A Programmer's Reduction Semantics for Classes and Mixins

TR 97-293: Corrected Version, June 8, 1999
Original in *Formal Syntax and Semantics of Java*, LNCS volume 1523 (1999)

Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen

Rice University

Abstract. While class-based object-oriented programming languages provide a flexible mechanism for re-using and managing related pieces of code, they typically lack linguistic facilities for specifying a uniform extension of many classes with one set of fields and methods. As a result, programmers are unable to express certain abstractions over classes. In this paper we develop a model of class-to-class functions that we refer to as *mixins*. A mixin function maps a class to an extended class by adding or overriding fields and methods. Programming with mixins is similar to programming with single inheritance classes, but mixins more directly encourage programming to interfaces. The paper develops these ideas within the context of Java. The results are

1. an intuitive model of an essential Java subset;
2. an extension that explains and models mixins; and
3. type soundness theorems for these languages.

1 Organizing Programs with Functions and Classes

Object-oriented programming languages offer classes, inheritance, and overriding to parameterize over program pieces for management purposes and re-use. Functional programming languages provide various flavors of functional abstractions for the same purpose. The latter model was developed from a well-known, highly developed mathematical theory. The former grew in response to the need to manage large programs and to re-use as many components as possible.

Each form of parameterization is useful for certain situations. With higher-order functions, a programmer can easily define many functions that share a similar core but differ in a few details. As many language designers and programmers readily acknowledge, however, the functional approach to parameterization is best used in situations with a relatively small number of parameters. When a function must consume a large number of arguments, the approach quickly

[0] This research was partially supported by a Lodieska Stockbridge Vaughan Fellowship, NSF Graduate Research Fellowship, NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

becomes unwieldy, especially if many of the arguments are the same for most of the function's uses.[1]

Class systems provide a simple and flexible mechanism for managing collections of highly parameterized program pieces. Using class extension (inheritance) and overriding, a programmer derives a new class by specifying only the elements that change in the derived class. Nevertheless, a pure class-based approach suffers from a lack of abstractions that specify uniform extensions and modifications of classes. For example, the construction of a programming environment may require many kinds of text editor frames, including frames that can contain multiple text buffers and frames that support searching. In Java, for example, we cannot implement all combinations of multiple-buffer and searchable frames using derived classes. If we choose to define a class for all multiple-buffer frames, there can be no class that includes only searchable frames. Hence, we must repeat the code that connects a frame to the search engine in at least two branches of the class hierarchy: once for single-buffer searchable frames and again for multiple-buffer searchable frames. If we could instead specify a mapping from editor frame classes to searchable editor frame classes, then the code connecting a frame to the search engine could be abstracted and maintained separately.

Some class-based object-oriented programming languages provide multiple inheritance, which permits a programmer to create a class by extending more than one class at once. A programmer who also follows a particular protocol for such extensions can mimic the use of class-to-class functions. Common Lisp programmers refer to this protocol as *mixin programming* [21, 22], because it roughly corresponds to mixing in additional ingredients during class creation. Bracha and Cook [7] designed a language of class manipulators that promote mixin thinking in this style and permit programmers to build mixin-like classes. Unfortunately, multiple inheritance and its cousins are semantically complex and difficult to understand for programmers.[2] As a result, implementing a mixin protocol with these approaches is error-prone and typically avoided.

For the design of MzScheme's class and interface system [16], we experimented with a different approach. In MzScheme, classes form a single inheritance hierarchy, but are also first-class values that can be created and extended at run-time. Once this capability was available, the programmers of our team used it extensively for the construction of DrScheme [15], a Scheme programming environment. A thorough analysis, however, reveals that the code only contains first-order functions on classes.

In this paper, we present a typed model of such "class functors" for Java [18]. We refer to the functors as *mixins* due to their similarity to Common Lisp's multiple inheritance mechanism and Bracha's class operators. Our proposal is superior in that it isolates the useful aspects of multiple inheritance yet retains the simple, intuitive nature of class-oriented, single-inheritance Java program-

[1] Function entry points *à la* Fortran or keyword arguments *à la* Common Lisp are a symptom of this problem, not a remedy.

[2] Dan Friedman determined in an informal poll in 1996 that almost nobody who teaches C++ teaches multiple inheritance [pers. com.].

ming. In the following section, we develop a calculus of Java classes. In the third section, we motivate mixins as an extension of classes using a small but illuminating example. The fourth section extends the type-theoretic model of Java to mixins. The last section considers implementation strategies for mixins and puts our work in perspective.

```
interface Place^i ...
interface Barrier^i ...
interface Door^i
    extends Place^i, Barrier^i ...
...
class Door^c extends Object
      implements Door^i {
  ...
  Room^c Enter(Person^c p) { ... }
  ...
}
class LockedDoor^c extends Door^c ...
class ShortDoor^c extends Door^c ...
```

$\Rightarrow$

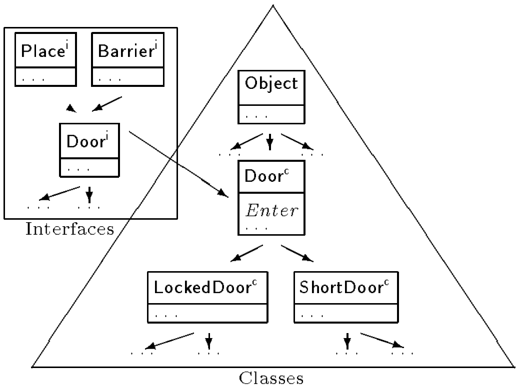

Fig. 1. A program determines a static directed acyclic graph of types

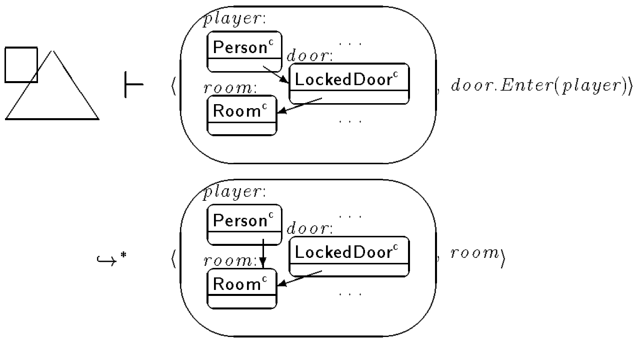

Fig. 2. Given a type graph, reductions map a store-expression pair to a new pair

2 A Model of Classes

ClassicJava is a small but essential subset of sequential Java. To model its type structure and semantics, we use well-known type elaboration and rewriting techniques for Scheme and ML [14, 19, 30]. Figures 1 and 2 illustrate the essence of our strategy. Type elaboration verifies that a program defines a static tree of classes and a directed acyclic graph (DAG) of interfaces. A type is simply a node in the combined graph. Each type is annotated with its collection of fields and methods, including those inherited from its ancestors.

Evaluation is modeled as a reduction on expression-store pairs in the context of a static type graph. Figure 2 demonstrates reduction using a pictorial representation of the store as a graph of objects. Each object in the store is a tagged record of field values, where the tag indicates the class of the object and its field values are references to other objects. A single reduction step may extend the store with a new object, or it may modify a field for an existing object in the store. Dynamic method dispatch is accomplished by matching the class tag of an object in the store with a node in the static class tree; a simple relation on this tree selects an appropriate method for the dispatch.

The class model relies on as few implementation details as possible. For example, the model defines a mathematical relation, rather than a selection algorithm, to associate fields with classes for the purpose of type-checking and evaluation. Similarly, the reduction semantics only assumes that an expression can be partitioned into a proper redex and an (evaluation) context; it does not provide a partitioning algorithm. The model can easily be refined to expose more implementation details [13, 19].

$$
\begin{array}{rcl}
P & = & \mathit{defn}^{*}\ e \\
\mathit{defn} & = & \mathbf{class}\ c\ \mathbf{extends}\ c\ \mathbf{implements}\ i^{*}\ \{\ \mathit{field}^{*}\ \mathit{meth}^{*}\ \} \\
 & & \mid\ \mathbf{interface}\ i\ \mathbf{extends}\ i^{*}\ \{\ \mathit{meth}^{*}\ \} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{meth} & = & t\ \mathit{md}\ (\ \mathit{arg}^{*}\ )\ \{\ \mathit{body}\ \} \\
\mathit{arg} & = & t\ \mathit{var} \\
\mathit{body} & = & e \mid \mathbf{abstract} \\
e & = & \mathbf{new}\ c \mid \mathit{var} \mid \mathsf{null} \mid \underline{e : c}.\mathit{fd} \mid \underline{e : c}.\mathit{fd} = e \\
 & & \mid e.\mathit{md}\ (e^{*}\ ) \mid \mathbf{super}\,\underline{\equiv \mathbf{this} : c}.\mathit{md}\ (e^{*}\ ) \\
 & & \mid \mathbf{view}\ t\ e \mid \mathbf{let}\ \mathit{var} = e\ \mathbf{in}\ e \\
\mathit{var} & = & \text{a variable name or } \mathbf{this} \\
c & = & \text{a class name or } \mathsf{Object} \\
i & = & \text{interface name or } \mathsf{Empty} \\
\mathit{fd} & = & \text{a field name} \\
\mathit{md} & = & \text{a method name} \\
t & = & c \mid i
\end{array}
$$

Fig. 3. ClassicJava syntax; underlined phrases are inserted by elaboration and are not part of the surface syntax

2.1 ClassicJava Programs

The syntax for ClassicJava is shown in Figure 3. A program P is a sequence of class and interface definitions followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations, while an interface consists of methods only. A method body in a class can be **abstract**, indicating that the method must be overridden in a subclass before the class is instantiated. A method body in an interface must be **abstract**. As in Java, classes are instantiated with the **new** operator, but there are no class constructors in ClassicJava; instance variables are always initialized to null. Finally, the **view** and **let** forms represent Java's casting expressions and local variable bindings, respectively.

The evaluation rules for ClassicJava are defined in terms of individual expressions, but certain rules require information about the context of the expression in the original program. For example, the evaluation rule for a field use depends on the syntactic type of the object position, which is determined by the expression's type environment in the original program. To remove such context dependencies before evaluation, the type-checker annotates field uses and **super** invocations with extra source-context information (see the underlined parts of the syntax).

A valid ClassicJava program satisfies a number of simple predicates and relations; these are described in Figures 4 and 5. For example, the ClassesOnce(P) predicate states that each class name is defined at most once in the program P. The relation $\prec^c_P$ associates each class name in P to the class it extends, and the (overloaded) $\Subset^c_P$ relations capture the field and method declarations of P.

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the subclass relation $\leq^c_P$, which is a partial order if the CompleteClasses(P) and WellFoundedClasses(P) predicates hold. In this case, the classes declared in P form a tree that has Object at its root.

If the program describes a tree of classes, we can "decorate" each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the *minimum* (*i.e.*, farthest from the root) superclass that declares the field or method. This algorithm is described precisely by the $\in^c_P$ relations. The $\in^c_P$ relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations. The superinterface declaration relation $\prec^i_P$ induces a subinterface relation $\leq^i_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The set methods of an interface, as described by $\in^i_P$, is the union of the interface's declared methods and the methods of its superinterfaces.

Finally, classes and interfaces are related by **implements** declarations, as captured in the $\prec\!\!\!\prec^c_P$ relation. This relation is a set of edges joining the class tree and the interface graph, completing the sub*type* picture of a program. A type in the full graph is a subtype of all of its ancestors.

The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable T is used for method signatures $(t \ldots \longrightarrow t)$, V for variable lists $(var \ldots)$, and Γ for environments mapping variables to types. Ellipses on the baseline $(\ldots)$ indicate a repeated pattern or continued sequence, while centered ellipses $(\cdots)$ indicate arbitrary missing program text (not spanning a class or interface definition).

ClassesOnce(P)	Each class name is declared only once **class** $c \cdots$ **class** $c' \cdots$ is in $P \implies c \neq c'$
FieldOncePerClass(P)	Field names in each class declaration are unique **class** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$
MethodOncePerClass(P)	Method names in each class declaration are unique **class** $\cdots \{ \cdots md\ (\cdots)\ \{\cdots\} \cdots md'\ (\cdots)\ \{\cdots\} \cdots \}$ is in $P \implies md \neq md'$
InterfacesOnce(P)	Each interface name is declared only once **interface** $i \cdots$ **interface** $i' \cdots$ is in $P \implies i \neq i'$
InterfacesAbstract(P)	Method declarations in an interface are **abstract** **interface** $\cdots \{ \cdots md\ (\cdots)\ \{e\} \cdots \}$ is in $P \implies e$ is **abstract**
$\prec^{c}_{P}$	Class is declared as an immediate subclass $c \prec^{c}_{P} c' \Leftrightarrow$ **class** c **extends** $c' \cdots \{ \cdots \}$ is in P
$\in\!\!\in^{c}_{P}$	Field is declared in a class $\langle c.fd,\ t\rangle \in\!\!\in^{c}_{P} c \Leftrightarrow$ **class** $c \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\in\!\!\in^{c}_{P}$	Method is declared in class $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1 \ldots var_n),\ e\rangle \in\!\!\in^{c}_{P} c$ $\Leftrightarrow$ **class** $c \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec^{i}_{P}$	Interface is declared as an immediate subinterface $i \prec^{i}_{P} i' \Leftrightarrow$ **interface** i **extends** $\cdots i' \cdots \{ \cdots \}$ is in P
$\in\!\!\in^{i}_{P}$	Method is declared in an interface $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1 \ldots var_n),\ e\rangle \in\!\!\in^{i}_{P} i$ $\Leftrightarrow$ **interface** $i \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec\!\!\prec^{c}_{P}$	Class declares implementation of an interface $c \prec\!\!\prec^{c}_{P} i \Leftrightarrow$ **class** $c \cdots$ **implements** $\cdots i \cdots \{ \cdots \}$ is in P
$\leq^{c}_{P}$	Class is a subclass $\leq^{c}_{P} \equiv$ the transitive, reflexive closure of $\prec^{c}_{P}$
CompleteClasses(P)	Classes that are extended are defined $\mathrm{rng}(\prec^{c}_{P}) \subseteq \mathrm{dom}(\prec^{c}_{P}) \cup \{\mathsf{Object}\}$
WellFoundedClasses(P)	Class hierarchy is an order $\leq^{c}_{P}$ is antisymmetric
ClassMethodsOK(P)	Method overriding preserves the type $(\langle md,\ T,\ V,\ e\rangle \in\!\!\in^{c}_{P} c$ and $\langle md,\ T',\ V',\ e'\rangle \in\!\!\in^{c}_{P} c') \implies (T = T'$ or $c \not\leq^{c}_{P} c')$
$\in^{c}_{P}$	Field is contained in a class $\langle c'.fd,\ t\rangle \in^{c}_{P} c$ $\Leftrightarrow \langle c'.fd,\ t\rangle \in\!\!\in^{c}_{P} c'$ and $c' = \min\{c'' \mid c \leq^{c}_{P} c''$ and $\exists t'$ s.t. $\langle c''.fd,\ t'\rangle \in\!\!\in^{c}_{P} c''\}$
$\in^{c}_{P}$	Method is contained in a class $\langle md,\ T,\ V,\ e\rangle \in^{c}_{P} c$ $\Leftrightarrow (\langle md,\ T,\ V,\ e\rangle \in\!\!\in^{c}_{P} c'$ and $c' = \min\{c'' \mid c \leq^{c}_{P} c''$ and $\exists e', V'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \in\!\!\in^{c}_{P} c''\})$

Fig. 4. Predicates and relations in the model of ClassicJava (Part I)

2.2 ClassicJava Type Elaboration

The type elaboration rules for ClassicJava are defined by the following judgements:

$\vdash_{\mathsf{p}} P \Rightarrow P' : t$	P elaborates to P' with type t
$P \vdash_{\mathsf{d}} defn \Rightarrow defn'$	$defn$ elaborates to $defn'$
$P, t \vdash_{\mathsf{m}} meth \Rightarrow meth'$	$meth$ in t elaborates to $meth'$
$P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$	e elaborates to e' with type t in Γ
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e has type t using subsumption in Γ
$P \vdash_{\mathsf{t}} t$	t exists

