Modular Mixin-Based Inheritance for Application Frameworks

Dominic Duggan
Dept of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07040
dduggan@cs.stevens-tech.edu

Ching-Ching Techaubol
Dept of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07040
ctechaub@cs.stevens-tech.edu

ABSTRACT

Mixin modules are proposed as an extension of a class-based programming language. Mixin modules combine parallel extension of classes, including extension of the self types for those classes, with mixin-based inheritance. For soundness of subtyping purposes, they require an explicit distinction between mixin-based objects and class-based objects. Applications of mixin modules are in statically type-safe monad-based aspect-oriented programming, and in modular mixin-based Internet programming.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data Types and Structures*

General Terms

Languages

Keywords

Mixin-Based Inheritance, Application Frameworks, Modularity

1. INTRODUCTION

Component-based programming has become the central part of a paradigm shift in software development, moving programming away from "from-the-ground-up" application development and towards a model of plugging together off-the-shelf components. Application frameworks have emerged in the object-oriented programming community as a basis for facilitating component-based programming. In its most rudimentary form an application framework is a library of classes, that are intended to be specialized by application programmers with application semantics. Specification techniques such as formal methods and pattern languages may be used to facilitate the use of such libraries.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 01 Tampa Florida USA
Copyright ACM 2001 1-58113-335-9/01/10…$5.00

While component-based programming at the class level has had some success, there is a growing recognition of the limitations of such a low-level view in developing large applications from off-the-shelf components. While module languages have been known at least since the days of languages such as Mesa, Modula and Ada [29, 26, 32, 22, 25, 30], there is now growing interest in developing module languages specifically for composing class libraries in object-oriented languages. Since Java is now a popular object-oriented language for both practitioners and researchers, some of this work has at least implicitly been aimed at the Java language.

It can be argued that the essence of class-based object-oriented languages is the incremental extension of recursive definitions via inheritance. This was the motivation for the fixed point semantics of inheritance by Cook et al, that in turn gave rise to research on self types that allowed method types to be specialized in subclasses derived via inheritance. The types being extended are object interfaces, which can be considered as a form of record type. So object interfaces can be viewed as recursively defined record types, and inheritance allows the incremental extension both of these types and of object implementations.

New applications suggest another view of recursive definitions, that of recursively defined labelled trees. This perspective is popularized by XML, which (at the risk of gross oversimpification) can be viewed as a language for defining recursive tree types. Recursive tree types can be viewed as dual to recursive record types; indeed, the former correspond to variant types and discriminated union types in the terminology of programming language types.

The ultimate application of our work is in developing a framework for type-safe mixin-based inheritance for components of XML applications. In designing this framework we assume that processors for XML applications will be written in class-based languages such as Java or C#, since such processors will be easily downloadable over the Web. Presumably some, perhaps even many, XML processors will be written in XML-specific dialects such as XDuce and XML-

SQL, but it appears safe to assume that some large number will be written in class-based languages.

Variant types can be represented using only object types and subtyping: the tree type is defined as an abstract class or interface, and specific node types of this tree type are represented as concrete classes that subclass this abstract class. However this agglomeration of recursive record types and recursive variant types into the single concept of object types confuses two concepts, and this confusion becomes problematic when we consider extensions.

There are three extensions that we consider useful for the kind of reuse of XML applications that we are interested in supporting:

1. *Mixin-based inheritance*: Traditional inheritance involves the incremental extension of an existing application. Mixin-based inheritance, popularized by CLOS, allows fragments of applications to be modularly composed into complete applications. We envision an environment for XML applications where schema fragments, and processors for those fragments, are defined in a modular fashion and then combined as necessary for particular applications. A central design problem with mixin-based inheritance is that of choosing field names. For example unless care is taken it may be difficult to combine two mixins, only because they have unrelated fields with the same name. A facility for renaming fields in mixins could avoid this problem; however this approach breaks with subtyping, since renaming prevents the derivative class from being a subclass of the original class. Approaches based on distinguishing "internal" and "external" field names [31] become problematic with parallel extensions of classes, and it is also not clear how fields can be shared between mixins with this approach.

2. *Parallel extensions*: Traditional inheritance, and method type specialization, focus on the extension of a single class. XML schemas typically involve the definition of several mutually recursive document types, and processors for such document types will also involve mutually recursive methods recursing over XML data structures. In the style of processor definition envisioned for our framework, the inputs and outputs of these methods would be represented as objects (to support mixin-based combination of processors [12]). Therefore support for modular XML applications must be able to support the incremental extension and combination of collections of mutually recursive types and processors.

3. *Method type specialization*: To be useful, our framework for combining XML applications should ensure type safety. Method type specialization is a useful tool towards this end, perhaps even essential with parallel extensions of types and processors. A standard example is given by abstract classes for the subject-observer pattern:

```
abstract class Subject {
        void notify (Observer x); }
abstract class Observer {
        void register (Subject x); }
class WindowSubject extends Subject { ... }
class WindowObserver extends Observer { ... }
```

   It would be very useful in the derivative classes to have the instance variables specialized to types `WindowObserver` and `WindowSubject`, respectively. Otherwise objects of the derived classes must use downcasting at runtime to coerce objects to the required types, potentially reducing the reliability of the resulting applications. On the other hand, allowing the types to be specialized in this case leads to a loss of type safety: subtyping is incompatible with method type specialization in scenarios such as this.

Our vehicle for adding these extensions to a class-based language is *mixin modules*. A mixin module is a collection of mutually recursive type and implementation definitions. One might expect that a mixin module is then a collection of mutually recursive class definitions, similar to the approach advocated by Bruce et al for a statically safe alternative to virtual types [7]. However because of issues (1) and (3) described above, it is advantageous to break with the traditional confusion of record and variant types in object-oriented languages:

1. We retain classes in the traditional sense. Classes support extensions and overriding via inheritance, and an object of a class is also an object of any superclass; there is subtyping based on class extensions. However classes do not support field renaming or method type specialization, since these are incompatible with subtyping. In effect classes are the mechanism by which we define variant types, and so class-based objects are the basis for representing XML data structures as trees.

2. We add mixins, as an alternative to classes, for defining object types and object implementations. Mixins can be combined in a modular fashion; however unlike the traditional approach to mixin-based combination, this combination is done at the level of modules rather than at the level of mixins. Mixins support both explicit field renaming, to avoid field name conflicts, and also method type specialization. The difficulties described earlier are avoided by *not* allowing subtyping for mixins; it is unnecessary, since the applications of subtyping are already handled by class-based objects, and it avoids the aforesaid problems.

As an example of the approach, assume we have two XML schemas `foo` and `bar`, defining tags `t1` and `t2` respectively as components of the tag `t`. We want to modularly combine these schemas into a single schema that defines a single tag `t` that can contain either a `t1` tag or a `t2` tag as its component. We represent this by two mixin modules `foo` and `bar`. The intended union tag `t` is defined as an abstract class, with `t1`

and t2 as subclasses of this class. We also attach a processor method proc to objects of the t class. We represent this in our framework by two mixin modules:

```
module foo {
   abstract class t { Output proc (Input x); };
   class t1 extends t { ... };
   mixin Input { ... };
   mixin Output { ... };
}
module bar {
   abstract class t { Output proc (Input x); };
   class t2 extends t { ... };
   mixin Input { ... };
   mixin Output { ... };
}
```

Then the intended union of these schemas is obtained by combination of mixin modules:

```
module baz combines foo, bar;
```

The resulting combination module has an abstract class t, and two subclasses t1 and t2. The module also has two mixins, Input and Output, corresponding to the input and output of the processor application. The reason for defining these inputs and outputs as mixins is to support the mixin-based combination of the processor definitions. So the input arguments to the processor for the t1 tag are combined with the input arguments to the processor for the t2 tag, and similarly for the outputs. If one thinks of XML documents as abstract syntax trees, and processors as attribute grammars, the inputs can be thought of as inherited attributes and the outputs as synthesized attributes. So this can be viewed as a mixin-based approach to modularly combining attribute grammars. This can be generalized to a monadic approach to aspect-oriented programming described in [12], where modules include "plugging" code for weaving together inputs and outputs for each modular processor fragment. Each module provides an "aspect" of the XML application, and mixin module combination uses the monadic definition of processor fragments to combine these aspects.

As can be seen by example, names are an important part of combination. It is important that the base class have the same name (t) in both modules, that the input and output mixins, Input and Output respectively, are called the same in both modules, and that the processor method proc have the same name in both modules. Therefore mixin modules support renaming of both classes and mixins, and also of field names in mixins.

In the next section we give an informal overview of a mini-language demonstrating the extension of a class-based language with mixin modules. In Sect. 3 we provide the key parts of the type system, while in Sect. 4 we provide the operational semantics. We consider related work in Sect. 5, while Sect. 6 provides conclusions.

2. INFORMAL DESCRIPTION

In describing the extension of a class-based language with mixin modules, we adopt the approach of FJ [23], a small and economical kernel language that describes the essential core of Java. We provide an operational semantics that can be viewed as an abstract interpreter for a class-based object-oriented language extended with mixin modules. We avoid basing mixin modules on Java inner classes, because the latter introduce some rather frightening complications [24]. We assume a collection of program variables $x, y, z, w, \ldots$, and a collection of class, mixin and module names $X, Y, Z, W, \ldots$. This latter class includes a special module name *This*, a reference in a class or mixin definition to the final form of the module (after extensions and combinations) in which that definition occurs. The abstract syntax of our mini-language is defined in Fig. 1.

