A Programmer's Reduction Semantics for Classes and Mixins

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

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

```
interface Placei ...
interface Barrieri ...
interface Doori
  extends Placei, Barrieri ...
...
class Doorc extends Object
    implements Doori {
  ...
  Roomc Enter(Personc p) { ... }
  ...
}
class LockedDoorc extends Doorc ...
class ShortDoorc extends Doorc ...
```

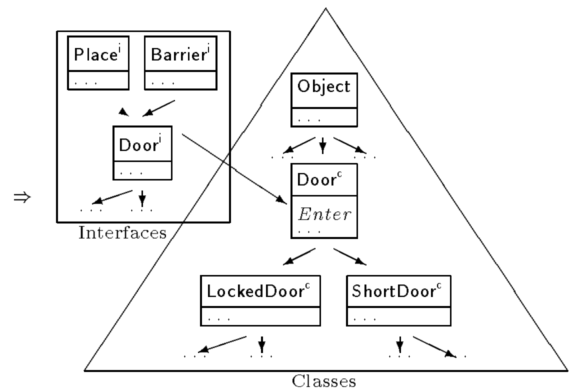

Fig. 1. A program determines a static directed acyclic graph of types

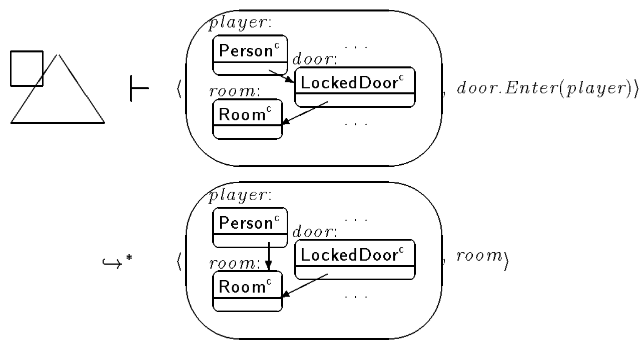

Fig. 2. Given a type graph, reductions map a store-expression pair to a new pair

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 [6] 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 [15], 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 [14], a Scheme programming environment. However, a thorough analysis 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 Java programming. 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.

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 [13, 19, 29]. Figures 1 and 2 illustrate 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

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

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

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

representation of the store as a graph of objects. Each object in the store is a class-tagged record of field values, where the tag indicates the run-time type 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 [12, 19].

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. In the evaluation language for ClassicJava, field uses and **super** invocations are annotated by the type-checker with extra information (see the underlined parts of the syntax). Finally, the **view** and **let** forms represent Java's casting expressions and local variable bindings, respectively.

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 of the form $(t \ldots \longrightarrow t)$, V is used for variable lists of the form $(var \ldots)$, and Γ is used for environments mapping variables to types. Ellipses on the baseline $(\ldots)$ indicate a repeated pattern or continued sequence, while centered ethe ellipses $(\cdots)$ indicate arbitrary missing program text (without straddling a class or interface definition).

ClassesOnce(P)	Each class name is declared only once $\forall c,c'$ **class** $c \cdots$ **class** $c' \cdots$ is in $P \implies c \neq c'$
FieldOncePerClass(P)	Field names in each class declaration are unique $\forall fd,fd'$ **class** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$
MethodOncePerClass(P)	Method names in each class declaration are unique $\forall md,md'$ **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 $\forall i,i'$ **interface** $i \cdots$ **interface** $i' \cdots$ is in $P \implies i \neq i'$
InterfacesAbstract(P)	Method declarations in an interface are **abstract** $\forall md,e$ **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 $\forall c,c',e,e',md,T,T',V,V'$ $(\langle md,\ T,\ V,\ e\rangle \in\!\!\in^{c}_{P} c$ 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. $\langle md,\ T,\ V',\ e'\rangle \in\!\!\in^{c}_{P} c''\})$

Table continues in Figure 5.

Fig. 4. Predicates and relations in the model of ClassicJava

A valid ClassicJava program satisfies a number of simple predicates and relations; these are described in Figure 4. For example, the predicate ClassesOnce(P) 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) $\in\!\!\in^{c}_{P}$ relations capture the field and method declarations of P.

$\leq^{\mathsf{i}}_P$	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
	$\mathrm{rng}(\prec^{\mathsf{i}}_P) \cup \mathrm{rng}(\prec\!\!\prec^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}$
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 $c' \prec\!\!\prec^{\mathsf{c}}_P i'$
InterfaceMethodsOK(P)	Redeclarations of methods are consistent
$\forall i, i', md, T, T', V, V'$	
$\langle md, T, V, \textbf{abstract}\rangle \in^{\mathsf{i}}_P i$ and $\langle md, T', V', \textbf{abstract}\rangle \in^{\mathsf{i}}_P i' \implies (T = T'$ or $i \not\leq^{\mathsf{i}}_P i')$	
$\in^{\mathsf{i}}_P$	Method is contained in an interface
	$\langle md, T, V, \textbf{abstract}\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md, T, V, \textbf{abstract}\rangle \in^{\mathsf{i}}_P i'$
ClassesImplementAll(P)	Classes supply methods to implement interfaces
$\forall i, c\ c \prec\!\!\prec^{\mathsf{c}}_P i \implies (\forall md, T, V$	$\langle md, T, V, \textbf{abstract}\rangle \in^{\mathsf{i}}_P i \implies \exists e, V'$ s.t. $\langle md, T, V', e\rangle \in^{\mathsf{c}}_P c)$
NoAbstractMethods(P, c)	Class has no **abstract** methods (can be instantiated)
	$\forall md, T, V, e\ \ \langle md, T, V, e\rangle \in^{\mathsf{c}}_P c \implies e \neq \textbf{abstract}$
$\leq_P$	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 continued from Figure 4