$\leq^{\mathsf{i}}_P$	Interface is a subinterface $\leq^{\mathsf{i}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{i}}_P$
CompleteInterfaces(P)	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_P) \cup \mathrm{rng}(\not\ll^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}$
WellFoundedInterfaces(P)	Interface hierarchy is an order $\leq^{\mathsf{i}}_P$ is antisymmetric
$\ll^{\mathsf{c}}_P$	Class implements an interface $c \ll^{\mathsf{c}}_P i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_P c'$ and $i' \leq^{\mathsf{i}}_P i$ and $c' \not\ll^{\mathsf{c}}_P i'$
InterfaceMethodsOK(P)	Interface inheritance or redeclaration of methods is consistent $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i$ and $\langle md,\ T',\ V',\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i'$ $\Longrightarrow (T = T'$ or $\forall i''(i'' \not\leq^{\mathsf{i}}_P i$ or $i'' \not\leq^{\mathsf{i}}_P i'))$
$\in^{\mathsf{i}}_P$	Method is contained in an interface $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i'$
ClassesImplementAll(P)	Classes supply methods to implement interfaces $c \not\ll^{\mathsf{c}}_P i \Longrightarrow (\forall md, T, V\ \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \Longrightarrow \exists e, V'$ s.t. $\langle md,\ T,\ V',\ e\rangle \in^{\mathsf{c}}_P c)$
NoAbstractMethods(P, c)	Class has no **abstract** methods (can be instantiated) $\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{c}}_P c \Longrightarrow e \neq \mathbf{abstract}$
$\leq_P$	Type is a subtype $\leq_P \equiv \leq^{\mathsf{c}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{c}}_P$
$\in_P$	Field or method is in a type $\in_P \equiv \in^{\mathsf{c}}_P \cup \in^{\mathsf{i}}_P$

Fig. 5. Predicates and relations in the model of ClassicJava (Part II)

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{l}\textsc{ClassesOnce}(P)\ \ \textsc{InterfacesOnce}(P)\ \ \textsc{MethodOncePerClass}(P)\ \ \textsc{FieldOncePerClass}(P)\ \ \textsc{CompleteClasses}(P)\\ \textsc{WellFoundedClasses}(P)\ \ \textsc{CompleteInterfaces}(P)\ \ \textsc{WellFoundedInterfaces}(P)\ \ \textsc{InterfaceMethodsOK}(P)\\ \textsc{InterfacesAbstract}(P)\ \ \textsc{ClassesImplementAll}(P)\ \ P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n]\\ P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \qquad \text{where } P = defn_1 \ldots defn_n\ e\end{array}}{\vdash_{\mathsf{p}} defn_1 \ldots defn_n\ e \Rightarrow defn'_1 \ldots defn'_n\ e' : t}[\mathbf{prog^{c}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} meth_k \Rightarrow meth'_k \text{ for each } k \in [1,p]}{\begin{array}{l}P \vdash_{\mathsf{d}} \mathbf{class}\ c \cdots \{\ t_1\ fd_1 \ldots t_n\ fd_n \quad \Rightarrow \mathbf{class}\ c \cdots \{\ t_1\ fd_1 \ldots t_n\ fd_n\\ \qquad\qquad meth_1 \ldots meth_p\ \} \qquad\qquad\qquad meth'_1 \ldots meth'_p\ \}\end{array}}[\mathbf{defn^{c}}]$$

$$\frac{P, i \vdash_{\mathsf{m}} meth_j \Rightarrow meth_j \text{ for each } j \in [1,p]}{P \vdash_{\mathsf{d}} \mathbf{interface}\ i \cdots \{\ meth_1 \ldots meth_p\ \} \Rightarrow \mathbf{interface}\ i \cdots \{\ meth_1 \ldots meth_p\ \}}[\mathbf{defn^{i}}]$$

$\vdash_{\mathsf{m}}$

$$\frac{P \vdash_{\mathsf{t}} t \quad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \quad P, [\mathbf{this} : t_o,\ var_1 : t_1, \ldots var_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, t_o \vdash_{\mathsf{m}} t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e\ \} \Rightarrow t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e'\ \}}[\mathbf{meth}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} c \qquad \textsc{NoAbstractMethods}(P, c)}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{new}\ c \Rightarrow \mathbf{new}\ c : c}[\mathbf{new^{c}}] \qquad \frac{var \in \mathrm{dom}(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} var \Rightarrow var : \Gamma(var)}[\mathbf{var}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Rightarrow \mathsf{null} : t}[\mathbf{null}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad \langle c.fd,\ t\rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.fd \Rightarrow e' \underline{\ :\ c\ }.fd : t}[\mathbf{get^{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.fd,\ t\rangle \in_P t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.fd = e_v \Rightarrow e' \underline{\ :\ c\ }.fd = e'_v : t}[\mathbf{set^{c}}]$$

Fig. 6. Context-sensitive checks and type elaboration rules for ClassicJava (Part I)

$$\frac{\begin{array}{l} P,\Gamma \vdash_{e} e \Rightarrow e' : t' \quad \langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e_b \rangle \in_P t' \\ P,\Gamma \vdash_{s} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \end{array}}{P,\Gamma \vdash_{e} e.md\ (e_1 \ldots e_n) \Rightarrow e'.md\ (e'_1 \ldots e'_n) : t} [\mathbf{call}^c]$$

$$\frac{\begin{array}{l} P,\Gamma \vdash_{e} \mathbf{this} \Rightarrow \mathbf{this} : c' \quad c' \prec^c_P c \quad \langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e_b \rangle \in_P c \\ P,\Gamma \vdash_{s} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \quad e_b \neq \mathbf{abstract} \end{array}}{P,\Gamma \vdash_{e} \mathbf{super}.md(e_1 \ldots e_n) \Rightarrow \mathbf{super}\underline{\equiv \mathbf{this} : c}.md(e'_1 \ldots e'_n) : t} [\mathbf{super}^c]$$

$$\frac{P,\Gamma \vdash_{s} e \Rightarrow e' : t}{P,\Gamma \vdash_{e} \mathbf{view}\ t\ e \Rightarrow e' : t} [\mathbf{wcast}^c] \qquad \frac{P \vdash_{t} t}{P,\Gamma \vdash_{e} \mathbf{abstract} \Rightarrow \mathbf{abstract} : t} [\mathbf{abs}]$$

$$\frac{P,\Gamma \vdash_{e} e \Rightarrow e' : t' \quad t \leq_P t' \text{ or } t \in \mathrm{dom}(\prec^i_P) \text{ or } t' \in \mathrm{dom}(\prec^i_P)}{P,\Gamma \vdash_{e} \mathbf{view}\ t\ e \Rightarrow \mathbf{view}\ t\ e' : t} [\mathbf{ncast}^c]$$

$$\frac{P,\Gamma \vdash_{e} e_1 \Rightarrow e'_1 : t_1 \quad P,\Gamma[var : t_1] \vdash_{e} e_2 \Rightarrow e'_2 : t}{P,\Gamma \vdash_{e} \mathbf{let}\ var = e_1\ \mathbf{in}\ e_2 \Rightarrow \mathbf{let}\ var = e'_1\ \mathbf{in}\ e'_2 : t} [\mathbf{let}]$$

$\vdash_s, \vdash_t$

$$\frac{P,\Gamma \vdash_{e} e \Rightarrow e' : t' \quad t' \leq_P t}{P,\Gamma \vdash_{s} e \Rightarrow e' : t} [\mathbf{sub}^c] \qquad \frac{t \in \mathrm{dom}(\prec^c_P) \cup \mathrm{dom}(\prec^i_P) \cup \{\mathsf{Object}, \mathsf{Empty}\}}{P \vdash_{t} t} [\mathbf{type}^c]$$

Fig. 7. Context-sensitive checks and type elaboration rules for ClassicJava (Part II)

The type elaboration rules translate expressions that access a field or call a **super** method into annotated expressions (see the underlined parts of Figure 3). For field uses, the annotated expression contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. For **super** method invocations, the annotated expression contains the compile-time type of **this**, which determines the class that contains the declaration of the method to be invoked.

The complete typing rules are shown in Figures 6 and 7. A program is well-typed if its class definitions and final expression are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types. For example, the $\mathbf{get}^c$ and $\mathbf{set}^c$ rules for fields first determine the type of the instance expression, and then calculate a class-tagged field name using $\in_P$; this yields both the type of the field and the class for the installed annotation. In the $\mathbf{set}^c$ rule, the right-hand side of the assignment must match the type of the field, but this match may exploit subsumption to coerce the type of the value to a supertype. The other expression typing rules are similarly intuitive.

2.3 ClassicJava Evaluation

The operational semantics for ClassicJava is defined as a contextual rewriting system on pairs of expressions and stores. A store $\mathcal{S}$ is a mapping from *objects* to class-tagged field records. A field record $\mathcal{F}$ is a mapping from elaborated field

$$e = \ldots \mid object \qquad v = object \mid \mathsf{null}$$

$$\begin{aligned} \mathsf{E} = {} & [\,] \mid \mathsf{E}\underline{\,:c\,}.fd \mid \mathsf{E}\underline{\,:c\,}.fd = e \mid v\underline{\,:c\,}.fd = \mathsf{E} \\ \mid {} & \mathsf{E}.md(e \ldots) \mid v.md(v \ldots \mathsf{E}\ e \ldots) \\ \mid {} & \textbf{super}\,\underline{\equiv v : c}\,.md(v \ldots \mathsf{E}\ e \ldots) \\ \mid {} & \textbf{view}\ t\ \mathsf{E} \mid \textbf{let}\ var = \mathsf{E}\ \textbf{in}\ e \end{aligned}$$

$$\begin{array}{lr}
P \vdash \langle \mathsf{E}[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[object], \mathcal{S}[object \mapsto \langle c, \mathcal{F}\rangle]\rangle & [new] \\
\quad \text{where } object \notin \operatorname{dom}(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.fd \mapsto \mathsf{null} \mid c \leq^{c}_{P} c' \text{ and } \exists t \text{ s.t. } \langle c'.fd, t\rangle \in^{c}_{P} c'\} & \\
P \vdash \langle \mathsf{E}[object\underline{\,:c'\,}.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle & [get] \\
\quad \text{where } \mathcal{S}(object) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(c'.fd) = v & \\
P \vdash \langle \mathsf{E}[object\underline{\,:c'\,}.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[object \mapsto \langle c, \mathcal{F}[c'.fd \mapsto v]\rangle]\rangle & [set] \\
\quad \text{where } \mathcal{S}(object) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle \mathsf{E}[object.md(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[object/\textbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S}\rangle & [call] \\
\quad \text{where } \mathcal{S}(object) = \langle c, \mathcal{F}\rangle \text{ and } \langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \in^{c}_{P} c & \\
P \vdash \langle \mathsf{E}[\textbf{super}\,\underline{\equiv object : c'}\,.md(v_1, \ldots v_n)], \mathcal{S}\rangle & [super] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\overline{object/\textbf{this}}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S}\rangle & \\
\quad \text{where } \langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \in^{c}_{P} c' & \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[object], \mathcal{S}\rangle & [cast] \\
\quad \text{where } \mathcal{S}(object) = \langle c, \mathcal{F}\rangle \text{ and } c \leq_P t' & \\
P \vdash \langle \mathsf{E}[\textbf{let}\ var = v\ \textbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v/var]], \mathcal{S}\rangle & [let] \\
& \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle & [xcast] \\
\quad \text{where } \mathcal{S}(object) = \langle c, \mathcal{F}\rangle \text{ and } c \not\leq_P t' & \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathsf{null}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle & [ncast] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\,:c\,}.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [nget] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\,:c\,}.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [nset] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.md(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [ncall]
\end{array}$$

Fig. 8. Operational semantics for ClassicJava

names to values. The evaluation rules are a straightforward modification of those for imperative Scheme [14].

The complete evaluation rules are in Figure 8. For example, the *call* rule invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the *super* reduction performs **super** method selection using the class annotation that is statically determined by the type-checker.

2.4 ClassicJava Soundness

For a program of type t, the evaluation rules for ClassicJava produce either a value that has a subtype of t, or one of two errors. Put differently, an evaluation cannot go wrong, which our model model means getting stuck. This property can be formulated as a type soundness theorem.

Theorem 1 (Type Soundness). *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ldots defn_n\ e$, *then either*

- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle object, \mathcal{S}\rangle$ *and* $\mathcal{S}(object) = \langle t', \mathcal{F}\rangle$ *and* $t' \leq_P t$*; or*
- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S}\rangle$*; or*

– $P' \vdash \langle e,\ \emptyset\rangle \hookrightarrow^* \langle \textsf{error: bad cast},\ \mathcal{S}\rangle$; *or*
– $P' \vdash \langle e,\ \emptyset\rangle \hookrightarrow^* \langle \textsf{error: dereferenced null},\ \mathcal{S}\rangle$.

The main lemma in support of this theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair (relative to the program) [30]. Specifically, for a configuration on the left-hand side of an evaluation step, there exists a type environment that establishes the expression's type as some t. This environment must be consistent with the store.

Definition 2 (Environment-Store Consistency).

$$
\begin{array}{ll}
 & P,\Gamma \vdash_{\sigma} \mathcal{S} \\
 & \quad \Leftrightarrow (\mathcal{S}(object) = \langle c,\mathcal{F}\rangle \\
\Sigma_1: & \qquad \Rightarrow \Gamma(object) = c \\
\Sigma_2: & \qquad\quad \text{and } \mathrm{dom}(\mathcal{F}) = \{c_1.fd \mid \langle c_1.fd, c_2\rangle \in^{\mathsf{c}}_P c\} \\
\Sigma_3: & \qquad\quad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4: & \qquad\quad \text{and } (\mathcal{F}(c_1.fd) = object' \text{ and } \langle c_1.fd, c_2\rangle \in^{\mathsf{c}}_P c) \\
 & \qquad\qquad \Rightarrow ((\mathcal{S}(object') = \langle c',\mathcal{F}'\rangle) \Rightarrow c' \leq_P c_2)) \\
\Sigma_5: & \quad \text{and } object \in \mathrm{dom}(\Gamma) \Rightarrow object \in \mathrm{dom}(\mathcal{S})^3 \\
\Sigma_6: & \quad \text{and } \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)
\end{array}
$$

Since the rewriting rules reduce *annotated* terms, we derive new type judgements $\vdash_{\underline{\mathsf{s}}}$ and $\vdash_{\underline{\mathsf{e}}}$ that relate annotated terms to show that reductions preserve type correctness. Each of the new rules performs the same checks as the rule it is derived from without removing or adding annotation. Thus, $\vdash_{\underline{\mathsf{s}}}$ is derived from $\vdash_{\mathsf{s}}$, and so forth.

The judgement on **view** expressions is altered slightly; we retain the **view** operation in all cases, and we collapse the [**wcast**$^{\mathsf{c}}$] and [**ncast**$^{\mathsf{c}}$] relations to a new [**cast**$^{\mathsf{c}}$] relation that permits any casting operation:

$$\frac{P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'}{P,\Gamma \vdash_{\underline{\mathsf{e}}} \textbf{view}\ t\ e : t}[\textbf{cast}^{\mathsf{c}}]$$

The new [**cast**$^{\mathsf{c}}$] relation lets us prove that every intermediate expresion in a reduction is well-typed, whereas [**wcast**$^{\mathsf{c}}$] and [**ncast**$^{\mathsf{c}}$] more closely approximate Java, which rejects certain expressions because they would certainly produce error: bad cast. For example, assuming that $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$ extend $\mathsf{Door}^{\mathsf{c}}$ separately, a legal source program

$$\textbf{let}\ x = o.GetDoor()\ \textbf{in}\ (\textbf{view}\ \mathsf{LockedDoor}^{\mathsf{c}}\ x)$$

might reduce to

$$\textbf{view}\ \mathsf{LockedDoor}^{\mathsf{c}}\ shortDoorObject$$

where *shortDoorObject* is an instance of $\mathsf{ShortDoor}^{\mathsf{c}}$. Unlike the [**wcast**$^{\mathsf{c}}$] and [**ncast**$^{\mathsf{c}}$] rules in $\vdash_{\mathsf{e}}$, the [**cast**$^{\mathsf{c}}$] rule in $\vdash_{\underline{\mathsf{e}}}$ assigns a type to the reduced expression.

[3] In Σ_5, it would be wrong to write $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\mathcal{S})$ because Γ may contain bindings for lexical variables.

