A Core Calculus for Mixin-Types

Tetsuo Kamina and Tetsuo Tamai
University of Tokyo
{kamina,tamai}@graco.c.u-tokyo.ac.jp

Abstract

The programming construct *mixin* was invented to implement modules that provide the mechanism of abstracting uniform extensions and modifications to superclasses. One approach to implement a mixin is to parameterize a superclass of a generic class using a type parameter; however, this approach lacks the ability to declare a mixin that is also used as a type.

In this paper, we propose a programming language McJava, an extension of Java that is equipped with mixin-types, a mechanism to declare a mixin that is also used as a type. Then, we develop Core McJava, a core language for McJava, and show its type soundness theorem. This core language is based on Featherweight Java (FJ), a minimum core calculus for Java. FJ is a very small subset of Java. Focusing on a few key issues, we have developed a flexible subtyping relation among mixin compositions.

1 Introduction

Object-oriented programming languages like Java and C# offer class systems that provide a simple and flexible mechanism for reusing collections of program pieces. Using inheritance and overriding, programmers may derive a new class by specifying only the elements that are extended and modified from the original class. However, a pure class-based approach lacks the mechanism of abstracting uniform extensions and modifications to classes.

The programming construct *mixin* was invented to implement modules that provide such uniform extensions and modifications [5]. A mixin is a partially implemented subclass whose superclass is not provided in its declaration. To use a mixin, we compose an actual superclass with the mixin to create a new class. For example, we may declare a mixin `Color` that is intended to be composed with GUI components like `Label` or `TextField`, to produce new classes `Color::Label`[1] or `Color::TextField`, respectively. For reusing implementation, it is also useful to compose a mixin `Color` with another mixin, e.g., `Font`, a mixin that provides "font" feature. This composition, written `Color::Font`, is considered as a mixin that has both feature of `Color` and `Font`. We call this mechanism *mixin-mixin composition*.

One approach to implementing mixins is to parameterize a superclass of a generic class using a type parameter [22, 15]. A mechanism of generic classes (also known as templates) is provided by C++ [16], [1], and GJ, an extension of Java with generic types [6]. Even though a generic type in GJ does not support parameterization of a superclass, a recent research shows an implementation and a type soundness proof of an extension of GJ that allows parameterization of superclasses [2].

One of the limitations of a generic type is that it may not be used as a type unless it is instantiated by substituting type variables with real types. For example, the following GJ-like code shows a mixin `Font` that is supposed to be composed with some other classes:

```
class Font<Widget> extends Widget {
  String font;
  void paint(Graphics g) {
    g.setFont(this);
    super.paint(g);
    ...
  }
  void setFontName(String font) {
    this.font=font; }
  String getFontName() { return font; }
}
```

In this approach, mixins are simply a coding convention and have no formal status. The `Font` may not be used as a type. Instead, by using a type parameter for methods, we may write a method like

```
<X> void setFont(Font<X> f) { ... }
```

that is intended to be applied to instances of all the results of composing `Font` with some classes. However, this approach does not provide the power of mixin-mixin composition where `Color::Font` has both feature of `Color` and `Font`. The above method actually cannot take an instance of, for example, `Color<Font<Label>>` that seems to be allowed if mixin-mixin compositions are supported.

In this paper, we propose a programming language McJava, an extension of Java that is equipped with *mixin-types*, a mechanism of declaring a mixin that may also be used as a type. Then, we develop Core McJava, a core calculus for McJava, and show its type sound-

[1] We use an operator `::` to denote mixin compositions.

ness theorem. This calculus is based on Featherweight Java (FJ), a minimum core calculus for Java [11]. FJ is a very small subset of Java. Focusing on a few key issues, we have developed a flexible subtyping relation among mixin compositions with mixin-mixin compositions.

The rest of this paper is structured as follows. In section 2, we propose a programming language McJava and explain how the use of mixin-types solves the realistic problems by showing an example. In section 3, we develop Core McJava, a core calculus for mixin-types, and show its type soundness theorem. Then, we discuss the relationship between this work and other related work in section 4. Finally, in section 5 we conclude this paper with some further research directions.

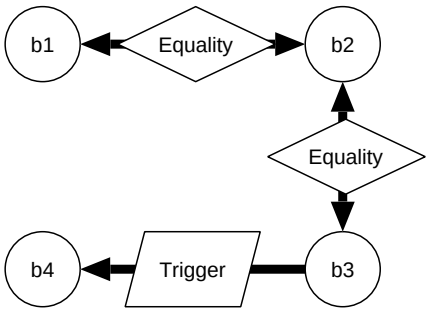
2 Programming Language McJava

To explain the expressive power of McJava, we start with introducing an interesting example of integrated systems. In [18], an integrated system is defined as "a collection of software tools that work together, freeing the user from having to coordinate them manually." For example, an integrated system with tools for text editing, compiling, and debugging will ensure that when the debugger reaches a breakpoint, the editor scrolls to the corresponding source statement.

One of the main problems in implementing an integrated system is its difficulty for evolution. Managing the complexity of integrated systems is hard. The solution of this problem is separating the components (i.e. the integrated software tools) and their relations at the design and implementation level; however, Sullivan et al. argued that an integrated system implemented by a traditional object-oriented language and even by an aspect-oriented language like AspectJ [12] hardly evolves [17]. In this section, we propose a solution to this problem with McJava, and show how the mechanism of mixin-types is used in this solution.

We show a simplified example of integrated systems originally described in [17]. In this example, the software tools that are subject to integration are the binary objects that have two states, *on* and *off*. We call these objects *Bits*. An instance of Bit has operations named *set* and *clear*, to change its state to "on" and "off," respectively. Binary relations, *Equality* and *Trigger*, are defined among Bits. The Equality relation always makes the states of the related Bits the same, while the Trigger relation activates the *target* Bit to be "on" if the *source* Bit becomes "on," but takes no action on the other situations.

For example, let us assume the following structure:



In this system, the four objects, b1, b2, b3 and b4, are instances of Bit; b1 and b2, and b2 and b3 are connected by Equality relations; b3 is a trigger of b4. If b1 receives a message "set," then the "set" message is sent to b2, that also activates sending the "set" message to b3. Furthermore, the "set" message is sent to b4 because b3 is a trigger of b4. However, no matter what is sent to b4, nothing happens to b3.

