline{\mathtt{L}}'\mathtt{;L''})\mathtt{,}\overline{\mathtt{L}}\mathtt{>}\ \mathtt{ok}$ and $\mathtt{L''} <:_w \mathtt{L}'$. Also, $\mathit{pmtype}(\mathtt{m}, \mathtt{C}, \mathtt{L}') = \overline{\mathtt{T}}'\to\mathtt{T}$, by assumption. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\,\overline{\mathtt{e}} : \overline{\mathtt{S}}'$ and, by assumption, $\overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$. Finally, T-InvkAL finishes the case.

Case T-With**:** $\mathtt{e} = \mathtt{with}\ \mathtt{e}_l\ \mathtt{e}_0$ $\quad \mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{e}_l : \mathtt{L}$ $\quad \mathtt{L\ req}\ \Lambda_0$
$\Lambda <:_w \Lambda_0$ $\quad \mathtt{L.C.m}; \Lambda \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}$

Since $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$, we have $\Lambda \cup \{\mathtt{L}\} <:_{sw} \{\overline{\mathtt{L}}\}$ by LSSW-Intro. By the induction hypothesis, $\bullet; \Lambda; \Gamma \vdash S\,\mathtt{e}_l : \mathtt{L}$ and $\bullet; \Lambda \cup \{\mathtt{L}\}; \Gamma \vdash S\,\mathtt{e}_0 : \mathtt{T}$. T-With finishes the case.

Case T-Swap**:** $\mathtt{e} = \mathtt{swap}\ (\mathtt{e}_l\mathtt{,L}_{sw})\ \mathtt{e}_0$ $\quad \mathtt{L.C.m}; \Lambda; \Gamma \vdash \mathtt{e}_l : \mathtt{L}$ $\quad \mathtt{L\ req}\ \Lambda_0$
$\mathtt{L}_{sw}\ \mathtt{swappable}$ $\quad \mathtt{L} <:_w \mathtt{L}_{sw}$
$\Lambda_{rm} = \Lambda \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\}$ $\quad \Lambda_{rm} <:_w \Lambda_0$
$\mathtt{L.C.m}; \Lambda_{rm} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}$

Since $\Lambda <:_{sw} \{\overline{\mathtt{L}}\}$, we have $\Lambda_{rm} \cup \{\mathtt{L}\} <:_{sw} \{\overline{\mathtt{L}}\}$ by LSSW-Intro. By the induction hypothesis, $\bullet; \Lambda; \Gamma \vdash S\,\mathtt{e}_l : \mathtt{L}$ and $\bullet; \Lambda_{rm} \cup \{\mathtt{L}\}; \Gamma \vdash S\,\mathtt{e}_0 : \mathtt{T}$. T-Swap finishes the case.

2. By induction on $\mathtt{C.m}; \Lambda; \Gamma \vdash \mathtt{e} : \mathtt{T}_0$ with case analysis on the last typing rule used. We show only main cases below (note that none of the cases T-Proceed and T-SuperP and T-SuperProceed can happen).

Case T-SuperB: $\texttt{e} = \texttt{super.m}'(\overline{\texttt{e}})$ $\quad$ $\mathit{mtype}(\texttt{m}', \texttt{D}, \emptyset) = \overline{\texttt{T}}' \to \texttt{T}_0$
$\texttt{C.m}; \Lambda; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}}'$ $\quad$ $\overline{\texttt{S}}' <: \overline{\texttt{T}}'$

Let $S = [\texttt{new C}_0(\overline{\texttt{v}})\texttt{<D,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>/super}]$. It suffices to show that $\bullet; \Lambda; \Gamma \vdash \texttt{new C}_0(\overline{\texttt{v}})\texttt{<D,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m}'(S\,\overline{\texttt{e}}) : \texttt{T}_0$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \texttt{new C}_0(\overline{\texttt{v}}) : \texttt{C}_0$. Next, we show $\texttt{C}_0\texttt{.m} \vdash \texttt{<D,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{> ok}$. By $\texttt{C}_0\texttt{.m} \vdash \texttt{<C,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{> ok}$, we have $\texttt{C}_0 <: \texttt{C}$, from which $\texttt{C}_0 <: \texttt{D}$ follows, and $\{\overline{\texttt{L}}\}$ *wf*. By $\mathit{mtype}(\texttt{m}', \texttt{D}, \emptyset) = \overline{\texttt{T}}' \to \texttt{T}_0$ and Lemma A.2, we have $\mathit{mtype}(\texttt{m}', \texttt{D}, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}}' \to \texttt{T}_0$; moreover, by Lemma A.13, $\mathit{ndp}(\texttt{m}, \texttt{D}, \overline{\texttt{L}}, \overline{\texttt{L}})$. So, $\texttt{C}_0\texttt{.m} \vdash \texttt{<D,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{> ok}$. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\,\overline{\texttt{e}} : \overline{\texttt{S}}'$ and, by assumption, $\overline{\texttt{S}}' <: \overline{\texttt{T}}'$. Finally, T-InvkA finishes the case. □

Lemma A.18 (Inversion for method body, Lemma 7). *Suppose* $\{\overline{\texttt{L}}\}$ *wf* *and* $\mathit{mbody}(\texttt{m}, \texttt{C}, \overline{\texttt{L}}', \overline{\texttt{L}}) = \overline{\texttt{x}}\texttt{.e}_0$ *in* $\texttt{C}', \overline{\texttt{L}}''$ *and* $\mathit{mtype}(\texttt{m}, \texttt{C}, \{\overline{\texttt{L}}'\}, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}} \to \texttt{T}_0$ *and* $\mathit{ndp}(\texttt{m}, \texttt{C}, \overline{\texttt{L}}', \overline{\texttt{L}})$.