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 $\leq^{\mathsf{c}}_P$, 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 $\in^{\mathsf{c}}_P$ relations. The $\in^{\mathsf{c}}_P$ 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 $\prec^{\mathsf{i}}_P$ induces a subinterface relation $\leq^{\mathsf{i}}_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The methods of an interface, as described by $\in^{\mathsf{i}}_P$, are 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 $\prec\!\!\prec^{\mathsf{c}}_P$ 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.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{llll}\textsc{ClassesOnce}(P) & \textsc{InterfacesOnce}(P) & \textsc{MethodOncePerClass}(P) & \textsc{FieldOncePerClass}(P) \\ \textsc{CompleteClasses}(P) & \textsc{WellFoundedClasses}(P) & \textsc{CompleteInterfaces}(P) & \textsc{WellFoundedInterfaces}(P) \\ \textsc{ClassFieldsOK}(P) & \textsc{ClassMethodsOK}(P) & \textsc{InterfaceMethodsOK}(P) & \textsc{InterfacesAbstract}(P) \\ \textsc{ClassesImplementAll}(P) & \multicolumn{2}{c}{P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n]} & P,[\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \multicolumn{4}{l}{\text{where } P = defn_1 \ldots defn_n\ e}\end{array}}{\vdash_{\mathsf{p}} defn_1 \ldots defn_n\ e \Rightarrow defn'_1 \ldots defn'_n\ e' : t}[\mathbf{prog}^{\mathsf{c}}]$$

$\vdash_{\mathsf{d}}$

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

$$\frac{P, i \vdash_{\mathsf{m}} meth_j \Rightarrow meth'_j \text{ for each } j \in [1,p]}{P \vdash_{\mathsf{d}} \mathbf{interface}\ i \cdots \{\ meth_1 \ldots meth_p\ \} \Rightarrow \mathbf{interface}\ i \cdots \{\ meth'_1 \ldots meth'_p\ \}}[\mathbf{defn}^{\mathsf{i}}]$$

$\vdash_{\mathsf{m}}$

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

$\vdash_{\mathsf{e}}$

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

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

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

Rules continue in Figure 7

Fig. 6. Context-sensitive checks and type elaboration rules for ClassicJava

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
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e has type t using subsumption
$P \vdash_{\mathsf{t}} t$	t exists

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 annotation 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 annotation contains the compile-time

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

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

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

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

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

$\vdash_{\mathrm{s}}, \vdash_{\mathrm{t}}$

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

Fig. 7. Rules continued from Figure 6

type of **this**, which determines the class that contains the declaration of the method to be invoked.

The complete typing rules are shown in Figure 6. 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 $\mathbf{get}^{\mathsf{c}}$ and $\mathbf{set}^{\mathsf{c}}$ 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 $\mathbf{set}^{\mathsf{c}}$ 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 *object*s to class-tagged field records. A field record is a mapping from elaborated field names to values. The evaluation rules are a straightforward modification of those for imperative Scheme [13].

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

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

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle & [\mathit{new}] \\
\quad \text{where } \mathit{object} \notin \mathrm{dom}(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.\mathit{fd} \mapsto \mathsf{null} \mid c \leq^{\mathsf{c}}_P c' \text{ and } \exists t \text{ s.t. } \langle c'.\mathit{fd}, t\rangle \in^{\mathsf{c}}_P c'\} & \\
P \vdash \langle \mathsf{E}[\mathit{object}\underline{\ :\ c'\ }.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle & [\mathit{get}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(c'.\mathit{fd}) = v & \\
P \vdash \langle \mathsf{E}[\mathit{object}\underline{\ :\ c'\ }.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle & [\mathit{set}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle & [\mathit{call}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c & \\
P \vdash \langle \mathsf{E}[\mathbf{super}\underline{\ \equiv\ \mathit{object}\ :\ c'\ }.\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle & [\mathit{super}] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle & \\
\quad \text{where } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}\rangle & [\mathit{cast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ \mathit{var} = v\ \mathbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v/\mathit{var}]], \mathcal{S}\rangle & [\mathit{let}] \\
 & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \textsf{error: bad cast}, \mathcal{S}\rangle & [\mathit{xcast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \not\leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\ :\ c\ }.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S}\rangle & [\mathit{nget}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\ :\ c\ }.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S}\rangle & [\mathit{nset}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \textsf{error: dereferenced null}, \mathcal{S}\rangle & [\mathit{ncall}]
\end{array}
$$

Fig. 8. Operational semantics for ClassicJava

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 get stuck. This property can be formulated as a type soundness theorem.

Theorem 1 (Type Soundness). *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = \mathit{defn}_1 \ldots \mathit{defn}_n\ e$, *then either*

- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathrm{object}, \mathcal{S}\rangle$ *and* $\mathcal{S}(\mathrm{object}) = \langle t', \mathcal{F}\rangle$ *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) [29]. 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).

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

Note that the environment may contain bindings for lexical variables, which are not store objects.

Since the rewriting rules reduce *annotated* terms, we derive new type judgements that relate annotated terms. Each of the new rules performs exactly the same checks as the rule it is derived from, but does not add any annotation. Thus $\vdash_{\underline{\mathsf{s}}}$ is derived from $\vdash_{\mathsf{s}}$, and so forth. Only the judgement on expressions ($\vdash_{\underline{\mathsf{e}}}$) is altered slightly: we retain the **view** operation in all cases and elide the [$\mathbf{wcast}^{\mathsf{c}}$] relation, which is only an optimization that removes an unnecessary check. This relaxation obviously does not change the type-checking or extensional behavior of any programs.

The following lemmata are used to prove the main lemma.

Lemma 3 (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. This follows by reasoning about the shape of the derivation. □

Lemma 4 (Replacement). *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. This follows by a replacement argument in the derivation tree. □

Lemma 5 (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 $[v_1/var_1, \ldots\; v_n/var_n]$, and $e' = \sigma(e)$. The proof proceeds by induction over the shape of the derivation showing that $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e : t$. We perform a case analysis on the last step.