A module definition P specifies a collection of classes C and mixins M. Each class and mixin definition includes the specification of fields, methods and a single constructor. A class or mixin is defined by extending an existing class or mixin *that is defined in the same module.* To reuse definitions from other mixin modules, we allow a module to extend another module, which introduces all of the (class and mixin) definitions in the latter module into the former module. So we have single inheritance for modules, class and mixins. Inheritance for modules and mixins serves a dual purpose: in addition to reusing definitions, it is also the operation for renaming classes and mixins (for modules) and renaming fields and methods (for mixins)

Since all class and mixin definitions are relative to a module, a type has the form $X.Y$, where X is a module name and Y is the name of a class or mixin defined in that module (directly or indirectly via inheritance). A type of the form *This.Y* may be used to refer to the final module of the class or mixin Y when an object of this class or mixin is instantiated. For example[1]:

```
module X extends Root {} {} {
   class Y extends Object {
      This.Y self () { return this; }
   }
}
module Z extends X {} {} { ... }
Z.Y y1 = new Z.Y();
Z.Y y2 = y1.self();
```

There are two forms of mixin module definitions. An atomic mixin module has the form

$$module\ X\ extends\ Y \rho\ \theta\ \{\ \overline{C};\ \overline{M};\ \}$$

The mixin module named X includes all of the definitions from the mixin module named Y, and extends it with new class definitions $\overline{C}$ and new mixin definitions $\overline{M}$. There is no notion of override; the additional classes and mixins must

[1]We assume a special module named Root from which all other modules inherit. This module defines a root class Object and a root mixin MObject, analogous to the Object class in Java.

$e \in$ Expression	::=	$new\ A(e_1, \ldots, e_k)$	Object
	|	$e.x$	Class-based variable access
	|	$e.x(e_1, \ldots, e_k)$	Class-based method invoke
	|	$A{::}e.x$	Mixin-based variable access
	|	$A{::}e.x = e'$	Mixin-based variable update
	|	$A{::}e.x(e_1, \ldots, e_k)$	Mixin-based method invoke
	|	$this$	This or self
	|	x	Variable
	|	$null$	Null
	|	$A{::}super.x(e_1, \ldots, e_k)$	Mixin super
	|	$A{::}inner.x(e_1, \ldots, e_k)$	Mixin inner
	|	$return\ e$	Method return
$A, B \in$ Type	::=	$X.Y$	Type
	|	$This.Y$	Self Type
$K \in$ Constructor	::=	$X\ (\overline{A}\ \overline{x},\ \overline{B}\ \overline{y})\ \{\ super(\overline{x});\ this.\overline{y} = \overline{y};\ \}$	Class constructor
	|	$X\ ()\ \{\ super();\ this.\overline{x_m} = \overline{e_m};\ \}$	Mixin constructor
$C \in$ Class	::=	$class\ X\ extends\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}$	Class definition
$M \in$ Mixin	::=	$mixin\ X\ extends\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}$	Mixin definition
$F \in$ Method	::=	$A\ f(\overline{B}\ \overline{x})\ \{\ return\ (e);\ \}$	Method definition
$P \in$ Module	::=	$module\ X\ extends\ Y\ \rho\ \theta\ \{\ \overline{C};\ \overline{M};\ \}$	Module definition
	|	$module\ X\ combines\ Y,\ Z$	Module combination
ρ	::=	$\{X_1 \mapsto Y_1, \ldots, X_k \mapsto Y_k\}$	Definition Renaming
θ	::=	$\{X_1.x_1 \mapsto y_1, \ldots, X_k.x_k \mapsto y_k\}$	Label Renaming

Figure 1: Abstract Syntax

not share names with the existing classes and mixins in Y. To ensure flexibility, extending a mixin module allows class and mixin names in the old module to be renamed in the new extended module, where this renaming is specified by the definition renaming ρ. For example in the code fragment:

```
module X extends Root {} {} { class Z ... }
module Y extends X {Z↦W} {} { class Z ... }
```

Then the module `Y` contains two classes[2]: one named `Y.W` and one named `Y.Z`. The class named `Y.W` is actually a renaming of `X.Z`, however there is no relationship between objects of the classes of `X` and objects of the classes of `Y`. There is no subtyping relation between `X.Z` and `Y.W`. Even if the class `Y.Z` inherits from `Y.W`, there is no subtype relation between `Y.Z` and `X.Z` (although there is between `Y.Z` and `Y.W`). It is not possible for `Y.Z` to be defined via inheritance from `X.Z`. This restriction is enforced simply because it would be unsound to do otherwise; well-typed programs could encounter run-time failure due to invalid method calls without this restriction. The source of the unsoundness is that it is possible to specialize the types of mixins as a result of inheritance, as shown on the next page. So each mixin module defines a collection of mutually recursive classes and mixins, with subtype relations between classes only valid within that module. This is compatible with the stated objective of mixin modules: to provide support for building safe reusable application frameworks, based on specializing classes "in parallel." As an example of the allowable subtype relations, we have:

[2]From now on, we ignore the root class and mixin.

```
module X extends Root {} {} {
   class Y extends Object { void foo (This.Y y); }
}
module Z extends X {} {} {
   class W extends Y { void bar (); }
}
Z.W w = new Z.W();
w.foo (new X.Y()); // type error
w.foo (new Z.Y()); // type-checks
w.foo (w);         // type-checks
```

The second form of mixin module allows a new form of module to be defined using *module combination*

$$module\ X\ combines\ Y,\ Z$$

This generalizes the combination of individual mixins, as in other approaches to mixin-based inheritance, to allow the simultaneous combination of collections of mixins. In this combination similarly named classes must be identical. Such classes arise from a common parent module W that the combined modules Y and Z extend, and in the expected usage of

a mixin module the combined modules contribute extensions of this common base class. Module combination is defined as a binary operation for simplicity; it is straightforward to extend it to an n-ary operation for $n \geq 0$.

As an example of module combination, fragments of an abstract syntax tree can be represented as:

```
module Interp extends Root {} {} {
   class AST {...} ...
}
module FuncInterp extends Interp {} {} {
   class Var extends AST { String x; }
   class Abs extends AST { String fml; AST body; }
   class App extends AST { AST rator; AST rand; }
}
module ImperativeInterp extends Interp {} {} {
   class Assign extends AST { AST lhs; AST rhs; }
   class Deref extends AST { AST exp; }
}
module FuncImpInterp
   combines FuncInterp, ImperativeInterp;
```

Since our claim (and that of others [36]) is that XML schemas correspond in some sense to AST definitions, this example demonstrates how XML schemas can be combined using mixin module combination.

On the other hand, in mixin module combination, similarly named mixins are regarded as fragments that should be coalesced in the combination of the modules. In this coalescing, similarly named instance variables must have the same type and are merged, so instance variables may be shared between mixins. Similarly named methods must also have the same type. In their coalescence, the method implementation in the left-hand combinand is executed when the method is invoked on a mixin object. It is possible for this method implementation to delegate to the implementation in the right-hand combinand; this is done using the *inner* construct described below. For example:

```
module Interp extends Root {} {} {
   mixin Input extends MObject ...  { }
   mixin Output extends MObject ...  { }
   class AST extends Object {Output eval(Input x);}
}
module FuncInterp extends Interp {} {} {
   class Env extends Object {...}
   mixin Input extends Input { Env env; }
}
module ImperativeInterp extends Interp {} {} {
   class Store extends Object {...}
   mixin Input extends Input { Store store; }
}
module FuncImpInterp
   combines FuncInterp, ImperativeInterp;
```

In the resulting combination mixin module, an object of the mixin `Input` contains both an `env` field and a `store` field. The class renaming operation that is part of mixin module extension appears to be essential to make this approach workable. Since part of our claim is that mixins can be used to define "attributes" for evaluators that walk over abstract syntax trees, this example demonstrates how such attribute definitions can be combined using mixin module combination. An approach to modularly combining attribute evaluators, supported by our minilanguage, is described in [12].

A class definition has the form

$$class\ X\ extends\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}$$

This is exactly the same as a class definition in FJ. The definition introduces new instance variables $\overline{x}$, and new and overriding method definitions $\overline{F}$, and also defines a constructor K. Every class extends another class; we assume a special root class analogous to `Object` in Java (although each module has its own version of `Object`, since subtyping is local to a module). The class being extended must either be defined in the current mixin module, or else have already been defined in the mixin module that the current module extends.

A mixin definition has the form

$$mixin\ X\ extends\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}$$

As with class definitions, a mixin definition extends an existing mixin, contributing new instance variables, and new and overriding method definitions. Unlike with classes, only two forms of mixin extension are allowed:

$$mixin\ X\ extends\ MObject\ \ldots$$

$$mixin\ X\ extends\ X\ \ldots$$

The first form defines a new mixin, while the second form defines the extension of an already-defined mixin (inherited from another module).

