Delegation vs Inheritance for Typestate Analysis

Du Li, Alex Potanin,[1] and Jonathan Aldrich

Carnegie Mellon University and Victoria University of Wellington[1]

{duli, aldrich}@cs.cmu.edu, alex@ecs.vuw.ac.nz[1]

Abstract

For several decades, inheritance and delegation have been widely adopted for code reuse in object-oriented languages. Though extensive research has explored the expressiveness of these techniques, little is known about how the choice between them affects formal reasoning. In this paper, we explore this question by describing two core languages that are identical except for the use of inheritance and delegation, respectively. We add support for formal reasoning about typestate to both languages, and evaluate the complexity of the formal semantics and compare the example specifications. Our study suggests that our variant of delegation can substantially simplify typestate reasoning, while inheritance makes code more succinct in the case where open recursion is used.

1. Introduction

Inheritance and delegation are two major mechanisms for code reuse and program extension in object-oriented programming languages [5, 9]. In addition to longstanding work studying the semantics of these constructs, recent empirical studies have investigated their use in practice [7, 8]. However, we are not aware of any work that investigates whether and how the two notions affect formal reasoning.

In this paper, we explore how inheritance and delegation affect a particular kind of formal reasoning: typestate checking [6]. We choose typestate checking because it is a well-established yet relatively simple kind of formal reasoning. Typestate is richer than a standard type system and enables reasoning about temporal behavior, as do many other formal reasoning systems. However, it is simple enough that we can easily understand and compare typestate reasoning in the presence of the two different reuse mechanisms. We hope that the insights obtained during our study can be generalized to other formal reasoning systems, although demonstrating this is future work.

We design two Java-like programming languages: one with inheritance and the other with delegation. Both languages support static reasoning about typestate [3]. In the typestate model, each object goes through a series of conceptual *states* during its lifetime. A **state** is defined by the set of methods that can be invoked on objects that are in that state; a method may transition the object into another state. Programmers can use **state invariants** to specify predicates over the object's fields that must hold for each state. Our languages use **classes** as the unit of code, as in most Java-like languages, and our inheritance language allows one class to extend another. In our delegation language, we replace inheritance with a **delegate** construct that delegates a set of methods to a second object reachable through a designated field. We deliberately make the two languages as similar as possible in order to focus purely on the differences caused by inheritance and delegation. Our study suggests:

1. Delegation can reduce the complexity of typestate reasoning because the relationship between delegator and delegatee is specified using the same mechanisms and are used for any other pair of related objects. Since the delegator and delegatee are separate objects, they are more loosely coupled than in inheritance, but at the cost of a more verbose specification relating them.
2. Inheritance provides more succinct code and specifications when there is open recursion between a superclass and subclass. However, the coupling that supports this succinctness also creates the need for tight coordination between the states of the superclass and subclass, leading to complex solution approaches such as sliding methods [2] or frame permissions [1].

To the best of our knowledge, this is the first work that compares inheritance with delegation in the context of formal reasoning. In order to make the study thorough and accessible, we use concrete examples as well as formal rules to highlight the differences between these two mechanisms.

2. Examples

We use two simple programs to illustrate the usage of the two languages. Both programs perform exactly the same task–

setting up and coloring a Flower, which reuses generic Plant code–and involve both up-calls and down-calls, which gives the examples non-trivial structure. The programs start with declarations that show the states of the program objects and the methods that can be called in each state. In the method signatures, the syntax after the argument list describes the state transition that the receiver object *this* goes through. Then come the class declarations; each class has a set of fields, followed by a set of *state invariants* that describe, for each state of the object, whether the fields are null or what state the fields are in. Finally, the methods of the class are listed.

Listing 1 shows the program written in the language with inheritance. The primary challenge in this version is coordinating the states of the superclass and subclass. The required relationship is shown in the state invariants for Flower: for instance, when the Flower is in the Colored state, the Plant must also be; there are additional requirements on the fields of both classes. To allow Flower's state invariants to talk about the state of its superclass, we introduce the syntax '**super instate** S' (lines 39-41).