The problem is to make this system evolvable, separating the implementation of the Bit objects and the Equality and Trigger relations, and make this system modular and scalable. Modularity means implementation of relations should be able to adapt to other implementations of Bit objects, and the implementation of the Bit objects should be reusable in other contexts. Scalability means that we may add new Bit objects and even new relations other than Equality or Trigger to that system with no difficulty.

A sample solution for this problem written in McJava is given in Figures 1 and 2. A statement beginning with `mixin` is a *mixin declaration*. A mixin declaration has the following form:

```
class Equality {
  public boolean busy;
  EqAdaptor role1, role2;

  public void join1(EqAdaptor e) {
    role1=e;
    e.equalities.add(this);
  }
  public void join2(EqAdaptor e) {
    role2=e;
    e.equalities.add(this);
  }
  public EqAdaptor getOpponent(EqAdaptor e){
    if (role1 == e) return role2;
    else if (role2 == e) return role1;
    else return null;
  }
}
```

Figure 1. Equality in McJava

```
interface eqI {
  void set();
  void clear();
}

mixin EqAdaptor requires eqI {
  public Vector equalities = new Vector();

  public void set() {
    super.set();
    for (Iterator i=equailties.iterator();
         i.hasNext(); ) {
      Equality e = (Equality)i.next();
      if (!e.busy) {
        e.busy = true;
        e.getOpponent(this).set();
        e.busy = false; }}}

  public void clear() {
    super.clear();
    for (Iterator i=equalities.iterator();
         i.hasNext(); ) {
      Equality e = (Equality)i.next();
      if (!e.busy) {
        e.busy = true;
        e.getOpponent(this).clear();
        e.busy = false; }}}
}
```

Figure 2. A role for equality in McJava

```
class Main {
  public static void main(String[] args) {
    EqAdaptor::Bit b1=new EqAdaptor::Bit();
    EqAdaptor::Bit b2=new EqAdaptor::Bit();
    TrAdaptor::EqAdaptor::Bit b3 =
        new TrAdaptor::EqAdaptor::Bit();
    Bit b4 = new Bit();
    Equality e1 = new Equality();
    Equality e2 = new Equality();
    Trigger t1 = new Trigger();

    e1.join1(b1); e1.join2(b2);
    e2.join1(b2); e2.join2(b3);
    t1.join1(b3); t1.join2(b4);
    ...
  }
}
```

Figure 3. An example program of integrated systems

```
mixin X requires I { ... }
```

where X denotes the name of mixin and I denotes the interface that the mixin *requires*. A mixin may invoke methods declared in the superclass, even though the superclass is not specified when the mixin is declared. The required interface is used to ensure that no "message not understood" error occurs at run-time; if a mixin invokes its superclasses' methods, they must be declared in the interface that the mixin requires. More details can be found in section 3.

In McJava, a mixin cannot be instantiated. Instead, a mixin may be composed with other classes. This *mixin-class composition* may make instances. A mixin may also be composed with other mixins; however, this *mixin-mixin composition* cannot be instantiated. The syntax of composition is concatenating mixin names and class names by `::`, like $Id_1 :: \cdots :: Id_n$.

Figure 1 gives an implementation of `Equality` relation. An `Equality` is a binary relation, so it has two instance variables `role1` and `role2` to hold the Bit objects that are linked with the `Equality` relation. But we would like to apply this `Equality` to other implementations of Bit objects. Therefore, the type of `role1` and `role2` is declared as `EqAdaptor` that abstracts a set of operations the `Equality` is interested in.

`EqAdaptor` is declared as a mixin in Figure 2. It declares methods `set()` and `clear()`. Because those methods invoke `super.set()` and `super.clear()` respectively, `EqAdaptor` requires the interface `eqI` that declares `set()` and `clear()`. `EqAdaptor` may be composed with any class that implements the methods declared in `eqI`. For example, the following class `Bit` may be composed with `EqAdaptor`.

```
class Bit {
  boolean state=false;
  void set() { state=true; }
  void clear() { state=false; }
  boolean get() { return state; }
}
```

At first, the method `set()`/`clear()` of `EqAdaptor` invokes the same method declared in the superclass (for example, the `set()`/`clear()` of `Bit` class). Then, it sends the `set()`/`clear()` message to all the objects that have the `Equality` relation linkage with the sender. The instance variable `busy` declared in `Equality` is a flag that ensures the transition of these method invocations does not end up with an infinite loop.

The Trigger relation is also implemented in the same way. Then, the integrated system may be implemented as in Figure 3. Because b1 and b2 only join in the `Equality` relation, they are created as instances of the composition of `EqAdaptor` and `Bit` (as mentioned before, in McJava the syntax for mixin composition is `::`). On the other hand, b3 is created as an instance of the composition of `TrAdaptor`, `EqAdaptor` and `Bit` (`TrAdaptor` is a mixin for `Trigger`), because it joins in both the `Equality` relation and the `Trigger` relation.

This solution is modular because the implementation of relations may be adapted to other implementations of Bit objects, if they implement the methods declared in `eqI`. Of course, the implementation of the Bit objects may be reused in other contexts. Furthermore, this solution is scalable because we may add new Bit objects easily via `join` methods declared in the relations. Adding new relations is also easy. The key of this solution is using mixin `EqAdaptor` that abstracts the operations in which the `Equality` is interested, and ability to use the name `EqAdaptor` as the type names in formal parameters and field declarations.

At the moment we have developed a preliminary version of McJava compiler that has many restrictions including it still does not have the capability of accessing Java standard libraries. The latest version of McJava compiler is downloadable at `http://kumiki.c.u-tokyo.ac.jp/~kamina/mcjava/`. In the next section, we define formal semantics of the core of McJava.

3 Core McJava: A Core Calculus for McJava

The design of Core McJava is based on FJ [11], a minimum core language for Java. FJ is a very small subset of Java, focusing on just a few key constructs. For example, in FJ constructors always take the same stylized form: there is one parameter for each field, with the same name as the field. FJ provides no side-effective operations, that means a method body always consists of `return` statement followed by an expression. Because FJ provides no side-effects, the only place where the assignment operations may appear is a constructor declaration. In FJ, all the fields are initialized at the object instantiation time. Once initialized, the FJ objects never change its state.

Core McJava shares the same features of FJ explained above. In the following subsections, we present the syntax and operational semantics of Core McJava and its type soundness theorem.