A mixin extension must carry the same name as the mixin being extended. This restriction is in order to ensure that references to the original mixin in a module being extended should rebind to the new mixin. As a result, mixins provide support for method type specialization. For example, a module defining the interfaces for subjects and observers could be specified as follows:

```
module SubjObs extends Root {} {} {
   mixin Subject extends MObject {
      void register (This.Observer x) { ... }
   }
   mixin Observer extends MObject {
      void notify (This.Subject x) { ... }
   }
}
```

A windowing version of this module is defined by:

```
module WindowSubjObs extends SubjObs {} {} {
  mixin Subject extends Subject {
   void notify (This.Observer x)
        { ...  This.Observer::x.windowMeth() ... }
  }
  mixin Observer extends Observer {
   void register (This.Subject x) {...}
   void windowMeth () {...}
  }
}
```

The annotations for accessing (and updating) mixin object fields (`This.Observer::x.windowMeth()` in this example) are explained below. This results in a module `WindowSubjObs` with two mixins: `Subject` and `Observer` (and also the default `MObject` mixin). Then we can create objects of these mixins:

```
WindowSubjObs.Subject ws =
      new WindowSubjObs.Subject(...);
WindowSubjObs.Observer wo =
      new WindowSubjObs.Observer(...);
WindowSubjObs.Subject::ws.register(wo);
```

This example demonstrates method type specialization, since for example an object from the `WindowSubjObs.Subject` mixin has fields contributed by both the base mixin and the derivative mixin, while the argument type of the `notify` method of an observer object has rebound to the new extended `Subject` type. We disallow a subtype relation between mixins in order to avoid unsoundness due to method type specialization. For example with such subtyping the following erroneous example would type-check:

```
void bad (SubjObs.Subject s, SubjObs.Observer o)
         { SubjObs.Subject::s.register(o); }
WindowSubjObs.Subject ws =
      new WindowSubjObs.Subject();
SubjObs.Observer o = new SubjObs.Observer();
bad (ws, o);
```

It is because of method type specialization for mixins that we do not allow a subtype relation between a class in one mixin module and a class in a derivative mixin module. The following example demonstrates the unsoundness that would result otherwise:

```
module X extends Root {} {} {
  mixin foo { int x; }
  class bar {
    void g (This.foo f) { This.foo::f.x=3; }
  }
}
module Y extends X {} {} {
  mixin foo extends foo { int y; }
  class baz extends bar {
    void g (This.foo f) { This.foo::f.y=4; }
  }
}
void bad (X.bar b, X.foo f) { X.bar::b.g(f); }
bad (new Y.baz(), new X.foo());
```

Field names are fundamentally important in mixins, because they are the basis for coalescing mixin definitions during module combination. Therefore module extension allows fields to be renamed in the mixins of the module being extended, using the field renaming θ. For example:

```
module X extends Root {} {} { mixin Z { int x; } }
module Y extends X {Z↦W} {Z.x↦y} {
   mixin W extends W { String x; }
}
```

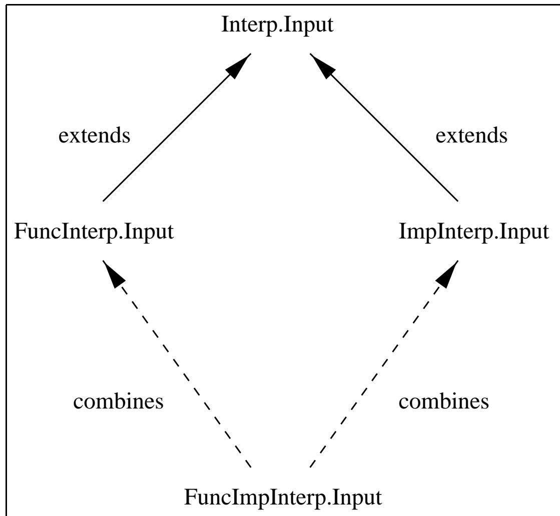
After these definitions, the module `Y` has a single mixin, `Y.W`, with two fields, a field `x` of type `String` and a field `y` of type `int`.

In summary we have three forms of extensions:

1. Module extension allows the definitions of one module to be included in another module. This operation also supports the renaming of definitions in the original module, and the relabelling of fields in mixins. This latter renaming is useful to support mixin combination, where it may not always be possible to ensure that two mixins have the same name for a field that they are expected to share.

2. Class extension is used to define a variant or union type, where a base abstract class defines a union type and concrete subclasses define the variants of the union type. Therefore subtyping between subclass and base class are a fundamental part of class extensions. For this reason, class extension does not support method type specialization or any renaming.

3. Mixin extension allows a new mixin to be defined via derivation from a base mixin. To understand the relationship between this and mixin combination, consider the example of the `Input` mixin defined above:



So mixin extension is intended to be used to derive a concrete mixin from an abstract mixin or mixin interface, while mixin combination is intended to modularly combine mixin implementations.

Mixin module combination combines both classes and mixins:

1. An abstract base class and its subclasses define a union type in a module. If both of the combined modules have the same base class, then combination forms the union of the corresponding union types in the two modules, since all variants (derivative classes) will be subclasses of the base class in the combination module.
2. If the same mixin is defined in both modules, then module combination combines the two mixins into one. Similarly named fields and mehtods must have the same types. Fields are shared between the mixins, and methods are combined using the *inner* construct, as explained below.

New objects are created by *new*. We follow the example of FJ and assume a stylized form of constructor definitions, where each class has a single constructor and that constructor has an argument for each instance variable of the class. This restriction is only required for a simplified description of the state of an object. Lifting the restriction is straightforward but leads to a tedious complication of the semantics. With this restriction, the state of an object is represented by an expression of the from

$$new\ X.Y(e_1, \ldots, e_k)$$

where the expressions $e_1, \ldots, e_k$ denote the values of the k instance variables of an object of class or mixin $X.Y$. We assume some canonical ordering of the instance variable names.

The form of a class constructor is required to be

$$X(\overline{A}\ \overline{x},\ \overline{B}\ \overline{y})\ \{\ super(\overline{x});\ this.\overline{y} = \overline{y};\ \}$$

where $\{\overline{x}\}$ are the fields defined in the superclass, and $\{\overline{y}\}$ are the fields added by the current class definition. With this stylized constructor definition, the state of a class-based object can be represented as explained above.

For mixin constructors, matters are complicated by the fact that not all fields are known when a constructor is defined; fields may be contributed by another mixin with which the current mixin is subsequently combined. Therefore the form of a mixin constructor is

$$X()\ \{\ super();\ this.\overline{y} = \overline{e};\ \}$$

where $\{\overline{y}\}$ are the fields contributed by the current mixin definition. The expressions $\{\overline{e}\}$ are the *default initial values* for these fields. An uninitialized object is one of the form

$$This.Y()$$

where Y is a mixin in the current module X. This object is initialized when it "evolves" to one of the form

$$X.Y(e_1, \ldots, e_k)$$

where the mixin $X.Y$ has k fields, and e_i is the default initial value for the ith field. Therefore constructors for mixins are always nullary; they set the values of instance variables to default values, and rely on field update to provide more specialized values.

For objects from classes, there are the usual operations for accessing variables and invoking methods. For objects from mixins, there are these operations and in addition there is a field update operation. This field update operation is essential because (as explained) we cannot rely on the constructor to initalize the variables. We make this update operation functional (it returns a new object with the updated field) only to simplify the operational semantics in Sect. 4, otherwise it would be necessary to provide a semantics involving stores etc.

For mixins, the operations for field access and update, and for method invocation, are annotated with the mixin type of the object on which the operation is being invoked. This is not necessary either for typing programs or for the operational semantics. It is only necessary because mixin extension allows some of the fields of a mixin to be renamed, and the names used in the mixin code must then be relabelled in the derivative mixin. The type annotation identifies those places in the code where the renaming must be done. For example:

```
module X extends Root {} {} {
   mixin Y extends MObject {
      int x;
      int get() { return This.Y::this.x; }
   }
}
module Z extends X {} {Y.x↦z} { }
Z.Y w = new Z.Y();
Z.Y::w.get();
```

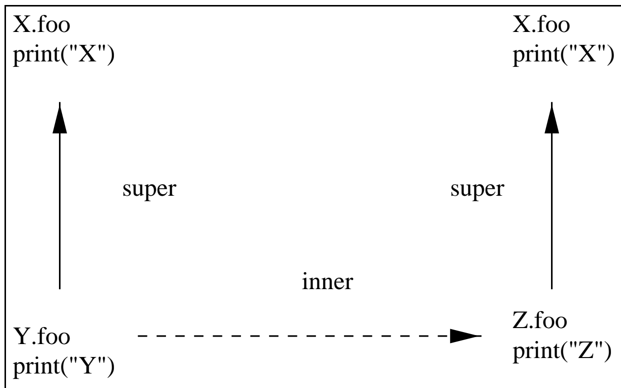
When `Z.Y` is derived from `X.Y`, the field `x` is renamed to `z`. The `get` method that is inherited from `X.Y` accesses the

field using the old name of `x`. Therefore when the method is invoked from a `Z.Y` object, the attempted access to the `x` field is translated to an access to the `z` field. The type annotations on field access and update and method invocation for mixin-based objects serve to delineate the places in method code where this renaming is necessary.