Because of the relationships between superclass and subclass states, it is critical that the superclass and subclass transition between states in a coordinated way. To enforce this requirement, we follow DeLine and Fahndrich in designating certain methods as *sliding methods* [2] using the **sliding** keyword. A sliding method is used to transition an object from one state to another. In each sliding method body, we can manipulate fields, and we can make **super** calls to transition the superclass between the appropriate states (for example, Flower.dye() does both of these things). However, at any given point in the inheritance hierarchy, the object has only been transitioned "halfway" into a new state. Thus, calling other methods on the receiver object ('this') is forbidden, because the call might dispatch to a method implementation in a subclass that expects the object to be in a consistent state.

A non-sliding method cannot touch the fields of the object, for fear of making the object's state inconsistent with the current state (even reading a field can do so, because we work in a linear type system in which field reads use up the permission to the field). However, a non-sliding method can call other methods on this, including sliding methods.

In our language, sliding methods must be overridden in all subclasses. The reason of overriding sliding methods is to avoid the scenario in which different frames [1] of an object are in inconsistent states. For instance, if the programmer does not implement the *dye()* method in class *Flower*, the object state transition will be as follows:

```
f=new Flower;
// f@Raw
f.initialize();
// f@Initialized
f.dye()
// super@Colored && petalColor == null,
// so f is not in any legitimate state.
```

As shown above, since *dye()* is not overridden in class *Flower*, *f.dye()* executes method *dye()* in class *Plant*, and object *f* will end up being an inconsistent state, that is, it is not in any state defined in this program. More specifically, if we treat an object as a collection of frames from different classes, *f* consists of two frames: *Plant* and *Flower* frames. At this point, the *Plant* frame of *f* is in *Colored* state while the *Flower* frame of *f* is in *Initialized* state.

```
1   state Raw {
2     unit paint() Raw→ Colored
3     unit initialize() Raw→ Initialized
4   }
5
6   state Initialized {
7     unit dye() Initialized→ Colored
8   }
9
10  state Colored {}
11
12  class Plant {
13    body;
14    color;
15
16    invariants:
17      Raw: body == null;
18      Initialized: body != null;
19      Colored: color != null;
20
21    unit paint() Raw→ Colored {
22      initialize();
23      dye();
24    }
25
26    sliding unit initialize() Raw → Initialized {
27      body = new Plant();
28    }
29
30    sliding unit dye() Initialized → Colored{
31      color = RED;
32    }
33  }
34
35  class Flower extends Plant {
36    petalColor;
37
38    invariants:
39      Raw: super instate Raw
40      Initialized: super instate Initialized
41      Colored: super instate Colored && petalColor != null
42
43    sliding unit initialize() Raw → Initialized {
44      super.initialize();
45    }
46
47    sliding unit dye() Initialized → Colored {
48      super.dye();
49      petalColor = YELLOW;
50    }
51  }
52
53  // Client code
54  f = new Flower;
55  f.paint();
```

Listing 1. Inheritance Example

As illustrated in this example, the complexity of the inheritance language lies in sliding window methods and dealing with inconsistent object states. The system with inheritance also has to reason about the state of *super*. On the other hand, the inheritance language makes upcalls and dynamic dispatch relatively easy and convenient, which results in more concise program code.

Listing 3 demonstrates the delegation-based language. Our semantics for delegation is slightly different from the classic one [5, 9]; rather than bind the original receiver as 'this' in the delegatee's method, we pass the original receiver as an additional parameter to the delegatee's method. This choice makes our typing rules both simpler and more expressive, because we can treat the current object and the original receiver as separate arguments to a method rather than try to find a type that fits both.

Our **delegate** construct replaces inheritance. For example, in Listing 3, line 49 delegates the method $paint()$ to $target$. In order to elaborate how the delegation mechanism works, we show in Listing 2 how it could be translated into equivalent code without delegation.

```
unit paint() Paintable→ Colored {
    // this@Paintable
    // UNPACK this
    // this unpacked && target@Raw
    tmp = target;
    // this unpacked && tmp@Raw
    // PACK this to PaintableCallback
    // this@PaintableCallback && tmp@Raw
    tmp.paint(this);
    // this@ColoredCallback && tmp@Colored
    // UNPACK this
    // this unpacked && petalColor != null && tmp@Colored
    this.target=tmp
    // target@Colored
    // PACK this to Colored
}
```

Listing 2. Translation of Delegation

The comments show the typestate of objects at each point in the method. We can exchange knowledge that an object is in a particular state (e.g. 'this@Paintable') for knowledge of its fields by conceptually UNPACKing the object [1, 2]. Here UNPACKing 'this' results in 'this unpacked && target@Raw', where target@Raw comes from the state invariant of Paintable in Flower. PACK goes the other direction.