1. *If* $\overline{\texttt{L}}'' = \overline{\texttt{L}}'''\texttt{;L}_0$, *then* $\texttt{L}_0$ `req` Λ *and* $\texttt{L}_0\texttt{.C}'\texttt{.m}; \Lambda \cup \{\texttt{L}_0\}; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C}' \vdash \texttt{e}_0 : \texttt{U}_0$ *and* $\texttt{C} <: \texttt{C}'$ *and* $\texttt{U}_0 <: \texttt{T}_0$ *and* $\mathit{ndp}(\texttt{m}, \texttt{C}', \overline{\texttt{L}}'', \overline{\texttt{L}})$ *for some* Λ *and* $\texttt{U}_0$.
2. *If* $\overline{\texttt{L}}'' = \bullet$, *then* $\texttt{C}'\texttt{.m}; \emptyset; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C}' \vdash \texttt{e}_0 : \texttt{U}_0$ *and* $\texttt{C} <: \texttt{C}'$ *and* $\texttt{U}_0 <: \texttt{T}_0$ *and* $\mathit{ndp}(\texttt{m}, \texttt{C}', \bullet, \overline{\texttt{L}})$ *for some* $\texttt{U}_0$.

Proof. Both 1 and 2 are proved simultaneously by induction on $\mathit{mbody}(\texttt{m}, \texttt{C}, \overline{\texttt{L}}', \overline{\texttt{L}}) = \overline{\texttt{x}}\texttt{.e}_0$ `in` $\texttt{C}', \overline{\texttt{L}}''$.

Case MB-Class: $\texttt{class C} \lhd \texttt{D \{...}\ \ \texttt{S}_0\ \texttt{m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\texttt{\{return e}_0\texttt{;\}}\ \texttt{...\}}$
$\texttt{C}' = \texttt{C}$ $\quad$ $\overline{\texttt{L}}' = \bullet$ $\quad$ $\overline{\texttt{L}}'' = \bullet$

By T-Class, T-Method, MT-Class, it must be the case that

$$\texttt{T}_0, \overline{\texttt{T}} = \texttt{S}_0, \overline{\texttt{S}} \qquad \texttt{C.m}; \emptyset; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{U}_0 \qquad \texttt{U}_0 <: \texttt{T}_0$$

for some $\texttt{U}_0$. We have $\mathit{ndp}(\texttt{m}, \texttt{C}, \bullet, \overline{\texttt{L}})$ by NDP-Class, finishing the case.

Case MB-Layer: $\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}_0) = \overline{\texttt{x}}\texttt{.e}_0$ `in` $\texttt{L}_1$ $\quad$ $\texttt{C}' = \texttt{C}$ $\quad$ $\overline{\texttt{L}}'' = \overline{\texttt{L}}'$

By the definition of *pmbody*, there exists some $\texttt{L}_1$ such that $\mathit{LT}(\texttt{L}_1)(\texttt{C.m}) = \texttt{S}_0\ \texttt{C.m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\texttt{\{ return e; \}}$ and $\texttt{L}_0 <:_w \texttt{L}_1$. By T-PMethod, it must be the case that

$$\texttt{T}_0, \overline{\texttt{T}} = \texttt{S}_0, \overline{\texttt{S}} \qquad \texttt{L}_1\ \texttt{req}\ \Lambda_1 \qquad \texttt{L}_1\texttt{.C.m}; \Lambda_1 \cup \{\texttt{L}_1\}; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{U}_0 \qquad \texttt{U}_0 <: \texttt{T}_0$$

for some $\texttt{U}_0$ and Λ_1. It is easy to show by induction on $\texttt{L}_0 <:_w \texttt{L}_1$ using Lemma A.7 and T-Layer and T-LayerSW that

$$\texttt{L}_0\texttt{.C.m}; \Lambda \cup \{\texttt{L}_0\}; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{U}_0$$

for some Λ such that $\texttt{L}_0$ `req` Λ. Finally, we have $\mathit{ndp}(\texttt{m},\texttt{C}',\overline{\texttt{L}}'',\overline{\texttt{L}})$ by assumption, finishing the case.

Case MB-Super: $\overline{\texttt{L}}' = \bullet$ $\quad$ $\texttt{class C} \lhd \texttt{D \{ ...}\ \ \overline{\texttt{M}}\ \texttt{\}}$ $\quad$ $\texttt{m} \notin \overline{\texttt{M}}$
$\mathit{mbody}(\texttt{m}, \texttt{D}, \overline{\texttt{L}}, \overline{\texttt{L}}) = \overline{\texttt{x}}\texttt{.e}_0$ `in` $\texttt{C}', \overline{\texttt{L}}''$

By MT-Super, it must be the case that $\mathit{mtype}(\texttt{m}, \texttt{D}, \{\overline{\texttt{L}}\}, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}} \to \texttt{T}_0$. By Lemma A.13, we have $\mathit{ndp}(\texttt{m},\texttt{D},\overline{\texttt{L}},\overline{\texttt{L}})$. The induction hypothesis and transitivity of subtyping finish the case.

Case MB-NEXTLAYER: $\overline{\mathtt{L}}' = \overline{\mathtt{L}}_b\,;\mathtt{L}_1 \qquad \mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}_1)$ undefined
$\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_b,\overline{\mathtt{L}}) = \overline{\mathtt{x}}.\,\mathtt{e}_0$ in $\mathtt{C}',\overline{\mathtt{L}}''$

We show $\mathit{ndp}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_b,\overline{\mathtt{L}})$ holds by case analysis on $\mathit{ndp}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_b;\mathtt{L}_1,\overline{\mathtt{L}})$. The cases NDP-SUPER and NDP-CLASS are easy. The case NDP-LAYER is easy, too: since $\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}_1)$ undefined, by NDP-LAYER, we have $\mathit{ndp}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_b,\overline{\mathtt{L}})$. Since $\mathit{pmbody}(\mathtt{m},\mathtt{C},\mathtt{L}_1)$ is undefined and $\mathit{mtype}(\mathtt{m},\mathtt{C},\{\overline{\mathtt{L}}'\},\{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}}\to\mathtt{T}_0$, it must be the case that $\mathit{mtype}(\mathtt{m},\mathtt{C},\{\overline{\mathtt{L}}_b\},\{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}}\to\mathtt{T}_0$. Then, the induction hypothesis finishes the case. □

Theorem A.1 (Subject Reduction). *Suppose* $\vdash (CT, LT)$ ok. *If* $\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{e}:\mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf and* $\overline{\mathtt{L}}\vdash\mathtt{e}\longrightarrow\mathtt{e}'$, *then* $\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{e}':\mathtt{S}$ *for some* S *such that* $\mathtt{S}<:\mathtt{T}$.