3.1 Syntax

$$
\begin{array}{rcl}
T & ::= & \vec{X} :: C \mid \vec{X} \\
L_C & ::= & \texttt{class } C \texttt{ extends } \vec{X} :: C \ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\} \\
L_X & ::= & \texttt{mixin } X \texttt{ requires } I \ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\} \\
L_I & ::= & \texttt{interface } I \ \{\ \vec{M_I};\ \} \\
K_C & ::= & C(\vec{S}\ \vec{g},\ \vec{T}\ \ \vec{f})\{\texttt{super}(\vec{g});\ \texttt{this.}\vec{f}=\vec{f};\} \\
K_X & ::= & X(\vec{T}\ \ \vec{f})\{\ \texttt{this.}\vec{f}=\vec{f};\} \\
M & ::= & T\ m(\vec{T}\ \vec{x})\{\ \texttt{return } e;\} \\
M_I & ::= & T\ m(\vec{T}\ \vec{x}) \\
e & ::= & x \mid e.f \mid e.m(\vec{e}) \mid \texttt{new } \vec{X} :: C(\vec{e}) \mid (T)e
\end{array}
$$

Figure 4. Abstract syntax of Core McJava

The abstract syntax of Core McJava is given in Figure 4. In this paper, the metavariables d and e range over expressions; K_C and K_X range over constructor declarations; m ranges over method names;

M ranges over method declarations; C and D range over class names; X and Y range over mixin names; R, S, T, U and V range over type names; I ranges over interface names; x ranges over variables; f and g range over field names. As in FJ, we assume that the set of variables includes the special variable `this`, that is considered to be implicitly bound in every method declaration.

We write $\vec{f}$ as a shorthand for a possibly empty sequence $f_1, \cdots, f_n$ and write $\vec{M}$ as a shorthand for $M_1 \cdots M_n$. The length of a sequence $\vec{x}$ is written as $\#(\vec{x})$. Empty sequences are denoted by $\cdot$. Similarly, we write "$\vec{T}\ \vec{f}$" as a shorthand for "$T_1\ f_1, \cdots, T_n\ f_n$", "$\vec{T}\ \vec{f}$;" as a shorthand for "$T_1\ f_1; \cdots T_n\ f_n;$", "$\texttt{this}.\vec{f} = \vec{f};$" as a shorthand for "$\texttt{this}.f_1 = f_1; \cdots \texttt{this}.f_n = f_n;$", $\vec{X}$ as a shorthand for $X_1 :: \cdots :: X_n$.

As in Figure 4, there are two kinds of types: $\vec{X}$ and $\vec{X} :: C$. The former denotes a *mixin-mixin composition* that is generated by composing mixin names, while the latter denotes *mixin-class composition* that is a result of composing mixin names (possibly empty sequence) and a class name. We may instantiate a mixin-class composition by `new` expression but may not instantiate mixin-mixin composition.

We write $T <: U$ when T is a subtype of U. Subtype relation between classes, mixins, and compositions is defined in Figure 5, i.e., subtyping is a reflexive and transitive relation of the immediate subclass relation given by the `extends` clauses in class declarations.

$$T <: T \qquad \text{(S-REFL)}$$

$$\forall T \in \mathit{subsequences}(U) \qquad U <: T \qquad \text{(S-COMP)}$$

$$\frac{T <: S \qquad S <: U}{T <: U} \qquad \text{(S-TRANS)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ \vec{X} :: D\ \{\ldots\}}{C <: \vec{X} :: D} \qquad \text{(S-CLASS)}$$

subsequences is defined as follows:

$$\begin{array}{lcl} \mathit{subsequences}(C) & = & \{C\} \\ \mathit{subsequences}(X) & = & \{X\} \\ \mathit{subsequences}(X :: T) & = & \{X :: U \mid U \in \mathit{subsequences}(T)\} \\ & & \cup \mathit{subsequences}(T) \cup \{X\} \end{array}$$

Figure 5. Subtype relation

One of the novel features of Core McJava is the flexibility of subtyping relation for compositions. A composition is a subtype of all its subsequences. For example, `TrAdaptor::EqAdaptor::Bit` is a subtype of `Bit`, `EqAdaptor`, `TrAdaptor`, `EqAdaptor::Bit`, `TrAdaptor::EqAdaptor`, and `TrAdaptor::Bit`. This subtype rules provide more chances of code reuse. For example, a method whose formal parameter type is a composition type T may be applied to an expression with a composition type that "mixes" some mixins with T.

3.2 Class Table

A Core McJava program is interpreted by a pair of (CT, e) of a *class table* CT and an expression e. A class table is a map from class names and mixin names to class declarations and mixin declarations. The expression e may be considered as the `main` method of the "real" McJava program. The class table is assumed to satisfy the following conditions: (1) $CT(C) = \texttt{class}\ C\ \ldots$ for every $C \in \mathit{dom}(CT)$; (2) $CT(X) = \texttt{mixin}\ X\ \ldots$ for every $X \in \mathit{dom}(CT)$; (3)$\texttt{Object} \notin \mathit{dom}(CT)$; (4) $T \in \mathit{dom}(CT)$ for every class name and mixin name appearing in $\mathit{ran}(CT)$; (5) there are no cycles in the subtype relation induced by CT.

In the induction hypothesis, we abbreviate $CT(C) = \texttt{class}\ C\ \ldots$ and $CT(X) = \texttt{mixin}\ X\ \ldots$ as $\texttt{class}\ C\ \ldots$ and $\texttt{mixin}\ X\ \ldots$, respectively.

3.3 Auxiliary functions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 6, 7 and 8.

The fields of type T, written $\mathit{fields}(T)$, is defined in Figure 6 as a sequence $\vec{T}\ \vec{f}$ pairing the type of each field with its name. If T is a class, $\mathit{fields}(T)$ is a sequence for all the fields declared in class T and all of its superclasses, placing the fields declared in T before the fields declared in T's superclass. If T is a mixin, $\mathit{fields}(T)$ is a sequence for all the fields declared in that mixin. If T is a composition, $\mathit{fields}(T)$ is a sequence for all the fields declared in all of its constituent mixins and a class, placing the fields declared in the left operand of composition after the fields declared in the right operand. For the field lookup, we also have the definition of $\mathit{ftype}(f_i, T)$ that is a type of field f_i declared in T. In contrast with Java, field hiding is not allowed in Core McJava.