Note we use local variable *tmp* to store the delegation target during the method call. We do this because we must pass 'this' to the paint method of the delegatee, so 'this' must be packed in a consistent state, and we cannot use the fields of 'this' while 'this' is packed. Note that we must pack 'this' into a new state, PaintableCallback, which is the same as Paintable except that the state invariant has no knowledge of the target field. We do this because, in our linear type system, reading the target field destroys knowledge of its contents; this is standard for typestate tracking systems, although there are ways to relax it that we do not consider here [1, 2].

```
1   state Initialized {
2     unit setColor() Initialized → Colored
3   }
4
5   state Raw {
6     unit initialize() Raw → Initialized
7     unit paint(PaintableCallback→ ColoredCallback p) Raw → Colored
8   }
9
10  state Paintable {
11    unit paint() Paintable → Colored
12  }
13
14  state Colored {}
15
16  state PaintableCallback {
17    unit dye(Initialized→ Colored t) PaintableCallback → ColoredCallback
18  }
19
20  state ColoredCallback {}
21
22  class Plant {
23    body;
24    color;
25
26    invariants:
27      Initialized: body != null;
28      Raw: body == null;
29      Colored: color != null;
30
31    unit initialize() Raw → Initialized {
32      body = new Plant();
33    }
34
35    unit paint(PaintableCallback→ ColoredCallback p) Raw → Colored {
36      initialize();
37      p.dye(this);
38    }
39
40    unit setColor() Initialized → Colored {
41      color = RED;
42    }
43  }
44
45  class Flower {
46    target = new Plant;
47    petalColor;
48
49    delegate Paintable to target
50      mapping Paintable −> PaintableCallback, Colored −> ColoredCallback;
51
52    invariants:
53      Paintable: target instate Raw
54      PaintableCallback: true
55      ColoredCallback: petalColor != null
56      Colored: target instate Colored && petalColor != null
57
58    unit dye(Initialized→ Colored t) PaintableCallback → ColoredCallback {
59      t.setColor(RED);
60      petalColor = YELLOW;
61    }
62  }
63
64  // Client code
65  f = new Flower;
66  f.paint();
```

Listing 3. Delegation Example

This example shows how our delegation system is able to provide open recursion, just as inheritance does. The method call *p.dye()* in class Plant (line 37) invokes method *dye()* in class Flower (line 58). As shown in Listing 2, the original object(”*f*”) is passed to the delegatee (”*target*”). Our approach simplifies reasoning compared to a conventional delegation semantics since we can treat the original receiver object as a regular parameter to the delegatee’s method.

In the delegation example, *f* and *target* are two independent objects instead of two frames in inheritance mechanism, which turns the problem of coordinating across superclass and subclass into simply coordinating the state of two objects–something both systems must be able to do anyway. For the same reason, we don’t need the sliding window mechanism either. In this regard, we argue delegation is actually simpler than inheritance in the setting of typestate reasoning. We recognize in this example the delegation code is noticeably longer than that of inheritance–mainly because of the additional states and accompanying state invariants. Essentially, our delegation approach is more explicit about the relationship between the delegator and delegatee, which has advantages and disadvantages: On the positive side, the objects are more loosely coupled, and the reasoning is simpler and clearer, but on the negative side the specification is longer.