PROOF. By induction on $\overline{\mathtt{L}}\vdash\mathtt{e}\longrightarrow\mathtt{e}'$ with case analysis on the last reduction rule used. We show only main cases.

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

By T-FIELD and T-NEW, it must be the case that

$$\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\overline{\mathtt{v}}:\overline{\mathtt{D}} \qquad \overline{\mathtt{D}}<:\overline{\mathtt{C}} \qquad \mathtt{C}=\mathtt{C}_i$$

Then, we have $\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{v}_i:\mathtt{D}_i$ and $\mathtt{D}_i<:\mathtt{C}_i$, finishing the case.

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

By T-INVK and T-NEW, it must be the case that

$$\begin{array}{lll}\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\overline{\mathtt{v}}:\overline{\mathtt{S}} & \mathit{fields}(\mathtt{C}_0)=\overline{\mathtt{T}}\ \overline{\mathtt{f}} & \overline{\mathtt{S}}<:\overline{\mathtt{T}}\\ \mathit{mtype}(\mathtt{m},\mathtt{C}_0,\{\overline{\mathtt{L}}\})=\overline{\mathtt{T}}'\to\mathtt{T} & \bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\overline{\mathtt{w}}:\overline{\mathtt{S}}' & \overline{\mathtt{S}}'<:\overline{\mathtt{T}}'.\end{array}$$

By Lemma A.13, $\mathit{ndp}(\mathtt{m},\mathtt{C}_0,\overline{\mathtt{L}},\overline{\mathtt{L}})$ and so $\mathtt{C}_0.\mathtt{m}\vdash\texttt{<}\mathtt{C}_0,\overline{\mathtt{L}},\overline{\mathtt{L}}\texttt{>}$ ok holds. Since $\{\overline{\mathtt{L}}\}<:_{sw}\{\overline{\mathtt{L}}\}$, we have

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

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

Case R-INVKP: $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}'\texttt{>}.\mathtt{m}(\overline{\mathtt{w}})$
$\mathit{mbody}(\mathtt{m},\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\,\mathtt{e}_0$ in $\mathtt{C}'',(\overline{\mathtt{L}}''';\mathtt{L}_0)$
$\mathtt{C}''\ \triangleleft\ \mathtt{D}$
$\mathtt{L}_0\ \triangleleft\ \mathtt{L}_1$

$$\mathtt{e}' = \left[\begin{array}{ll}\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}) & /\mathtt{this},\\ \overline{\mathtt{w}} & /\overline{\mathtt{x}},\\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{C}'',\overline{\mathtt{L}}''',\overline{\mathtt{L}}'\texttt{>}.\mathtt{m} & /\mathtt{proceed},\\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{D},\overline{\mathtt{L}}',\overline{\mathtt{L}}'\texttt{>} & /\mathtt{super},\\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{C}'',\mathtt{L}_1,(\overline{\mathtt{L}}''';\mathtt{L}_0),\overline{\mathtt{L}}'\texttt{>}.\mathtt{m} & /\mathtt{superproceed}\end{array}\right]\mathtt{e}_0$$

By T-INVKA, it must be the case that

$$\begin{array}{lll}\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}):\mathtt{C}_0 & \mathtt{C}_0.\mathtt{m}\vdash\texttt{<}\mathtt{C}',\overline{\mathtt{L}}'',\overline{\mathtt{L}}'\texttt{>}\ \mathtt{ok} & \{\overline{\mathtt{L}}\}<:_{sw}\{\overline{\mathtt{L}}'\}\\ \mathit{mtype}(\mathtt{m},\mathtt{C}',\{\overline{\mathtt{L}}''\},\{\overline{\mathtt{L}}'\})=\overline{\mathtt{T}}'\to\mathtt{T} & \bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\overline{\mathtt{w}}:\overline{\mathtt{S}}' & \overline{\mathtt{S}}'<:\overline{\mathtt{T}}'\end{array}$$

for some $\overline{\mathtt{T}}'$ and $\overline{\mathtt{S}}'$.

By Lemma A.18,

$$\begin{array}{l}
\mathtt{L_0.C''.m};\Lambda\cup\{\mathtt{L_0}\};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:\mathtt{C''}\vdash\mathtt{e_0}:\mathtt{S}\\
\mathtt{L_0\ req\ \Lambda}\\
\mathtt{C'}<:\mathtt{C''}\\
\mathtt{S}<:\mathtt{T}\\
\mathit{ndp}(\mathtt{m},\mathtt{C''},(\overline{\mathtt{L}}''';\mathtt{L_0}),\overline{\mathtt{L}}')
\end{array}$$

and for some Λ and $\mathtt{S}$.

By S-Trans, $\mathtt{C_0}<:\mathtt{C''}$. From $\mathit{ndp}(\mathtt{m},\mathtt{C''},(\overline{\mathtt{L}}''';\mathtt{L_0}),\overline{\mathtt{L}}')$ and $\mathtt{C_0.m}\vdash\mathtt{<C',\overline{L}'',\overline{L}'>\ ok}$, it follows that $\mathtt{C_0.m}\vdash\mathtt{<C'',(\overline{L}''';L_0),\overline{L}'>\ ok}$.