The $\vdash_{\underline{e}}$ relation also generalizes the [**super**^{c}] judgement to allow an arbitrary expression within a **super** expression's annotation (in place of **this**). The generalized judgement permits replacement and substitution lemmas that treat **super** annotations in the same manner as other expression. Nevertheless, at each reduction step, every **super** expression's annotation contains either **this** or an object. This fact is crucial to proving the soundness of ClassicJava, so we formalize it as a SuperOk predicate.

Definition 3 (Well-Formed Super Calls).

$$\textsc{SuperOk}(e) \Leftrightarrow \text{For all } \mathbf{super}\ \underline{\equiv e_0 : c}\ .md(e_1, \ldots\ e_n) \text{ in } e,$$
$$\text{either } e_0 = \mathbf{this} \text{ or } \exists\ object \text{ s.t. } e_0 = object.$$

Although $\vdash_{\underline{e}}$ types more expressions than $\vdash_{e}$, we are only concerned with source expressions typed by $\vdash_{e}$. The following lemma establishes that the new typing judgements conserve the result of the original typing judgements.

Lemma 4 (Conserve). *If* $\vdash_{p} P \Rightarrow P' : t$ *and* $P' = defn_1\ \ldots\ defn_n\ e$*, then* $P', \emptyset \vdash_{\underline{e}} e' : t$.

Proof. The claim follows from a copy-and-patch argument. □

Lemma 5 (Subject Reduction). *If* $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, SuperOk$(e)$, *and* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*, then* e' *is an* error *configuration or* $\exists\ \Gamma'$ *such that*

1. $P,\Gamma' \vdash_{\underline{s}} e' : t$,
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$, *and*
3. SuperOk(e').

Proof. The proof examines reduction steps. For each case, if execution has not halted with an error configuration, we construct the new environment Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment. See Appendix A for the complete proof. □

Lemma 6 (Progress). *If* $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, *and* SuperOk$(e)$*, then either* e *is a value or there exists an* $\langle e',\mathcal{S}'\rangle$ *such that* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$.

Proof. The proof is by analysis of the possible cases for the current redex in e (in the case that e is not a value). See Appendix A for the complete proof. □

By combinining the Subject Reduction and Progress lemmas, we can prove that every non-value ClassicJava program reduces while preserving its type, thus estrablishing the soundness of ClassicJava.

2.5 Related Work on Classes

Our model for class-based object-oriented languages is similar to two recently published semantics for Java [10, 29], but entirely motivated by prior work on Scheme and ML models [14, 19, 30]. The approach is fundamentally different from most of the previous work on the semantics of objects. Much of that work has focused on interpreting object systems and the underlying mechanisms via record extensions of lambda calculi [12, 20, 25, 23, 26] or as "native" object calculi (with a record flavor) [1–3]. In our semantics, types are simply the names of entities declared in the program; the collection of types forms a DAG, which is specified by the programmer. The collection of types is static during evaluation[4] and is only used for field and method lookups and casts. The evaluation rules describe how to transform statements, formed over the given type context, into plain values. The rules work on plain program text such that each intermediate stage of the evaluation is a complete program. In short, the model is as simple and intuitive as that of first-order functional programming enriched with a language for expressing hierarchical relationships among data types.

3 From Classes to Mixins: An Example

Implementing a maze adventure game [17, page 81] illustrates the need for adding mixins to a class-based language. A player in the adventure game wanders through rooms and doors in a virtual world. All locations in the virtual world share some common behavior, but also differ in a wide variety of properties that make the game interesting. For example, there are many kinds of doors, including locked doors, magic doors, doors of varying heights, and doors that combine several varieties into one. The natural class-based approach for implementing different kinds of doors is to implement each variation with a new subclass of a basic door class, Door^c. The left side of Figure 9 shows the Java definition for two simple Door^c subclasses, $\mathsf{LockedDoor}^c$ and $\mathsf{ShortDoor}^c$. An instance of $\mathsf{LockedDoor}^c$ requires a key to open the door, while an instance of $\mathsf{ShortDoor}^c$ requires the player to duck before walking through the door.

A subclassing approach to the implementation of doors seems natural at first because the programmer declares only what is different in a particular door variation as compared to some other door variation. Unfortunately, since the superclass of each variation is fixed, door variations cannot be composed into more complex, and thus more interesting, variations. For example, the $\mathsf{LockedDoor}^c$ and $\mathsf{ShortDoor}^c$ classes cannot be combined to create a new $\mathsf{LockedShortDoor}^c$ class for doors that are both locked and short.

A mixin approach solves this problem. Using mixins, the programmer declares how a particular door variation differs from an *arbitrary* door variation. This creates a function from door classes to door classes, using an interface as the

[4] Dynamic class loading could be expressed in this framework as an addition to the static context. Still, the context remains the same for most of the evaluation.

```
class LockedDoor^c extends Door^c {
  boolean canOpen(Person^c p) {
    if (!p.hasItem(theKey)) {
      System.out.println("You don't have the Key");
      return false;
    }
    System.out.println("Using key...");
    return super.canOpen(p);
  }
}
class ShortDoor^c extends Door^c {
  boolean canPass(Person^c p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}

/* Cannot merge for LockedShortDoor^c */
```

```
interface Door^i {
  boolean canOpen(Person^c p);
  boolean canPass(Person^c p);
}
mixin Locked^m extends Door^i {
  boolean canOpen(Person^c p) {
    if (!p.hasItem(theKey)) {
      System.out.println("You don't have the Key");
      return false;
    }
    System.out.println("Using key...");
    return super.canOpen(p);
  }
}
mixin Short^m extends Door^i {
  boolean canPass(Person^c p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}
class LockedDoor^c = Locked^m(Door^c);
class ShortDoor^c = Short^m(Door^c);
class LockedShortDoor^c = Locked^m(Short^m(Door^c));
```

Fig. 9. Some class definitions and their translation to composable mixins

input type. Each basic door variation is defined as a separate mixin. These mixins are then functionally composed to create many different kinds of doors.

A programmer implements mixins in exactly the same way as a derived class, except that the programmer cannot rely on the *implementation* of the mixin's superclass, only on its *interface*. We consider this an advantage of mixins because it enforces the maxim "program to an interface, not an implementation" [17, page 11].

The right side of Figure 9 shows how to define mixins for locked and short doors. The mixin $\mathsf{Locked}^{\mathsf{m}}$ is nearly identical to the original $\mathsf{LockedDoor}^{\mathsf{c}}$ class definition, except that the superclass is specified via the interface $\mathsf{Door}^{\mathsf{i}}$. The new $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$ classes are created by applying $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Short}^{\mathsf{m}}$ to the class $\mathsf{Door}^{\mathsf{c}}$, respectively. Similarly, applying $\mathsf{Locked}^{\mathsf{m}}$ to $\mathsf{ShortDoor}^{\mathsf{c}}$ yields a class for locked, short doors.

Consider another door variation: $\mathsf{MagicDoor}^{\mathsf{c}}$, which is similar to $\mathsf{LockedDoor}^{\mathsf{c}}$ except the player needs a book of spells instead of a key. We can extract the common parts of the implementation of $\mathsf{MagicDoor}^{\mathsf{c}}$ and $\mathsf{LockedDoor}^{\mathsf{c}}$ into a new mixin, $\mathsf{Secure}^{\mathsf{m}}$. Then, key- or book-specific information is composed with $\mathsf{Secure}^{\mathsf{m}}$ to produce $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$, as shown in Figure 10. Each of the new mixins extends $\mathsf{Door}^{\mathsf{i}}$ since the right hand mixin in the composition, $\mathsf{Secure}^{\mathsf{m}}$, extends $\mathsf{Door}^{\mathsf{i}}$.

The $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$ mixins can also be composed to form $\mathsf{LockedMagic}^{\mathsf{m}}$. This mixin has the expected behavior: to open an instance of $\mathsf{LockedMagic}^{\mathsf{m}}$, the player must have both the key and the book of spells. This combinational effect

```
interface Secure^i extends Door^i {
  Object neededItem();
}
mixin Secure^m extends Door^i implements Secure^i {
   Object neededItem() { return null; }
   boolean canOpen(Person^c p) {
      Object item = neededItem();
      if (!p.hasItem(item)) {
        System.out.println("You don't have the " + item);
        return false;
      }
      System.out.println("Using " + item + "...");
      return super.canOpen(p);
   }
}
mixin NeedsKey^m extends Secure^i {
   Object neededItem() {
      return theKey;
   }
}
mixin NeedsSpell^m extends Secure^i {
   Object neededItem() {
      return theSpellBook;
   }
}
mixin Locked^m = NeedsKey^m compose Secure^m;
mixin Magic^m = NeedsSpell^m compose Secure^m;
mixin LockedMagic^m = Locked^m compose Magic^m;
mixin LockedMagicDoor^m = LockedMagic^m compose Door^m;
class LockedDoor^c = Locked^m(Door^c); ...
```

Fig. 10. Composing mixins for localized parameterization

is achieved by a chain of **super**.*canOpen*() calls that use distinct, non-interfering versions of *neededItem*. The *neededItem* declarations of $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$ do not interfere with each other because the interface extended by $\mathsf{Locked}^{\mathsf{m}}$ is $\mathsf{Door}^{\mathsf{i}}$, which does not contain *neededItem*. In contrast, $\mathsf{Door}^{\mathsf{i}}$ does contain *canOpen*, so the *canOpen* method in $\mathsf{Locked}^{\mathsf{m}}$ overrides and chains to the *canOpen* in $\mathsf{Magic}^{\mathsf{m}}$.

4 Mixins for Java

MixedJava is an extension of ClassicJava with mixins. In ClassicJava, a class is assembled as a chain of **class** expressions. Specifically, the content of a class is defined by its immediate field and method declarations *and* by the declarations of its superclasses, up to Object.[5] In MixedJava, a "class" is assembled by composing a chain of mixins. The content of the class is defined by the field and method declarations in the entire chain.

MixedJava provides two kinds of mixins:

[5] We use boldfaced **class** to refer to the content of a single **class** expression, as opposed to an actual class.

– An *atomic* mixin declaration is similar to a **class** declaration. An atomic mixin declares a set of fields and methods that are extensions to some inherited set of fields and methods. In contrast to a class, an atomic mixin specifies its inheritance with an *inheritance interface*, not a static connection to an existing class. By abuse of terminology, we say that a mixin *extends* its inheritance interface.

  A mixin's inheritance interface determines how method declarations within the mixin are combined with inherited methods. If a mixin declares a method x that is not contained in its inheritance interface, then that declaration never overrides another x.

  An atomic mixin *implements* one or more interfaces as specified in the mixin's definition. In addition, a mixin always implements its inheritance interface.
– A *composite* mixin does not declare any new fields or methods. Instead, it composes two existing mixins to create a new mixin. The new composite mixin has all of the fields and methods of its two constituent mixins. Method declarations in the left-hand mixin override declarations in the right-hand mixin according to the left-hand mixin's inheritance interface. Composition is allowed only when the right-hand mixin implements the left-hand mixin's inheritance interface.

  A composite mixin extends the inheritance interface of its right-hand constituent, and it implements all of the interfaces that are implemented by its constituents. Composite mixins can be composed with other mixins, producing arbitrarily long chains of atomic mixin compositions.[6]

Figure 11 illustrates how the mixin $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ from the previous section corresponds to a chain of atomic mixins. The arrows connecting the tops of the boxes represent mixin compositions; in each composition, the inheritance interface for the left-hand side is noted above the arrow. The other arrows show how method declarations in each mixin override declarations in other mixins according to the composition interfaces. For example, there is no arrow from the first $\mathsf{Secure}^{\mathsf{m}}$'s *neededItem* to $\mathsf{Magic}^{\mathsf{m}}$'s method because *neededItem* is not included in the $\mathsf{Door}^{\mathsf{i}}$ interface. The *canOpen* method is in both $\mathsf{Door}^{\mathsf{i}}$ and $\mathsf{Secure}^{\mathsf{i}}$, so that corresponding arrows connect all declarations of *canOpen*.

Mixins completely subsume the role of classes. A mixin can be instantiated with **new** when the mixin does not inherit any services. In MixedJava, this is indicated by declaring that the mixin extends the special interface Empty. Consequently, we omit classes from our model of mixins, even though a realistic language would include both mixins and classes.

[6] Our composition operator is associative semantically, but not type-theoretically. The type system could be strengthened to make composition associative—giving MixedJava a categorical flavor—by letting each mixin declare a set of interfaces for inheritance, rather than a single interface. Each required interface must then either be satisfied or propagated by a composition. We have not encountered a practical use for the extended type system.

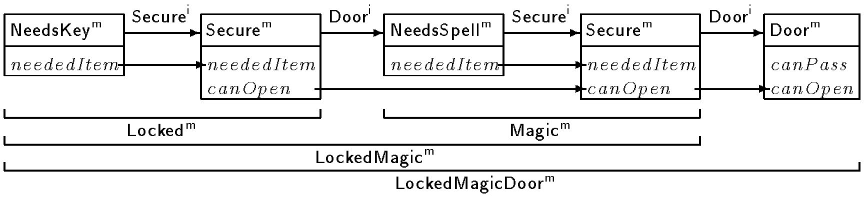

Fig. 11. The LockedMagicDoor^m mixin corresponds to a sequence of atomic mixins

The following subsections present a precise description of MixedJava. Section 4.1 describes the syntax and type structure of MixedJava programs, followed by the type elaboration rules in Section 4.2. Section 4.3 explains the operational semantics of MixedJava, which is significantly different from that of ClassicJava. Section 4.4 presents a type soundness theorem, Section 4.5 briefly considers implementation issues, and Section 4.6 discusses related work.

4.1 MixedJava Programs

Figure 12 contains the syntax for MixedJava; the missing productions are inherited from the grammar of ClassicJava in Figure 3. The primary change to the syntax is the replacement of **class** declarations with **mixin** declarations. Another change concerns the annotations added by type elaboration. First, **view** expressions are annotated with the syntactic type of the object expression. Second, a type is no longer included in the **super** annotation or the field use annotations. In addition, type elaboration inserts extra **view** expressions into a program to implement subsumption.

$$
\begin{array}{rcl}
\mathit{defn} & = & \textbf{mixin}\ m\ \textbf{extends}\ i\ \textbf{implements}\ i^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
 & & \mid\ \textbf{mixin}\ m\ \textbf{=}\ m\ \textbf{compose}\ m \\
 & & \mid\ \textbf{interface}\ i\ \textbf{extends}\ i^*\ \{\ \mathit{meth}^*\ \} \\
e & = & \textbf{new}\ m \mid \mathit{var} \mid \mathsf{null} \mid e.\mathit{fd} \mid e.\mathit{fd}\ \textbf{=}\ e \\
 & & \mid\ e.\mathit{md}\ (e^*) \mid \textbf{super}\underline{\ \equiv \textbf{this}\ }.\mathit{md}\ (e^*) \\
 & & \mid\ \textbf{view}\underline{\ t\ \textbf{as}\ }\,t\ e \mid \textbf{let}\ \mathit{var}\ \textbf{=}\ e\ \textbf{in}\ e \\
m & = & \text{mixin name} \\
t & = & m \mid i
\end{array}
$$

Fig. 12. Syntax extensions for MixedJava

The predicates and relations in Figures 13 and 14 (along with the interface-specific parts of Figures 4 and 5) summarize the syntactic content of a MixedJava program. A well-formed program induces a subtype relation $\leq_P^{\mathsf{m}}$ on its mixins such that a composite mixin is a subtype of each of its constituent mixins.