$$\mathit{fields}(\texttt{Object}) = \cdot$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ \vec{X} :: D\ \{\vec{T}\ \vec{f};\ K_C\ \vec{M}\} \qquad \mathit{fields}(\vec{X} :: D) = \vec{S}\ \vec{g}}{\mathit{fields}(C) = \vec{S}\ \vec{g},\ \vec{T}\ \vec{f}}$$

$$\frac{\texttt{mixin}\ X\ \texttt{requires}\ I\ \{\vec{T}\ \vec{f};\ K_X\ \vec{M}\}}{\mathit{fields}(X) = \vec{T}\ \vec{f}}$$

$$\frac{\mathit{fields}(X) = \vec{T}\ \vec{f} \qquad \mathit{fields}(T) = \vec{S}\ \vec{g}}{\mathit{fields}(X :: T) = \vec{S}\ \vec{g},\ \vec{T}\ \vec{f}}$$

$$\frac{\mathit{fields}(T) = \vec{T}\ \vec{f}}{\mathit{ftype}(f_i, T) = T_i}$$

Figure 6. Field lookup

The type of method m in type T is given by $\mathit{mtype}(m, T)$. The function mtype is defined in Figure 8 by a pair $\vec{S} \to S$, where $\vec{S}$ is a sequence of argument types and S is a result type. If T is a composition, the left operand of :: is searched first. If m is not found in T, we define it `nil`. The type of method m in interface I is also defined in the same way. Similarly, the body of method m in type T, written $\mathit{mbody}(m, T)$, is a pair, written $\vec{x}.e$ of a sequence of parameters $\vec{x}$ and an expression e. In contrast with Java, method overloading is not allowed in Core McJava.

$$mtype(m,\texttt{Object}) = \texttt{nil}$$

$$\frac{\texttt{class } C \texttt{ extends } \vec{X} :: D\ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\} \quad S\ m(\vec{S}\ \vec{x})\{\ \texttt{return } e;\ \} \in \vec{M}}{mtype(m,C) = \vec{S} \to S}$$

$$\frac{\texttt{class } C \texttt{ extends } \vec{X} :: D\ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\} \qquad m \notin \vec{M}}{mtype(m,C) = mtype(m,\vec{X} :: D)}$$

$$\frac{\texttt{mixin } X \texttt{ requires } I\ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\} \quad S\ m(\vec{S}\ \vec{x})\{\ \texttt{return } e;\ \} \in \vec{M}}{mtype(m,X) = \vec{S} \to S}$$

$$\frac{\texttt{mixin } X \texttt{ requires } I\ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\} \qquad m \notin \vec{M}}{mtype(m,X) = mtype(m,I)}$$

$$\frac{\texttt{interface } I\ \{\ \vec{M_I};\ \} \qquad T\ m(\vec{T}\ \vec{x}) \in \vec{M_I}}{mtype(m,I) = \vec{T} \to T}$$

$$\frac{\texttt{interface } I\ \{\ \vec{M_I};\ \} \qquad m \notin \vec{M}}{mtype(m,I) = \texttt{nil}}$$

$$\frac{mtype(m,X) = \vec{T} \to T}{mtype(m,X :: T_0) = \vec{T} \to T}$$

$$\frac{mtype(m,X) = \texttt{nil} \qquad mtype(m,T_0) = \vec{T} \to T}{mtype(m,X :: T_0) = \vec{T} \to T}$$

Figure 7. Method type lookup

$$mbody(m,\texttt{Object}) = \texttt{nil}$$

$$\frac{\texttt{class } C \texttt{ extends } \vec{X} :: D\ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\} \quad S\ m(\vec{S}\ \vec{x})\{\ \texttt{return } e;\ \} \in \vec{M}}{mbody(m,C) = \vec{x}.e}$$

$$\frac{\texttt{class } C \texttt{ extends } \vec{X} :: D\ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\} \qquad m \notin \vec{M}}{mbody(m,C) = mbody(m,\vec{X} :: D)}$$

$$\frac{\texttt{mixin } X \texttt{ requires } I\ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\} \quad S\ m(\vec{S}\ \vec{x})\{\ \texttt{return } e;\ \} \in \vec{M}}{mbody(m,X) = \vec{x}.e}$$

$$\frac{\texttt{mixin } X \texttt{ requires } I\ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\} \qquad m \notin \vec{M}}{mbody(m,X) = \texttt{nil}}$$

$$\frac{mbody(m,X) = \vec{x}.e}{mbody(m,X :: T) = \vec{x}.e}$$

$$\frac{mbody(m,X) = \texttt{nil} \qquad mbody(m,T) = \vec{x}.e}{mbody(m,X :: T) = \vec{x}.e}$$

Figure 8. Method body lookup

3.4 Dynamic Semantics

The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step". We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The reduction rules are given in Figure 9. There are three reduction rules, one for field access, one for method invocation, and one for casting. The field access reduces to the corresponding argument for the constructor. Due to the stylized form of object constructors, the constructor has one parameter for each field, in the same order as the fields are declared. The method invocation reduces to the expression of the method body, substituting all the parameter $\vec{x}$ with the argument expressions $\vec{d}$ and the special variable `this` with the receiver (we write $[\vec{d}/\vec{x}, e/y]e_0$ for the result of substituting x_1 by d_1,...,x_n by d_n and y by e in e_0).

3.5 Typing

The typing rules for class declarations, mixin declarations, compositions and expressions are given in Figure 10 and 11. An environment Γ is a finite mapping from variables to types, written $\vec{x} : \vec{T}$. The typing judgment for expressions has the form $\Gamma \vdash e : T$, read "in the environment Γ, expression e has type T".

Figure 10 shows the typing rules for methods, classes, mixins, and compositions. The type of the body of the method declaration is a subtype of the declared type, and, for the method in a class, the static type of the overriding method is the same as that of the overriden method. A class definition is well-formed if all the methods declared in that class and the constructor are well-formed. Similarly, a mixin is well-formed if all the method declared in that mixin are well-formed.

The typing rule for compositions checks that the following requirements are met. First, there are no fields declared with the same name between the left component and the right component of the composition. Second, there is no method collision, that is, if some methods are declared with the same name in the left and the right, the static type of both methods is the same. Finally, for all the methods declared in the interface that is required by the left mixin, if the right operand of the composition is a class, it declares the methods named and typed as the same as the interface.

Figure 11 shows the typing rules for expressions. These rules are syntax directed, with one rule for each form of expression, except that there are three rules for casts. Most of them are straightforward extension of the rules in FJ. The typing rules for constructor and method invocations check that the type of each argument is a subtype of the corresponding formal parameter. The typing rule for constructor invocations also checks that there are no instances of mixins and mixin-mixin compositions.