By $\{\overline{\mathtt{L}}'\}$ *wf* and Lemma A.11 and $\mathtt{L_0}\in\overline{\mathtt{L}}'$ and $\mathtt{L_0\ req\ \Lambda}$, we have $\forall\mathtt{L}\in\Lambda,\exists\mathtt{L}'\in\overline{\mathtt{L}}'.\mathtt{L}'<:_w\mathtt{L}$. So, by LSS-Intro, we have $\{\overline{\mathtt{L}}'\}=\{\overline{\mathtt{L}}'\}\cup\{\mathtt{L_0}\}<:_w\Lambda\cup\{\mathtt{L_0}\}$. By this fact and $\{\overline{\mathtt{L}}\}<:_{sw}\{\overline{\mathtt{L}}'\}$, we get $\{\overline{\mathtt{L}}\}<:_{sw}\Lambda\cup\{\mathtt{L_0}\}$. By Lemma A.7,

$$\mathtt{L_0.C''.m};\{\overline{\mathtt{L}}\};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:\mathtt{C''}\vdash\mathtt{e_0}:\mathtt{S}$$

By $\mathit{ndp}(\mathtt{m},\mathtt{C''},(\overline{\mathtt{L}}''';\mathtt{L_0}),\overline{\mathtt{L}}')$ and the definition of *ndp*, $\mathtt{proceed}\in\mathtt{e_0}$ implies $\mathit{ndp}(\mathtt{m},\mathtt{C'},\overline{\mathtt{L}}''',\overline{\mathtt{L}}')$. Then, by Lemmas A.8 and Lemma A.17(1),

$$\bullet;\{\overline{\mathtt{L}}\};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:\mathtt{C''}\vdash\left[\begin{array}{ll}
\mathtt{new\ C_0(\overline{v})<C'',\overline{L}''',\overline{L}'>.m} & \mathtt{/proceed,}\\
\mathtt{new\ C_0(\overline{v})<D,\overline{L}',\overline{L}'>} & \mathtt{/super,}\\
\mathtt{new\ C_0(\overline{v})<C'',L_1,(\overline{L}''';L_0),\overline{L}'>.m} & \mathtt{/superproceed}
\end{array}\right]\mathtt{e_0}:\mathtt{S}$$

By Lemmas A.8, A.5 and A.9, $\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{e'}:\mathtt{S'}$ for some $\mathtt{S'}<:\mathtt{S}$. By S-Trans, $\mathtt{S'}<:\mathtt{T}$, finishing the case.

Case R-InvkSP**:**
$$\begin{array}{l}
\mathtt{e}=\mathtt{new\ C_0(\overline{v})<C',L_1,(\overline{L}'';L_0),\overline{L}'>.m(\overline{w})}\\
\mathit{pmbody}(\mathtt{m},\mathtt{C'},\mathtt{L_1})=\overline{\mathtt{x}}.\mathtt{e_0\ in\ L_2}\\
\mathtt{C'}\lhd\mathtt{D}\\
\mathtt{L_2}\lhd\mathtt{L_3}\\
\mathtt{e'}=\left[\begin{array}{ll}
\mathtt{new\ C_0(\overline{v})} & \mathtt{/this}\\
\overline{\mathtt{w}} & \mathtt{/\overline{x}}\\
\mathtt{new\ C_0(\overline{v})<C'',\overline{L}'',\overline{L}'>.m} & \mathtt{/proceed}\\
\mathtt{new\ C_0(\overline{v})<D,\overline{L}',\overline{L}'>} & \mathtt{/super,}\\
\mathtt{new\ C_0(\overline{v})<C'',L_3,(\overline{L}'';L_0),\overline{L}'>.m} & \mathtt{/superproceed}
\end{array}\right]\mathtt{e_0}
\end{array}$$

By T-InvkAL, it must be the case that

$$\begin{array}{llll}
\bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\mathtt{new\ C_0(\overline{v})}:\mathtt{C_0} & \mathtt{C.m}\vdash\mathtt{<C',(\overline{L}'';L_0),\overline{L}'>\ ok} & \{\overline{\mathtt{L}}\}<:_{sw}\{\overline{\mathtt{L}}'\} & \\
\mathtt{L_0}<:_w\mathtt{L_1} & \mathit{pmtype}(\mathtt{m},\mathtt{C'},\mathtt{L_1})=\overline{\mathtt{T}}'\to\mathtt{T} & \bullet;\{\overline{\mathtt{L}}\};\Gamma\vdash\overline{\mathtt{w}}:\overline{\mathtt{S}}' & \overline{\mathtt{S}}'<:\overline{\mathtt{T}}'
\end{array}$$

for some $\overline{\mathtt{T}}'$ and $\overline{\mathtt{S}}'$. Let Λ be the layer set such that $\mathtt{L_1\ req\ \Lambda}$. By Lemma A.16,

$$\mathtt{L_1.C'.m};\Lambda\cup\{\mathtt{L_1}\};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:\mathtt{C'}\vdash\mathtt{e_0}:\mathtt{S}$$

and $\mathtt{S}<:\mathtt{T}$ for some $\mathtt{S}$.