MixinsOnce(P) Each mixin name is declared only once

$$\textbf{mixin}\ m \cdots \textbf{mixin}\ m' \cdots \text{ is in } P \Longrightarrow m \neq m'$$

FieldOncePerMixin(P) Field names in each mixin declaration are unique

$$\textbf{mixin} \cdots \{ \cdots fd \cdots fd' \cdots \} \text{ is in } P \Longrightarrow fd \neq fd'$$

MethodOncePerMixin(P) Method names in each mixin declaration are unique

$$\textbf{mixin} \cdots \{ \cdots md\ (\cdots)\ \{ \cdots \} \cdots md'\ (\cdots)\ \{ \cdots \} \cdots \} \text{ is in } P \Longrightarrow md \neq md'$$

NoAbstractMixins(P) Methods in a mixin are not **abstract**

$$\textbf{mixin} \cdots \{ \cdots md\ (\cdots)\ \{\ e\ \} \cdots \} \text{ is in } P \Longrightarrow e \neq \textbf{abstract}$$

$\prec^{\mathsf{m}}_{P}$ Mixin declares an inheritance interface

$$m \prec^{\mathsf{m}}_{P} i \Leftrightarrow \textbf{mixin}\ m\ \textbf{extends}\ i \cdots \{ \cdots \} \text{ is in } P$$

$\prec\!\!\prec^{\mathsf{m}}_{P}$ Mixin declares implementation of an interface

$$m \prec\!\!\prec^{\mathsf{m}}_{P} i \Leftrightarrow \textbf{mixin}\ m \cdots \textbf{implements} \cdots i \cdots \{ \cdots \} \text{ is in } P$$

$\bullet \doteq^{\mathsf{m}}_{P} \bullet \circ \bullet$ Mixin is declared as a composition

$$m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Leftrightarrow \textbf{mixin}\ m\ \mathbf{=}\ m'\ \textbf{compose}\ m'' \text{ is in } P$$

$\Subset^{\mathsf{m}}_{P}$ Method is declared in a mixin

$$\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1\ \ldots\ var_n),\ e\rangle \Subset^{\mathsf{m}}_{P} m$$
$$\Leftrightarrow \textbf{mixin}\ m \cdots \{ \cdots t\ md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{\ e\ \} \cdots \} \text{ is in } P$$

$\Subset^{\mathsf{m}}_{P}$ Field is declared in a mixin

$$\langle m.fd,\ t\rangle \Subset^{\mathsf{m}}_{P} m \Leftrightarrow \textbf{mixin}\ m \cdots \{ \cdots t\ fd \cdots \} \text{ is in } P$$

$\leq^{\mathsf{m}}_{P}$ Mixin is a submixin

$$m \leq^{\mathsf{m}}_{P} m' \Leftrightarrow m = m' \text{ or } (\exists m'', m''' \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m'' \circ m''' \text{ and } (m'' \leq^{\mathsf{m}}_{P} m' \text{ or } m''' \leq^{\mathsf{m}}_{P} m'))$$

$\unlhd^{\mathsf{m}}_{P}$ Mixin is viewable as a mixin

$$m \unlhd^{\mathsf{m}}_{P} m' \Leftrightarrow m = m' \text{ or } (\exists m'', m''' \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m'' \circ m''' \text{ and } (m'' \leq^{\mathsf{m}}_{P} m' \text{ xor } m''' \leq^{\mathsf{m}}_{P} m')$$
$$\text{and } (m'' \unlhd^{\mathsf{m}}_{P} m' \text{ xor } m''' \unlhd^{\mathsf{m}}_{P} m'))$$

CompleteMixins(P) Mixins that are composed are defined

$$\mathrm{rng}(\doteq^{\mathsf{m}}_{P}) \subseteq \{m \circ m' \mid m, m' \in \mathrm{dom}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_{P})\}$$

WellFoundedMixins(P) Mixin hierarchy is an order

$$\leq^{\mathsf{m}}_{P} \text{ is antisymmetric}$$

CompleteInterfaces(P) Extended/implemented interfaces are defined

$$\mathrm{rng}(\prec^{\mathsf{i}}_{P}) \cup \mathrm{rng}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{rng}(\prec\!\!\prec^{\mathsf{m}}_{P}) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_{P}) \cup \{\mathsf{Empty}\}$$

$\prec\!\!\mid^{\mathsf{m}}_{P}$ Mixin extends an interface

$$m \prec\!\!\mid^{\mathsf{m}}_{P} i \Leftrightarrow m \prec^{\mathsf{m}}_{P} i \text{ or } (\exists m', m'' \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m' \circ m'' \text{ and } m'' \prec\!\!\mid^{\mathsf{m}}_{P} i)$$

$\ll^{\mathsf{m}}_{P}$ Mixin implements an interface

$$m \ll^{\mathsf{m}}_{P} i \Leftrightarrow \exists m', i' \text{ s.t. } m \leq^{\mathsf{m}}_{P} m' \text{ and } i' \leq^{\mathsf{i}}_{P} i \text{ and } (m' \prec^{\mathsf{m}}_{P} i' \text{ or } m' \prec\!\!\prec^{\mathsf{m}}_{P} i')$$

$\lhd\!\!\lhd^{\mathsf{m}}_{P}$ Mixin is viewable as an interface

$$m \lhd\!\!\lhd^{\mathsf{m}}_{P} i \Leftrightarrow (\exists i' \text{ s.t. } i' \leq^{\mathsf{i}}_{P} i \text{ and } (m \prec^{\mathsf{m}}_{P} i' \text{ or } m \prec\!\!\prec^{\mathsf{m}}_{P} i'))$$
$$\text{or } (\exists m', m'' \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m' \circ m'' \text{ and } (m' \ll^{\mathsf{m}}_{P} i \text{ xor } m'' \ll^{\mathsf{m}}_{P} i)$$
$$\text{and } (m' \lhd\!\!\lhd^{\mathsf{m}}_{P} i \text{ xor } m'' \lhd\!\!\lhd^{\mathsf{m}}_{P} i))$$

MixinCompositionsOK(P) Mixins are composed safely

$$m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Longrightarrow \exists i \text{ s.t. } m' \prec\!\!\mid^{\mathsf{m}}_{P} i \text{ and } m'' \lhd\!\!\lhd^{\mathsf{m}}_{P} i$$

MixinMethodsOK(P) Method definitions match inheritance interface

$$(\langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_{P} m \text{ and } \langle md,\ T',\ V',\ \textbf{abstract}\rangle \Subset^{\mathsf{m}}_{P} i) \Longrightarrow (T = T' \text{ or } m \not\prec\!\!\mid^{\mathsf{m}}_{P} i)$$

$\in^{\mathsf{m}}_{P}$ Field is contained and visible in a mixin

$$\langle m'.fd,\ t\rangle \in^{\mathsf{m}}_{P} m \Leftrightarrow m \unlhd^{\mathsf{m}}_{P} m'$$
$$\text{and } \{\langle m'.fd,\ t\rangle\} = \{\langle m'.fd,\ t\rangle \mid m \leq^{\mathsf{m}}_{P} m' \text{ and } \langle m'.fd,\ t\rangle \Subset^{\mathsf{m}}_{P} m'\}$$

$\underline{\in}^{\mathsf{m}}_{P}$ Method is potentially visible in a mixin (used for $\in^{\mathsf{m}}_{P}$)

$$\langle md,\ T\rangle \underline{\in}^{\mathsf{m}}_{P} m \Leftrightarrow (\exists V, e \text{ s.t. } \langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_{P} m)$$
$$\text{or } (\exists i, V \text{ s.t. } m \prec^{\mathsf{m}}_{P} i \text{ and } \langle md,\ T,\ V,\ \textbf{abstract}\rangle \in^{\mathsf{i}}_{P} i)$$
$$\text{or } (\exists m', m'' \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m' \circ m''$$
$$\text{and } (\langle md,\ T\rangle \underline{\in}^{\mathsf{m}}_{P} m' \text{ or } \langle md,\ T\rangle \underline{\in}^{\mathsf{m}}_{P} m''))$$

$\in^{\mathsf{m}}_{P}$ Method is visible in a mixin

$$\langle md,\ T\rangle \in^{\mathsf{m}}_{P} m \Leftrightarrow \langle md,\ T\rangle \underline{\in}^{\mathsf{m}}_{P} m$$
$$\text{and } ((\exists i \text{ s.t. } m \prec^{\mathsf{m}}_{P} i)$$
$$\text{or } (\exists m', m'', i \text{ s.t. } m \doteq^{\mathsf{m}}_{P} m' \circ m'' \text{ and } m' \prec\!\!\mid^{\mathsf{m}}_{P} i$$
$$\text{and } (\langle md,\ T'\rangle \underline{\in}^{\mathsf{m}}_{P} m' \Rightarrow \langle md,\ T'\rangle \in^{\mathsf{m}}_{P} m')$$
$$\text{and } (\langle md,\ T''\rangle \underline{\in}^{\mathsf{m}}_{P} m'' \Rightarrow \langle md,\ T''\rangle \in^{\mathsf{m}}_{P} m'')$$
$$\text{and } (((\langle md,\ T'\rangle \in^{\mathsf{m}}_{P} m' \text{ and } \langle md,\ T''\rangle \in^{\mathsf{m}}_{P} m'')$$
$$\Rightarrow (\exists V \text{ s.t. } \langle md,\ T,\ V,\ \textbf{abstract}\rangle \in^{\mathsf{i}}_{P} i))))$$

Fig. 13. Predicates and relations in the model of MixedJava (Part I)

$\text{MixinsImplementAll}(P)$ Mixins supply methods to implement interfaces

$$m \ll^{\mathsf{m}}_P i \implies (\forall md, T\ \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i$$
$$\implies (\exists\ e \text{ s.t. } \langle md,\ T,\ V,\ e\rangle \in\!\!\!\in^{\mathsf{m}}_P m$$
$$\text{or } \exists\ i' \ s.t.\ (m \lhd\!\!\prec^{\mathsf{m}}_P i'$$
$$\text{and } \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i')))$$

$\in^{\mathsf{i}}_P$ Method with type in an interface

$$\langle md,\ T\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists V \text{ s.t. } \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i$$

$\leq_P$ Type is a subtype $\quad \leq_P\ \equiv\ \leq^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{m}}_P$

$\unlhd_P$ Type is viewable as another type $\quad \unlhd_P\ \equiv\ \unlhd^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \lhd\!\!\lhd^{\mathsf{m}}_P$

$\in_P$ Field or method is in a type $\quad \in_P\ \equiv\ \in^{\mathsf{m}}_P \cup \in^{\mathsf{i}}_P$

$::$ and @ Chain constructors

$::$ adds an element to the beginning of a chain; @ appends two chains

$\longrightarrow_P$ Mixin corresponds to a chain of atomic mixins

$$m \longrightarrow_P M$$
$$\Leftrightarrow (\exists i \text{ s.t. } m \prec^{\mathsf{m}}_P i \text{ and } M = [m])$$
$$\text{or } (\exists m', m'', M', M'' \text{ s.t. } m \doteq^{\mathsf{m}}_P m' \circ m'' \text{ and } m' \longrightarrow_P M'$$
$$\text{and } m'' \longrightarrow_P M'' \text{ and } M = M'@M'')$$

$\leq^{\mathsf{M}}$ Chains have an inverted subsequence order

$$M \leq^{\mathsf{M}} M' \Leftrightarrow \exists M'' \text{ s.t. } M = M''@M'$$

$\bullet/\bullet \triangleright \bullet/\bullet$ Mixin view operation selects a new chain

$$M/m \triangleright M'/m' \Leftrightarrow (m = m' \text{ and } M = M')$$
$$\text{or } (\exists m'', m''' \text{ s.t. } m \doteq^{\mathsf{m}}_P m'' \circ m'''$$
$$\text{and } ((m'' \leq^{\mathsf{m}}_P m' \text{ and } M/m'' \triangleright M'/m')$$
$$\text{or } (m''' \leq^{\mathsf{m}}_P m' \text{ and } \exists M_l, M_r \text{ s.t. } m'' \longrightarrow_P M_l \text{ and } M = M_l@M_r$$
$$\text{and } M_r/m''' \triangleright M'/m'))$$

$\bullet/\bullet \triangleright \bullet/\bullet$ Interface view operation selects a new chain

$$M/t \triangleright M'/i \Leftrightarrow M' = \min\{m{::}M'' \mid m \ll^{\mathsf{m}}_P i \text{ and } M \leq^{\mathsf{M}} m{::}M''\}$$

$\bullet.\bullet \propto \bullet.\bullet$ Method in a chain is the same as in another chain

$$m{::}M.md \propto M'.md \Leftrightarrow m{::}M = M' \text{ or } (\exists i, T, V, M'' \text{ s.t. } m \prec^{\mathsf{m}}_P i \text{ and } \langle md,\ T\rangle \in^{\mathsf{i}}_P i$$
$$\text{and } M/i \triangleright M''/i \text{ and } M''.md \propto M'.md)$$

$\bullet \in^{\mathsf{rt}}_P \bullet$ in $\bullet$ Method selects a view within a chain and subchain

$$\langle md,\ T,\ V,\ e,\ m{::}M/m\rangle \in^{\mathsf{rt}}_P M_v \text{ in } M_o$$
$$\Leftrightarrow \langle md,\ T,\ V,\ e\rangle \in\!\!\!\in^{\mathsf{m}}_P m$$
$$\text{and } m_x{::}M_x = \min\{m_x{::}M_x \mid M_v \leq^{\mathsf{M}} m_x{::}M_x \text{ and } \langle md,\ T\rangle \in^{\mathsf{m}}_P m_x\}$$
$$\text{and } M_b = \max\{M' \mid m_x{::}M_x.md \propto M'.md\}$$
$$\text{and } m{::}M = \min\{m{::}M \mid M_o \leq^{\mathsf{M}} m{::}M$$
$$\text{and } m{::}M.md \propto M_b.md$$
$$\text{and } \exists V', e' \text{ s.t. } \langle md,\ T,\ V',\ e'\rangle \in\!\!\!\in^{\mathsf{m}}_P m\}$$

Fig. 14. Predicates and relations in the model of MixedJava (Part II)

Since each composite mixin has two supertypes, the type graph for mixins is a DAG, rather than a tree as for classes. This DAG would result in ambiguities if subsumption were based on subtypes. For example, $\mathsf{LockedMagic}^{\mathsf{m}}$ is a subtype of $\mathsf{Secure}^{\mathsf{m}}$, but it contains two copies of $\mathsf{Secure}^{\mathsf{m}}$ (see Figure 11). Hence, interpreting an instance of $\mathsf{LockedMagic}^{\mathsf{m}}$ as an instance of $\mathsf{Secure}^{\mathsf{m}}$ is ambiguous. More concretely, the fragment

$$\mathsf{LockedMagicDoor}^{\mathsf{m}}\ door = \mathbf{new}\ \mathsf{LockedMagicDoor}^{\mathsf{m}};$$
$$(\mathbf{view}\ \mathsf{Secure}^{\mathsf{m}}\ door).neededItem();$$

is ill-formed because $\mathsf{LockedMagic}^{\mathsf{m}}$ is not uniquely viewable as $\mathsf{Secure}^{\mathsf{m}}$. To eliminate such ambiguities, we introduce the "viewable as" relation $\unlhd_P$, which is a restriction on the subtype relation. Subsumption is thus based on $\unlhd_P$ rather than $\leq_P$. The relations $\in_P^{\mathsf{m}}$, which collect the fields and methods contained in each mixin, similarly eliminates ambiguities.