3. Languages

3.1 Syntax

Both languages are inspired by *Featherweight Java(FJ)* [4]. The major difference with FJ are related to the addition of states. Every object has a **state** as its type. We use an overbar to abbreviate a sequence (e.g. $\overline{MD} = MD_1, ..., MD_n$) and write • to indicate an empty sequence. A program (PG) consists of a list of classes and an expression. A class declaration (CL) gives a state or a class a unique name. A state is defined by a collection of methods that can be invoked at this state. A method declaration (MD) specifies the states of the receiver object, the return value and each parameter before and after method execution. Besides method and field definitions, a class also includes a set of invariants that are predicates for a state. For instance, in Listing 1, ”*Initialized: body != null*” is a predicate for state *Initialized*, which means an object is in state *Initialized* if and only if field *body* is not null. For a state name *S*, an object can be in packed state $C@S$ or unpacked state $C@\{\overline{f : S}\}$. An object can only have downcalls at a packed state and a field access can only be performed for an object in an unpacked state. We will further explain the subtyping relation between unpacked and packed states later.

Figure 1 shows the syntax of the delegation language while Figure 2 highlights the syntactic differences in the inheritance language. In the delegation language, the class declaration may delegate all the methods in a state S to some field *f*. In the inheritance language, the keyword *extends* works similarly to Java. Predicates can also include the state of *super*. More importantly, we distinguish between *normal* and *sliding* methods. *Sliding* methods can access object fields and must be overridden in subclasses. *Normal* method cannot access fields.

$$\begin{aligned}
PG &::= \langle \overline{CL}, t\rangle \\
CL &::= \textbf{class } C\ \{\overline{f};\ \overline{M};\ \textbf{invariant } \overline{S \Rightarrow P}; \\
&\qquad\qquad \textbf{delegate } S \text{ to } f \text{ mapping } \overline{S} \mapsto \overline{S}\ \} \\
&\quad\ \mid \textbf{state } S\ \{\overline{MD}\ \} \\
M &::= MD\ \{\textbf{return } t;\} \\
MD &::= S\ m(\overline{[S \to S]\ x})\ [S \to S] \\
P &::= \overline{f \textbf{ instate } S} \\
b &::= x \mid l \\
t &::= b \mid v \mid b.f \mid b.m(b) \mid \textbf{new } C \mid (S)b \mid b.f = b;\ t \\
&\quad\ \mid \textbf{let } x = t \textbf{ in } t \\
\tau &::= \gamma \mid C@S \mid C@\{\overline{f\ :\ S}\} \\
\gamma &::= S \mid \textbf{none} \\
v &::= C(\overline{f \to v}) \\
\Gamma &::= \bullet \mid \Gamma,\ b : \tau
\end{aligned}$$

C, D ∈ CLASSNAMES S ∈ STATENAMES
m ∈ METHODNAMES v ∈ VALUES
x, y, z, this ∈ IDENTIFIERNAMES
f ∈ FIELDNAMES l ∈ INDIRECTREFERENCES

Figure 1. Syntax for Delegation Language

$$\begin{aligned}
CL &::= \textbf{class } C\ \textbf{extends}\ \text{D}\ \{\overline{f};\ \overline{M};\ \textbf{invariant } \overline{S} \Rightarrow \overline{P};\} \mid \dots \\
MD &::= S\ m_w\ (\overline{[S \to S]\ x})\ [S \to S] \\
\text{w} &::= \textbf{sliding} \mid \textbf{normal} \\
P &::= \overline{f \textbf{ instate } S},\ \textbf{super instate}\ \text{S}
\end{aligned}$$

Figure 2. Syntax for Inheritance Language

3.2 Semantics

This section provides static typing rules of the two languages. We list the typing rules of the delegation language in Figure 3, and present inheritance-specific rules in Figure 4 for comparison. Most of them are straightforward; we explain the most interesting ones. The T-Read and T-Write rules ensure read/write can only be conducted on objects that are in an unpacked state ($C@\{\overline{f : S}\}$). The Sub-Pack and Sub-unpack rules allow us to use subtyping to pack and unpack objects in the delegation system (in the inheritance system, these rules are missing; the method rule does packing and unpacking instead). T-Method rule requires the method

$$\frac{CT(C) = \mathbf{class}\ C\ ...}{\Gamma \vdash \mathbf{new}\ C : C@\{\} \dashv \Gamma}\ \textit{T-New} \qquad \frac{f_i \in fields(C)}{\Gamma, b : C@\{\overline{f : S}\} \vdash b.f_i : S_i \dashv \Gamma, b : C@\{\overline{f : S}\}\backslash\{f_i : S_i\}}\ \textit{T-Read}$$