Since $\mathtt{L_0}<:_w\mathtt{L_1}$, $\mathtt{L_0}$ requires all the layers that $\mathtt{L_1}$ requires (including Λ). By $\{\overline{\mathtt{L}}'\}$ *wf* and Lemma A.11 and $\mathtt{L_0}\in\overline{\mathtt{L}}'$, we have $\forall\mathtt{L}\in\Lambda,\exists\mathtt{L}'\in\{\overline{\mathtt{L}}'\}$ such that $\mathtt{L}'<:_w\mathtt{L}$. So, $\{\overline{\mathtt{L}}'\}=\{\overline{\mathtt{L}}'\}\cup\{\mathtt{L_0}\}<:_w\Lambda\cup\{\mathtt{L_1}\}$. By this and $\{\overline{\mathtt{L}}\}<:_{sw}\{\overline{\mathtt{L}}'\}$, we have $\{\overline{\mathtt{L}}\}<:_{sw}\Lambda\cup\{\mathtt{L_1}\}$. By Lemma A.7,

$$\mathtt{L_1.C'.m};\{\overline{\mathtt{L}}\};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:\mathtt{C'}\vdash\mathtt{e_0}:\mathtt{S}.$$

By $ndp(\mathtt{m}, \mathtt{C}', (\overline{\mathtt{L}}'';\mathtt{L}_0), \overline{\mathtt{L}}')$ (which follows from $\mathtt{C.m} \vdash \mathtt{<C'},(\overline{\mathtt{L}}'';\mathtt{L}_0),\overline{\mathtt{L}}'\mathtt{>\ ok}$) and the definition of *ndp*, $\mathtt{proceed} \in \mathtt{e}_0$ implies $ndp(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}'', \overline{\mathtt{L}}')$ holds. Then, by Lemmas A.8, A.5, A.9 and A.17(1), $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{S}'$ for some $\mathtt{S}' <: \mathtt{S}$. By S-Trans, $\mathtt{S}' <: \mathtt{T}$, finishing the case.

Case R-InvkB: $\mathtt{e} = \mathtt{new\ C_0(\overline{v})<C',\overline{L}'',\overline{L}'>.m(\overline{w})}$
$mbody(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}'', \bullet$
$\mathtt{C}'' \lhd \mathtt{D}$
$$\mathtt{e}' = \begin{bmatrix} \mathtt{new\ C_0(\overline{v})} & /\mathtt{this} \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}} \\ \mathtt{new\ C_0(\overline{v})<D,\overline{L}',\overline{L}'>} & /\mathtt{super} \end{bmatrix} \mathtt{e}_0$$

By T-InvkA, it must be the case that

$$\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{new\ C_0(\overline{v})} : \mathtt{C_0} \qquad \mathtt{C_0.m} \vdash \mathtt{<C',\overline{L}'',\overline{L}'>\ ok} \qquad \{\overline{\mathtt{L}}\} <:_{sw} \{\overline{\mathtt{L}}'\}$$
$$mtype(\mathtt{m}, \mathtt{C}', \{\overline{\mathtt{L}}''\}, \{\overline{\mathtt{L}}'\}) = \overline{\mathtt{T}}' \to \mathtt{T} \qquad \bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{S}}' \qquad \overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$$

for some $\overline{\mathtt{T}}'$ and $\overline{\mathtt{S}}'$. By Lemma A.18,

$$\mathtt{C}''.\mathtt{m}; \emptyset; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}'' \vdash \mathtt{e}_0 : \mathtt{S}$$

and $\mathtt{C}' <: \mathtt{C}''$ and $\mathtt{S} <: \mathtt{T}$ and $ndp(\mathtt{m}, \mathtt{C}'', \bullet, \overline{\mathtt{L}}')$ for some S. By S-Trans, $\mathtt{C_0} <: \mathtt{C}''$. By Lemma A.5,

$$\mathtt{C}''.\mathtt{m}; \{\overline{\mathtt{L}}\}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}'' \vdash \mathtt{e}_0 : \mathtt{S}$$

By Lemmas A.8, A.5, A.9 and A.17(2), $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{S}'$ for some $\mathtt{S}' <: \mathtt{S}$. By S-Trans, $\mathtt{S}' <: \mathtt{T}$, finishing the case.

Case RC-With: $\mathtt{e} = \mathtt{with\ new\ L()\ e_0}$ $\mathtt{e}' = \mathtt{with\ new\ L()\ e_0}'$
$with(\mathtt{L}, \overline{\mathtt{L}}) = \overline{\mathtt{L}}'$ $\overline{\mathtt{L}}' \vdash \mathtt{e}_0 \longrightarrow \mathtt{e_0}'$

By T-With, it must be the case that

$$\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{new\ L()} : \mathtt{L} \qquad \mathtt{L\ req}\ \Lambda \qquad \{\overline{\mathtt{L}}\} <:_w \Lambda \qquad \bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}$$

for some Λ. Here, $\{\overline{\mathtt{L}}'\} = \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}$ *wf* by Wf-With. By the induction hypothesis, $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e_0}' : \mathtt{S}$ for some $\mathtt{S} <: \mathtt{T}$. By T-With, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e_0}' : \mathtt{S}$, finishing the case.

Case RC-WithArg: $\mathtt{e} = \mathtt{with\ e}_l\ \mathtt{e}_0$ $\mathtt{e}' = \mathtt{with\ e}_l'\ \mathtt{e}_0$ $\overline{\mathtt{L}} \vdash \mathtt{e}_l \longrightarrow \mathtt{e}_l'$

By T-With, it must be the case that

$$\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_l : \mathtt{L} \qquad \mathtt{L\ req}\ \Lambda \qquad \{\overline{\mathtt{L}}\} <:_w \Lambda \qquad \bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}$$

for some Λ. By the induction hypothesis, we have $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}_l' : \mathtt{L}'$ for some $\mathtt{L}' <: \mathtt{L}$. By LS-Extends, $\mathtt{L}'$ and L have the same require clause Λ. Since $\mathtt{L}' <: \mathtt{L}$, we have $\mathtt{L}' <:_w \mathtt{L}$, and $\{\overline{\mathtt{L}}\} \cup \{\mathtt{L}'\} <:_w \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}$. By Lemma A.7 and T-With, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{e}' : \mathtt{T}$. Reflexivity of <: finishes the case.

Case R-WithVal: $\mathtt{e} = \mathtt{with\ new\ L()\ v_0}$ $\mathtt{e}' = \mathtt{v_0}$

By T-With, it must be the case that $\bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{v_0} : \mathtt{T}$. By Lemma A.8, $\bullet; \{\overline{\mathtt{L}}\}; \Gamma \vdash \mathtt{v_0} : \mathtt{T}$, finishing the case.