4.2 MixedJava Type Elaboration

Despite the replacement of the subtype relation with the "viewable as" relation, ClassicJava's type elaboration strategy applies equally well for typing MixedJava. The typing rules in Figure 15 are combined with the **defn**$^{\mathsf{i}}$, **meth**, **let**, **var**, **null**, and **abs** rules from Figures 6 and 7.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{llll} \textsc{MixinsOnce}(P) & \textsc{MethodOncePerMixin}(P) & \textsc{InterfacesOnce}(P) & \textsc{CompleteMixins}(P) \\ \textsc{WellFoundedMixins}(P) & \textsc{CompleteInterfaces}(P) & \textsc{WellFoundedInterfaces}(P) & \textsc{MixinFieldsOK}(P) \\ \textsc{MixinMethodsOK}(P) & \textsc{InterfaceMethodsOK}(P) & \textsc{InterfacesAbstract}(P) & \textsc{NoAbstractMixins}(P) \\ \textsc{MixinsImplementAll}(P) & P \vdash_{\mathsf{d}} \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n] & & P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \multicolumn{4}{l}{\text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e} \end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e' : t}\,[\mathbf{prog}^{\mathsf{m}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \qquad P, m \vdash_{\mathsf{m}} \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for each } k \in [1,p]}{\begin{array}{ll} P \vdash_{\mathsf{d}} \mathbf{mixin}\ m \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n & \Rightarrow \mathbf{mixin}\ m \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n \\ \qquad \mathit{meth}_1 \ldots \mathit{meth}_p\ \} & \qquad \mathit{meth}'_1 \ldots \mathit{meth}'_p\ \} \end{array}}\,[\mathbf{defn}^{\mathsf{m}}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} m \quad m \prec\!\!\!\!\cdot_P^{\mathsf{m}} \mathsf{Empty}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{new}\ m \Rightarrow \mathbf{new}\ m : m}\,[\mathbf{new}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \qquad \langle m'.\mathit{fd}, t\rangle \in_P m}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} \Rightarrow e'.\mathit{fd} : t}\,[\mathbf{get}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \qquad \langle m'.\mathit{fd}, t\rangle \in_P m \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} = e_v \Rightarrow e'.\mathit{fd} = e'_v : t}\,[\mathbf{set}^{\mathsf{m}}]$$

$$\frac{\begin{array}{ll} P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' & \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t)\rangle \in_P t' \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] & \end{array}}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{md}\,(e_1 \ldots e_n) \Rightarrow e'.\mathit{md}\,(e'_1 \ldots e'_n) : t}\,[\mathbf{call}^{\mathsf{m}}]$$

$$\frac{\begin{array}{lll} P, \Gamma \vdash_{\mathsf{e}} \mathbf{this} \Rightarrow \mathbf{this} : m & m \prec\!\!\!\!\cdot_P^{\mathsf{m}}\ i & \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t)\rangle \in_P i \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] & & \end{array}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{super}.\mathit{md}(e_1 \ldots e_n) \Rightarrow \mathbf{super}\underline{\ \equiv \mathbf{this}\ }.\mathit{md}(e'_1 \ldots e'_n) : t}\,[\mathbf{super}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow e' : t}\,[\mathbf{wcast}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \not\trianglelefteq_P t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow \mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ e' : t}\,[\mathbf{ncast}^{\mathsf{m}}]$$

$\vdash_{\mathsf{s}}$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \trianglelefteq_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow \mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ e' : t}\,[\mathbf{sub}^{\mathsf{m}}]$$

$\vdash_{\mathsf{t}}$

$$\frac{t \in \mathrm{dom}(\prec_P^{\mathsf{m}}) \cup \mathrm{dom}(\doteq_P^{\mathsf{m}}) \cup \mathrm{dom}(\prec_P^{\mathsf{i}}) \cup \{\mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}\,[\mathbf{type}^{\mathsf{m}}]$$

Fig. 15. Context-sensitive checks and type elaboration rules for MixedJava

---

Three of the new rules deserve special attention. First, the **super**$^{\mathsf{m}}$ rule allows a **super** call only when the method is declared in the current mixin's inheritance interface, where the current mixin is determined by looking at the type of **this**. Second, the **wcast**$^{\mathsf{m}}$ rule strips out the **view** part of the expression and delegates all work to the subsumption rules. Third, the **sub**$^{\mathsf{m}}$ rule for subsumption inserts a **view** operator to make subsumption coercions explicit.

4.3 MixedJava Evaluation

The operational semantics for MixedJava differs substantially from that of ClassicJava. The rewriting semantics of the latter relies on the uniqueness of each method name in the chain of **class**es associated with an object. This uniqueness is not guaranteed for chains of mixins. Specifically, a composition m_1 **compose** m_2 contains two methods named x if both m_1 and m_2 declare x and m_1's inheritance interface does not contain x. Both x methods are accessible in an instance of the composite mixin since the object can be viewed specifically as an instance of m_1 or m_2.

One strategy to avoid the duplication of x is to rename it in m_1 and m_2. At best, this is a global transformation on the program, since x is visible to the entire program as a public method. At worst, renaming triggers an exponential explosion in the size of the program, which occurs when m_1 and m_2 are actually the same mixin m. Since the mixin m represents a type, renaming x in each use of m splits it into two different types, which requires type-splitting at every expression in the program involving m.

Our MixedJava semantics handles the duplication of method names with run-time context information: the current *view* of an object.[7] During evaluation, each reference to an object is bundled with its view of the object, so that values are of the form $\langle object \| view \rangle$. A reference's view can be changed by subsumption, method calls, or explicit casts.

Each view is represented as a chain of mixins. The chain is always a sub-chain of the object's full chain of mixins, *i.e.*, the chain of mixins for the object's instantiation type. For example, when an instance of $\mathsf{LockedMagicDoor^m}$ is used as a $\mathsf{Magic^m}$ instance, the object's view corresponds to the boxed part of the following chain:

$$[\mathsf{NeedsKey^m}\ \mathsf{Secure^m}\ \boxed{\mathsf{NeedsSpell^m}\ \mathsf{Secure^m}}\ \mathsf{Door^m}]$$

The full chain correponds to $\mathsf{LockedMagicDoor^m}$ and the boxed part correponds to $\mathsf{Magic^m}$. The view designates a specific point in the full mixin chain for selecting methods during dynamic dispatch. With the above view, a search for the *neededItem* method of the object begins in the $\mathsf{NeedsSpell^m}$ element of the chain.

Our notation for views exploits the fact that an object in MixedJava encodes its full chain of mixins (in the same way that an object in ClassicJava encodes

[7] A view is analogous to a "subobject" in languages with multiple inheritance, but without the complexity of shared superclasses [27].

its class). Thus, the part of the chain before the box is not needed to describe the view:

$$[\boxed{\textsf{NeedsSpell}^{\textsf{m}}\ \textsf{Secure}^{\textsf{m}}}\ \textsf{Door}^{\textsf{m}}]$$

Since the view is always at the start of the remaining chain, we can replace the box with the name of the type it represents, which provides a purely textual notation for views:[8]

$$[\textsf{NeedsSpell}^{\textsf{m}}\ \textsf{Secure}^{\textsf{m}}\ \textsf{Door}^{\textsf{m}}]/\textsf{Magic}^{\textsf{m}}.$$

The view-based dispatching algorithm, described by the $\in^{\mathrm{rt}}_{P}$ relation, proceeds in two phases. The first phase of a search for method x locates the *base declaration* of x, which is the unique non-overriding declaration of x that is visible in the current view. This declaration is found by traversing the view from left to right, using the inheritance interface at each step as a guide for the next step (via the $\propto$ and $\triangleright$ relations). When the search reaches a mixin whose inheritance interface does not include x, the base declaration of x has been found. But the base declaration is not the destination of the dispatch; the destination is determined by the second phase, which locates an overriding declaration of x that is contained in the object's instantiated mixin. Among the declarations that override the base declaration, the leftmost declaration is selected as the destination, following customary overriding conventions. The location of the overriding declaration determines both the method definition that is invoked and the view of the object within the destination method body (*i.e.*, the view for **this**).

The dispatching algorithm explains how $\textsf{Secure}^{\textsf{m}}$'s *canOpen* method calls the appropriate *neededItem* method in an instance of $\textsf{LockedMagicDoor}^{\textsf{m}}$, sometimes dispatching to the method in $\textsf{NeedsKey}^{\textsf{m}}$ and sometimes to the one in $\textsf{NeedsSpell}^{\textsf{m}}$. The following example illustrates the essence of dispatching from $\textsf{Secure}^{\textsf{m}}$'s *canOpen*:

$$\textsf{Object}\ \mathit{canOpen}(\textsf{Secure}^{\textsf{m}}\ o)\ \{\ \ldots\ o.\mathit{neededItem}()\ \ldots\ \}$$

$$\begin{array}{l}
\textbf{let}\ \mathit{door} = \textbf{new}\ \textsf{LockedMagicDoor}^{\textsf{m}} \\
\ \textbf{in}\ \mathit{canOpen}(\underline{\textbf{view}\ \textsf{Secure}^{\textsf{m}}}\ \textbf{view}\ \textsf{Locked}^{\textsf{m}}\ \mathit{door})\ \ldots \\
\qquad \mathit{canOpen}(\underline{\textbf{view}\ \textsf{Secure}^{\textsf{m}}}\ \textbf{view}\ \textsf{Magic}^{\textsf{m}}\ \mathit{door})
\end{array}$$

The **new** $\textsf{LockedMagicDoor}^{\textsf{m}}$ expression produces *door* as an $\langle \mathit{object} \| \mathit{view} \rangle$ pair, where *object* is a new object in the store and *view* is (recall Figure 11)

$$[\textsf{NeedsKey}^{\textsf{m}}\ \textsf{Secure}^{\textsf{m}}\ \textsf{NeedsSpell}^{\textsf{m}}\ \textsf{Secure}^{\textsf{m}}\ \textsf{Door}^{\textsf{m}}]/\textsf{LockedMagicDoor}^{\textsf{m}}.$$

The **view** expressions shift the view part of *door*. Thus, for the first call to *canOpen*, o is replaced by a reference with the view

$$[\textsf{Secure}^{\textsf{m}}\ \textsf{NeedsSpell}^{\textsf{m}}\ \textsf{Secure}^{\textsf{m}}\ \textsf{Door}^{\textsf{m}}]/\textsf{Secure}^{\textsf{m}}.$$

[8] We could also use numeric position pairs to denote sub-chains, but the tail/type encoding works better for defining the operational semantics and soundness of MIXEDJAVA.

In this view, the base declaration of *neededItem* is in the leftmost $\mathsf{Secure^m}$ since *neededItem* is not in the interface extended by $\mathsf{Secure^m}$. The overriding declaration is in $\mathsf{NeedsKey^m}$, which appears to the left of $\mathsf{Secure^m}$ in the instantiated chain and extends an interface that contains *neededItem*.

In contrast, the second call to *canOpen* receives a reference with the view

$$[\mathsf{Secure^m}\ \mathsf{Door^m}]/\mathsf{Secure^m}.$$

In this view, the base definition of *neededItem* is in the rightmost $\mathsf{Secure^m}$ of the full chain, and it is overridden in $\mathsf{NeedsSpell^m}$. Neither the definition of *neededItem* in $\mathsf{NeedsKey^m}$ nor the one in the leftmost occurrence of $\mathsf{Secure^m}$ is a candidate relative to the given view, because $\mathsf{Secure^m}$ extends an interface that hides *neededItem*.

MixedJava not only differs from ClassicJava with respect to method dispatching, but also in its treatment of **super**. In MixedJava, **super** dispatches are dynamic, since the "supermixin" for a **super** expression is not statically known. The **super** dispatch for mixins is implemented like regular dispatches with the $\in_P^{rt}$ relation, but using a tail of the current view in place of both the instantiation and view chains; this ensures that a method is selected from the leftmost mixin that follows the current view.

Figure 16 contains the complete operational semantics for MixedJava as a rewriting system on expression-store pairs, similar to the class semantics described in Section 2.3. In the MixedJava semantics, an *object* in the store is tagged with a mixin instead of a class, and the values are null and $\langle object \| view \rangle$ pairs.

4.4 MixedJava Soundness

The type soundness theorem for MixedJava is *mutatis mutandis* the same as the soundness theorem for ClassicJava as described in Section 2.4.

Theorem 7 (Type Soundness for MixedJava). *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ldots defn_n\ e$*, then either*

- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \langle object \| M/t \rangle, \mathcal{S} \rangle$ *and* $\mathcal{S}(object) = \langle t', \mathcal{F} \rangle$ *and* $t' \leq_P t$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error: bad\ cast}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error: dereferenced\ null}, \mathcal{S} \rangle$*.*

The proof of soundness for MixedJava is analogous to the proof for ClassicJava, but we must update the type of the environment and the environment-store consistency relation ($\vdash_\sigma$) to reflect the differences between ClassicJava and MixedJava. In MixedJava, the environment Γ maps object-view pairs to the type part of the view, *i.e.*, $\Gamma(\langle object \| M/t \rangle) = t$. The updated consistency relation is defined as follows:

$$
\begin{array}{ll}
e = \ldots \mid \langle object \| M/t \rangle & \mathsf{E} = [\,] \mid \mathsf{E}.fd \mid \mathsf{E}.fd = e \mid v.fd = \mathsf{E} \\
v = \langle object \| M/t \rangle \mid \mathsf{null} & \quad \mid \mathsf{E}.md(e \ldots) \mid v.md(v \ldots \mathsf{E}\ e \ldots) \\
 & \quad \mid \mathbf{super}\underline{\ \equiv v\ }.md(v \ldots \mathsf{E}\ e \ldots) \\
 & \quad \mid \mathbf{view}\underline{\ t\ \mathbf{as}\ }t\ \mathsf{E} \mid \mathbf{let}\ var = \mathsf{E}\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{lr}
P \vdash \langle \mathsf{E}[\mathbf{new}\ m], \mathcal{S} \rangle & [new] \\
\quad \hookrightarrow \langle \mathsf{E}[\langle object \| M/m \rangle], \mathcal{S}[object \mapsto \langle m, [M_1.fd_1 \mapsto \mathsf{null}, \ldots M_n.fd_n \mapsto \mathsf{null}] \rangle] \rangle & \\
\quad \text{where } object \notin \mathrm{dom}(\mathcal{S}) \text{ and } m \longrightarrow_P M & \\
\qquad \{M_1.fd_1, \ldots M_n.fd_n\} = \{m'{::}M'.fd \mid M \leq^{\mathsf{M}} m'{::}M' & \\
\qquad\qquad \text{and } \exists t \text{ s.t. } \langle m'.fd, t \rangle \in^{\mathsf{m}}_P m' \} & \\
P \vdash \langle \mathsf{E}[\langle object \| M/m \rangle.fd], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S} \rangle & [get] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } \langle m'.fd, t \rangle \in_P m \text{ and } M/m \triangleright M'/m' \text{ and } \mathcal{F}(M'.fd) = v & \\
P \vdash \langle \mathsf{E}[\langle object \| M/m \rangle.fd = v], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[object \mapsto \langle m, \mathcal{F}[M'.fd \mapsto v] \rangle] \rangle & [set] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } \langle m'.fd, t \rangle \in_P m \text{ and } M/m \triangleright M'/m' & \\
P \vdash \langle \mathsf{E}[\langle object \| M/t \rangle.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [call] \\
\quad \hookrightarrow \langle \mathsf{E}[e[\langle object \| m'{::}M'/m' \rangle/\mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \longrightarrow_P M_o & \\
\qquad \text{and } \langle md, T, (var_1 \ldots var_n), e, m'{::}M'/m' \rangle \in^{\mathsf{rt}}_P M \text{ in } M_o & \\
P \vdash \langle \mathsf{E}[\mathbf{super}\underline{\ \equiv \langle object \| m{::}M/m \rangle\ }.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [super] \\
\quad \hookrightarrow \langle \mathsf{E}[e[\langle object \| m'{::}M'/m' \rangle/\mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } m \prec^{\mathsf{m}}_P i \text{ and } M/i \triangleright M''/i & \\
\qquad \text{and } \langle md, T, (var_1 \ldots var_n), e, m'{::}M'/m' \rangle \in^{\mathsf{rt}}_P M'' \text{ in } M'' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M'/t \rangle], \mathcal{S} \rangle & [view] \\
\quad \text{where } t' \trianglelefteq_P t \text{ and } M/t' \triangleright M'/t & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M''/t \rangle], \mathcal{S} \rangle & [cast] \\
\quad \text{where } t' \not\trianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \trianglelefteq_P t \text{ and } m \longrightarrow_P M' \text{ and } M'/m \triangleright M''/t & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ var = v\ \mathbf{in}\ e], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[e[v/var]], \mathcal{S} \rangle & [let]
\end{array}
$$

$$
\begin{array}{lr}
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \textsf{error: bad cast}, \mathcal{S} \rangle & [xcast] \\
\quad \text{where } t' \not\trianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \not\trianglelefteq_P t & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \mathsf{null}], \mathcal{S} \rangle \hookrightarrow \langle \textsf{error: bad cast}, \mathcal{S} \rangle & [ncast] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.fd], \mathcal{S} \rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S} \rangle & [nget] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.fd = v], \mathcal{S} \rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S} \rangle & [nset] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.md(v_1, \ldots v_n)], \mathcal{S} \rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S} \rangle & [ncall]
\end{array}
$$

Fig. 16. Operational semantics for MixedJava

Definition 8 (Environment-Store Consistency).