$$\frac{f_i \in fields(C) \quad \Gamma, b : C@\{\overline{f : S}\} \cup \{f_i : S_{b'}\}, b' : \mathbf{none} \vdash t : S \dashv \Gamma'}{\Gamma, b : C@\{\overline{f : S}\}, b' : S_{b'} \vdash b.f_i = b'; \quad t : S\ \dashv \Gamma'}\ \textit{T-Write} \qquad \frac{}{\Gamma, b : \tau \vdash b : \tau \dashv \Gamma}\ \textit{T-Binding}$$

$$\frac{\forall l \in dom(\Gamma).\Gamma \vdash \Theta(l) : Gamma(l) \dashv \Gamma}{\Gamma \vdash \Theta}\ \textit{T-Ref} \qquad \frac{(S \Rightarrow \overline{f\ \mathbf{instate}\ S}) \in \mathbf{class}\ C \quad \Gamma \vdash v'_i : S_i \dashv \Gamma}{\Gamma \vdash C(\overline{f = v'}) : S \dashv \Gamma}\ \textit{T-Val}$$

$$\frac{\Gamma \vdash \overline{v : S} \dashv \Gamma}{\Gamma \vdash C(\overline{f = v}) : C@\{\overline{f : S}\} \dashv \Gamma}\ \textit{T-UnpackedVal} \qquad \frac{}{S <: S}\ \textit{Sub-Reflex} \qquad \frac{S_1 <: S_2 \quad S_2 <: S_3}{S_1 <: S_3}\ \textit{Sub-Trans}$$

$$\frac{S_1 = \{\overline{MD_1}\} \quad S_2 = \{\overline{MD_2}\} \quad \overline{MD_2} \supseteq \overline{MD_1}}{S_2 <: S_1}\ \textit{Sub-Struct1} \qquad \frac{S_1 <: S_2}{C@S_1 <: C@S_2}\ \textit{Sub-Struct2}$$

$$\frac{\overline{f\ \mathbf{instate}\ S} \subseteq \overline{f'\ \mathbf{instate}\ S'}}{C@\{\overline{f : S}\} <: C@\{\overline{f' : S'}\}}\ \textit{Sub-Struct3} \qquad \frac{(S' \Rightarrow \overline{f\ \mathbf{instate}\ S}) \in \mathbf{class}\ C}{C@\{\overline{f : S}\} <: C@S'}\ \textit{Sub-Pack}$$

$$\frac{(S' \Rightarrow \overline{f\ \mathbf{instate}\ S}) \in \mathbf{class}\ C}{C@S' <: C@\{\overline{f : S}\}}\ \textit{Sub-Unpack} \qquad \frac{\Gamma \vdash b : S \dashv \Gamma}{\Gamma \vdash (S')b : S' \dashv \Gamma}\ \textit{T-Cast} \qquad \frac{\begin{array}{c}\Gamma \vdash t_1 : S_1 \dashv \Gamma' \\ \Gamma', x : S_1 \vdash t_2 : S_2 \dashv \Gamma\text{”}\end{array}}{\Gamma \vdash \mathbf{let}\ x = t_1 \mathbf{in}\ t_2 : S_2 \dashv \Gamma\text{”}\backslash x}\ \textit{T-Let}$$

$$\frac{\begin{array}{c}\mathbf{class}\ C\ \{\overline{f};\ \overline{M};\ \mathbf{invariant}\ \overline{S \Rightarrow P};\ \mathbf{delegate}\ S'\ \mathbf{to}\ f\ \ldots\} \\ MD \in S\ \wedge\ MD\ \{\mathbf{return}\ t\}\ \notin\ \overline{M}\ \Rightarrow\ MD \in S'\end{array}}{S\ is\ OK\ in\ C}\ \textit{T-State}$$