Case $e = \mathbf{new}\ c$. $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e : c$ and $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : c$.

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

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

Case $e = e_1\underline{\ :\ c\ }.fd$. $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e_1 : t'$ and $\langle c.fd,t\rangle \in_P t'$ follow from the antecedents. By induction, $P,\Gamma \vdash_{\underline{\mathsf{s}}} \sigma(e_1) : t'$. Therefore $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_1) : t''$, where t'' is a sub-type of t'. Hence, $\langle c.fd,t\rangle \in_P t''$. Thus $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_1)\underline{\ :\ c\ }.fd : t$.

Case $e = e_1\underline{\ :\ c\ }.fd = e_2$. This case is similar to the one above.

Case $e = \textbf{view}\ t\ e_1$. $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e_1 : t'$ and $t \leq_P t'$ follow from the antecedent. Inductively, $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_1) : t''$ for $t'' \leq_P t'$. If $t \leq_P t''$ or $t'' \leq_P t$, P, $\Gamma \vdash_{\underline{\mathsf{e}}}$ $\textbf{view}\ t\ \sigma(e_1) : t$ (by our relaxed [$\textbf{ncast}^{\mathsf{c}}$] rule).

Case $e = \textbf{let}\ var = e_1\ \textbf{in}\ e_2$. Let $\sigma_1 = \sigma$. From [**let**], we get $P,\Gamma\sigma_1 \vdash_{\underline{\mathsf{e}}} e_1 : t_1$. Let σ_2 be the substitution $[var : t_1]$. Then $P,\Gamma\sigma_2\sigma_1 \vdash_{\underline{\mathsf{e}}} e_2 : t$. By induction, $P,\Gamma \vdash_{\underline{\mathsf{s}}} \sigma_1(e_1) : t_1$ and $P,\Gamma\sigma_2 \vdash_{\underline{\mathsf{s}}} \sigma_1(e_2) : t$. By using Lemma 6 for each term, $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma_1(\textbf{let}\ var = e_1\ \textbf{in}\ e) : t$.

Case $e = e_0.md\,(e_1, \ldots\ e_n)$. Typability of the expression implies $P,\Gamma\sigma \vdash_{\underline{\mathsf{s}}} e_i : t_i$ for $i \in [1,n]$ and $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e_0 : t_0$ where $\langle md, (t_1 \ldots\ t_n \rightarrow t), (var_1, ..., var_n), e\rangle$ $\in^{\mathsf{c}}_P t_0$. 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 $t_0{}' \leq_P t_0$, which implies that $\langle md, (t_1 \ldots\ t_n \rightarrow t), (var_1, ..., var_n), e\rangle \in^{\mathsf{c}}_P t_0{}'$. Thus $P,\Gamma \vdash_{\underline{\mathsf{e}}} \sigma(e_0.md\,(e_1, \ldots\ e_n)) : t$.

Case $e = \textbf{super}\underline{\equiv \textbf{this} : c\ }.md\,(e_1, \ldots\ e_n)$. This follows in a similar fashion to the rule above. Since the class c is embedded in the expression, and the induction yields a subtype of the original type, **this** can be subsumed appropriately to instantiate the method in the superclass. □

Lemma 6. *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 doesn't matter: the result is the type of the field.

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

Case $v\underline{\ :\ c\ }.fd = \bullet$. Since $t'' \leq_P t'$, the type of $\bullet$ is t' by subsumption and the type of the expression is unchanged.

Case $\bullet.md(e \ldots)$. Since $t'' \leq_P t'$ and methods in an inheritance chain must preserve the type, the result of method application is the same type.

Case $v.md(v \ldots \bullet\ e \ldots)$. By subsumption, arguments have the declared type by [**meth**]; t'' can be t' by subsumption.

Case $\textbf{super}\underline{\equiv v : c\ }.md\,(v \ldots \bullet\ e \ldots)$. Analogous to the previous case.

Case $\textbf{view}\ t\ \bullet$. The type of this expression is the same regardless of $\bullet$. Since $\vdash_{\underline{\mathsf{e}}}$ has the less restrictive condition for [$\textbf{ncast}^{\mathsf{c}}$] that t and t'' be comparable by $\leq_P$, the typing proceeds even if t'' is a subtype of t.

Case **let** $var = \bullet$ **in** e_2. We are given $P,\Gamma \vdash_{\underline{e}} e : t'$, so from [**let**], $P,\Gamma\sigma_1 \vdash_{\underline{e}} e_2 : t_1$ for some type t_1 where σ_1 is $[var : t']$. We must show that $P,\Gamma\sigma_2 \vdash_{\underline{s}} e_2 : t_1$ where $\sigma_2 = [var : t'']$. This follows from Lemma 8. □

Definition 7. $\Gamma \leq_\Gamma \Gamma'$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall\ v \in \mathrm{dom}(\Gamma)$, $\Gamma'(v) \leq_P \Gamma(v)$.

Lemma 8. *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 simple adaptation of that of Lemma 5. □

We can now prove the subject reduction lemma. Since ClassicJava does not include any primitives, its type soundness follows by induction over this result.

Definition 9 (Error Configuration). An *error configuration* is any one of [*xcast*], [*nget*], [*nset*] and [*ncall*].

Lemma 10 (Subject Reduction). *If $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_\sigma \mathcal{S}$, $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$, and $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$, then e' is an answer, e' is an error configuration, or $\exists\ \Gamma'$ such that*

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

Proof. The proof examines the structure of the reduction step. For each case, we construct the new environment Γ' and show that, if execution has not halted with an answer or in an error configuration, 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 $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ c] : t$. From Σ_5, $object \notin \mathrm{dom}(\mathcal{S}) \Rightarrow object \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ c] : t$ by Lemma 3. Since $P,\Gamma' \vdash_{\underline{e}} \mathbf{new}\ c : c$ and $P,\Gamma' \vdash_{\underline{e}} object : c$ we use Lemma 4 to get $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[object] : t$.
2. Let $\mathcal{S}'(object) = \langle c,\mathcal{F}\rangle$. *object* is the only new element in $\mathrm{dom}(\mathcal{S}')$.
   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*.