$$
\begin{array}{ll}
 & P, \Gamma \vdash_\sigma \mathcal{S} \\
 & \Leftrightarrow (\mathcal{S}(object) = \langle m, \mathcal{F} \rangle \\
\Sigma_1: & \qquad \Rightarrow (\Gamma(\langle object \| M/t \rangle) = t' \\
 & \qquad\qquad \Rightarrow (\mathrm{WF}(M/t) \text{ and } t = t' \text{ and } m \leq_P t)) \\
\Sigma_2: & \qquad \text{and } \mathrm{dom}(\mathcal{F}) = \{m'{::}M'.fd \mid |m| \leq^{\mathsf{M}} m'{::}M' \text{ and} \\
 & \qquad\qquad\qquad \exists\ t \text{ s.t. } \langle m'.fd, t \rangle \in^{\mathsf{m}}_P m' \} \\
\Sigma_3: & \qquad \text{and } \{object \mid \langle object \| \_\_ \rangle \in \mathrm{rng}(\mathcal{F})\} \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4: & \qquad \text{and } (\mathcal{F}(m'{::}M'.fd) = \langle object' \| M''/t' \rangle \text{ and } \langle m'.fd, t \rangle \in^{\mathsf{m}}_P m') \\
 & \qquad\qquad \Rightarrow (t' = t)) \\
\Sigma_5: & \quad \text{and } \langle object \| \_\_ \rangle \in \mathrm{dom}(\Gamma) \Rightarrow object \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6: & \quad \text{and } object \in \mathrm{dom}(\mathcal{S}) \Rightarrow \langle object \| \_\_ \rangle \in \mathrm{dom}(\Gamma)
\end{array}
$$

This definition of $\vdash_\sigma$ relies on the WF predicate on views, which is true of well-formed views. A well-formed view combines 1) a chain that is a tail of some mixin's chain, and 2) a type, either a mixin whose chain is a prefix of the view's chain or an interface implemented by the first mixin in the view's chain. Formally, WF is defined as follows:

Definition 9 (Well-Formed View).

$$\begin{aligned} \mathrm{WF}(M/t) \Leftrightarrow\ & \exists\ m_o, M_o \text{ s.t. } m_o \longrightarrow_P M_o \text{ and } M_o \leq^{\mathsf{M}} M \\ & \text{and } ((\exists\ M', M'' \text{ s.t. } M = M' \ @\ M'' \text{ and } t \longrightarrow_P M') \\ & \quad \text{or } (\exists\ m, M' \text{ s.t. } M = m{::}M' \text{ and } m \trianglelefteq_P t)) \end{aligned}$$

The lemmata for proving MixedJava soundness are mostly the same as for ClassicJava, based on a revised typing relation $\vdash_{\underline{\mathsf{e}}}$. The annotations in MixedJava programs eliminate implicit subsumption by inserting explicit **view** expressions, so the $\vdash_{\underline{\mathsf{s}}}$ relation for proving MixedJava soundness is the same as $\vdash_{\underline{\mathsf{e}}}$. The $\vdash_{\underline{\mathsf{e}}}$ relation is like $\vdash_{\mathsf{e}}$, except for the handling of **view** expressions:

$$\frac{P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'}{P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathbf{view}\underline{\ t'\ \mathbf{as}\ } t\ e : t}[\mathbf{cast}^{\mathsf{m}}]$$

Also, as in ClassicJava, we define a SuperOk predicate for validating the shape of **super** calls:

Definition 10 (Well-Formed Super Calls).

$$\begin{aligned} \mathrm{SuperOk}(e) \Leftrightarrow\ & \text{For all } \mathbf{super}\underline{\ \equiv e_0\ }.md(e_1, \ldots\ e_n) \text{ in } e, \\ & \text{either } e_0 = \mathbf{this} \text{ or } \exists\ object, M, m \text{ s.t. } e_0 = \langle object \| m{::}M/m \rangle \end{aligned}$$

Lemma 11 (Conserve for MixedJava). *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ldots defn_n\ e$*, then* $P',\emptyset \vdash_{\underline{\mathsf{e}}} e' : t$.

Proof. The claim follows from a copy-and-patch argument. □

Lemma 12 (Subject Reduction for MixedJava). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, SuperOk$(e)$, *and* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*, then* e' *is an* error *configuration or* $\exists\ \Gamma'$ *such that*

1. $P,\Gamma' \vdash_{\underline{\mathsf{e}}} e' : t$,
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$, *and*
3. SuperOk(e').

Proof. The proof examines reduction steps. For each case, if execution has not halted with an answer or in an error configuration, we construct the new environment Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment. See Appendix B for the complete proof. □

Lemma 13 (Progress for MixedJava). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, *and* SuperOk$(e)$, *then either* e *is a value or there exists an* $\langle e',\mathcal{S}'\rangle$ *such that* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$.

Proof. The proof is by analysis of the possible cases for the current redex in e (in the case that e is not a value). See Appendix B for the complete proof. □

By combinining the Subject Reduction and Progress lemmas, we can prove that every non-value MixedJava program reduces while preserving its type, thus estrablishing the soundness of MixedJava.

4.5 Implementation Considerations

The MixedJava semantics is formulated at a high level, leaving open the question of how to implement mixins efficiently. Common techniques for implementing classes can be applied to mixins, but two properties of mixins require new implementation strategies. First, each object reference must carry a view of the object. This can be implemented using double-wide references, one half for the object pointer and the other half for the current view. Second, method invocation depends on the current view as well as the instantiation mixin of an object, as reflected in the $\in_P^{\mathsf{rt}}$ relation. Although this relation depends on two inputs, it nevertheless determines a static, per-mixin method table that is analogous to the virtual method tables typically generated for classes.

The overall cost of using mixins instead of classes is equivalent to the cost of using interface-typed references instead of class-typed references. The justification for this cost is that mixins are used to implement parts of a program that cannot be easily expressed using classes. In a language that provides both classes and mixins, portions of the program that do not use mixins do not incur any extra overhead.

4.6 Related Work on Mixins

Mixins first appeared as a CLOS programming pattern [21, 22]. Unfortunately, the original linearization algorithm for CLOS's multiple inheritance breaks the encapsulation of class definitions [11], which makes it difficult to use CLOS for proper mixin programming. The CommonObjects [28] dialect of CLOS supports multiple inheritance without breaking encapsulation, but the language does not provide simple composition operators for mixins.

Bracha has investigated the use of "mixin modules" as a general language for expressing inheritance and overriding in objects [6–8]. His system is based on earlier work by Cook [9], and its underlying semantics was more recently reformulated in categorical terms by Ancona and Zucca [5]. Bracha's system gives the programmer a mechanism for defining *modules* (**class**es, in our sense) as a collection of *attributes* (methods). Modules can be combined into new modules through various merging operators. Roughly speaking, these operators provide an assembly language for expressing class-to-class functions and, as such, permit programmers to construct mixins. The language, however, forces the programmer to resolve attribute name conflicts *manually* and to specify attribute overriding *explicitly* at a mixin merge site. As a result, the programmer is faced with the same problem as in Common Lisp, *i.e.*, the low-level management of details. In contrast, our system provides a language to specify both the content of a mixin and its interaction with other mixins for mixin compositions. The latter gives each mixin an explicit role in the construction of programs so that only sensible mixin compositions are allowed. It distinguishes method overriding from accidental name collisions and thus permits the system to resolve name collisions automatically in a natural manner.

Agesen *et al.* [4] suggest that a Java variant with type parameterization can support mixins. Their approach provides a form of separately-compiled mixins. The resulting mixins are less powerful than ours because they do not resolve name collisions. Instead, any name collision introduced by a mixin application triggers a compile-time error.

5 Conclusion

We have presented a language of mixins that relies on the same programming intuition as single inheritance classes. Indeed, a mixin declaration in our language hardly differs from a class declaration since, from the programmer's local perspective, there is little difference between knowing the properties of a superclass as described by an interface and knowing the exact implementation of a superclass. From the programmer's global perspective, however, mixins free each collection of field and method extensions from the tyranny of a single superclass, enabling new abstractions and increasing the re-use potential of code.

Using mixins is inherently more expensive than using classes, but the additional cost is justified, reasonable, and offset by gains in code re-use. Future work on mixins must focus on exploring compilation strategies that lower the cost of mixins, and on studying how designers can exploit mixins to construct better design patterns.

Acknowledgements: Thanks to Corky Cartwright, Robby Findler, Cormac Flanagan, and Dan Friedman for their comments on early drafts of this paper.

References

1. Abadi, M., and Cardelli, L. A theory of primitive objects — untyped and first-order systems. In *Theoretical Aspects of Computer Software*, M. Hagiya and J. C. Mitchell, Eds., vol. 789 of *Lecture Notes in Computer Science*. Springer-Verlag, Apr. 1994, pp. 296–320.
2. Abadi, M., and Cardelli, L. A theory of primitive objects: second-order systems. In *Proc. European Symposium on Programming* (1994), D. Sannella, Ed., vol. 788 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–25.
3. Abadi, M., and Cardelli, L. An imperative object calculus. In *Theory and Practice of Software Development* (May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 471–485.
4. Agesen, O., Freund, S., and Mitchell, J. C. Adding type parameterization to Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1997), pp. 49–65.
5. Ancona, D., and Zucca, E. An algebraic approach to mixins and modularity. In *Proc. Conference on Algebraic and Logic Programming* (1996), M. Hanus and M. Rodríguez-Artalejo, Eds., vol. 1139 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 179–193.
6. Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

7. Bracha, G., and Cook, W. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990).
8. Bracha, G., and Lindstrom, G. Modularity meets inheritance. In *Proc. IEEE Computer Society International Conference on Computer Languages* (Apr. 1992), pp. 282–290.
9. Cook, W. R. *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.
10. Drossopolou, S., and Eisenbach, S. Java is typesafe – probably. In *Proc. European Conference on Object Oriented Programming* (June 1997).
11. Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic conflict resolution mechanisms for inheritance. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1992), pp. 16–24.
12. Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. Application of OOP type theory: State, decidability, integration. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1994), pp. 16–30.
13. Felleisen, M. Programming languages and lambda calculi. `www.cs.rice.edu/~matthias/411web/mono.ps`.
14. Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. 100, Rice University, June 1989. *Theoretical Computer Science*, volume 102, 1992, pp. 235–271.
15. Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S., and Felleisen, M. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs* (Sept. 1997), pp. 369–388.
16. Flatt, M. PLT MzScheme: Language manual. Tech. Rep. TR97-280, Rice University, 1997.
17. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
18. Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.
19. Harper, R., and Stone, C. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 1998.
20. Kamin, S. Inheritance in smalltalk-80: a denotational definition. In *Proc. ACM Symposium on Principles of Programming Languages* (Jan. 1988).
21. Kessler, R. R. *LISP, Objects, and Symbolic Programming*. Scott, Foresman and Company, Glenview, IL, USA, 1988.
22. Koschmann, T. *The Common LISP Companion*. John Wiley and Sons, New York, N.Y., 1990.
23. Mason, I. A., and Talcott, C. L. Reasoning about object systems in VTLoE. *International Journal of Foundations of Computer Science 6*, 3 (Sept. 1995), 265–298.
24. Palsberg, J., and Schwartzbach, M. I. *Object-oriented Type Systems*. John Wiley & Sons, 1994.
25. Reddy, U. S. Objects as closures: Abstract semantics of object oriented languages. In *Proc. Conference on Lisp and Functional Programming* (July 1988), pp. 289–297.

26. Rémy, D. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software* (New York, N.Y., Apr. 1994), M. Hagiya and J. C. Mitchell, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 321–346.
27. Rossie, J. G., Friedman, D. P., and Wand, M. Modeling subobject-based inheritance. In *Proc. European Conference on Object-Oriented Programming* (July 1996), P. Cointe, Ed., vol. 1098 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 248–274.
28. Snyder, A. Inheritance and the development of encapsulated software components. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 165–188.
29. Syme, D. Proving Java type soundness. Tech. Rep. 427, University of Cambridge, July 1997.
30. Wright, A., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.

Appendix A: ClassicJava Proofs

Lemma 5 (Subject Reduction). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, $\textsc{SuperOk}(e)$, *and* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$, *then* e' *is an* error *configuration or* $\exists\ \Gamma'$ *such that*

1. $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e' : t$,
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$, *and*
3. $\textsc{SuperOk}(e')$.

Proof. The proof examines reduction steps. For each case, if execution has not halted with an error configuration, we construct the new environment Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case $[new]$. Set $\Gamma' = \Gamma\,[object : c]$.

1. We have $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ c] : t$. From $[new]$, $object \notin \mathrm{dom}(\mathcal{S})$. Then, by Σ_5, $object \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ c] : t$ by Lemma 14. Since $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathbf{new}\ c : c$ and $P,\Gamma' \vdash_{\underline{\mathsf{e}}} object : c$, Lemma 15 implies $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[object] : t$.
2. Let $\mathcal{S}'(object) = \langle c,\mathcal{F}\rangle$. $object$ is the only new element in $\mathrm{dom}(\mathcal{S}')$ and $\mathrm{dom}(\Gamma')$.

   Σ_1**:** $\Gamma'(object) = c$.

   Σ_2**:** $\mathrm{dom}(\mathcal{F})$ is correct by construction.

   Σ_3**:** $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$.

   Σ_4**:** Since $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$, this property is unaffected.

   Σ_5 **and** Σ_6**:** The only change to Γ and $\mathcal{S}$ is $object$.
3. Since $\mathsf{E}[object]$ contains the same **super** expressions as $\mathsf{E}[\mathbf{new}\ c]$, and no instance of **this** or $object$ is replaced in the new expression, $\textsc{SuperOk}(e')$ holds.

Case [*get*].

1. Let t' be such that $P,\Gamma \vdash_{\underline{\mathsf{e}}} object\underline{\ : c'\ }.fd : t'$. If v is null, it can be typed as t', so $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$ by Lemma 15. If v is not null, then by Σ_4, $\mathcal{S}(v) = \langle c'',\_\rangle$ where $c'' \leq_P t'$. By Lemma 17, $P,\Gamma' \vdash_{\underline{\mathsf{s}}} \mathsf{E}[v] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. SuperOk(e') holds because no **super** expression is changed.

Case [*set*].

1. The proof is by a straightforward extension of the proof for [*get*].
2. The only change to the store is a field update; thus only Σ_3 and Σ_4 are affected. Let v be the assigned value, and assume that v is not null.
   Σ_3**:** Since v is typable, it must be in dom(Γ). By Σ_5, it is therefore in dom($\mathcal{S}$).
   Σ_4**:** The typing of the assignment expression demands that the type of v can be treated as the type of the field *fd* by subsumption. Combining this with Σ_1 indicates that the type tag of v will preserve Σ_4.
3. SuperOk(e') holds because no **super** expression is changed.

Case [*call*].

1. From $P,\Gamma \vdash_{\underline{\mathsf{e}}} object.md(v_1, \ldots\ v_n) : t$ we know $P,\Gamma \vdash_{\underline{\mathsf{e}}} object : t'$, $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ for i in $[1, n]$, and $\langle md, (t_1 \ldots\ t_n \rightarrow t), (var_1, ..., var_n), e_m\rangle \in^{\mathsf{c}}_P t'$. The type-checking of P proves that $P,t_0 \vdash_{\underline{\mathsf{m}}} t\ md\ (t_1\ var_1, \ldots\ t_n\ var_n)\ \{e_m\}$, which implies that $P,[\mathbf{this} : t_0,\ var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{s}}} e_m : t$ where t_0 is the defining class of *md*. Further, we know that $t' \leq_P t_0$ from $\in^{\mathsf{c}}_P$ for methods and ClassMethodsOk(P). Thus, Lemma 16 shows that $P,\Gamma \vdash_{\underline{\mathsf{s}}} e_m[object/\mathbf{this}, v_1/var_1, \ldots\ v_n/var_n] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. The reduction may introduce new **super** expressions into the complete expression, but each new **super** must originate directly from P, which contains **super** expressions with **this** annotations only. The [*object*/**this**] part of the substitution may replace **this** in a **super** annotation with *object*, but no other part of the substitution can affect **super** annotations. Thus, SuperOk(e') holds.

Case [*super*]. The proof is essentially the same as the proof for [*call*].

Case [*cast*].

1. By assumption, $\mathcal{S}(object) = \langle c,\_\rangle$ where $c \leq_P t$. Since $c \leq_P t$, $P,\Gamma \vdash_{\underline{\mathsf{s}}} object : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. SuperOk(e') holds because no **super** expression is changed.