$$\frac{\begin{array}{c}P\ =\ \overline{f'\ \mathbf{instate}\ \gamma'} \quad \overline{f}' \subseteq \overline{f} \\ \mathbf{class}\ C\ \{\overline{f};\ \overline{M};\ \mathbf{invariant}\ \overline{\gamma \Rightarrow P};\ \mathbf{delegate}\ S\ \mathbf{to}\ f\ \ldots\}\end{array}}{P\ is\ OK\ in\ C}\ \textit{T-Formula}$$

$$\frac{mdecl(m, \gamma) = \gamma'\ m(\overline{[\gamma_y \to \gamma'_y]\ y})[\gamma_t \to \gamma'_t] \quad \gamma <: \gamma_t \quad \overline{\gamma_y\text{”} <: \gamma_y}}{\Gamma,\ b : \gamma,\ \overline{b' : \gamma_y\text{”}}\ \vdash\ b.m(\overline{b'}) : \gamma'\ \dashv\ \Gamma,\ b : \gamma_{t'},\ \overline{b' : \gamma'_y}}\ \textit{T-Invoke}$$

$$\frac{\begin{array}{c}\overline{M}\ is\ OK\ in\ C \quad \overline{S}\ is\ OK\ in\ C \quad \overline{P}\ is\ OK\ in\ C \quad \mathbf{state}\ S\ \{\overline{MD}\} \\ \mathbf{delegate}\ S\ \mathbf{to}\ f\ \mathbf{mapping}\ \overline{S'} \mapsto \overline{S''}\ is\ OK\ in\ C \quad names(\overline{M})\ is\ disjoint\ from\ names(\overline{MD})\end{array}}{\mathbf{class}\ C\ \{\overline{f};\ \overline{M};\ \mathbf{invariant}\ \overline{S \Rightarrow P};\ \mathbf{delegate}\ S\ \mathbf{to}\ f\ \mathbf{mapping}\ \overline{S'} \mapsto \overline{S''}\ \} \quad OK}\ \textit{T-Class}$$

$$\frac{\begin{array}{c}\mathbf{this} : C@S_t,\ \overline{y : S_y}\ \vdash t : S' \dashv\ \mathbf{this} : C@S'_t,\ \overline{y : S'_y} \\ \mathbf{class}\ C\ \{\overline{f};\ \overline{M};\ \mathbf{invariant}\ \overline{S \Rightarrow P};\ \mathbf{delegate}\ S\ \mathbf{to}\ f\ \ldots\} \quad m \in \overline{M}\end{array}}{S' \quad m(\overline{[S_y \to S'_y]\ y})\ [S_t \to S'_t]\ \{\mathbf{return}\ t;\} \quad OK\ \mathbf{in}\ C}\ \textit{T-Method}$$

$$\frac{\begin{array}{c}\forall S_i \in \overline{S} \quad invariant_C(S_i) = invariant_C(S'_i), f\ \mathbf{instate}\ S''_i \\ \forall MD \in methods(S) \quad \exists S_j, S_k \in \overline{S} \quad pre(MD) = S_j \wedge post(MD) = S_k \wedge \exists MD' \in methods(S''_j) \\ where\ MD = S_r\ m(\overline{[S^a \to S^b]x})[S_j \to S_k] \quad and\ MD' = S_r\ m(\overline{[S^a \to S^b]x}, [S'_j \to S'_k]y)[S''_j \to S''_k]\end{array}}{\mathbf{delegate}\ S\ \mathbf{to}\ f\ \mathbf{mapping}\ \overline{S} \mapsto \overline{S'}\ OK\ in\ C}\ \textit{T-Delegation}$$

Figure 3. Static Semantics of Delegation Language

$$\frac{\overline{M}\ \mathit{is\ OK\ in}\ C \quad \overline{S}\ \mathit{is\ OK\ in}\ C \quad \overline{P}\ \mathit{is\ OK\ in}\ C \\ \overline{C}\ \textbf{override}\ \mathit{method}\ \overline{M}\ \mathit{in}\ D \quad \mathit{all\ sliding\ methods\ in}\ D\ \mathit{overridden}}{\textbf{class}\ C\ \textbf{extends}\ D\{\overline{f};\ \overline{M};\ \textbf{invariant}\ \overline{S} \Rightarrow \overline{P};\}\quad OK}\ \textit{T-Class}$$