Case [*get*]. Set $\Gamma' = \Gamma$. Let t' be the type such that $P,\Gamma \vdash_{\underline{e}} object\underline{\ :c'\ }.fd : t'$. $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[object\underline{\ :c'\ }.fd] : t$ implies that $\Gamma(object) \leq_P c'$. Thus $\mathcal{S}(object) = \langle c,\mathcal{F}\rangle$ with $c'.fd \in \mathrm{dom}(\mathcal{F})$.

1. If v is null, it can be cast to t', so $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[v] : t$ by Lemma 4. If v is not null, by Σ_4, $\mathcal{S}(v) = \langle c'',\_\rangle$ where $c'' \leq_P t'$. By Lemma 6, $P,\Gamma' \vdash_{\underline{s}} \mathsf{E}[v] : t$.

2. $\mathcal{S}$ and Γ are unchanged.

Case [*set*].

1. The proof is by a straight-forward 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. Assume v is non-null.
   - Σ_3**:** Since v is typable, it must be in $\mathrm{dom}(\Gamma)$. By Σ_5, it is therefore in $\mathrm{dom}(\mathcal{S})$.
   - Σ_4**:** The typing of the active expression indicates 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.

Case [*call*]. 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 $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e\rangle \in^{\mathsf{c}}_P t'$. The type-checking of P proves that $P,t_0 \vdash_{\underline{\mathsf{m}}} t\ md\ (t_1\ var_1, \ldots\ t_n\ var_n)\ \{e\}$, which implies that $P,[\mathbf{this} : t_0,\ var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{s}}} e : t$ 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).

1. Lemma 5 shows that $P,\Gamma \vdash_{\underline{\mathsf{s}}} e[object/\mathbf{this},\ v_1/var_1, \ldots\ v_n/var_n] : t$.
2. $\mathcal{S}' = \mathcal{S}$ and Γ' does not bind new addresses, so $\vdash_\sigma$ is preserved.

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

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'. Set $\Gamma' = \Gamma\ [var : t']$. From [**let**], $P,\Gamma' \vdash_{\underline{\mathsf{e}}} e : t$.

1. By Lemma 5, $P,\Gamma \vdash_{\underline{\mathsf{s}}} e\ [v/var] : t$.
2. The store is unchanged and the only addition to the environment is not an *object*, so the store relation holds. □

2.5 Related Work on Classes

Our model for class-based object-oriented languages is similar to two recently published semantics for Java [9, 28], but entirely motivated by prior work on Scheme and ML models [13, 19, 29]. 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 [11, 20, 24, 23, 25] 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[3] 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] 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.

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, LockedDoor^c and ShortDoor^c. An instance of LockedDoor^c requires a key to open the door, while an instance of 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 LockedDoor^c and ShortDoor^c classes cannot be combined to create a new LockedShortDoor^c class for doors that are both locked and short.

```
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

```
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

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 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 a $\mathsf{LockedDoor}^{\mathsf{c}}$, 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, Secure^m. Then, key- or book-specific information is composed with Secure^m to produce Locked^m and Magic^m, as shown in Figure 10. Each of the new mixins extends Door^i since the right hand mixin in the composition, Secure^m, extends Door^i.

The new Locked^m and Magic^m mixins can also be composed to form LockedMagic^m. This mixin has the expected behavior: to open an instance of LockedMagic^m, the player must have both the key and the book of spells. This combinational effect is achieved by a chain of **super**.*canOpen*() calls that use distinct, non-interfering versions of *neededItem*. The *neededItem* declarations of Locked^m and Magic^m do not interfere with each other because the interface extended by Locked^m is Door^i, which does not contain *neededItem*. In contrast, Door^i does contain *canOpen*, so the *canOpen* method in Locked^m overrides and chains to the *canOpen* in Magic^m.

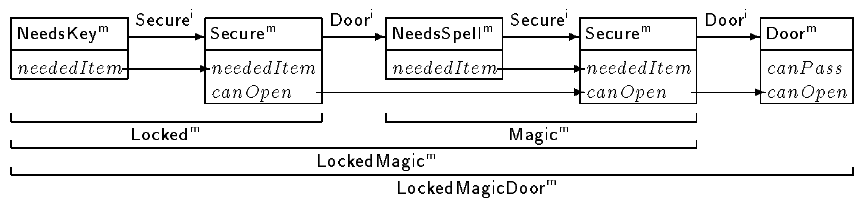

Fig. 11. The LockedMagicDoor^m mixin corresponds to a sequence of atomic mixins

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.[4] 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:

– 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.

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

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.[5]

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.

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.

[5] 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.

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

Fig. 12. Syntax extensions for MixedJava

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 is in the annotations added by type elaboration. First, **view** expressions are annotated with the source type of the expression. Second, a type is no longer included in the **super** annotation. Type elaboration also inserts extra **view** expressions into a program to implement subsumption.

The predicates and relations in Figure 13 (along with the interface-specific parts of Figure 4) 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.