Case RC-SWAP: $\texttt{e} = \texttt{swap (new L(),}\texttt{L}_{sw}\texttt{) e}_0$ $\quad \texttt{e}' = \texttt{swap (new L(),}\texttt{L}_{sw}\texttt{) e}_0'$
$\mathit{swap}(\texttt{L}, \texttt{L}_{sw}, \overline{\texttt{L}}) = \overline{\texttt{L}}'$ $\quad \overline{\texttt{L}}' \vdash \texttt{e}_0 \longrightarrow \texttt{e}_0'$

By T-SWAP, it must be the case that

$$\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{new L()} : \texttt{L} \qquad \texttt{L}_{sw}\ \texttt{swappable} \qquad \texttt{L} <:_w \texttt{L}_{sw} \qquad \texttt{L req } \Lambda$$
$$\Lambda_{rm} = \{\overline{\texttt{L}}\} \setminus \{\texttt{L}' \mid \texttt{L}' <:_w \texttt{L}_{sw}\} \qquad \Lambda_{rm} <:_w \Lambda \qquad \bullet; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$$

for some L, Λ, and Λ_{rm}. Here, $\{\overline{\texttt{L}}'\} = \Lambda_{rm} \cup \{\texttt{L}\}$. Then, $\{\overline{\texttt{L}}'\}$ *wf* by WF-SWAP. By the induction hypothesis, $\bullet; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0' : \texttt{S}$ for some $\texttt{S} <: \texttt{T}$. By T-SWAP, $\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_0' : \texttt{S}$, finishing the case.

Case RC-SWAPARG: $\texttt{e} = \texttt{swap (e}_l\texttt{,L}_{sw}\texttt{) e}_0 \quad \texttt{e}' = \texttt{swap (e}_l'\texttt{,L}_{sw}\texttt{) e}_0$
$\overline{\texttt{L}} \vdash \texttt{e}_l \longrightarrow \texttt{e}_l'$

By T-SWAP, it must be the case that

$$\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_l : \texttt{L} \qquad \texttt{L}_{sw}\ \texttt{swappable} \qquad \texttt{L} <:_w \texttt{L}_{sw} \qquad \texttt{L req } \Lambda$$
$$\Lambda_{rm} = \{\overline{\texttt{L}}\} \setminus \{\texttt{L}' \mid \texttt{L}' <:_w \texttt{L}_{sw}\} \qquad \Lambda_{rm} <:_w \Lambda \qquad \bullet; \Lambda_{rm} \cup \{\texttt{L}\}; \Gamma \vdash \texttt{e}_0 : \texttt{T}$$

for some L, Λ, and Λ_{rm}. By the induction hypothesis, we have $\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_l' : \texttt{L}'$ for some $\texttt{L}' <: \texttt{L}$. By LS-EXTENDS, L′ and L have the same require clause Λ. Since $\texttt{L}' <: \texttt{L}$, we have $\texttt{L}' <:_w \texttt{L}$, $\texttt{L}' <:_w \texttt{L}_{sw}$, and $\Lambda_{rm} \cup \{\texttt{L}'\} <:_w \Lambda_{rm} \cup \{\texttt{L}\} <:_w \Lambda$. By Lemma A.7 and T-SWAP, $\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}' : \texttt{T}$. Reflexivity of <: finishes the case.

Case R-SWAPVAL:

Similar to Case R-WITHVAL.

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

By T-INVK, it must be the case that

$$\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \mathit{mtype}(\texttt{m}, \texttt{C}_0, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}} \to \texttt{T} \qquad \bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \overline{\texttt{S}} <: \overline{\texttt{T}}.$$

for some $\overline{\texttt{T}}$ and $\overline{\texttt{S}}$. By the induction hypothesis, $\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_0' : \texttt{D}_0$ for some $\texttt{D}_0 <: \texttt{C}_0$. By Lemma A.6, $\mathit{mtype}(\texttt{m}, \texttt{D}_0, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}} \to \texttt{S}$ and $\texttt{S} <: \texttt{T}$ for some S. By T-INVK, $\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_0'\texttt{.m}(\overline{\texttt{e}}) : \texttt{S}$, finishing the case.

Case RC-INVKARG: $\texttt{e} = \texttt{e}_0\texttt{.m(}\ldots\texttt{,e}_i\texttt{,}\ldots\texttt{)} \quad \overline{\texttt{L}} \vdash \texttt{e}_i \longrightarrow \texttt{e}_i' \quad \texttt{e}' = \texttt{e}_0\texttt{.m(}\ldots\texttt{,e}_i'\texttt{,}\ldots\texttt{)}$

By T-INVK, it must be the case that

$$\bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \qquad \mathit{mtype}(\texttt{m}, \texttt{C}_0, \{\overline{\texttt{L}}\}) = \overline{\texttt{T}} \to \texttt{T} \qquad \bullet; \{\overline{\texttt{L}}\}; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \overline{\texttt{S}} <: \overline{\texttt{T}}.$$

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

Case RC-NEW, RC-INVKAARG1, RC-INVKAARG2:

Similar to the case above. □

Lemma A.19 (Lemma 12). *If* $\mathit{pmtype}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{T}} \to \texttt{T}_0$, *then there exist* $\overline{\texttt{x}}$ *and* $\texttt{e}_0$ *and* $\texttt{L}'$ *(*$\neq$ Base*) such that* $\mathit{pmbody}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{x}}\texttt{.e}_0$ *in* $\texttt{L}'$ *and the lengths of* $\overline{\texttt{x}}$ *and* $\overline{\texttt{T}}$ *are equal and* $\texttt{L} <:_w \texttt{L}'$.

PROOF. By induction on $\mathit{pmtype}(\texttt{m}, \texttt{C}, \texttt{L}) = \overline{\texttt{T}} \to \texttt{T}_0$.