$$\frac{\textbf{class}\ C\ \textbf{extends}\ D\{\overline{f};\ \overline{M};\ \textbf{invariant}\ \overline{S} \Rightarrow \overline{P};\}\quad m \in \overline{M} \\ \textbf{this} : S_t,\ \overline{y : S_y}\ \vdash t : S' \dashv\ \textbf{this} : S'_t,\ \overline{y : S'_y}}{S' \quad m_{\textbf{normal}}(\overline{[S_y \to S'_y]\ y})\ [S_t \to S'_t]\ \{\textbf{return}\ t;\}\quad OK\ \textbf{in}\ C}\ \textit{T-Method-Normal}$$

$$\frac{\textbf{class}\ C\ \textbf{extends}\ D\{\overline{f};\ \overline{M};\ \textbf{invariant}\ \overline{S} \Rightarrow \overline{P};\}\quad m \in \overline{M} \\ S_i, S_j \in \overline{S} \quad \{\overline{f\ \textbf{instate}\ S_f},\ \textbf{super instate}\ S_s\} \subseteq P_i \quad \{\overline{f\ \textbf{instate}\ S'_f},\ \textbf{super instate}\ S'_s\} \subseteq P_j \\ \textbf{this} : C@\{\overline{f : S_f}\},\ \textbf{super} : S_s,\ \overline{y : S_y}\ \vdash t : S' \dashv\ \textbf{this} : C@\{\overline{f : S'_f}\},\ \textbf{super} : S'_s,\ \overline{y : S'_y}}{S' \quad m_{\textbf{sliding}}(\overline{[S_y \to S'_y]\ y})\ [S_i \to S_j]\ \{\textbf{return}\ t;\}\quad OK\ \textbf{in}\ C}\ \textit{T-Method-Sliding}$$

$$\frac{mdecl(m, D) = \gamma'\ m(\overline{[\gamma_y \to \gamma'_y]\ y})[\gamma_t \to_w \gamma'_t] \quad C\ \textbf{extends}\ D \\ \Longrightarrow\ \ mdecl(m, C) = \gamma'\ m(\overline{[\gamma_y \to \gamma'_y]\ y})[\gamma_t \to_w \gamma'_t],}{C\ \textbf{override}\ \mathit{method}\ m\ \mathit{in}\ D}\ \textit{T-Override}$$

Figure 4. Static Semantics of Inheritance Language

body to match the declaration. We use *indirect references* to match variables to values as in Garcia et al [3] and *T-Ref* checks the types of indirect references are right.

Next, we examine the difference between two type systems. In Figure 3, the rules within the frame are those shared by both delegation and inheritance, the rest are rules specific to delegation. We have the T-Delegation rule to check if the delegation target, f, is in compatible pre- and post-states. We also have T-class rule to restrict the names of delegated method to be disjoint from names of methods that are implemented in this class.

Figure 4 lists rules that only are used by inheritance, which are more complex than delegation. The major overhead in the delegation type system is the sliding method mechanism, requiring separate rules for sliding nand normal methods. In addition, the inheritance system also needs to specify the state of *super* in state invariants, as shown in the rule T-Method-Sliding.

First, We need the T-Override rule to deal with method overriding. Second, the inheritance language distinguishes between normal methods and sliding methods by T-Method-Normal and T-Method-Sliding, which are the most complicated part in this type system.

Conclusion. Overall, we argue the type system of the inheritance language is more complex than that of the delegation language. This confirms our observations in Section 2. Our delegation system also retains the traditional advantage that the delegatee can be chosen at run time. However, the simplicity of formal reasoning comes at the cost of greater verbosity in the specification.

Acknowledgements. This work was supported by NSF grant #CCF-1116907. We thank the reviewers for their suggestions.

References

[1] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.

[2] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.

[3] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, Oct. 2014.

[4] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[5] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA*, 1986.

[6] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[7] E. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *ECOOP*, 2008.

[8] E. Tempero, H. Y. Yang, and J. Noble. What programmers do with inheritance in Java. In *ECOOP*, 2013.

[9] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, 1987.