Besides variable access and update, and method invocation, there are two other operations that may be formed on mixin-based objects. The *super* construct allows a method implementation to delegate to the implementation in the parent mixin. On the other hand the *inner* construct allows a method implementation in a mixin to delegate to the implementation in a similarly named mixin with which it is combined, as a result of module combination. For example:

```
module X extends Root {} {} {
   mixin foo extends MObject {
      void f() { print("X "); }
   }
}
module Y extends X {} {} {
   mixin foo extends foo {
      void f() { print("Y "); super.f(); inner.f(); }
   }
}
module Z extends X {} {} {
   mixin foo extends foo {
      void f() { print("Z "); super.f(); }
   }
}
module W combines Y, Z;
W.foo x = new W.foo();
W.foo::x.f();
```

The output is `Y X Z X`. The chain of delegations is described by this diagram:



For simplicity we do not track when it is valid to invoke *inner* in a mixin method implementation. So it would be type-correct to add a call to `inner.f()` to `Z` above, and this would fail at run-time. We can fix this by extending mixins to specify an interface they expect of any mixins with which they are combined, to ensure that calls to *inner* do not fail. We omit this check for simplicity.

$$\frac{\begin{array}{c} CT(X) = (module\ X\ extends\ Y\ \rho\ \theta\ \{\ \overline{C};\ \overline{M};\ \}) \\ ME = \{(defname(M) \mapsto M) \mid M \in \overline{M}\} \\ (\overline{C'}, ME') = defins(Y) \quad C'' = \rho(\theta(C')) \\ ME'' = \{(\rho(X) \mapsto \rho(\theta(ME'(X)))) \mid X \in \mathtt{dom}(ME')\} \end{array}}{defins(X) = ((\overline{C}, \overline{C'}), INH(ME, ME''))} \quad \text{(Classes Extends)}$$

$$\frac{\begin{array}{c} CT(X) = (module\ X\ combines\ Y,\ Z) \\ (\overline{C}, ME) = defins(Y) \quad (\overline{C'}, ME') = defins(Z) \end{array}}{defins(X) = ((\overline{C}, \overline{C'}), COMB(ME, ME'))} \quad \text{(Classes Combines)}$$

Figure 2: Flattened Inheritance Hierarchy

3. TYPE SYSTEM

Both the static and dynamic semantics make use of various metafunctions. In both of these semantics we assume a fixed global module table CT, analogous to the global class table CT in the FJ semantics. The most important of the metafunctions is the function $defins(X)$, where X is a module name. This metafunction essentially flattens the inheritance hierarchy for a mixin module (arising both from extension and from combination) into a pair of a sequence of classes and a mapping ME from each mixin name to a tree of the inherited and combined definitions for that mixin. The inheritance hierarchy for a mixin is described by the following data structure:

$$\begin{array}{rcl} m & ::= & inh(M, m) \\ & | & comb(m, m') \\ & | & mobj \\ & | & M \end{array}$$

The case of $m ::= M$ is only included as a convenience for the Classes Extends rule in Fig. 2. This figure gives the definition of the *defins* metafunction, that performs the flattening of the inheritance hierarchy for a mixin module. This operation combines mixin environments using the INH and $COMB$ metafunctions defined in Fig. 3.

In the case of Classes Extends, where we compute the classes and mixins defined in the base module Y, the definition renaming ρ and label renaming θ are applied to these classes and mixins before they are combined with the new classes and mixins defined in the extension X. Some representative cases in the definition of the application of a definition renaming and a label renaming are given in Fig. 3.

The flattened inheritance hierarchy is used in the definition of metafunctions for computing method bodies and method types in Fig. 4. For a method defined in a class, we search for its definition in a class and then in the base class from which that class is derived. This corresponds to the two cases Method Class Base and Method Class Extend in Fig. 4. For a method defined in a mixin, there are three cases: either the method is defined in a mixin definition (Method Mixin InhL), or it is defined in a mixin from

$$
\begin{array}{rcl}
INH(ME, ME') & = & \{(X \mapsto inh(ME(X), ME'(X))) \mid X \in \texttt{dom}(ME) \cap \texttt{dom}(ME')\} \\
& & \cup\{(X \mapsto inh(ME(X), mobj)) \mid X \in \texttt{dom}(ME) - \texttt{dom}(ME')\} \\
& & \cup\{(X \mapsto ME'(X)) \mid X \in \texttt{dom}(ME') - \texttt{dom}(ME)\} \\
COMB(ME, ME') & = & \{(X \mapsto comb(ME(X), ME'(X))) \mid X \in \texttt{dom}(ME) \cap \texttt{dom}(ME')\} \\
& & \cup\{(X \mapsto ME(X)) \mid X \in \texttt{dom}(ME) - \texttt{dom}(ME')\} \\
& & \cup\{(X \mapsto ME'(X)) \mid X \in \texttt{dom}(ME') - \texttt{dom}(ME)\}
\end{array}
$$

$$
\begin{array}{rcl}
\theta_X(x) & = & \begin{cases} y & \text{if } (X.x \mapsto y) \in \theta \\ x & \text{otherwise} \end{cases} \\
\theta(C) & = & class\ X\ extends\ Y\ \{\overline{A}\ \overline{x};\ \theta(K);\ \overline{\theta(F)}\} \\
& & \text{if } C \equiv (class\ X\ extends\ Y\ \{\overline{A}\ \overline{x};\ K;\ \overline{F}\}) \\
\theta(F) & = & (A\ f(\overline{B}\ \overline{x})\{return(\theta(e))\}) \\
& & \text{if } F \equiv (A\ f(\overline{B}\ \overline{x})\{return(e)\}) \\
\theta(M) & = & mixin\ X\ extends\ Y\ \{\overline{A}\ \overline{\theta_X(x)};\ \theta(K);\ \overline{\theta_X(F)}\} \\
& & \text{if } M \equiv (mixin\ X\ extends\ Y\ \{\overline{A}\ \overline{x};\ K;\ \overline{F}\}) \\
\theta_X(F) & = & (A\ \theta_X(f)(\overline{B}\ \overline{x})\{return(\theta(e))\}) \\
& & \text{if } F \equiv (A\ f(\overline{B}\ \overline{x})\{return(e)\}) \\
\theta(W.Y{::}e.x) & = & \begin{cases} W.Y{::}\theta(e).\theta_Y(x) & \text{if } W \equiv This \\ W.Y{::}\theta(e).x & \text{otherwise} \end{cases} \\
\rho(C) & = & class\ \rho(X)\ extends\ \rho(Y)\ \{\overline{\rho(A)}\ \overline{x};\ \rho(K);\ \overline{\rho(F)}\} \\
& & \text{if } C \equiv (class\ X\ extends\ Y\ \{\overline{A}\ \overline{x};\ K;\ \overline{F}\}) \\
\rho(M) & = & mixin\ \rho(X)\ extends\ \rho(Y)\ \{\overline{\rho(A)}\ \overline{x};\ \rho(K);\ \overline{\rho(F)}\} \\
& & \text{if } M \equiv (mixin\ X\ extends\ Y\ \{\overline{A}\ \overline{x};\ K;\ \overline{F}\}) \\
\rho(F) & = & (\rho(A)\ f(\overline{\rho(B)}\ \overline{x})\{return(\rho(e))\}) \\
& & \text{if } F \equiv (A\ f(\overline{B}\ \overline{x})\{return(e)\}) \\
\rho(W.Y{::}e.x) & = & \begin{cases} W.\rho(Y){::}\rho(e).x & \text{if } W \equiv This \\ W.Y{::}\rho(e).x & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3: Metafunctions

$$\frac{\mathit{classMeth}(X.Y, f) = (A\ f(\overline{B}\ \overline{x})\{\ldots\},\ m)}{\mathit{classMethTy}(X.Y, f) = \overline{B} \rightarrow A} \quad \text{(Method CType)}$$

$$\frac{(\overline{C}, \mathit{ME}) = \mathit{defins}(X) \quad (\mathit{class}\ Y\ \mathit{extends}\ Z\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}) \in \overline{C} \quad F \equiv (A\ f(\overline{B}\ \overline{x})\ \{\ \mathit{return}\ (e);\ \}) \in \overline{F}}{\mathit{classMeth}(X.Y, f) = F} \quad \text{(Method Class Base)}$$

$$\frac{(\overline{C}, \mathit{ME}) = \mathit{defins}(X) \quad (\mathit{class}\ Y\ \mathit{extends}\ Z\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}) \in \overline{C} \quad f \notin \mathit{methnames}(\overline{F})}{\mathit{classMeth}(X.Y, f) = \mathit{classMeth}(X.Z, f)} \quad \text{(Method Class Extend)}$$

$$\frac{\mathit{mixinMeth}(X.Y, f) = (A\ f(\overline{B}\ \overline{x})\{\ldots\},\ m)}{\mathit{mixinMethTy}(X.Y, f) = \overline{B} \rightarrow A} \quad \text{(Method MType)}$$

$$\frac{(\overline{C}, \mathit{ME}) = \mathit{defins}(X) \quad Y \in \mathtt{dom}(\mathit{ME})}{\mathit{mixinMeth}(X.Y, f) = \mathit{mixinMeth}(f, \mathit{ME}(X))} \quad \text{(Method Mixin)}$$