Case PMT-Layer: $LT(\mathtt{L})(\mathtt{C.m}) = \mathtt{T}_0\ \mathtt{C.m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}$

By T-PMethod, the lengths of $\overline{\mathtt{T}}$ and $\overline{\mathtt{x}}$ are equal. $\mathtt{L} <:_w \mathtt{L}$ by Reflexivity of $<:_w$. Then, PMB-Layer finishes the case.

Case PMT-Super: $LT(\mathtt{L})(\mathtt{C.m})$ undefined $\mathtt{L} \lhd \mathtt{L}'$ $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}') = \overline{\mathtt{T}} \to \mathtt{T}_0$

The induction hypothesis and PMB-Layer and LSw-Extends and LSw-Extends finish the case. □

Lemma A.20 (Lemma 13). *If* $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *and* $\overline{\mathtt{L}'}$ *is a prefix of* $\overline{\mathtt{L}}$ *and* $\{\overline{\mathtt{L}}\}$ *wf, then there exist* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *and* $\overline{\mathtt{L}''}$ *and* $\mathtt{C}'$ *(*$\neq$ `Object`*) such that* $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}'}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathit{in}\ \mathtt{C}', \overline{\mathtt{L}''}$ *and the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{T}}$ *are equal and, if* $\overline{\mathtt{L}''}$ *is not empty, the last layer name of* $\overline{\mathtt{L}''}$ *is not* `Base`*.*

Proof. By lexicographic induction on $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ and the length of $\overline{\mathtt{L}'}$.

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

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

Case: $\overline{\mathtt{L}'} = \bullet$ $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\ \ \overline{\mathtt{M}}\}$ $\mathtt{m} \notin \overline{\mathtt{M}}$

It must be the case that $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ is derived by MT-Super and $mtype(\mathtt{m}, \mathtt{D}, \{\overline{\mathtt{L}}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$. The induction hypothesis and MB-Super finish the case.

Case: $\overline{\mathtt{L}'} = \overline{\mathtt{L}'''}, \mathtt{L}_0$ $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \overline{\mathtt{T}} \to \mathtt{T}_0$

By Lemma A.19 and MB-Layer.

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

Since $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)$ undefined, it must be the case that $mtype(\mathtt{m}, \mathtt{C}, \{\overline{\mathtt{L}'''}\}, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}_0$. By the induction hypothesis, there exist $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ and $\overline{\mathtt{L}''}$ and $\mathtt{C}'$ ($\neq$ `Object`) such that $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'''}, \overline{\mathtt{L}'}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in\ C}', \overline{\mathtt{L}''}$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{T}}$ are equal. It follows that $pmbody(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)$ is undefined from $pmtype(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)$ undefined. MB-NextLayer finishes the case. □

Theorem A.2 (Progress). *Suppose* $\vdash (CT, LT)$ `ok`*. If* $\bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\{\overline{\mathtt{L}}\}$ *wf, then* `e` *is a value or* $\overline{\mathtt{L}} \vdash \mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$*.*

Proof. By induction on $\bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e} : \mathtt{T}$ with case analysis on the last typing rule used.

Case T-Var, T-Super, T-Proceed, T-SuperProceed**:**

Cannot happen.

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

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

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

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

Case T-Invk: $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}})$ $\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e}_0 : \mathtt{C}_0$
$\quad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0, \{\overline{\mathtt{L}}\}) = \overline{\mathtt{T}} \to \mathtt{T}$ $\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ $\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

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, $\mathtt{v}_0 = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{w}})$ for some values $\overline{\mathtt{w}}$. By Lemma A.20, there exist $\overline{\mathtt{x}}$, $\mathtt{e}_0'$, $\overline{\mathtt{L}}''$ and $\mathtt{C}'$ ($\neq \mathtt{Object}$) such that $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}_0\ \mathtt{in}\ \mathtt{C}', \overline{\mathtt{L}}''$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{T}}$ are the same. Since $\mathtt{C}' \neq \mathtt{Object}$, there exists $\mathtt{D}'$ such that $\mathtt{class}\ \mathtt{C}' \lhd \mathtt{D}'\ \{\ldots\}$. 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 $\overline{\mathtt{L}}'''$. Since $\mathtt{L}_0 \neq \mathtt{Base}$, there exists $\mathtt{L}_1$ such that $\mathtt{layer}\ \mathtt{L}_0 \lhd \mathtt{L}_1\ \{\ldots\}$. Then, the expression

$$\mathtt{e}' = \begin{bmatrix} \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{w}}) & /\mathtt{this} \\ \overline{\mathtt{v}} & /\overline{\mathtt{x}} \\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{w}})\mathtt{<C'},\overline{\mathtt{L}}''',\overline{\mathtt{L}}\mathtt{>.m} & /\mathtt{proceed} \\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{w}})\mathtt{<D'},\overline{\mathtt{L}},\overline{\mathtt{L}}\mathtt{>} & /\mathtt{super} \\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{w}})\mathtt{<C'},\mathtt{L}_1,\overline{\mathtt{L}},\overline{\mathtt{L}}\mathtt{>} & /\mathtt{superproceed} \end{bmatrix} \mathtt{e}_0'$$

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: $\mathtt{e} = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ $\quad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ $\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

By the induction hypothesis, either (1) $\overline{\mathtt{e}}$ are all values, in which case $\mathtt{e}$ is also a value; or (2) there exists 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-NewL:

Trivial.

Case T-With: $\mathtt{e} = \mathtt{with}\ \mathtt{e}_l\ \mathtt{e}_0$ $\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e}_l : \mathtt{L}$ $\quad \bullet; \{\overline{\mathtt{L}}\} \cup \{\mathtt{L}\}; \bullet \vdash \mathtt{e}_0 : \mathtt{T}$
$\quad \mathtt{L}\ \mathtt{req}\ \Lambda$ $\quad \{\overline{\mathtt{L}}\} <:_w \Lambda$

By the induction hypothesis, either $\mathtt{e}_l$ is not a value, in which case RC-WithArg finishes the case; or $\mathtt{e}_0$ is a value, in which case R-WithVal finishes the case; or there exists $\mathtt{e}_0'$ such that $\mathit{with}(\mathtt{L}, \overline{\mathtt{L}}) \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$, in which case RC-With finishes the case (notice that $\{\mathit{with}(\mathtt{L}, \overline{\mathtt{L}})\}\ \mathit{wf}$, by Wf-With).