Case [*let*].

1. $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathbf{let}\ var = v\ \mathbf{in}\ e : t$ implies $P,\Gamma \vdash_{\underline{\mathsf{e}}} v : t'$ for some type t' and $P,\Gamma\ [var : t'] \vdash_{\underline{\mathsf{e}}} e : t$. By Lemma 16, $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e\ [v/var] : t$.

2. $\mathcal{S}$ and Γ are unchanged.

3. SuperOk(e') holds because no **super** expression is changed.

Case [*xcast*], [*ncast*], [*nget*], [*nset*], and [*ncall*]. e' is an error configuration. □

Lemma 6 (Progress). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, *and* SuperOk($e$), *then either* e *is a value or there exists an* $\langle e',\mathcal{S}'\rangle$ *such that* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$.

Proof. The proof is by analysis of the possible cases for the current redex in e (in the case that e is not a value).

Case new c. The [*new*] reduction rules constructs the appropriate e' and $\mathcal{S}'$.
Case $v\underline{\ :\ c\ }.fd$. If v is null, then the [*nget*] reduction rule applies. Otherwise, v = *object*, and we show that [*get*] applies.
Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle c,\mathcal{F}\rangle$ for some c and $\mathcal{F}$. Type-checking also implies that $\langle c'.fd,t\rangle \in_P c$ for some c' by [**get**$^{\mathsf{c}}$]. By the definition of $\in_P$ ($\in^{\mathsf{c}}_P$ in this case), we have $c \leq_P c'$. Finally, by Σ_2, $c'.fd \in \mathrm{dom}(\mathcal{F})$.
Case $v\underline{\ :\ c\ }.fd = v'$. Similar to $v\underline{\ :\ c\ }.fd$, either [*nset*] or [*set*] applies.
Case $v.md(v_1, \ldots\ v_n)$. If v is null, then the [*ncall*] reduction rule applies. Otherwise, v = *object*, and [*call*] applies: type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle c,\mathcal{F}\rangle$ for some c and $\mathcal{F}$, and type-checking also implies $\langle md,T,V,e_m\rangle \in^{\mathsf{c}}_P c$.
Case super$\underline{\equiv v\ :\ c\ }(v_1, \ldots\ v_n)$. By SuperOk($e$), v must be of the form *object*. Type-checking ensures $\langle md,T,V,e_m\rangle \in^{\mathsf{c}}_P c$.
Case view $t\ v$. If v is null, then [*ncast*] applies. Otherwise, v = *object*, and by Σ_5, $\mathcal{S}(object) = \langle c,\mathcal{F}\rangle$ for some c and $\mathcal{F}$. Either [*cast*] or [*xcast*] applies, depending on whether $c \leq_P t$.
Case let $var = v \in e$. The [*let*] reduction always applies, constructing an e' and $\mathcal{S}'$ ($= \mathcal{S}$). □

Lemma 14 (Free). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $a \notin \mathrm{dom}(\Gamma)$, *then* $P,\Gamma\,[a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.

Proof. The claim follows by reasoning about the shape of the derivation. □

Lemma 15 (Replacement). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$, *then* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.

Proof. The proof is a replacement argument in the derivation tree. □

Lemma 16 (Substitution). *If* $P,\Gamma\,[var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\{var_1, \ldots\ var_n\} \cap \mathrm{dom}(\Gamma) = \emptyset$ *and* $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1,n]$, *then* $P,\Gamma \vdash_{\underline{\mathsf{s}}} e\,[v_1/var_1, \ldots\ v_n/var_n] : t$.

Proof. Let σ denote the substitution $[v_1/var_1, \ldots\ v_n/var_n]$, let γ denote the type environment $[var_1 : t_1, \ldots\ var_n : t_n]$, and let $e' = \sigma(e)$. The proof proceeds by induction on the structure of the derivation showing that $P,\Gamma\gamma \vdash_{\underline{\mathsf{e}}} e : t$. We perform a case analysis on the last step in the derivation.

Case $e = \textbf{new}\ c$. Since $e' = \textbf{new}\ c$ and its type does not depend on Γ, then $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : c$.

Case $e = var$. If $var \notin \mathrm{dom}(\sigma)$, then $e' = var$ and $\Gamma(var) = t$, so $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t$. Otherwise, $var = var_i$ for some $i \in [1,n]$, and $e' = \sigma(var_i) = v_i$. By assumption, $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$, so $P,\Gamma \vdash_{\underline{\mathsf{s}}} e' : t_i$.

Case $e = \mathsf{null}$. By [**null**], any type is derivable for null.

Case $e = e_1 \underline{\ :c\ }.fd$. By [**get**$^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{\mathsf{e}}} e_1 : t'$ and $\langle c.fd,t \rangle \in_P t'$ from some t'. By induction, $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_1) : t''$ where t'' is a sub-type of t'. Since $\in_P$ for fields is closed over subtypes on the right-hand side, $\langle c.fd,t \rangle \in_P t''$. Thus, by [**get**$^{\mathsf{c}}$] on $e' = \sigma(e_1)\underline{\ :c\ }.fd$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t$.

Case $e = e_1 \underline{\ :c\ }.fd = e_2$. This case is similar to the previous case, relying on subsumption for the right-hand side of an assignment as allowed by [**set**$^{\mathsf{c}}$].

Case $e = \textbf{view}\ t\ e_1$. By [**cast**$^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{\mathsf{e}}} e_1 : t'$ for some t'. By induction, $P,\Gamma \vdash_{\underline{\mathsf{s}}} \sigma(e_1) : t'$. Since $e' = \sigma(\textbf{view}\ t\ e_1) = \textbf{view}\ t\ \sigma(e_1)$, [**cast**$^{\mathsf{c}}$] gives $P, \Gamma \vdash_{\underline{\mathsf{e}}} e' : t$.

Case $e = \textbf{let}\ var = e_1\ \textbf{in}\ e_2$. Let $\sigma_1 = \sigma$ and $\gamma_1 = \gamma$, and lexically rename var in e so that $var \notin \mathrm{dom}(\gamma_1)$. By [**let**], $P,\Gamma\gamma_1 \vdash_{\underline{\mathsf{e}}} e_1 : t_1$. By induction, $P,\Gamma \vdash_{\underline{\mathsf{s}}} \sigma_1(e_1) : t_1$. Let $\gamma_2 = [var : t_1]$, so that $P,\Gamma\gamma_1\gamma_2 \vdash_{\underline{\mathsf{e}}} e_2 : t$. Since $var \notin \mathrm{dom}(\gamma_1)$, we can reverse the order of the γ_1 and γ_2 extensions to Γ, so $P,\Gamma\gamma_2\gamma_1 \vdash_{\underline{\mathsf{e}}} e_2 : t$. By induction, $P,\Gamma\gamma_2 \vdash_{\underline{\mathsf{s}}} \sigma_1(e_2) : t$. Finally, by [**let**] on $e' = \sigma_1(\textbf{let}\ var = e_1\ \textbf{in}\ e)$, $P,\Gamma \vdash_{\underline{\mathsf{s}}} e' : t$.

Case $e = e_0.md\,(e_1, \ldots\ e_n)$. By [**call**$^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{\mathsf{s}}} e_i : t_i$ for $i \in [1,n]$ and $P,\Gamma\gamma \vdash_{\underline{\mathsf{e}}} e_0 : t_0$ such that $\langle md, (t_1 \ldots\ t_n \rightarrow t), (var_1, ..., var_n), e_m \rangle \in_P t_0$. By induction, $P,\Gamma \vdash_{\underline{\mathsf{s}}} \sigma(e_i) : t_i$ for each e_i, and $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_0) : t_0{}'$ where $t_0{}' \leq_P t_0$. Since $\in_P$ must preserve the type of methods over subtypes on the right-hand side (by ClassMethodsOk), $\langle md,(t_1\ \ldots\ t_n \rightarrow t),(var_1{}',\ \ldots var_n{}'),e_m{}' \rangle \in_P t_0{}'$. Thus, by [**call**$^{\mathsf{c}}$] on $e' = \sigma(e_0.md\,(e_1,\ \ldots\ e_n))$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t$.

Case $e = \textbf{super}\underline{\ \equiv \textbf{this} : c\ }.md\,(e_1, \ldots\ e_n)$. This case is similar to the previous case. $\square$

Lemma 17 (Replacement with Subtyping). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$*,* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$*, and* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t''$ *where* $t'' \leq_P t'$*, then* $P,\Gamma \vdash_{\underline{\mathsf{s}}} \mathsf{E}[e'] : t$*.*

Proof. The proof is by induction on the depth of the evaluation context E. If E is the empty context $[\,]$ we are done. Otherwise, partition $\mathsf{E}[e] = \mathsf{E}_1[\mathsf{E}_2[e]]$ where E_2 is a singular evaluation context, *i.e.*, a context whose depth is one. Consider the shape of $\mathsf{E}_2[\bullet]$, which must be one of:

Case $\bullet\underline{\ :c\ }.fd$. Since c is fixed, $\bullet$'s type does not matter; the expression's type is the type of the field.

Case $\bullet\underline{\ :c\ }.fd = e$. Same as the previous case.

Case $v\underline{\ :c\ }.fd = \bullet$. Since [**set**$^{\mathsf{c}}$] allows subsumption on the right-hand side of the assignment, the type of the expression is the same replacing $\bullet$ with e or e'.

Case $\bullet.md(e\ \ldots)$. Since $t'' \leq_P t'$ and methods in an inheritance chain preserve the return type, the type of the expression is the same replacing $\bullet$ with e or e'.

Case $v.md(v\ \ldots\ \bullet\ e\ \ldots)$. Since [**call**^{c}] allows subsumption on method arguments, the type of the expression is the same replacing $\bullet$ with e or e'.

Case **super**$\underline{\equiv v : c}$ $.md\ (v\ \ldots\ \bullet\ e\ \ldots)$. Same as the previous case.

Case **view** t $\bullet$. Since t is fixed, $\bullet$'s type does not matter in [**cast**^{c}] (our less restrictive typing rule); the expression's type is t.

Case **let** $var = \bullet$ **in** e_2. By [**let**] with $P,\Gamma \vdash_{\underline{e}} e : t'$, $P,\Gamma\gamma_1 \vdash_{\underline{e}} e_2 : t_1$ for some type t_1 where γ_1 is $[var : t']$. We must show that $P,\Gamma\gamma_2 \vdash_{\underline{s}} e_2 : t_1$ where $\gamma_2 = [var : t'']$, which follows from Lemma 19. □

Definition 18. $\Gamma \leq_\Gamma \Gamma'$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall\ v \in \mathrm{dom}(\Gamma)$, $\Gamma'(v) \leq_P \Gamma(v)$.

Lemma 19. *If* $P,\Gamma \vdash_{\underline{e}} e : t$ *and* $\Gamma \leq_\Gamma \Gamma'$*, then* $P,\Gamma' \vdash_{\underline{s}} e : t$.

Proof. The proof is a straightforward adaptation of the proof for Lemma 16. □

Appendix B: MixedJava Proofs

Lemma 12 (Subject Reduction for MixedJava). *If* $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_\sigma \mathcal{S}$, SuperOk$(e)$, *and* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*, then* e' *is an* error *configuration or* $\exists\ \Gamma'$ *such that*

1. $P,\Gamma' \vdash_{\underline{e}} e' : t$,
2. $P,\Gamma' \vdash_\sigma \mathcal{S}'$, *and*
3. SuperOk(e').

Proof. The proof examines reduction steps. For each case, if execution has not halted with an answer or in an error configuration, we construct the new environment Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case [*new*]. Set $\Gamma' = \Gamma\ [\langle object||M/m\rangle : m]$.

1. We have $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ m] : t$. From Σ_5, $object \notin \mathrm{dom}(\mathcal{S}) \Rightarrow \langle object||\_\_\rangle \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ m] : t$ by Lemma 20. Since $P,\Gamma' \vdash_{\underline{e}}$ $\mathbf{new}\ m : m$ and $P,\Gamma' \vdash_{\underline{e}} \langle object||M/m\rangle : m$, Lemma 21 implies $P,\Gamma' \vdash_{\underline{e}}$ $\mathsf{E}[\langle object||M/m\rangle] : t$.
2. Let $\mathcal{S}'(object) = \langle m,\mathcal{F}\rangle$, so $object$ is the only new element in $\mathrm{dom}(\mathcal{S}')$ and $\langle object||M/m\rangle$ is the only new element in $\mathrm{dom}(\Gamma')$.
   Σ_1**:** $\Gamma'(\langle object||M/m\rangle) = m$ and $m \leq_P m$. Since $m \longrightarrow_P M$, $\mathrm{WF}(M/m)$.

   Σ_2**:** $\mathrm{dom}(\mathcal{F})$ is correct by construction.

Σ_3**:** $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$.

Σ_4**:** Since $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$, this property is unaffected.

Σ_5 **and** Σ_6**:** The only addition to the domains of Γ and $\mathcal{S}$ is *object*.

3. Since $\mathsf{E}[\langle object \| M/m \rangle]$ contains the same **super** expressions as $\mathsf{E}[$**new** $m]$, and no instance of **this** or *object* is replaced in the new expression, $\textsc{SuperOk}(e')$ holds.

Case $[get]$. Set $\Gamma' = \Gamma$.

1. If v is null, it can be typed as t, so $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$ by Lemma 21. If v is not null, then by Σ_4, $v = \langle object \| M/t \rangle$ for some *object* and M. By Σ_1, $\Gamma(v) = t$, so by Lemma 21, $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case $[set]$. Set $\Gamma' = \Gamma$.

1. The proof is by a straightforward extension of the proof for $[get]$.
2. The only change to the store is a field update; thus only Σ_3 and Σ_4 are affected. Let v be the assigned value, and assume that v is not null.

   Σ_3**:** Since v is typable, it must be in $\mathrm{dom}(\Gamma)$. By Σ_5, its object part is therefore in $\mathrm{dom}(\mathcal{S})$.

   Σ_4**:** The typing of the assignment expression indicates that the type of v is t, so v must be of the form $\langle object' \| M''/t \rangle$.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case $[call]$. Set $\Gamma' = \Gamma\ [\langle object \| m'{::}M'/m' \rangle : m']$.

1. We are given $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e_m, m' :: M'/m' \rangle \in^{\mathsf{rt}}_P M_v$ in M_o, which implies $\langle md, T, V, e_m \rangle \in^{\mathsf{m}}_P m'$ by the definition of $\in^{\mathsf{rt}}_P$, where $T = (t_1\ \ldots\ t_n \to t)$ and $V = (var_1\ \ldots\ var_n)$.
   We are also given $P,\Gamma \vdash_{\underline{\mathsf{e}}} \langle object \| M_v/t' \rangle.md(v_1, \ldots\ v_n) : t$, which implies $\langle md, T' \rangle \in_P t'$. By Lemma 25, $T' = T$. Since the method call type-checks and $T' = T$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} v_i : t_i$ for i in $[1, n]$.
   Type-checking for the program P ensures that $P,m' \vdash_{\underline{\mathsf{m}}} t\ md\ (t_1\ var_1, \ldots\ t_n\ var_n)\ \{e_m\}$, and thus $P,[\mathbf{this} : m', var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$. Hence, by Lemma 22, $P,\Gamma' \vdash_{\underline{\mathsf{e}}} e_m[\langle object \| m'{::}M'/m' \rangle/\mathbf{this}, v_1/var_1, \ldots\ v_n/var_n] : t$.
2. $\mathcal{S}' = \mathcal{S}$. If Γ' contained a mapping for $\langle object \| m'{::}M'/m' \rangle$, it was m' by Σ_1, so $\Gamma' = \Gamma$. Otherwise, $\langle object \| m'{::}M'/m' \rangle$ is new in Γ', which might affect Σ_1, Σ_5, and Σ_6:

   Σ_1**:** $\Gamma'(\langle object \| m'{::}M'/m' \rangle) = m'$. The $\in^{\mathsf{rt}}_P$ relation ensures that $m \leq_P m'$ because $M_o \leq^{\mathsf{M}} m'{::}M'$. $\mathrm{WF}(m'{::}M'/m')$ is immediate.