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

$$
\begin{array}{l}
\mathsf{LockedMagicDoor^m}\ \mathit{door} = \textbf{new}\ \mathsf{LockedMagicDoor^m}; \\
(\textbf{view}\ \mathsf{Secure^m}\ \mathit{door}).\mathit{neededItem}();
\end{array}
$$

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

4.2 MixedJava Type Elaboration

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

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

$\textsc{MixinsOnce}(P)$	Each mixin name is declared only once $\forall m, m'$ **mixin** $m \cdots$ **mixin** $m' \cdots$ is in $P \Longrightarrow m \neq m'$
$\textsc{FieldOncePerMixin}(P)$	Field names in each mixin declaration are unique $\forall fd, fd'$ **mixin** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \Longrightarrow fd \neq fd'$
$\textsc{MethodOncePerMixin}(P)$	Method names in each mixin declaration are unique $\forall md, md'$ **mixin** $\cdots \{ \cdots md (\cdots) \{ \cdots \} \cdots md' (\cdots) \{ \cdots \} \cdots \}$ is in $P \Longrightarrow md \neq md'$
$\textsc{NoAbstractMixins}(P)$	Methods in a mixin are not **abstract** $\forall md, e$ **mixin** $\cdots \{ \cdots md (\cdots) \{ e \} \cdots \}$ is in $P \Longrightarrow e \neq$ **abstract**
$\prec^{\mathsf{m}}_{P}$	Mixin declares an inheritance interface $m \prec^{\mathsf{m}}_{P} i \Leftrightarrow$ **mixin** m **extends** $i \cdots \{ \cdots \}$ is in P
$\lll^{\mathsf{m}}_{P}$	Mixin declares implementation of an interface $m \lll^{\mathsf{m}}_{P} i \Leftrightarrow$ **mixin** $m \cdots$ **implements** $\cdots i \cdots \{ \cdots \}$ is in P
$\bullet \doteq^{\mathsf{m}}_{P} \bullet \circ \bullet$	Mixin is declared as a composition $m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Leftrightarrow$ **mixin** m **=** m' **compose** m'' is in P
$\in\!\!\in^{\mathsf{m}}_{P}$	Method is declared in a mixin $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e \rangle \in\!\!\in^{\mathsf{m}}_{P} m$ $\Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n) \{ e \} \cdots \}$ is in P
$\in\!\!\in^{\mathsf{m}}_{P}$	Field is declared in a mixin $\langle m.fd, t \rangle \in\!\!\in^{\mathsf{m}}_{P} m \Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\leq^{\mathsf{m}}_{P}$	Mixin is a submixin $m \leq^{\mathsf{m}}_{P} m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_{P} m'' \circ m'''$ and $(m'' \leq^{\mathsf{m}}_{P} m'$ or $m''' \leq^{\mathsf{m}}_{P} m'))$
$\trianglelefteq^{\mathsf{m}}_{P}$	Mixin is viewable as a mixin $m \trianglelefteq^{\mathsf{m}}_{P} m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_{P} m'' \circ m'''$ and $(m'' \trianglelefteq^{\mathsf{m}}_{P} m'$ xor $m''' \trianglelefteq^{\mathsf{m}}_{P} m'))$
$\textsc{CompleteMixins}(P)$	Mixins that are composed are defined $\mathrm{rng}(\doteq^{\mathsf{m}}_{P}) \subseteq \{ m \circ m' \mid m, m' \in \mathrm{dom}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_{P}) \}$
$\textsc{WellFoundedMixins}(P)$	Mixin hierarchy is an order $\leq^{\mathsf{m}}_{P}$ is antisymmetric
$\textsc{CompleteInterfaces}(P)$	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_{P}) \cup \mathrm{rng}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{rng}(\lll^{\mathsf{m}}_{P}) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_{P}) \cup \{\mathsf{Empty}\}$
$\dashv^{\mathsf{m}}_{P}$	Mixin extends an interface $m \dashv^{\mathsf{m}}_{P} i \Leftrightarrow m \prec^{\mathsf{m}}_{P} i$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $m'' \dashv^{\mathsf{m}}_{P} i)$
$\ll^{\mathsf{m}}_{P}$	Mixin implements an interface $m \ll^{\mathsf{m}}_{P} i \Leftrightarrow \exists m', i'$ s.t. $m \leq^{\mathsf{m}}_{P} m'$ and $i' \leq^{\mathsf{i}}_{P} i$ and $(m' \prec^{\mathsf{m}}_{P} i'$ or $m' \lll^{\mathsf{m}}_{P} i')$
$\ll\!\!\lhd^{\mathsf{m}}_{P}$	Mixin is viewable as an interface $m \ll\!\!\lhd^{\mathsf{m}}_{P} i \Leftrightarrow (\exists i'$ s.t. $i \leq^{\mathsf{i}}_{P} i'$ and $(m \prec^{\mathsf{m}}_{P} i'$ or $m \lll^{\mathsf{m}}_{P} i'))$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $(m' \ll\!\!\lhd^{\mathsf{m}}_{P} i$ xor $m'' \ll\!\!\lhd^{\mathsf{m}}_{P} i))$
$\textsc{MixinCompositionsOK}(P)$	Mixins are composed safely $\forall m, m', m''\ m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Longrightarrow \exists i\ s.t.\ m' \dashv^{\mathsf{m}}_{P} i$ and $m'' \ll\!\!\lhd^{\mathsf{m}}_{P} i$
:: and @	Sequence constructors :: adds an element to the beginning of a sequence; @ appends two sequences
$\longrightarrow_{P}$	Mixin corresponds to a chain of atomic mixins $m \longrightarrow_{P} M$ $\Leftrightarrow (\exists i$ s.t. $m \prec^{\mathsf{m}}_{P} i$ and $M = [m])$ or $(\exists m', m'', M', M''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $m' \longrightarrow_{P} M'$ and $m'' \longrightarrow_{P} M''$ and $M = M'@M'')$
$\leq^{\mathsf{M}}$	Views have an inverted subsequence order $M \leq^{\mathsf{M}} M' \Leftrightarrow \exists M''$ s.t. $M = M''@M'$

Table continues in Figure 14.

Fig. 13. Predicates and relations in the model of MixedJava

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.