3.6 Properties

We show that Core McJava is type sound. The proof is given in the accompanying Appendix A. We start by stating some lemmas used in the proof of type soundness.

LEMMA 3.1. *If $ftype(f,U) = T$, then $ftype(f,S) = T$ for all $S <: U$.*

Computation:

$$\frac{\mathit{fields}(\vec{X} :: C) = \vec{T}\ \vec{f}}{\texttt{new}\ \vec{X} :: C(\vec{e}).\texttt{f}_i \longrightarrow e_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(m,\ \vec{X} :: C)\ =\ \vec{x}.e_0}{\texttt{new}\ \vec{X} :: C(\vec{e}).m(\vec{d})\ \longrightarrow [\vec{d}/\vec{x}, \texttt{new}\ \vec{X} :: C(\vec{e})/\texttt{this}]e_0} \quad \text{(R-INVK)}$$

$$\frac{\vec{X} :: C\ <:\ T}{(T)\texttt{new}\ \vec{X} :: C(\vec{e}) \longrightarrow \texttt{new}\ \vec{X} :: C(\vec{e})} \quad \text{(R-CAST)}$$

Congruence:

$$\frac{e_0 \longrightarrow e_0^{'}}{e_0.f \longrightarrow e_0^{'}.f} \quad \text{(RC-FIELD)}$$

$$\frac{e_0 \longrightarrow e_0^{'}}{e_0.m(\vec{e}) \longrightarrow e_0^{'}.m(\vec{e})} \quad \text{(RC-INVK-RECV)}$$

$$\frac{e_i \longrightarrow e_i^{'}}{e_0.m(\cdots, e_i, \cdots) \longrightarrow e_0.m(\cdots, e_i^{'}, \cdots)} \quad \text{(RC-INVK-ARG)}$$

$$\frac{e_i \longrightarrow e_i^{'}}{\texttt{new}\ \vec{X} :: C(\cdots, e_i, \cdots) \longrightarrow \texttt{new}\ \vec{X} :: C(\cdots, e_i^{'}, \cdots)} \quad \text{(RC-NEW)}$$

$$\frac{e_0 \longrightarrow e_0^{'}}{(T)e_0 \longrightarrow (T)e_0^{'}} \quad \text{(RC-CAST)}$$

Figure 9. Operational semantics

LEMMA 3.2. *If $\mathit{mtype}(m,U) = \vec{T} \to T_0$, then $\mathit{mtype}(m,T) = \vec{T} \to T_0$ for all $T\ <:\ U$.*

LEMMA 3.3. *If $\Gamma, \vec{x} : \vec{S} \vdash e : U,\ \Gamma \vdash \vec{d} : \vec{R}$ where $\vec{R}\ <:\ \vec{S}$, then $\Gamma \vdash [\vec{d}/\vec{x}]e : T$ for some $T\ <:\ U$.*

LEMMA 3.4. *If $\Gamma \vdash e : T$ where Γ does not include x, then $\Gamma, x : U \vdash e : T$.*

LEMMA 3.5. *If $\mathit{mtype}(m, \vec{X} :: C) = \vec{U} \to U$ and $\mathit{mbody}(m, \vec{X} :: C) = \vec{x}.e$, then, for some U_0 with $\vec{X} :: C\ <:\ U_0$, there exists $T\ <:\ U$ such that $\vec{x} : \vec{U}, \texttt{this} : U_0 \vdash e : T$.*

From the lemmas established above, we derive the type soundness theorem for Core McJava:

THEOREM 3.1 (SUBJECT REDUCTION). *If $\Gamma \vdash e : T$ and $e \longrightarrow e^{'}$, then $\Gamma \vdash e^{'} : T^{'}$ for some $T^{'}\ <:\ T$.*

THEOREM 3.2 (PROGRESS). *Suppose e is a well-typed expression.*

1. *If e includes $\texttt{new}\ \vec{X} :: C(\vec{e}).f$ as a subexpression, then $\mathit{fields}(\vec{X} :: C) = \vec{T}\ \vec{f}$ and $f \in \vec{f}$ for some $\vec{T}$ and $\vec{f}$.*

2. *If e includes $\texttt{new}\ \vec{X} :: C(\vec{e}).m(\vec{d})$ as a subexpression, then $\mathit{mbody}(m, \vec{X} :: C) = \vec{x}.e_0$ and $\#(\vec{x}) = \#(\vec{d})$ for some $\vec{x}$ and e_0.*

$$\frac{\begin{array}{c}\vec{x} : \vec{T}, \texttt{this} : C \vdash e_0 : U_0 \qquad U_0\ <:\ T_0 \\ \texttt{class}\ C\ \texttt{extends}\ \vec{X} :: D\ \{\ldots\} \\ \text{if } \mathit{mtype}(m, \vec{X} :: D) = \vec{S} \to S_0, \text{ then } \vec{S} = \vec{T} \text{ and } S_0 = T_0\end{array}}{T_0\ m(\vec{T}\ \vec{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ C} \quad \text{(T-CMETHOD)}$$

$$\frac{\begin{array}{c}\vec{x} : \vec{T}, \texttt{this} : X \vdash e_0 : S_0 \qquad S_0\ <:\ T_0 \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ldots\}\end{array}}{T_0\ \texttt{m}(\vec{T}\ \vec{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ X} \quad \text{(T-XMETHOD)}$$

$$\frac{\begin{array}{c}K_C = C(\vec{S}\ \vec{g},\ \vec{T}\ \vec{f})\{\texttt{super}(\vec{g});\ \texttt{this}.\vec{f}=\vec{f};\} \\ \mathit{fields}(\vec{X} :: D) = \vec{S}\ \vec{g} \qquad \vec{M}\ \texttt{OK IN}\ C\end{array}}{\texttt{class}\ C\ \texttt{extends}\ \vec{X} :: D\ \{\vec{T}\ \ \vec{f};\ K_C\ \ \vec{M}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

$$\frac{\begin{array}{c}K_X = X(\vec{S}\ \vec{f})\{\ \texttt{this}.\vec{f}=\vec{f};\} \\ \vec{M}\ \texttt{OK IN}\ X\end{array}}{\texttt{mixin}\ X\ \{\vec{T}\ \ \vec{f};\ K_X\ \ \vec{M}\}\ \texttt{OK}} \quad \text{(T-MIXIN)}$$

$$\frac{\begin{array}{c}\mathit{fields}(X) \cap \mathit{fields}(T) = \emptyset \qquad \texttt{interface}\ I\ \{\vec{M}_I\} \\ \texttt{mixin}\ X\ \texttt{requires}\ I\ \{\ \ldots\ \ \vec{M}\ \} \\ \forall m \in \vec{M} \quad \mathit{mtype}(m,X) = \mathit{mtype}(m,T) \text{ or} \\ \mathit{mtype}(m,T) = \texttt{nil} \\ \text{If } T \text{ is a composition } \vec{X} :: C, \text{ then} \\ \forall n \in \vec{M}_I\ \mathit{mtype}(n,I) = \mathit{mtype}(n,T)\end{array}}{X :: T\ \text{ok}} \quad \text{(T-COMP)}$$