$$\frac{m \equiv \mathit{inh}(M, m') \quad M \equiv (\mathit{mixin}\ X\ \mathit{extends}\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}) \quad F \equiv (A\ f(\overline{B}\ \overline{x})\ \{\ \mathit{return}\ (e);\ \}) \in \overline{F}}{\mathit{mixinMeth}(f, m) = (F, m', ())} \quad \text{(Method Mixin InhL)}$$

$$\frac{m \equiv \mathit{inh}(M, m') \quad M \equiv (\mathit{mixin}\ X\ \mathit{extends}\ Y\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}) \quad f \notin \mathit{methnames}(\overline{F})}{\mathit{mixinMeth}(f, m) = \mathit{mixinMeth}(f, m')} \quad \text{(Method Mixin InhR)}$$

$$\frac{m \equiv \mathit{comb}(m_1, m_2) \quad \mathit{mixinMeth}(f, m_1) = (F, m', (\overline{m''}))}{\mathit{mixinMeth}(f, m) = (F, m', (\overline{m''}, m_2))} \quad \text{(Method Mixin CombL)}$$

$$\frac{m \equiv \mathit{comb}(m_1, m_2) \quad \mathit{mixinMeth}(f, m_1)\ \text{undefined} \quad \mathit{mixinMeth}(f, m_2) = (F, m', (\overline{m''}))}{\mathit{mixinMeth}(f, m) = (F, m', (\overline{m''}))} \quad \text{(Method Mixin CombR)}$$

$$\frac{\mathit{mixinMeth}(f, m) = (F, m'', (\overline{m'''}))}{\mathit{mixinMeth}(f, (m, \overline{m'})) = (F, m'', (\overline{m'''}, \overline{m'}))} \quad \text{(Method Mixin SeqH)}$$

$$\frac{\mathit{mixinMeth}(f, m)\ \text{undefined} \quad \mathit{mixinMeth}(f, \overline{m'}) = (F, m'', (\overline{m'''}))}{\mathit{mixinMeth}(f, (m, \overline{m'})) = (F, m'', (\overline{m'''}))} \quad \text{(Method Mixin SeqT)}$$

Figure 4: Method Body and Method Type

$$\frac{(\overline{C}, ME) = \mathit{defins}(X) \quad (\mathit{class}\ Y\ \mathit{extends}\ Z\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \}) \in \overline{C} \quad (\overline{B}\ \overline{y}) = \mathit{classVars}(X.Z)}{\mathit{classVars}(X.Y) = (\overline{A}\ \overline{x},\ \overline{B}\ \overline{y})} \quad \text{(Vars Class)}$$

$$\frac{(\overline{C}, ME) = \mathit{defins}(X) \quad Y \in \mathtt{dom}(ME)}{\mathit{mixinVars}(X.Y) = \mathit{removeDups}(\mathit{mixinVars}(ME(Y)))} \quad \text{(Vars Mixin)}$$

$$\frac{(\overline{B}\ \overline{y} = \overline{e}) = \mathit{mixinVars}(m) \quad K \equiv (Y()\{\mathit{super}();\ \mathit{this}.\overline{x} = \overline{e'}\}) \quad M \equiv (\mathit{mixin}\ Y\ \mathit{extends}\ Z\ \{\ \overline{A}\ \overline{x};\ K;\ \overline{F}\ \})}{\mathit{mixinVars}(\mathit{inh}(M, m)) = (\overline{A}\ \overline{x} = \overline{e'},\ \overline{B}\ \overline{y} = \overline{e})} \quad \text{(Vars Mixin Inh)}$$

$$\frac{(\overline{A}\ \overline{x} = \overline{e}) = \mathit{mixinVars}(m) \quad (\overline{B}\ \overline{y} = \overline{e'}) = \mathit{mixinVars}(m')}{\mathit{mixinVars}(\mathit{comb}(m, m')) = (\overline{A}\ \overline{x} = \overline{e},\ \overline{B}\ \overline{y} = \overline{e'})} \quad \text{(Vars Mixin Comb)}$$

$$\mathit{mixinVars}(\mathit{mobj}) = () \quad \text{(Vars Mixin MObject)}$$

$$\frac{\mathit{classVars}(X.Y) = (\overline{A}\ \overline{x})}{\mathit{varType}(X.Y, x_i) = A_i} \quad \text{(Var Class Type)}$$

$$\frac{\mathit{mixinVars}(X.Y) = (\overline{A}\ \overline{x} = \overline{e})}{\mathit{varType}(X.Y, x_i) = A_i} \quad \text{(Var Mixin Type)}$$

Figure 5: Instance Variables and Instance Variable Type

$$\frac{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash e : B}{TE; A, \overline{A}; W, \overline{W}; (m, \mathcal{P}); ((\overline{m}), \mathcal{M}) \vdash \mathit{return}(e) : B} \quad \text{(Val Return)}$$

$$\frac{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash \diamond}{TE; (A, \overline{A}); \overline{W}; \mathcal{P}; \mathcal{M} \vdash \mathit{this} : A} \quad \text{(Val This)}$$

$$\frac{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash \diamond}{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash x : TE(x)} \quad \text{(Val Var)}$$

$$\frac{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash \diamond}{TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash \mathit{null} : B} \quad \text{(Val Null)}$$

$$\frac{TE; \overline{A}; W, \overline{W}; \mathcal{P}; \mathcal{M} \vdash e : A \quad W \vdash A \leq B}{TE; \overline{A}; W, \overline{W}; \mathcal{P}; \mathcal{M} \vdash e : B} \quad \text{(Val Subsumption)}$$

Figure 7: Expression Type Rules

$$W \vdash A \leq A \quad \text{(Sub Refl)}$$

$$\frac{W \vdash A \leq B \quad W \vdash B \leq C}{W \vdash A \leq C} \quad \text{(Sub Trans)}$$

$$\frac{(\overline{C}, ME) = \mathit{defins}(X) \quad (\mathit{class}\ Y\ \mathit{extends}\ Z\ \{\dots\}) \in \overline{C}}{W \vdash X.Y \leq X.Z} \quad \text{(Sub Class)}$$

$$\frac{(\overline{C}, ME) = \mathit{defins}(W) \quad (\mathit{class}\ Y\ \mathit{extends}\ Z\ \{\dots\}) \in \overline{C}}{W \vdash \mathit{This}.Y \leq \mathit{This}.Z} \quad \text{(Sub This)}$$

Figure 8: Subtype Rules

which that mixin is derived by inheritance (Method Mixin InhR), or it is defined in another mixin (of the same name) with which it is combined by module combination (Method Mixin CombL or Method Mixin CombR). Computing a method type $\overline{B} \rightarrow A$ is defined in terms of the operation for computing a method body (Method CType for class methods, Method MType for mixin methods).

The Method Mixin rules in Fig. 4 for looking up methods in mixins return a triple as their result. The first element of this triple is the method definition. The second element of this triple is an inheritance hierarchy for the parent of the mixin in which the method definition is found; references to *super* in the method body are resolved using this

$$\frac{(\overline{B}\ \overline{y}) = \mathit{classVars}(X.Y) \quad X \not\equiv \mathit{This} \qquad TE;\overline{A};\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{\{X/\mathit{This}\}B}}{TE;\overline{A};\overline{W};\mathcal{P};\mathcal{M} \vdash \mathit{new}\ X.Y(\overline{e}) : X.Y} \quad \text{(Val New Conc)}$$

$$\frac{(\overline{B}\ \overline{y}) = \mathit{mixinVars}(X.Y) \quad X \not\equiv \mathit{This} \qquad TE;\overline{A};\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{\{X/\mathit{This}\}B}}{TE;\overline{A};\overline{W};\mathcal{P};\mathcal{M} \vdash \mathit{new}\ X.Y(\overline{e}) : X.Y} \quad \text{(Val New Conc)}$$

$$\frac{(\overline{B}\ \overline{y}) = \mathit{classVars}(W.Y) \quad TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{B}}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \mathit{new}\ \ \mathit{This}.Y(\overline{e}) : \mathit{This}.Y} \quad \text{(Val New Class)}$$

$$\frac{(\overline{C'}, ME) = \mathit{defins}(W) \quad Y \in \mathtt{dom}(ME) \quad TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \diamond}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \mathit{new}\ \ \mathit{This}.Y() : \mathit{This}.Y} \quad \text{(Val New Mixin)}$$

$$\frac{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e : X.Y \quad Z \equiv W \text{ if } X \equiv \mathit{This}, Z \equiv X \text{ otherwise} \qquad \mathit{varType}(Z.Y,x) = B}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e.x : \{X/\mathit{This}\}B} \quad \text{(Val Access)}$$

$$\frac{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e : X.Y \quad Z \equiv W \text{ if } X \equiv \mathit{This}, Z \equiv X \text{ otherwise} \qquad \mathit{classMethTy}(Z.Y,x) = \overline{B} \rightarrow B \quad TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{\{X/\mathit{This}\}B}}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e.x(\overline{e}) : \{X/\mathit{This}\}B} \quad \text{(Val Invoke)}$$