MIXINMETHODSOK(P) Method definitions match inheritance interface

$\forall m, i, e, md, T, T', V, V'$

$(\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_{P} m$ and $\langle md,\ T',\ V',\ \mathbf{abstract}\rangle \in^{\mathsf{m}}_{P} i) \Longrightarrow (T = T'$ or $m \not\prec^{\mathsf{m}}_{P} i)$

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

$\langle m'.fd,\ t\rangle \in^{\mathsf{m}}_{P} m$

$\Leftrightarrow \exists M, M'$ s.t. $m \longrightarrow_{P} M$ and $\langle m'.fd,\ t\rangle \in^{\mathsf{m}}_{P} m'$

and $\{m'::M'\} = \{m'::M' \mid M \leq^{\mathsf{M}} m'::M'$ and $\exists t'$ s.t. $\langle m'.fd,\ t'\rangle \in^{\mathsf{m}}_{P} m'\}$

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

$\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_{P} m$

$\Leftrightarrow \exists M, m', M'$ s.t. $m \longrightarrow_{P} M$ and $\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_{P} m'$

and $\{m'::M'\} = \{m'::M' \mid M \leq^{\mathsf{M}} m'::M'$ and $\exists V', e'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \in^{\mathsf{m}}_{P} m'\}$

MIXINSIMPLEMENTALL(P) Mixins supply methods to implement interfaces

$\forall m, i\ m \lll^{\mathsf{m}}_{P} i \Longrightarrow (\forall md, T\ \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_{P} i$

$\Longrightarrow (\exists\ e$ s.t. $\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_{P} m$

or $\exists\ i'\ s.t.\ (m \prec^{\mathsf{m}}_{P} i'$

and $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_{P} i')))$

$\leq_P$	Type is a subtype	$\leq_P \equiv \leq^{\mathsf{m}}_{P} \cup \leq^{\mathsf{i}}_{P} \cup \ll^{\mathsf{m}}_{P}$
$\trianglelefteq_P$	Type is viewable as another type	$\trianglelefteq_P \equiv \trianglelefteq^{\mathsf{m}}_{P} \cup \leq^{\mathsf{i}}_{P} \cup \lll^{\mathsf{m}}_{P}$
$\in_P$	Field or method is in a type	$\in_P \equiv \in^{\mathsf{m}}_{P} \cup \in^{\mathsf{i}}_{P}$

$\bullet/\bullet \triangleright \bullet$ Mixin selects a view in a chain

$M/m \triangleright M' \Leftrightarrow \{M'\} = \{M''@M''' \mid m \longrightarrow_{P} M''$ and $M \leq^{\mathsf{M}} M''@M'''\}$

$\bullet/\bullet \triangleright \bullet$ Interface selects a view in a chain

$M/i \triangleright M' \Leftrightarrow M' = \min\{m::M'' \mid m \prec^{\mathsf{m}}_{P} i$ and $M \leq^{\mathsf{M}} m::M''\}$

$\bullet/\bullet \propto \bullet$ Method in a sequence is the same as in a subsequence

$m::M/md \propto M' \Leftrightarrow$ $m::M = M'$ or $(\exists i, T, V, M''$ s.t. $m \prec^{\mathsf{m}}_{P} i$ and $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_{P} i$

and $M'/i \triangleright M''$ and $M''/md \propto M')$

$\bullet \in^{\mathsf{rt}}_{P} \bullet$ in $\bullet$ Method and view is selected by a view in a chain

$\langle md,\ T,\ V,\ e,\ m::M\rangle \in^{\mathsf{rt}}_{P} M_v$ in M_o

$\Leftrightarrow \langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_{P} m$ and $M_b = \max\{M' \mid M_v/md \propto M'\}$

and $\{m::M\} = \{m::M \mid m::M \leq^{\mathsf{M}} M_o$ and $m::M/md \propto M_b$

and $\exists V', e'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \in^{\mathsf{m}}_{P} m\}$

Fig. 14. Predicates and relations continued from Figure 13

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.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{llll}\textsc{MixinsOnce}(P) & \textsc{MethodOncePerMixin}(P) & \textsc{InterfacesOnce}(P) & \textsc{CompleteMixins}(P) \\ \textsc{WellFoundedMixins}(P) & \textsc{CompleteInterfaces}(P) & \textsc{WellFoundedInterfaces}(P) & \textsc{MixinFieldsOK}(P) \\ \textsc{MixinMethodsOK}(P) & \textsc{InterfaceMethodsOK}(P) & \textsc{InterfacesAbstract}(P) & \textsc{NoAbstractMixins}(P) \\ \textsc{MixinsImplementAll}(P) & P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n] & & P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \text{where } P = defn_1 \ldots defn_n \; e & & & \end{array}}{\vdash_{\mathsf{p}} defn_1 \ldots defn_n \; e \Rightarrow defn'_1 \ldots defn'_n \; e' : t}[\mathbf{prog}^{\mathsf{m}}]$$

$\vdash_{\mathsf{d}}$

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

$\vdash_{\mathsf{e}}$

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

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

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

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

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

$\vdash_{\mathsf{s}}$

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

$\vdash_{\mathsf{t}}$

$$\frac{t \in \mathrm{dom}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_{P}) \cup \mathrm{dom}(\prec^{\mathsf{i}}_{P}) \cup \{\mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}[\mathbf{type}^{\mathsf{m}}]$$

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

Our MixedJava semantics handles the duplication of method names with run-time context information: the current *view* of an object.[6] 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.

A view is represented as a chain of mixins. This chain is always a tail of the object's full chain of mixins, *i.e.*, the chain of mixins for the object's instantiation type. The tail designates a specific point in the full mixin chain

[6] A view is analogous to a "subobject" in languages with multiple inheritance, but without the complexity of shared superclasses [26].