Figure 10. Typing rules for classes, mixins, and compsitions

To state type soundness formally, we introduce a value v of an expression e by $v\ \ ::=\ \ \texttt{new}\ \vec{X} :: C(\vec{v})\ .$

THEOREM 3.3 (CORE MCJAVA TYPE SOUNDNESS). *If $\emptyset \vdash e : T$ and $e \longrightarrow^* e^{'}$ with $e^{'}$ a normal form, then $e^{'}$ is either (1) a value v of e with $\emptyset \vdash v : U$ and $U\ <:\ T$, or (2) an expression containing $(U)\texttt{new}\ T(\vec{e})$ where $U\ \not<:\ T$.*

4 Related Work

McJava is a *nominally typed* class-based language, that means the name of a class (or mixin) determines its subtype relationship. On the other hand, in object-oriented languages with *structural subtyping*, the subtype relation between classes is determined by their structures. A core calculus of classes and mixins for structurally typed language was proposed by Bono et al.[4]. Instead, we take a nominal approach, because the host language (Java) is nominaly typed. With nominal approach, we directly define flexible subtype rules regarding to mixin-mixin composition that is not supported by Bono's approach.

To our knowledge, core calculus for mixin types extending Java was originally developed by Flatt et al.[9]. The novel feature of this calculus, named MixedJava, is its ability to implement hygienic mixins [2, 13]. Hygienic mixins postpone the timing of method look up to run-time, avoiding the problem of method collision. This feature is achieved by changing the protocol of method lookup; in MixedJava, each reference to an object is bundled with its *view* of the object, the run-time context information. A view is represented as a chain of mixins for the object's instantiation type. It designates

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T \qquad T\ \mathit{ok}}{\Gamma \vdash e_0.f : T} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{mtype}(m, S) = \vec{S} \to T \qquad \Gamma \vdash \vec{e} : \vec{T} \qquad \vec{T} <: \vec{S} \qquad T\ \text{ok}}{\Gamma \vdash e_0.m(\vec{e}) : T} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\vec{X} :: C) = \vec{S}\ \vec{f} \qquad \Gamma \vdash \vec{e} : \vec{T} \qquad \vec{T} <: \vec{S} \qquad \vec{X} :: C\ \text{ok}}{\Gamma \vdash \mathsf{new}\ \vec{X} :: C(\vec{e}) : \vec{X} :: C} \qquad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad S <: T \qquad T\ \text{ok}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad T <: S \qquad T \neq S \qquad T\ \text{ok}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-DCAST)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad T \not<: S \qquad S \not<: T \qquad T\ \text{ok} \qquad \textit{stupid warning}}{\Gamma \vdash (T)e_0 : T} \qquad \text{(T-SCAST)}$$

Figure 11. Typing rules for expressions

a specific point in the full mixin chain, the static type of that object, for selecting methods during dynamic dispatch.

Even though the proposal of hygienic mixins itself is useful and feasible in the practical programming languages [2, 13], implementing the operational semantics and the type system of MixedJava (that supports mixin-types) on the JVM (that contains no information for views) is difficult. Furthermore, McJava defines very flexible subtyping relations. For example, the subtype relation $X :: Y :: C <: X :: C$ does not exist in MixedJava. McJava does not support hygienic mixins. Instead, the type system of McJava detects the method collisions statically, allowing programmers to treat them manually.

Jam [3] is a practical proposal for adding mixin-types to Java. Jam gives semantics of mixin compositions formally by translation to Java. Based on that semantics, mixin-types in Jam have some significant limitations; In Jam, a mixin-mixin composition is not allowed. Furthermore, using the keyword `this` in mixins is very restricted. For example, using `this` as the argument value for the method invocation is not allowed in Jam. Formulating operational semantics at an abstract level, these limitations are resolved in Core McJava.

Mixin modules [7], essentially motivated by the problem of interaction with recursive constructs that cross module boundaries in module systems of functional languages, mainly focus on facilitating reusing large scale programming constructs such as frameworks [8]. Our work, on the other hand, mainly focuses on integrating mixin-types and its flexible subtyping with real programming languages. The work [8] sacrifices mixin subtyping in favor of allowing method renaming.

MixJuice [10] is also independently proposed by Ichisugi et al. to modularize large scale compilation unit. MixJuice is designed as an extension of Java with *difference-based modules* that are separately compilable units of encapsulation. The design of mixins in MixJuice is actually different from our work. In MixJuice, the *providers* of mixins control encapsulation. In the case of diamond inheritance, the users have the responsibility of composing them without breaking encapsulation. In McJava, on the contrary, the *users* of mixins control encapsulation because these mixins are parametrized over their superclasses. Users add superclasses to mixins and there are no case of diamond inheritance.

Schärly et al. proposed *traits* [14], fine grained reusable components as building blocks for classes. Traits support method renaming that overcomes the problem of method collision. When traits are composed, the members of those traits are "flatterned" into one class, which also solves the ordering problem of mixins. Our work, in contrast with traits, has more focus on declaring a mixin as a type, and studying their subtype relations. We would also like to note that the ordering of mixins is useful particularly when we "extend" a parametrized superclass with the same name of method as the superclass, and invoke it via `super.`*`m`*, where *`m`* is a method name.

Mixins may be used as vehicles to directly implement *roles* in terms of role modeling [19]. Epsilon [21, 20], a role-based executable model, was also proposed for this purpose. Currently Epsilon lacks the feature of static typing. We consider McJava and its core calculus provides some basic understanding to study static typing on Epsilon.