$$\frac{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e : X.Y \quad Z \equiv W \text{ if } X \equiv \mathit{This}, Z \equiv X \text{ otherwise} \qquad \mathit{varType}(Z.Y,x) = B}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash (X.Y{::}e.x) : \{X/\mathit{This}\}B} \quad \text{(Val MAccess)}$$

$$\frac{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e : X.Y \quad Z \equiv W \text{ if } X \equiv \mathit{This}, Z \equiv X \text{ otherwise} \qquad \mathit{varType}(Z.Y,x) = B \quad TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e' : \{X/\mathit{This}\}B}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash (X.Y{::}e.x = e') : X.Y} \quad \text{(Val MUpdate)}$$

$$\frac{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash e : X.Y \quad Z \equiv W \text{ if } X \equiv \mathit{This}, Z \equiv X \text{ otherwise} \qquad \mathit{mixinMethTy}(Z.Y,x) = \overline{B} \rightarrow B \qquad TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{\{X/\mathit{This}\}B}}{TE;\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash (X.Y{::}e.x(\overline{e})) : \{X/\mathit{This}\}B} \quad \text{(Val MInvoke)}$$

$$\frac{\mathcal{P} \equiv (m,\overline{m}) \quad \mathit{mixinMeth}(x,m) = (B'\ x(\overline{B}\ \overline{y})\{\mathit{return}(e)\}) \qquad TE;A,\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{B}}{TE;A,\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash A{::}\mathit{super}.x(\overline{e}) : B'} \quad \text{(Val Super)}$$

$$\frac{A \equiv X.Y \qquad \mathit{mixinMethTy}(W.Y,x) = \overline{B} \rightarrow B' \quad TE;A,\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash \overline{e} : \overline{B}}{TE;A,\overline{A};W,\overline{W};\mathcal{P};\mathcal{M} \vdash A{::}\mathit{inner}.x(\overline{e}) : B'} \quad \text{(Val Inner)}$$

Figure 6: Expression Type Rules

hierarchy. The third element of the triple is a sequence of inheritance hierarchies, corresponding to mixins with which the mixin containing the method definition has been combined on the left. The latter sequence is used to resolve references to *inner* in the method body, in the operational semantics though not in the type system because not all of these combinands may be known at compile-time. This sequence of inheritance hierarchies while the inheritance hierarchy is searched for a method definition; if the definition is found in the left combinand of a mixin combination (Rule (Method Mixin CombL)), then the right combinand is added to this sequence of inheritance hierarchies.

Fig. 5 defines metafunctions for computing the set of all (unique) instance variables defined in a class or mixin, and the type of an instance variable. The rule Vars Class computes the instance variables for a class by chasing up the inheritance hierarchy. The rule Vars Mixin computes the instance variables for a mixin by forming the sequence of all instance variables declared for all mixins with that name, and then removing duplicates. We omit the obvious definition of the metafunction *removeDups*.

We omit type rules for modules, classes and mixins for lack of space; they are reasonably straightforward. We concentrate instead on the type rules for the core language, provided in Fig. 6 and Fig. 7. Here it is important to state that the type rules are intended to be used to type method bodies, but they are also intended to be used to type configurations of the operational semantics provided in the next section. In the operational semantics, a program evaluates relative to several stacks, required because of the *new*, *super* and *inner* constructs:

1. Evaluation of *new This*.$Y()$ for creating a mixin object requires knowledge of the "current" module, the module in which the currently running code is defined (perhaps by inheritance or combination from another module). Therefore the operational semantics maintains a stack of module names, where the top of the stack gives the current module name.

2. Evaluation of A::*super*.$x(\overline{e})$ and A::*inner*.$x(\overline{e})$ require access to the "next" appropriate mixin in the inheritance hierarchy. For *super* the operational semantics maintains a stack of inheritance hierarchies, each hierarchy corresponding to the parent mixin of the mixin in which the current method was defined. For *inner* the operational semantics maintains a stack of sequences of inheritance hierarchies, each sequence corresponding to the mixins with which the mixin containing the current method definition has been combined. It is also useful to have a stack of class/mixin names, where the top of the stack identifies the current class or mixin.

Stacks are required because, on each method call, the old module name, class name and remaining inheritance hierarchy must be saved until the method call terminates. The *return* construct is added to the language only as a placeholder for a pending method call that has not yet returned. If we give a type for an expression with stacks of size $k+1$, then there must be a surrounding context that contains *return* expressions nested to a depth of k. This is enforced by the type system in Fig. 6 and Fig. 7. These type rules use sequents of the form

$$TE; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash e : B$$

where *TE* is a type environment, a mapping from program variables (method parameters) to types. The sequence $\overline{A}$ is the sequence of class and mixin names, one per pending method call. The sequence $\overline{W}$ is the sequence of module names, while $\mathcal{P}$ is the stack of inheritance hierarchies for *super* and $\mathcal{M}$ is the stack of sequences of inheritance hierarchies for *inner*.

4. OPERATIONAL SEMANTICS

The operational semantics are provided in Fig. 9. The semantics is defined using "small-step" semantics, based on rewrite rules that map between 6-tuples

$$(e,\ \overline{v},\ \overline{B},\ \overline{W},\ \mathcal{P},\ \mathcal{M})$$

In this 6-tuple, e is an expression representing a running program, the sequence of values $\overline{v}$ is a stack of objects (each such object corresponds to the *this* parameter to a suspended method invocation), the sequence $\overline{B}$ is a stack of class and mixin names (each such name is the type of the corresponding *this* parameter) and $\overline{W}$ a stack of module names, $\mathcal{P}$ is a stack of inheritance hierarchies (a sequence $(m_1, \dots, m_k)$) and $\mathcal{M}$ a stack of sequences of inheritance hierarchies (a sequence of sequences $((m_{1,1}, \dots, m_{1,n_1}), \dots, (m_{m,1}, \dots, m_{m,n_m}))$, as described in the previous section. The stacks grow from the right. As method invocation is commenced, the *return* construct is used to save a "return point." The Red Ret Ctxt allows evaluation within a *return* (evaluation inside a method that has not yet returned). This rule ensures that only the tops of the various stacks are visible in the evaluation of a method body. When evaluation of the expression reduces to a value, a "return" is effected by popping the stacks (Red Return). Values are defined as a subset of expressions for which no further evaluation is possible:

$$\begin{array}{rcl} v & ::= & \textit{null} \\ & | & \textit{new}\ X.Y(v_1, \dots, v_k),\ X \not\equiv \textit{This} \end{array}$$

The stacks are pushed whenever a new method invocation is executed: Red Invoke, Red MInvoke, Red Super and Red Inner, corresponding to invocation of class object method, invocation of mixin object method, invocation via *super* and invocation via *inner*. When a class object method is invoked (Red Invoke), dummy entries are placed on the super and inner stacks, since neither *super* nor *inner* are available within the body of a class method. The Red MInvoke rule for invoking a mixin method pushes a new parent hierarchy m for super and a new sequence of hierarchies for inner $\overline{m'}$, both of them obtained from the method lookup.

The Red Super rule for invoking a method via *super* uses the top m of the super stack to search for the method definition in the parent mixin of the currently executing method's

mixin. This returns a new parent inheritance hierarchy m' and a new sequence of hierarchies $\overline{m''}$ for inner. The former is pushed onto the super stack, the latter is appended to the top of the current inner stack and the result then pushed onto the inner stack. The motivation for the latter is that a mixin found in the parent hierarchy has been combined with the same mixins as the current mixin on the right, and in addition may have been combined with other mixins in the parent inheritance hierarchy.

The RED INNER rule for invoking a method via *inner* searches the sequence of hierarchies $\overline{m}$ at the top of the *inner* stack. Method lookup returns a new parent hierarchy m' that is pushed onto the super stack, and a new sequence of hierarchies $\overline{m''}$ that is pushed onto the inner stack.

The RED CONTEXT rule allows evaluation within an evaluation context, defined by:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \\ & | & new\ A(\ldots, E[\,], \ldots) \\ & | & E[\,].x \\ & | & E[\,].x(e_1, \ldots, e_k) \\ & | & v.x(\ldots, E[\,], \ldots) \\ & | & A :: E[\,].x \\ & | & A :: E[\,].x = e \\ & | & A :: v.x = E[\,] \\ & | & A :: E[\,].x(e_1, \ldots, e_k) \\ & | & A :: v.x(\ldots, E[\,], \ldots) \\ & | & A :: super.x(\ldots, E[\,], \ldots) \\ & | & A :: inner.x(\ldots, E[\,], \ldots) \end{array}$$

We deliberately omit *return* from this definition, so the RED RET CTXT rule must be used to evaluate inside a return.

There is an auxiliary evaluation relation

$$X \vdash e \Longrightarrow e'$$

that is used to instantiate any occurrences of *This*. The most important rule in this relation, Rule CRED NEW THIS, is provided in Fig. 9.