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

$$
\begin{array}{lr}
P \vdash \langle \mathsf{E}[\mathbf{new}\ m], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| 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 & [new] \\
\quad \text{where } object \notin \mathrm{dom}(\mathcal{S}) \text{ and } m \longrightarrow_P M & \\
\qquad\quad \{M_1.fd_1, \ldots M_n.fd_n\} = \{m'{::}M'.fd \mid M \leq^{\mathsf{M}} m'{::}M' \quad \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 \rangle\underline{\;:\; m'\;}.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 } M/m' \triangleright M' \text{ and } \mathcal{F}(M'.fd) = v & \\
P \vdash \langle \mathsf{E}[\langle object \| M \rangle\underline{\;:\; m'\;}.fd = v], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], 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 } M/m' \triangleright M' & \\
P \vdash \langle \mathsf{E}[\langle object \| M \rangle.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [call] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\langle object \| 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\quad \text{and } \langle md, T, (var_1 \ldots var_n), e, M' \rangle \in^{\mathsf{rt}}_P M \text{ in } M_o & \\
P \vdash \langle \mathsf{E}[\mathbf{super}\underline{\;\equiv \langle object \| m{::}M \rangle\;}.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [super] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\langle object \| M' \rangle/\mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } m \prec\!\!\!\mid^{\mathsf{m}}_P i \text{ and } M/i \triangleright M'' \text{ and } \langle md, T, (var_1 \ldots var_n), e, 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 \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M' \rangle], \mathcal{S} \rangle & [view] \\
\quad \text{where } t' \trianglelefteq_P t \text{ and } M/t \triangleright M' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\;t'\ \mathbf{as}\;} t\ \langle object \| M \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M'' \rangle], \mathcal{S} \rangle & [cast] \\
\quad \text{where } t' \ntrianglelefteq_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'/t \triangleright M'' & \\
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] \\
 & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\;t'\ \mathbf{as}\;} t\ \langle object \| M \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S} \rangle & [xcast] \\
\quad \text{where } t' \ntrianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \ntrianglelefteq_P t & \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\;:\; m\;}.fd], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S} \rangle & [nget] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\;:\; m\;}.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. 16. Operational semantics for MixedJava

for selecting methods during dynamic dispatch. For example, when an instance of $\mathsf{LockedMagicDoor^m}$ is used as a $\mathsf{Magic^m}$ instance, the view of the object is $[\mathsf{NeedsSpell^m\ Secure^m\ Door^m}]$. With this view, a search for the *neededItem* method of the object begins in the $\mathsf{NeedsSpell^m}$ element of the chain.

The first phase of a search for some 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 an overriding declaration of x for this base that is contained in the object's instantiated mixin. Among the declarations that override this base, the leftmost declaration is selected as the destination. The location of that overriding declaration determines both the method definition that is invoked and the view of the object (*i.e.*, the representation of **this**) within the destination method body. This dispatching algorithm is encoded in the $\in^{\mathsf{rt}}_P$ relation.

Let us apply the algorithm to $o.getNeeded()$ in the following example:

$$
\begin{array}{l}
\textbf{mixin}\ \mathsf{Getter^m}\ \textbf{extends}\ \mathsf{Empty}\ \{ \\
\quad \mathsf{Object}\ \mathit{get}(\mathsf{Secure^m}\ o)\ \{\ o.\mathit{neededItem}()\ \} \\
\} \\
\textbf{let}\ \mathit{door} = \textbf{new}\ \mathsf{LockedMagicDoor^m} \\
\ \textbf{in let}\ g = \textbf{new}\ \mathsf{Getter^m} \\
\quad \textbf{in}\ g.\mathit{get}(\underline{\textbf{view}\ \mathsf{Secure^m}}\ \textbf{view}\ \mathsf{Locked^m}\ \mathit{door}); \\
\qquad g.\mathit{get}(\underline{\textbf{view}\ \mathsf{Secure^m}}\ \textbf{view}\ \mathsf{Magic^m}\ \mathit{door})
\end{array}
$$

For the first call to *g.get*, *o* is replaced by a reference with the view [$\mathsf{Secure^m}$ $\mathsf{NeedsSpell^m}$ $\mathsf{Secure^m}$ $\mathsf{Door^m}$]. 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 *g.get* receives a reference with the view [$\mathsf{Secure^m}$ $\mathsf{Door^m}$]. 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 $\in_P^{\mathsf{rt}}$ 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, like the class semantics described in Section 2.3. In this semantics, an *object* in the store is tagged with a mixin instead of a class, and the values are null and $\langle \mathit{object} \| \mathit{view} \rangle$ 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. To prove the soundness theorem, we introduce a conservative extension, MixedJava′, which is defined by revising some of the MixedJava relations (see Figure 17).

In the extended language, the subtype relation is used directly for the "viewable as" relation without eliminating ambiguities. Thus, MixedJava′ allows coercions and method calls that are rejected as ambiguous in MixedJava. This makes MixedJava′ less suitable as a programming language, but simplifies the proof of a type soundness theorem. The soundness theorem for MixedJava′ applies to MixedJava by the following two lemmas:

$\leq_P$	Type is a subtype
Extended for views:	$M \leq_P m \Leftrightarrow M$ contains m's sequence;
	$M \leq_P i \Leftrightarrow M$ contains an m s.t. $m \prec_P^{\mathsf{m}} i$
$\trianglelefteq_P$	Type is viewable as another type $\quad \trianglelefteq_P \equiv \leq_P$
$\in_P$	Field or method is contained in a type
	Choose the leftmost field/method instance
$\bullet / \bullet \triangleright \bullet$	Mixin selects a view in a chain
	Choose the leftmost instance in the chain
$\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$	Method and view is selected by a view in a chain
	Choose the minimum view with a method

Fig. 17. Revised relations for MixedJava′

1. Every MixedJava program is a MixedJava′ program.
2. $P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$ in MixedJava
   $\Rightarrow P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$ in MixedJava′.

The proof of the soundness theorem is divided into two parts: we first sketch the soundness of MixedJava′, then show why this result applies to MixedJava.