5 Conclusion and Directions for Further Research

As shown in section 2, adding mixin-types to a traditional object-oriented language significantly improves its expressive power. Based on FJ, the core language for Java, we have developed the core language for Java with mixin-types. We have shown the core language for mixin-types is type sound. We believe that these results provide a convincing way for adding mixin-types to nominally typed object-oriented languages such as Java and C#.

Finally, we point out some issues remained for the future work:

Formal reasoning of compilation We have developed a McJava compiler that translates McJava programs into Java programs. Using a formal method will enhance understanding on the correctness of that translation. To do this, a possible way is to design a core language of the target language and show the compilation from Core McJava to the core language is correct. We consider FJ is not adequate for the target language, because the compilation strongly depends on the existence of interfaces in the target language. Thus, we have to extend FJ to obtain an appropriate target core language.

Core McJava with generic types J2SE 1.5 is the next major revision to the Java platform and language that will include major enhancements such as generic types. It is interesting to integrate the feature of genericity with mixin-types.

McJava compiler for practical use Even though a preliminary version of McJava compiler is implemented, it is desirable to develop a McJava compiler for practical use. The compiler should support, for example, separate compilation that is not supported by the current version.

Acknowledgements: The authors would like to thank Atsushi

Igarashi, Hidehiko Masuhara and Etsuya Shibayama for their very helpful comments on the earlier version of this calculus.

6 References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Conference Proceedings of OOPSLA '97, Atlanta*, pages 49–65. ACM, 1997.

[2] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of OOPSLA2003*, pages 96–114, 2003.

[3] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – A smooth extension of java with mixins. In *ECOOP 2000*, pages 154–178, 2000.

[4] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In *Proceedings of ECOOP'99*, LNCS 1628, pages 43–66, 1999.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

[6] Gilad Bracha, Martin Odersky, David Stroutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA 1998*, pages 183–200, 1998.

[7] Dominic Duggan and Constantinous Sourelis. Mixin modules. In *ICFP'96*, pages 262–272, 1996.

[8] Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *OOPSLA 2001*, pages 223–240, 2001.

[9] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.

[10] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *Proceedings of ECOOP 2002*, pages 62–88, 2002.

[11] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[13] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA2001*, 2001.

[14] Nathanael Schärly, Steṕhane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *ECOOP 2003*, LNCS 2743, pages 248–274, 2003.

[15] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings ECOOP'98*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570, 1998.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[17] Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In *Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 19–26, 2002.

[18] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transaction on Software Engineering and Methodology*, 1(3):229–268, 1992.

[19] Tetsuo Tamai. Objects and roles: modeling based on the dualistic view. *Information and Software Technology*, 41(14):1005–1010, 1999.

[20] Tetsuo Tamai. Evolvable Programming based on Collaboration-Field and Role Model. In *International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 1–5, 2002.

[21] Naoyasu Ubayashi and Tetsuo Tamai. Separation of Concerns in Mobile Agent Applications. In *Metalevel Architectures and Separation of Crosscutting Conserns – Proceedings of the 3rd International Conference (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 89–109, 2001.

[22] Michael VanHislt and David Notkin. Using C++ templates to implement role-based designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.

A Proof of Core McJava Type Soundness Theorem

Intuitionally, the step of proving Core McJava type soundness is almost the same as that of FJ, but details vary a little.

A.1 Proof of Lemma 3.1

Straightforward induction on the derivation of subtype relation $<:$ and *ftype*. □

A.2 Proof of Lemma 3.2

Straightforward induction on the derivation of subtype relation $<:$, *mtype* and T-COMP. Note that whether m is defined in C or not, $mtype(m,C) = mtype(m,\vec{X} :: D)$ where `class` C `extends` $\vec{X} :: D$ $\{\ldots\}$. Similarly, note that whether m is defined in X or not, $mtype(m,X :: T) = mtype(m,X)$ (see the rule T-COMP). □

A.3 Proof of Lemma 3.3

By induction on the derivation of $\Gamma,\vec{x} : \vec{S} \vdash e : U$.

Case T-VAR.

$$e = x \quad U = \Gamma(x)$$

If $x \notin \vec{x}$, then the conclusion is immediate, since $[\vec{d}/\vec{x}]x = x$. If $x = x_i$, and $U = S_i$, then letting $R_i = T$ finishes the case because $[\vec{d}/\vec{x}]x = [\vec{d}/\vec{x}]x_i = d_i$, $d_i : R_i$ and $R_i <: S_i = U$.

Case T-FIELD.

$$\begin{array}{ll} e = e_0.f_i & \Gamma,\vec{x} : \vec{S} \vdash e_0 : \vec{X} :: C \\ fields(\vec{X} :: C) = \vec{T}\ \vec{f} & U = T_i \end{array}$$

By the induction hypothesis, there is some T_0 such that $\Gamma \vdash [\vec{d}/\vec{x}]e_0 : T_0$ and $T_0 <: \vec{X} :: C$. Then, by Lemma 3.1, $ftype(f_i,T_0) = ftype(f_i,\vec{X} :: C)$. Therefore, by the rule T-FIELD, $\Gamma \vdash ([\vec{d}/\vec{x}]e_0).f_i : T_i$.

Case T-INVK.

$$e = e_0.m(\vec{e}) \quad \Gamma,\vec{x}:\vec{S} \vdash e_0 : \vec{X} :: C \quad mtype(m,\vec{X} :: C) = \vec{V} \to U$$
$$\Gamma,\vec{x}:\vec{S} \vdash \vec{e} : \vec{U} \quad \vec{U} <: \vec{V}$$

By the induction hypothesis, there are some T_0 and $\vec{X}$ such that

$$\Gamma \vdash [\vec{d}/\vec{x}]e_0 : T_0 \quad T_0 <: \vec{X} :: C$$
$$\Gamma \vdash [\vec{d}/\vec{x}]\vec{e} : \vec{T} \quad \vec{T} <: \vec{U}$$

By Lemma 3.2, $mtype(m,T_0) = mtype(m,\vec{X} :: C) = \vec{V} \to U$. Then, by S-TRANS, $\vec{T} <: \vec{V}$. Therefore, by the rule T-INVK, $\Gamma \vdash [\vec{d}/\vec{x}]e_0.m([\vec{d}/\vec{x}]\vec{e}) : U$.