Theorem 1. *Suppose e does not have any subexpressions involving return. If*

$$\{\}; (A); (W); \mathcal{P}; \mathcal{M} \vdash e : B$$

and

$$W \vdash e \Longrightarrow e'$$

then

$$\{\}; (A); (W); \mathcal{P}; \mathcal{M} \vdash e' : \{W/This\}B.$$

Theorem 2 (SUBJECT REDUCTION). *Suppose B does not have any occurrences of This and*

$$\{\}; \overline{A}; \overline{W}; \mathcal{P}; \mathcal{M} \vdash e : B$$

and

$$\{\}; (); (); (); () \vdash \overline{v} : \overline{A}$$

and

$$(e, \overline{v}, \overline{A}, \overline{W}, \mathcal{P}, \mathcal{M}) \longrightarrow (e', \overline{v'}, \overline{A'}, \overline{W'}, \mathcal{P}', \mathcal{M}')$$

then

$$\{\}; (); (); (); () \vdash \overline{v'} : \overline{A'}$$

and

$$\{\}; \overline{A'}; \overline{W'}; \mathcal{P}'; \mathcal{M}' \vdash e' : B.$$

Both theorems are verified by induction on type derivations. For subject reduction, the interesting cases are for method invocations (RED INVOKE, RED MINVOKE, RED INNER and RED SUPER). For these we require a simple substitution lemma, and a lemma that verifies that if a method (in a class or mixin) has a particular type in a module, then it has the same type in any derivative of that module. Method types are parametric in *This*, the type of the module in which they are defined, and the auxiliary evaluation relation allows method bodies and their types to be instantiated appropriately at the point where they are used.

5. RELATED WORK

A great deal of work in module languages has been done in the last few years in the functional programming community, specifically in the context of the ML functional language [21, 20, 27, 28, 34, 14, 16]. ML provides a rich module structure where implementations are separated from interfaces (similarly to Modula-3), and where modules can parameterize over their imports. Such parameterized modules can then be instantiated by applying them to a collection of imports matching the required interface. So ML follows the model of programming-in-the-large as functional programming: a module that imports other modules is represented by a parameterized module, mapping from imports to exports, and linking is represented by the application of a parameterized module to its imports. This work builds on earlier work in algebraic specifications, where module interfaces are algebraic theories, module implementations are models of such theories, and modules are matched to interfaces by "view morphisms" from theory to model [19].

A weakness of the ML module system is that it does not handle circular imports, where a parameterized module may export some definitions that are required by its imports. So for example two mutually recursive types, or two mutually recursive functions, must be defined in the same module and cannot be defined in separate modules. This lack can be seen as symptomatic of a more general ability that would be desirable in a module language: to not only allow mutually recursive definitions to be split into separate modules, but to actually allow a recursive definition itself to be fragmented and the fragments to be defined in separate modules.

Duggan and Sourelis [14] proposed mixin modules as an extension of the ML module system that allowed exactly this kind of decomposition to be performed. In ML types

$$\frac{\mathit{mixinVars}(X.Y) = (\overline{B}\ \overline{x} = \overline{e})}{X \vdash (\mathit{new}\ \mathit{This}.Y()) \implies (\mathit{new}\ X.Y(\overline{e}))} \qquad \text{(CRed New This)}$$

$$(\mathit{this},\ (v),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (v,\ (v),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \qquad \text{(Red This)}$$

$$\frac{(\overline{B}\ \overline{x}) = \mathit{classVars}(X.Y)}{((\mathit{new}\ X.Y(\overline{v})).x_i,\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (v_i,\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M})} \qquad \text{(Red Access)}$$

$$\frac{\begin{array}{c}\mathit{classMeth}(X.Y, f) = (B\ f(\overline{B}\ \overline{x})\{\mathit{return}(e)\}) \\ v \equiv (\mathit{new}\ X.Y(\overline{v})) \quad X \vdash e \Longrightarrow e' \quad e'' \equiv (\mathit{return}(\{\overline{v'}/\overline{x}\}e'))\end{array}}{(v.f(\overline{v'}),\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e'',\ (v_{\mathit{this}}, v),\ (B, X.Y),\ (W, X),\ (\mathcal{P}, \mathit{mobj}),\ (\mathcal{M}, ()))}$$
(Red Invoke)

$$\frac{(\overline{B}\ \overline{x} = \overline{e}) = \mathit{mixinVars}(X.Y)}{(A{::}(\mathit{new}\ X.Y(\overline{v})).x_i,\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (v_i,\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M})} \qquad \text{(Red MAccess)}$$

$$\frac{v \equiv \mathit{new}\ X.Y(v_1, \ldots, v_i, \ldots, v_k) \quad (\overline{B}\ \overline{x} = \overline{e}) = \mathit{mixinVars}(X.Y)}{(A{::}v.x_i = v',\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (\mathit{new}\ X.Y(v_1, \ldots, v', \ldots, v_k)),\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M})}$$
(Red MUpdate)

$$\frac{\begin{array}{c}\mathit{mixinMeth}(X.Y, f) = (A\ f(\overline{A}\ \overline{x})\{\mathit{return}(e)\},\ m,\ \overline{m'}) \\ v \equiv (\mathit{new}\ X.Y(\overline{v})) \quad X \vdash e \Longrightarrow e' \quad e'' \equiv (\mathit{return}(\{\overline{v'}/\overline{x}\}e'))\end{array}}{(A{::}v.f(\overline{v'}),\ (v_{\mathit{this}}),\ (B),\ (W),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e'',\ (v_{\mathit{this}}, v),\ (B, X.Y),\ (W, X),\ (\mathcal{P}, m),\ (\mathcal{M}, (\overline{m'})))}$$
(Red MInvoke)

$$\frac{\begin{array}{c}\mathcal{P} \equiv (m) \quad \mathit{mixinMeth}(f, m) = (A\ f(\overline{A}\ \overline{x})\{\mathit{return}(e)\},\ m',\ (\overline{m''})) \quad \mathcal{P}' \equiv (m, m') \\ X \vdash e \Longrightarrow e' \quad e'' \equiv \mathit{return}(\{\overline{v}/\overline{x}\}e') \quad \mathcal{M} \equiv ((\overline{m'''})) \quad \mathcal{M}' \equiv ((\overline{m'''}), (\overline{m''}, \overline{m'''})))\end{array}}{(A{::}\mathit{super}.f(v, \overline{v}),\ (v_{\mathit{this}}),\ (B),\ (X),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e'',\ (v_{\mathit{this}}, v_{\mathit{this}}),\ (B, B),\ (X, X),\ \mathcal{P}',\ \mathcal{M}')}$$
(Red Super)

$$\frac{\begin{array}{c}\mathcal{M} \equiv ((\overline{m})) \quad \mathit{mixinMeth}(f, \overline{m}) = (A\ f(\overline{A}\ \overline{x})\{\mathit{return}(e)\},\ m',\ (\overline{m''})) \quad \mathcal{M}' \equiv ((\overline{m}), (\overline{m''})) \\ X \vdash e \Longrightarrow e' \quad e'' \equiv \mathit{return}(\{\overline{v}/\overline{x}\}e') \quad \mathcal{P} \equiv (m''') \quad \mathcal{P}' \equiv (m''', m'))\end{array}}{(A{::}\mathit{inner}.f(\overline{v}),\ (v_{\mathit{this}}),\ (B),\ (X),\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e'',\ (v_{\mathit{this}}, v_{\mathit{this}}),\ (B, B),\ (X, X),\ \mathcal{P}',\ \mathcal{M}')} \qquad \text{(Red Inner)}$$

$$\frac{e_0 \equiv \mathit{return}(e) \quad (e,\ \overline{v},\ \overline{A},\ \overline{W},\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e',\ \overline{v'},\ \overline{A'},\ \overline{W'},\ \mathcal{P}',\ \mathcal{M}') \quad e_0' \equiv \mathit{return}(e')}{(e_0,\ (v, \overline{v}),\ (A, \overline{A}),\ (W, \overline{W}),\ (m, \mathcal{P}),\ ((\overline{m}), \mathcal{M})) \longrightarrow (e_0',\ (v, \overline{v'}),\ (A, \overline{A'}),\ (W, \overline{W'}),\ (m, \mathcal{P}'),\ ((\overline{m}), \mathcal{M}'))}$$
(Red Ret Ctxt)

$$(\mathit{return}(v'),\ (\overline{v}, v),\ (\overline{A}, A),\ (\overline{W}, X),\ (\mathcal{P}, m),\ (\mathcal{M}, (\overline{m}))) \longrightarrow (v',\ \overline{v},\ \overline{A},\ \overline{W},\ \mathcal{P},\ \mathcal{M}) \qquad \text{(Red Return)}$$

$$\frac{(e,\ \overline{v},\ \overline{A},\ \overline{W},\ \mathcal{P},\ \mathcal{M}) \longrightarrow (e',\ \overline{v},\ \overline{A'},\ \overline{W'},\ \mathcal{P}',\ \mathcal{M}')}{(E[e],\ \overline{v},\ \overline{A},\ \overline{W},\ \mathcal{P},\ \mathcal{M}) \longrightarrow (E[e'],\ \overline{v},\ \overline{A'},\ \overline{W'},\ \mathcal{P}',\ \mathcal{M}')} \qquad \text{(Red Context)}$$

Figure 9: Reduction Semantics

are datatypes, essentially named variant (or discriminated union) types. Mixin modules allowed a datatype, and a function defined over that datatype, to be fragmented into separate modules. A new module composition operation then allowed the fragments of type and function definitions in two modules to be coalesced into a new module. The semantics of this were based on the semantics of mixin-based composition for object-oriented languages developed by Bracha and Cook [5]. Bracha and Lindstrom [6] also developed a generalization of the notion of mixins, which they called "modules" and which were collections of mutually recursive functions that could be combined via mixin composition with other modules. The latter work did not consider modules containing type components, nor did it consider the typing aspects of the language.