Case T-Swap: $\mathtt{e} = \mathtt{swap}\ (\mathtt{e}_l, \mathtt{L}_{sw})\ \mathtt{e}_0$ $\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{e}_l : \mathtt{L}$
$\quad \mathtt{L}_{sw}\ \mathtt{swappable}$ $\quad \mathtt{L} <:_w \mathtt{L}_{sw}$ $\quad \mathtt{L}\ \mathtt{req}\ \Lambda'$
$\quad \Lambda_{rm} = \{\overline{\mathtt{L}}\} \setminus \{\mathtt{L}' \mid \mathtt{L}' <:_w \mathtt{L}_{sw}\}$ $\quad \Lambda_{rm} <:_w \Lambda'$ $\quad \mathcal{L}; \Lambda_{rm} \cup \{\mathtt{L}\}; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0$

By the induction hypothesis, either $\mathtt{e}_l$ is not a value, in which case RC-SwapArg finishes the case; or $\mathtt{e}_0$ is a value, in which case R-SwapVal finishes the case; or there exists $\mathtt{e}_0'$ such that $\mathit{swap}(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}}) \vdash \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$, in which case RC-Swap finishes the case (notice that, by Wf-Swap, $\{\mathit{swap}(\mathtt{L}, \mathtt{L}_{sw}, \overline{\mathtt{L}})\}\ \mathit{wf}$).

Case T-InvkA:

Similar to the case for T-Invk.

Case T-InvkAL: $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}})\mathtt{<D}_0\mathtt{,L}_1\mathtt{,(}\overline{\mathtt{L}}''\mathtt{;L}_0\mathtt{),}\overline{\mathtt{L}}'\mathtt{>.m}(\overline{\mathtt{e}})$
$\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}) : \mathtt{C}_0$ $\quad \mathtt{C}_0\mathtt{.m} \vdash \mathtt{<D}_0\mathtt{,L}_1\mathtt{,(}\overline{\mathtt{L}}''\mathtt{;L}_0\mathtt{),}\overline{\mathtt{L}}'\mathtt{>}\ \mathtt{ok}$
$\quad \{\overline{\mathtt{L}}\} <:_{sw} \{\overline{\mathtt{L}}'\}$ $\quad \mathtt{L}_0 <:_w \mathtt{L}_1$
$\quad \mathit{pmtype}(\mathtt{m}, \mathtt{D}_0, \mathtt{L}_1) = \overline{\mathtt{T}}' \to \mathtt{T}_0$
$\quad \bullet; \{\overline{\mathtt{L}}\}; \bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}'$ $\quad \overline{\mathtt{S}}' <: \overline{\mathtt{T}}'$

By the induction hypothesis, either (1) there exists $i \geq 1$ and $\mathtt{e}_i'$ such that $\overline{\mathtt{L}} \vdash \mathtt{e}_i \longrightarrow \mathtt{e}_i'$, in which case RC-InvkArg finishes the case, or (2) all $\mathtt{e}_i$'s are values $\overline{\mathtt{w}}$. Then, by Lemma A.19, there exist $\overline{\mathtt{x}}$, $\mathtt{e}_0'$ and $\mathtt{L}_2$ ($\neq$ `Base`) such that $\mathit{pmbody}(\mathtt{m}, \mathtt{D}_0, \mathtt{L}_1) = \overline{\mathtt{x}}\,.\,\mathtt{e}_0$ `in` $\mathtt{L}_2$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{T}}'$ are the same. Since $\mathtt{L}_2' \neq$ `Base`, there exists $\mathtt{L}_3$ such that `layer` $\mathtt{L}_2$ `◁` $\mathtt{L}_3$ `{...}`.

By Sanity Condition (8), $\mathtt{D}_0$ is not `Object` and there exists $\mathtt{E}_0$ such that `class` $\mathtt{D}_0$ `◁` $\mathtt{E}_0$ `{...}`. Then, the expression

$$\mathtt{e}' = \left[\begin{array}{ll} \texttt{new } \mathtt{C}_0(\overline{\mathtt{v}}) & \texttt{/this} \\ \overline{\mathtt{w}} & /\overline{\mathtt{x}} \\ \texttt{new } \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{D}_0\texttt{,}\overline{\mathtt{L}''}\texttt{,}\overline{\mathtt{L}}\texttt{>.m} & \texttt{/proceed} \\ \texttt{new } \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{E}_0\texttt{,}\overline{\mathtt{L}}\texttt{,}\overline{\mathtt{L}}\texttt{>} & \texttt{/super} \\ \texttt{new } \mathtt{C}_0(\overline{\mathtt{v}})\texttt{<}\mathtt{D}_0\texttt{,}\mathtt{L}_3\texttt{,(}\overline{\mathtt{L}''}\texttt{;}\mathtt{L}_0\texttt{),}\overline{\mathtt{L}}\texttt{>.m} & \texttt{/superproceed} \end{array}\right] \mathtt{e}_0'$$

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

Theorem A.3 (Type Soundness). *If* $\vdash (CT, LT, \mathtt{e}) : \mathtt{T}$ *and* $\mathtt{e}$ *reduces to a normal form under the empty set of layers, then the normal form is* `new` $\mathtt{S}(\overline{\mathtt{v}})$ *for some* $\overline{\mathtt{v}}$ *and* $\mathtt{S}$ *such that* $\mathtt{S} <: \mathtt{T}$.

Proof. By T-Prog and Theorems A.1 and A.2. □