Case T-NEW.

$$e = \mathsf{new}\ \vec{X} :: C(\vec{e}) \quad fields(\vec{X} :: C) = \vec{U}\ \vec{f}$$
$$\Gamma,\vec{x}:\vec{S} \vdash \vec{e} : \vec{T} \quad \vec{T} <: \vec{U}$$

By the induction hypothesis, there are some $\vec{V}$ such that $\Gamma \vdash [\vec{d}/\vec{x}]\vec{e} : \vec{V}$ and $\vec{V} <: \vec{T}$. Then, by the rule S-TRANS, $\vec{V} <: \vec{U}$. Therefore, by the rule T-NEW, $\Gamma \vdash \mathsf{new}\ \vec{X} :: C([\vec{d}/\vec{x}]\vec{e}) : \vec{X} :: C$.

Case T-UCAST.

$$e = (U)e_0 \quad \Gamma,\vec{x}:\vec{S} \vdash e_0 : T \quad T <: U$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\vec{d}/\vec{x}]e_0 : V$ and $V <: T$. Then, by the rule S-TRANS, $V <: U$. Therefore, by the rule T-UCAST, $\Gamma \vdash (U)([\vec{d}/\vec{x}]e_0) : U$.

Case T-DCAST.

$$e = (U)e_0 \quad \Gamma,\vec{x}:\vec{S} \vdash e_0 : T \quad U <: T \quad U \neq T$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\vec{d}/\vec{x}]e_0 : V$ and $V <: T$. If $V <: U$ or $U <: V$, then $\Gamma \vdash (U)([\vec{d}/\vec{x}]e_0) : U$ by the rule T-UCAST or T-DCAST, respectively. On the other hand, by the rule T-SCAST, $\Gamma \vdash (U)([\vec{d}/\vec{x}]e_0) : U$ (with a *stupid warning*).

Case T-SCAST.

$$e = (U)e_0 \quad \Gamma,\vec{x}:\vec{S} \vdash e_0 : T \quad U \not<: T \quad T \not<: U$$

By the induction hypothesis, there are some V such that $\Gamma \vdash [\vec{d}/\vec{x}]e_0 : V$ and $V <: T$. If $V \not<: U$, then, by the rule T-SCAST, $\Gamma \vdash (U)([\vec{d}/\vec{x}]e_0) : U$ (with a *stupid warning*). If $V <: U$, then, by the rule T-UCAST, $\Gamma \vdash (U)([\vec{d}/\vec{x}]e_0) : U$. □

A.4 Proof of Lemma 3.4

Straightforward induction. □

A.5 Proof of Lemma 3.5

By induction on the derivation of *mbody*. In the base case (where m is defined in $CT(T_0)$), it is easy to prove by the rule T-CMETHOD, if T_0 is a class type, or by the rule T-XMETHOD, if T_0 is a mixin type. The induction step is also straightforward. □

A.6 Proof of Theorem 3.1

By induction on a derivation of $e \longrightarrow e'$.

Case R-FIELD.

$$e = (\mathsf{new}\ \vec{X}(C)(\vec{e})).f_i \quad e' = e_i \quad fields(\vec{X}(C)) = \vec{U}\ \vec{f}$$

By the rule T-FIELD, we have $\Gamma \vdash \mathsf{new}\ \vec{X} :: C(\vec{e}) : \vec{Y} :: D,\ T = U_i$ for some $\vec{Z} :: E$. Then, by the rule T-NEW, we have $\Gamma \vdash \vec{e} : \vec{T},\ \vec{T} <: \vec{U},\ \vec{Y} :: D = \vec{X} :: C$. In particular, $\Gamma \vdash e_i : T_i$, finishing the case, since $T_i <: U_i$.

Case R-INVK.

$$e = (\mathsf{new}\ \vec{X} :: C(\vec{e}).m(\vec{d}) \quad mbody(m,\vec{X} :: C) = \vec{x}.e_0$$
$$e' = [\vec{d}/\vec{x},(\mathsf{new}\ \vec{X} :: C(\vec{e}))/\mathtt{this}]e_0$$

By the rule T-INVK and T-NEW, we have

$$\Gamma \vdash \mathsf{new}\ \vec{X} :: C : \vec{X} :: C \quad mtype(m,\vec{X} :: C) = \vec{U} \to T$$
$$\Gamma \vdash \vec{d} : \vec{T} \quad \vec{T} <: \vec{U}$$

for some $\vec{T}$ and $\vec{U}$. By Lemma 3.5, $\vec{x}:\vec{U},\mathtt{this} : U_0 \vdash e_0 : S$ for some U_0 and S where $\vec{X} :: C <: U_0$ and $S <: T$. By Lemma 3.4, $\Gamma,\vec{x}:\vec{U},\mathtt{this} : U_0 \vdash e : S$. Then, by Lemma 3.3, $\Gamma \vdash [\vec{d}/\vec{x},(\mathsf{new}\ \vec{X} :: C(\vec{e}))/\mathtt{this}]e_0 : V$ for some $V <: S$. Then we have $V <: T$ by transitivity of $<:$. Finally, letting $V = T'$ finishes this case.

Case R-CAST.

$$e = (U)(\mathsf{new}\ \vec{X} :: C(\vec{e})) \quad \vec{X} :: C(\vec{e}) <: U \quad e' = \mathsf{new}\ \vec{X} :: C(\vec{e})$$

Because of the assumption $\vec{X} :: C <: T$, the proof of $\Gamma \vdash (T)\mathsf{new}\ \vec{X} :: C(\vec{e}) : U$ must end with the rule T-UCAST. By the rules T-UCAST and T-NEW, we have $\Gamma \vdash (U)\mathsf{new}\ \vec{X} :: C(\vec{e}) : U$.

The cases for congruence rules are easy. □

A.7 Proof of Theorem 3.2

If e has $\mathsf{new}\ \vec{X} :: C(\vec{e}).f$ as a subexpression, by well-typedness of the subexpression, it is easy to check that $fields(\vec{X} :: C)$ is well defined and f appears in it. Similarly, if e has $\mathsf{new}\ \vec{X} :: C(\vec{e}).m(\vec{d})$ as a subexpression, it is also easy to show $mbody(m,vecX :: C) = \vec{x}.e_0$ and $\#(\vec{x}) = \#(vecd)$, since $mtype(m,\vec{X} :: C) = \vec{T} \to U$ where $\#(\vec{x}) = \#(T)$. □

A.8 Proof of Theorem 3.3

Immediate from Theorem 3.1 and 3.2. □