The current form of mixin modules has at least one compelling advantage over the functional approach: in the functional approach it is difficult to ensure that a case statement examining a value of a variant type will never fail for an unmatched case, since the variant type may be extended by mixin module combination. This is not a problem in the object-oriented approach, since the branches of the case can be distributed to the classes that provide the components of such a union type (subclassing from a common abstract base class).

The aspect that is in common with both Duggan and Sourelis' mixin modules, and Bracha and Lindstrom's modules, is this: Inheritance is now understood semantically (thanks to the work of Cook [10, 9]) as the incremental extension of the fixed point of a recursive definition, The aforesaid work generalizes this to a form of inheritance that incrementally extends the fixed points of several mutually recursive definitions. In the work of Duggan and Sourelis, these are the fixed points of recursive types and recursive functions over values of those types. In the work of Bracha and Lindstrom, these are the fixed points of mutually recursive object generators.

The need to consider inheritance over more than one class has been recognized in the literature. A standard example is that of the subject-observer pattern, where the subject has a list of observers and each observer has a reference to the subject. If a generic subject-observer pattern is provided as an application framework, a user of this framework would like to specialize both the subject and observer classes for example to window subjects and window observers, and have the classes specialized accordingly (a window observer object can only observe a window subject). An informal proposal for an extension to Java was given by Bruce, Odersky and Wadler [7], and a formal semantics was given by Bruce [8].

Recursive modules have also been a focus of research [16, 18, 11, 2, 3, 37]. All of this work has been based on allowing mutually recursive definitions (sometimes including type definitions) to be defined in separate modules. This work has not considered issues concerned with inheritance and subtyping, and their interaction with the combination of mutually recursive modular definitions. Although Ancona and Zucca refer to their work as "mixin modules," in fact their mixin modules are collections of datatypes and functions defined over values of those datatypes, rather than the more general form of decomposition considered by Duggan and Sourelis [14]. Recent work on component languages for classes has been based on some of this work [33], with particular emphasis on dynamic loading. More recently Duggan [13] has considered an algebra for manipulating recursive modules, based on the operations of process algebras such as CCS. So combining modules is a form of parallel composition, and there are also operations for renaming module fields and for coercively hiding module fields. Analogous operations are considered by Ancona and Zucca [3] and Wells and Vestergaard [37]. The module calculus considered here has parallel composition (module combination) and renaming, but not coercive hiding, because the Java language has no notion of package types separate from package implementations. While being a fundamental limitation of the language, it also greatly simplifies the work presented here.

Besides the work of Bracha and Cook on mixin-based inheritance, more recent work has looked at flexible constructs for combining mixins. A central problem is that, because mixins are defined separately, it is difficult to avoid field name conflicts when combining mixins. Flatt et al [17] solve this problem by having every access to a mixin object be made relative to an interface. However this approach does not allow a field to be shared between different mixins, and the aforesaid work on mixin-based definition of modular interpreters makes critical use of shared fields. They also do not consider the issues with specializing the field types of mixins considered here. Findler and Flatt [15] consider the combination of mixins with recursive modules. Their approach amounts to the form of modular decomposition of variant types that was part of the motivation for Duggan and Sourelis' mixin modules. They do not consider the form of mixin-based combination of record definitions and the attendant issues of field type specialization that are a critical consideration of this work (providing a motivation for the distinction between classes and mixins that is not considered by others).

Ancona et al [1] have considered the addition of mixins (but not mixin modules) to the full Java languages, although there some critical issues regarding field name conflicts are left unresolved.

Duggan [12] describes an approach to defining modular interpreters in an object-oriented fashion, and speculates on the form of language support required. Mixin modules, as described here, are capable of representing that example. Specifically the types of abstract syntax trees can be represented using classes (abstract base class and derivatives) while the input and output types of the interpreter function can be represented as mixins. Each interpreter fragment can then be represented as a mixin module containing both class and mixin definitions. Mixin module combination coalesces the definitions in a type-safe manner.

Related to the latter work, the JTS project [4, 35] provides support for reusing abstract syntax trees and operations on those trees, with an emphasis on "hygienic" syntactic ma-

nipulations of ASTs. An interesting point of comparison is that the JTS project recognizes the importance of mixin-based inheritance, at the granularity of collections of classes rather than individual classes, for reusing DSLs. For example, if a language syntax has nonterminals for expressions and statements, then mixin-based inheritance for language fragments must be able to simultaneously inherit from both expression and statement classes. This is certainly a part of the framework described in [12]; however the latter framework goes beyond this, to considering inheritance for semantic definitions and the proper composition mechanisms that allow modular semantic definitions to be combined via inheritance. The current work in addition shows that this composition can be statically checked for type safety.

6. CONCLUSIONS

We have described a kernel language for an extension of a class-based language with mixin modules. This extension combines the parallel extension of classes via inheritance (with method type specialization for static type safety) with mixin-based inheritance. The intended application of this approach is in supporting application framework development and its use in a flexible and type-safe manner, using the module combination and extension operations of mixin modules. We are in the process of implementing this design in a compiler for Java, and we are investigating its application in a concrete application domain for which Java has a good chance of being an application development language, that of portable XML processors.

Regarding portability, it is certainly the case that our extensions, if carried through to the JVM instruction set, would make programs highly non-portable. However at runtime only a finished mixin module would be downloaded. Because of the self-contained nature of a mixin module (for example, all subtype relations are local), it is possible to translate from a mixin module to a collection of Java classes: class hierarchies are translated unchanged, while mixin hierarchies are collapsed into a Java class without superclasses. We are pursuing this approach in our implementation work.

7. ACKNOWLEDGEMENTS

This work was supported by a grant from the New Jersey Commission on Science and Technology. Thanks to Phil Wadler for an interesting NJPLS talk. Thanks to David Naumann for helpful discussions.

8. REFERENCES

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam: A smooth extension of Java with mixins. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Cannes, France, June 2000. Springer-Verlag.

[2] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[3] D. Ancona and E. Zucca. A primitive calculus for module systems. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, Paris, France, September 1999. Springer-Verlag.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the International Conference on Software Reuse*. ACM Press, 1998.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, pages 303–311. ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10.

[6] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages*, pages 282–290. IEEE, 1992.

[7] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, Brussels, July 1998. Extended abstract in: Workshop on Foundations of Object-Oriented Languages(FOOL 5), San Diego, January 1998.

[8] K. B. Bruce and J. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Mathematical Foundations of Programming Semantics (MFPS)*, 1999.

[9] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming.* The MIT Press, 1994.

[10] W. Cook and J. Palsberg. A denotational semantics for inheritance and its correctness. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1989.

[11] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999. ACM Press.

[12] D. Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Cannes, France, June 2000. Springer-Verlag.

[13] D. Duggan. Shared in typed module assembly language. In *Workshop on Types in Compilation*, Montreal, Quebec, Canada, September 2000.

[14] D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of ACM International Conference on Functional Programming*, pages 262–273, 1996.

[15] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ACM International Conference on Functional Programming*, 1999.

[16] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1998.

[18] N. Glew and G. Morrisett. Type-safe linking and modular assembly languages. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. ACM Press.

[19] J. A. Goguen. Principles of parameterized programming. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability I: Concepts and Models*, chapter 7, pages 159–226. ACM Press, 1989.

[20] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.

[21] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In P. Hudak, editor, *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 341–354. Association for Computing Machinery, 1990.

[22] J. D. Ichbiach, J. G. P. Barnes, R. J. Firth, and M. Woodger. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6B), 1979. Special issue.

[23] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A core calculus for Java and GJ. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, Denver, CO, 1999. ACM Press.

[24] A. Igarashi and B. C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming*, 2000.

[25] Intermetrics, Cambridge, Mass. *Ada-95 Reference Manual*, 1995. International standard ISO/IEC 8652:1995(E).

[26] B. Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[27] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994. acmp.

[28] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–163, San Francisco, California, January 1995. ACM Press.

[29] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Lecture Notes in Computer Science. Springer-Verlag, 1981.

[30] G. Nelson. *Systems Programming in Modula-3*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, 1991.

[31] J. Riecke and C. Stone. Privacy via subsumption. In *Workshop on Foundations of Object-Oriented Languages*, 1998. To appear in *Information and Computation.*

[32] P. Rovner. On extending Modula-2+ to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[33] J. Seco and L. Caires. A basic model of typed components. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[34] Z. Shao. Transparent modules with fully syntactic signatures. In *Proceedings of ACM International Conference on Functional Programming*, Paris, France, September 1999.

[35] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, 1998.

[36] P. Wadler. The next 700 markup languages. In *USENIX Conference on Domain-Specific Languages*, October 1999. Invited talk.

[37] J. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *European Symposium on Programming*, Berlin, Germany, April 2000. Springer-Verlag.