   Σ_5 **and** Σ_6**:** Since $\Gamma(\langle object \| M_v/t' \rangle) = t'$, $object \in \mathrm{dom}(\mathcal{S})$ by Σ_5 on $\mathcal{S}$ and Γ. Thus, adding $\langle object \| m'{::}M'/m' \rangle$ to Γ' does not require any new elements in $\mathcal{S}'$.

3. The reduction may introduce new **super** expressions into the complete expression, but each new **super** expression must originate directly from P, which contains **super** expressions with **this** annotations only. The $[\langle object||m'{::}M'/m'\rangle/\mathbf{this}]$ part of the substitution may replace **this** in a **super** annotation with $\langle object||m'{::}M'/m'\rangle$, but no other part of the substitution can affect **super** annotations. Thus, SUPEROK(e') holds.

Case [*super*]. Set $\Gamma' = \Gamma\ [\langle object||m'{::}M'/m'\rangle : m']$.

1. Similar to [*call*]. The object for dispatching is $\langle object||m{::}M/m\rangle$, and we are given $m \prec^{\mathsf{m}}_P i$ and $\langle md,T\rangle \in_P i$. (The i in [$\mathbf{super}^{\mathsf{m}}$] and the i in [*call*] are the same, since a mixin extends only one interface.) To apply Lemma 25, we need $\mathrm{WF}(M''/i)$, where $M/i \triangleright M''/i$. Lemma 23 is not strong enough to gurantee $\mathrm{WF}(M''/i)$, since M/i is not necessarily well-formed. However, $\triangleright$ with an interface always produces a well-formed view on the right-hand side by construction, so $\mathrm{WF}(M''/i)$. Thus, we can apply Lemma 25 as for [*call*].
2. Similar to [*call*]. If $\langle object||m'{::}M'/m'\rangle$ is new:
   Σ_1: $\Gamma'(\langle object||m'{::}M'/m'\rangle) = m'$. If $\mathcal{S}(object) = \langle m_o,\_\rangle$, $m_o \leq_P m'$ because Σ_1 on Γ ensures that the original view $m{::}M$ is part of m_o, $\triangleright$ selects a sub-view of M as M'', and $\in^{\mathsf{rt}}_P$ selects $m'{::}M'$ within M''.

   Σ_5 and Σ_6: Same as [*call*].
3. Same as [*call*].

Case [*view*]. Set $\Gamma' = \Gamma[\langle object||M'/t\rangle : t]$.

1. Since $\Gamma(\langle object||M'/t\rangle) = t$, by Lemma 21, $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\langle object||M'/t\rangle] : t$.
2. Similar to [*call*]. If $\langle object||M'/t\rangle \notin \Gamma$:
   Σ_1: $\Gamma'(\langle object||M'/t\rangle) = t$. The side condition for [*view*] requires $t' \trianglelefteq_P t$, which implies $t' \leq_P t$. Σ_1 on Γ ensures $m \leq_P t'$ when $\mathcal{S}(object) = \langle m,\_\rangle$, so $m \leq_P t$ by transitivity.

   Σ_5 and Σ_6: Same as [*call*].
3. SUPEROK(e') holds because no **super** expression is changed.

Case [*cast*]. Set $\Gamma' = \Gamma[\langle object||M''/t\rangle : t]$.

1. Same as [*view*].
2. Similar to [*call*]. If $\langle object||M''/t\rangle$ is new:
   Σ_1: $\Gamma'(\langle object||M''/t\rangle) = t$. The side condition for [*cast*] requires $m \trianglelefteq_P t$, which implies $m \leq_P t$.

   Σ_5 and Σ_6: Same as [*call*].
3. SUPEROK(e') holds because no **super** expression is changed.

Case [*let*]. $P,\Gamma \vdash_{\underline{\mathsf{e}}}$ **let** $var = v$ **in** $e : t$ implies $P,\Gamma \vdash_{\underline{\mathsf{e}}} v : t'$ for some type t' and $P,\Gamma\ [var : t'] \vdash_{\underline{\mathsf{e}}} e : t$. Set $\Gamma' = \Gamma$.

1. By Lemma 22, $P,\Gamma' \vdash_{\underline{\mathsf{e}}} e\ [v/var] : t$.

2. $\mathcal{S}$ and Γ are unchanged.
3. SUPEROK(e') holds because no **super** expression is changed.

Case [*xcast*], [*ncast*], [*nget*], [*nset*] and [*ncall*]. e' is an error configuration. □

Lemma 13 (Progress for MIXEDJAVA). *If $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, and* SUPEROK($e$), *then either e is a value or there exists an $\langle e',\mathcal{S}'\rangle$ such that $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$.*

Proof. The proof is by analysis of the possible cases for the current redex in e (in the case that e is not a value).

Case new m. The [*new*] reduction rules constructs the appropriate e' and $\mathcal{S}'$.
Case $v.fd$. If v is null, then the [*nget*] reduction rule applies. Otherwise, $v = \langle object||M/t\rangle$, and we show that [*get*] applies.
Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle m_o,\mathcal{F}\rangle$ for some m_o and $\mathcal{F}$. Type-checking also implies that $t = m$ for some mixin m, and $\langle m'.fd,t\rangle \in_P m$ for some m' by [$\mathbf{get}^{\mathsf{m}}$]. By the definition of $\in_P$ ($\in_P^{\mathsf{m}}$ in this case), we have $m \trianglelefteq_P m'$, so there is a unique m' such that $M/m \triangleright M'/m'$, and $M \leq^{\mathsf{M}} M'$.
Environment-store consistency implies $M_o \leq^{\mathsf{M}} M$, where $m_o \longrightarrow_P M_o$. By transitivity, $M_o \leq^{\mathsf{M}} M'$. Finally, by Σ_2, $M'.fd \in \mathrm{dom}(\mathcal{F})$.
Case $v.fd = v'$. Similar to $v.fd$; either [*nset*] or [*set*] applies.
Case $v.md(v_1, \ldots\ v_n)$. If v is null, then the [*ncall*] reduction rule applies. Otherwise, $v = \langle object||M/t'\rangle$, and we show that [*call*] applies.
Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle m_o,\mathcal{F}\rangle$ for some m_o and $\mathcal{F}$. Define M_o as $m_o \longrightarrow_P M_o$.
By [$\mathbf{call}^{\mathsf{m}}$], $\langle md, T\rangle \in_P t'$. The $\in_P^{\mathsf{rt}}$ relation necessarily selects some m'::M'/m' and e:

1. m_x::M_x exists because WF(M/t') and $\langle md,T\rangle \in_P t'$ implies that some atomic mixin in M contains *md*.
2. M_b exists because $\propto$ can at least relate m_x::$M_x.md$ to itself. More specifically, we know that M_b starts with an atomic mixin m_b such that $\langle md,T,V,e\rangle \Subset_P^{\mathsf{m}} m_b$:
   – If there is no i such that $m_x \prec_P^{\mathsf{m}} i$ and $\langle md,T\rangle \in_P i$, then $\langle md,T,V,e\rangle \Subset_P^{\mathsf{m}} m_x$ by the definition of $\Subset_P^{\mathsf{m}}$.
   – Otherwise, there must be some m_y::M_y such that $M_x/i \triangleright m_y$::M_y/i, or else m_o would not be a proper mixin composition. Thus, m_x::$M_x.md$ $\propto m_y$::$M_y.md$ where m_x::$M_x \leq^{\mathsf{M}} m_y$::$M_y$. This argument on m_x::M_x applies inductively to m_y::M_y, showing that $\langle md,T,V,e\rangle \Subset_P^{\mathsf{m}} m_b$.
3. By WF(M/t') and $M \leq^{\mathsf{M}} M_b$, then $M_o \leq^{\mathsf{M}} M_b$, so the final phase of the $\in_P^{\mathsf{rt}}$ calculation must succeed (although the selected view is not necessarily M_b).

Case super$\underline{\equiv v}$ $(v_1, \ldots\ v_n)$. By SUPEROK(e), v must be of the form $\langle object||m$::$M/m\rangle$. Thus, we show that the [*super*] reduction applies.
Since $m \prec_P^{\mathsf{m}} i$, there must be some atomic mixin in M that implements i, otherwise the object's instantiation mixin would not be a proper composition. Thus, there is some M' such that $M/i \triangleright M'/i$. Since type-checking ensures

that $\langle md,T\rangle \in_P i$, we can apply the same reasoning as for the $v.md(v_1, \ldots v_n)$ case, showing that the $\in_P^{\mathsf{rt}}$ relation necessarily selects some $m'::M'/m'$ and e.

Case view $\underline{\ t'\ \mathbf{as}\ }\ t\ v$. If v is null, then $[ncast]$ applies. Otherwise, $v = \langle object \| M/t'\rangle$ for some M, and by Σ_5 $\mathcal{S}(object) = \langle m_o,\mathcal{F}\rangle$ for some m_o and $\mathcal{F}$.
If $t' \trianglelefteq_P t$, then $[view]$ applies since M'/t clearly exists for $M/t \triangleright M'/t$. Assume that $t' \not\trianglelefteq_P t$. If $m_o \not\trianglelefteq_P t'$, then $[xcast]$ applies. Otherwise, $[cast]$ applies since $m_o \trianglelefteq_P t$ means that M''/t clearly exists for $M'/m \triangleright M''/t$, where $m \longrightarrow_P M'$.

Case let $var = v \in e$. The $[let]$ reduction always applies, constructing an e' and $\mathcal{S}'$ $(= \mathcal{S})$. □

By combinining the Subject Reduction and Progress lemmas, we can prove that every non-value MixedJava program reduces while preserving its type, thus estrablishing the soundness of MixedJava.

Lemma 20 (Free for MixedJava). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $a \notin \mathrm{dom}(\Gamma)$*, then* $P,\Gamma[a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.

Proof. This follows by reasoning about the shape of the derivation. □

Lemma 21 (Replacement for MixedJava). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, *and* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$*, then* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.

Proof. The proof is a replacement argument in the derivation tree. □

Lemma 22 (Substitution for MixedJava). *If* $P,\Gamma[var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\{var_1, \ldots\ var_n\} \cap \mathrm{dom}(\Gamma) = \emptyset$ *and* $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1,n]$*, then* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e\,[v_1/var_1, \ldots\ v_n/var_n] : t$.

Proof. Unlike the Substituion lemma for ClassicJava, the proof of this lemma follows simply from reasoning about the shape of the derivation, since it makes no claims about subsumption. The proof uses nested induction over the number of variables $var_1, \ldots\ var_n$ and over the number of replacements for each variable. □

Lemma 23 ($\bullet/\bullet \triangleright \bullet/\bullet$ Preserves Well-Formedness). *If* $\mathrm{WF}(M/t)$*, and* $M/t \triangleright M'/t'$*, then* $\mathrm{WF}(M'/t')$.

Proof. There are two cases, depending on whether t' is a mixin or an interface:

Case t and t' are mixins, m and m'. The proof is by lexicographic induction on the length of M and size of m (*i.e.*, the number atomic mixins composed to define m). If M is $[m]$ (the base case), then $t' = m$, $M' = M$, and $\mathrm{WF}([m']/m')$. Also, if $m = m'$ and $M = M'$, then $\mathrm{WF}(M/m) = \mathrm{WF}(M'/m')$.
Otherwise, $m' = m'' \circ m'''$, and there are two sub-cases:

– If $m'' \leq^{\mathsf{M}}_P m'$ and $M/m'' \triangleright M'/m'$, then $\mathrm{WF}(M'/m')$ by induction: m'' is smaller than m, and $\mathrm{WF}(M/m'')$ since m'' is a prefix of m.

– If $m''' \leq^{\mathsf{M}}_P m'$ and $M_r/m''' \triangleright M'/m'$, then $\mathrm{WF}(M'/m')$ by induction: M_r is smaller than M, and $\mathrm{WF}(M_r/m''')$ because m''' is a prefix of M_r.

Case t' is an interface, i. M' is constructed as m::M'' where $m \prec\!\!\prec^{\mathsf{m}}_P i$. Thus, $\mathrm{WF}(M'/i)$ because $M' = m$::M'' and $m \trianglelefteq_P i$. □

Lemma 24 (Consistency of •.• ∝ •.•). *If* $\langle md,T\rangle \in_P m$ *and* m::$M.md \propto m'$::$M'.md$*, then* $\langle md,T\rangle \in_P m'$*.*

Proof. The proof is by induction on the length of M. If $M = []$, then $m' = m$ and $M' = []$, because $\triangleright$ (used in the definition of $\propto$) cannot select any chain other than $[m]$.

Otherwise, $m \prec^{\mathsf{m}}_P i$ for some i where $\langle md,T\rangle \in^{\mathsf{i}}_P i$. Since $M/i \triangleright M''/i$ and $\triangleright$ preserves well-formedness of views, M'' is of the form m''::M''' for some m'' and M''' where $m'' \prec^{\mathsf{m}}_P i$. By MixinsImplementAll, MixinMethodsOk, $m'' \prec^{\mathsf{m}}_P i$, and $\langle md,T\rangle \in^{\mathsf{i}}_P i$, we have $\langle md,T\rangle \in_P m''$. Finally, m''::$M'''.md \propto m'$::M', so $\langle md,T\rangle \in_P m'$ by induction. □

Lemma 25 (Soundess of $\bullet \in^{\mathsf{rt}}_P \bullet$ in •). *If* $\langle md,T,\_\_,\_\_,m$::$M/m\rangle \in^{\mathsf{rt}}_P M_v$ *in* M_o*,* $\mathrm{WF}(M_v/t_v)$*, and* $\langle md,T'\rangle \in_P t_v$*, then* $T = T'$*.*

Proof. To get $\langle md,T,\_\_,\_\_,m$::$M/m\rangle$, the $\in^{\mathsf{rt}}_P$ relation first finds m_x::M_x such that $\langle md,T''\rangle \in_P m_x$. Since $\mathrm{WF}(M_v/t_v)$ and $\langle md,T'\rangle \in_P t_v$, then $T'' = T'$ by reasoning about the possible forms of t:

Case t_v is an interface i. Then, $\mathrm{WF}(M_v/t_v)$ implies m_x::$M_x = M_v$ and $m_x \trianglelefteq_P t$. Since $m_x \trianglelefteq_P t$, $T'' = T'$ by MixinMethodsOk and MixinsImplementAll.

Case t_v is an atomic mixin m'. Then, $\mathrm{WF}(M_v/t_v)$ implies m_x::$M_x = M_v$ and $m_x = t$, so $T'' = T'$ by MethodOncePerMixin.

Case t_v is a composite mixin m'. Then, $\langle md,T'\rangle \in_P t_v$ implies $\langle md,T'\rangle \in^{\mathsf{m}}_P m''$ for some m'' where $m' \leq_P m''$, which means that some canditate for m_x exists within m'. Furthermore, $\langle md,T'''\rangle \in^{\mathsf{m}}_P m''$ where $m' \leq_P m''$ implies $T''' = T''$ by the definition of $\in^{\mathsf{m}}_P$ (because $\in^{\mathsf{m}}_P$ eliminates ambiguities), so every candidate for m_x with m' gives the same type to method md. Since M_v starts with the chain of atomic mixins of m', then m_x must be part of m'. Finally, $T'' = T'$ because $m' \leq_P m_x$.

Next, $\in^{\mathsf{rt}}_P$ finds an M_b such that m_x::$M_x.md \propto M_b.md$. From $\mathrm{WF}(m_x$::$M_x/m_x)$, $\langle md,T'\rangle \in_P m_x$, and Lemma 24, the first element of M_b must be an atomic mixin m_b such that $\langle md,T'\rangle \in_P m_b$. Finally, the $\in^{\mathsf{rt}}_P$ relation selects a m::M such that $\langle md,T,\_\_,\_\_\rangle \Subset^{\mathsf{m}}_P m$ and m::$M.md \propto M_b.md$. Thus, by Lemma 24 again, $\langle md,T\rangle \in_P m_b$. Since m_b is an atomic mixin, $\in^{\mathsf{m}}_P$ implies $\Subset^{\mathsf{m}}_P$, so we have both $\langle md,T,\_\_,\_\_\rangle \Subset^{\mathsf{m}}_P m_b$ and $\langle md,T',\_\_,\_\_\rangle \Subset^{\mathsf{m}}_P m_b$. By MethodOncePerMixin, those must be the same method in m_b, so $T = T'$. □