Type Soundness of MixedJava′ To prove the type soundness of MixedJava′, we must first 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 $\langle object \| M \rangle$ pairs to the mixin type M. The updated consistency relation is defined as follows:

Definition 11 (Environment-Store Consistency).

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

The statements of the theorems and lemmata remain unchanged, but the proofs must be adjusted for differences between the two languages. We show how the subject reduction lemma is updated; the remaining proofs change along similar lines.

To prove the type soundness of MixedJava′, we must establish that field accesses and method invocations that have passed the type-checker will not fail at run-time. The salient differences in the proof of the subject reduction lemma are:

Case [*get*]. The typing rules show that $P,\Gamma \vdash_{\underline{\mathsf{e}}} \langle object||M\rangle \underline{\ : m'\ }.fd : t_1$ where $\langle m'.fd,t_1\rangle \in_P \Gamma(\langle object||M\rangle)$. By Σ_2, *object* has the field $m'.fd$. The rest of the proof follows as for ClassicJava.

Case [*call*]. $\Gamma(\langle object||M\rangle) = M$ combined with [**call$^{\mathsf{m}}$**] shows that the method is in M. The search algorithm seeks out the base class of the method definition, and then the leftmost definition of the method in the instantiated mixin. Since the search algorithm ($\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$) follows interfaces in both directions, we know that the method must exist. Further, both the "downward" and "upward" searches are type-preserving, since method overriding must preserve type (by MixinMethodsOk). Thus, the invoked method must exist and must have the same type. The rest of the proof is similar to that for ClassicJava.

The proof for the remaining language features is similar to the corresponding proofs for ClassicJava.

Relationship Between MixedJava and MixedJava′ Since the revised relations for MixedJava′ are conservative extensions of those for MixedJava, it is easy to see that every MixedJava program is also a MixedJava′ program. What remains to be shown is that for programs common to both languages, their evaluators produce analogous configurations for each reduction step.

The crucial difference between the languages is, for a given expression, which field or method is chosen by the run-time system of each language. Whereas in MixedJava the choice is unique (this is ensured by the "viewable as" relation, $\trianglelefteq_P$), MixedJava′ allows implicit and explicit views that can result in ambiguity, and then chooses the leftmost entity (in the linearization) from the set of options. These differences are captured in the $\in_P$, $\bullet/\bullet \propto \bullet$ and $\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$ relations.

Since we are only concerned with programs common to the two languages, we can ignore programs that select views that result in ambiguity. In the remaining programs there is only one field or method to be picked at each stage, which is also the leftmost choice. Hence the two evaluators coincide by making the same choices. As a result, they compute the same answers, and can be used interchangeably for programs common to the two languages. This establishes that the type soundness of MixedJava′ applies to 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. Nevertheless, this relation 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 [10], which makes it difficult to use CLOS for proper mixin programming. The CommonObjects [27] 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 [5–7]. His system is based on earlier work by Cook [8]; its underlying semantics was recently reformulated in categorical terms by Ancona and Zucca [4]. 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. However, this language 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.

5 Conclusion

We have presented a programming language of mixins that relies on the same 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. However, from the programmer's global perspective, 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.

While using mixins is inherently more expensive than using classes—because mixins enforce the distinction between implementation inheritance and subtyping—the cost is 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 *LNCS*. 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* (New York, N.Y., 1994), D. Sannella, Ed., Lecture Notes in Computer Science 788, Springer Verlag, pp. 1–25.
3. Abadi, M., and Cardelli, L. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development* (May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., Lecture Notes in Computer Science 915, Springer-Verlag, pp. 471–485.
4. Ancona, D., and Zucca, E. An algebraic approach to mixins and modularity. In *Proc. Conference on Algebraic and Logic Programming* (Berlin, 1996), M. Hanus and M. Rodríguez-Artalejo, Eds., Lecture Notes in Computer Science 1139, Springer Verlag, pp. 179–193.
5. Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.
6. 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).
7. Bracha, G., and Lindstrom, G. Modularity meets inheritance. In *Proc. IEEE Computer Society International Conference on Computer Languages* (Washington, DC, Apr. 1992), IEEE Computer Society, pp. 282–290.
8. Cook, W. R. *A Denotational Semantics of Inheritance.* Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.
9. Drossopolou, S., and Eisenbach, S. Java is typesafe – probably. In *Proc. European Conference on Object Oriented Programming* (June 1997).
10. Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic conflict resolution mechanisms for inheritance. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1992), pp. 16–24.
11. Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. Application of OOP type theory: State, decidability, integration. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1994), pp. 16–30.

12. Felleisen, M. Programming languages and lambda calculi. URL: `www.cs.rice.edu/~matthias/411web/mono.ps`.
13. 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.
14. 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* (1997), pp. 369–388.
15. Flatt, M. PLT MzScheme: Language manual. Tech. Rep. TR97-280, Rice University, 1997.
16. Flatt, M., Krishnamurthi, S., and Felleisen, M. Classes and mixins. Tech. Rep. TR97-293, 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 semantics for Standard ML 1996. Submitted for publication, 1997.
20. Kamin, S. Inheritance in smalltalk-80: a denotational definition. In *Proc. ACM Conference 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. 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.
25. 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 789, Springer-Verlag, pp. 321–346.
26. Rossie, J. G., Friedman, D. P., and Wand, M. Modeling subobject-based inheritance. In *Proc. European Conference on Object-Oriented Programming* (Berlin, Heidelberg, and New York, July 1996), P. Cointe, Ed., Lecture Notes in Computer Science 1098, Springer-Verlag, pp. 248–274.
27. Snyder, A. Inheritance and the development of encapsulated software components. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 165–188.
28. Syme, D. Proving Java type soundness. Tech. Rep. 427, University of Cambridge, July 1997.
29. Wright, A., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991. *Information and Computation*, volume 115, 1994.